



Wzorce projektowe

Łukasz Bednarski



Definicja

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

- **Kreacyjne (konstrukcyjne)** – opisujące proces tworzenia nowych obiektów; ich zadaniem jest tworzenie, inicializacja oraz konfiguracja obiektów, klas oraz innych typów danych.
- **Strukturalne** – opisujące struktury powiązanych ze sobą obiektów.
- **Czynnościowe** – opisujące zachowanie i odpowiedzialność współpracujących ze sobą obiektów.



Literatura

Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

WZORCE PROJEKTOWE

Elementy oprogramowania
obiektowego wielokrotnego użytku



NAUCZ SIĘ WYKORZYSTYWAĆ WZORCE
PROJEKTOWE I UŁATW SOBIE PRACĘ!
Jak wykorzystać projekty, które już wcześniej okazały się dobre?
Jak stworzyć elastyczny projekt obiektowy?
Jak sprawnie rozwiązywać typowe problemy projektowe?





Budowniczy

Wzorzec konstrukcyjny



Budowniczy

Budowniczy jest wzorcem, gdzie proces tworzenia obiektu podzielony jest na kilka mniejszych etapów, a każdy z nich może być implementowany na wiele sposobów. Dzięki takiemu rozwiązaniu możliwe jest tworzenie różnych reprezentacji obiektów w tym samym procesie konstrukcyjnym.

Jego celem jest rozdzielenie sposobu tworzenia obiektu od jego implementacji. Sposób tworzenia obiektów zamknięty jest w oddzielnego obiektach, tzw. konkretnych budowniczych.



Budowniczy - zalewy/wady

Zalety

1. Duża możliwość zróżnicowania wewnętrznych struktur klas.
2. Duża skalowalność (dodawanie nowych reprezentacji obiektów jest uproszczone).
3. Większa możliwość kontrolowania tego, w jaki sposób tworzony jest obiekt (proces konstrukcyjny jest niezależny od elementów, z których składa się tworzony obiekt).

Wady:

1. Duża liczba obiektów reprezentujących konkretne produkty.
2. Nieumiejęte użycwanie wzorca może spowodować nieczytelność kodu (jeden produkt może być tworzony przez zbyt wielu budowniczych).



Budowniczy - Przykład

```
public class Student {  
  
    private String name;  
    private String lastName;  
    private String salutation;  
    private int age;  
    private List<String> language;  
  
    public Student(String name, String lastName, String salutation, int age, List<String> language) {  
        this.name = name;  
        this.lastName = lastName;  
        this.salutation = salutation;  
        this.age = age;  
        this.language = language;  
    }  
  
    public Student(String name, String lastName, String salutation, int age) {  
        this.name = name;  
        this.lastName = lastName;  
        this.salutation = salutation;  
        this.age = age;  
    }  
  
    public Student(String name, String lastName, String salutation) {  
        this.name = name;  
        this.lastName = lastName;  
        this.salutation = salutation;  
    }  
}
```



Budowniczy - Przykład

```
public class Student {  
    private String name;  
    private String lastName;  
    private String salutation;  
    private int age;  
    private List<String> language;  
  
    public static class Builder {  
        private String name;  
        private String lastName;  
        private String salutation;  
        private int age;  
        private List<String> language;  
  
        public Builder name(String name) {  
            this.name = name;  
            return this;  
        }  
  
        public Builder lastname(String lastName) {  
            this.lastName = lastName;  
            return this;  
        }  
  
        public Builder salutation(String salutation) {  
            this.salutation = salutation;  
            return this;  
        }  
  
        public Builder age(int age) {  
            this.age = age;  
            return this;  
        }  
  
        public Builder language(List<String> language) {  
            this.language = language;  
            return this;  
        }  
  
        public Student build() {  
            return new Student(this);  
        }  
  
        private Student(Builder builder) {  
            name = builder.name;  
            lastName = builder.lastName;  
            salutation = builder.salutation;  
            age = builder.age;  
            language = builder.language;  
        }  
    }  
}
```



Budowniczy - Przykład

```
public static void main(String[] args) {  
  
    Student jan = new Student.Builder()  
        .name("Jan")  
        .age(18)  
        .language(Arrays.asList("chinese", "english"))  
        .build();  
  
    Student kamil = new Student.Builder()  
        .name("Kamil")  
        .lastname("Nowak")  
        .language(Arrays.asList("chinese", "english"))  
        .build();  
  
}
```



Budowniczy - zadanie

1. Utwórz klasę Room w której znajdzie się kilka pól: nr pokoju, liczba łóżek, czy pokój jest dostępny, czy pokój jest czysty, czy znajduje się łazienka, powierzchnia pokoju.
2. Utwórz klasę RoomBuilder i zaimplementuj ją.
3. Rozbuduj klasę budowniczego w taki sposób aby pola nr pokoju oraz powierzchnia były polami wymaganymi podczas tworzenia obiektu (parametry powinny być przyjęte w konstruktorze budowniczego).



Fabryka Abstrakcji

Wzorzec konstrukcyjny



Fabryka Abstrakcji

Fabryka abstrakcyjna jest wzorcem projektowym, którego zadaniem jest określenie interfejsu do tworzenia różnych obiektów należących do tego samego typu (rodziny). Interfejs ten definiuje grupę metod, za pomocą których tworzone są obiekty.

Celem jest przygotowanie interfejsu do tworzenie obiektów jednego typu.

Sposób tworzenia obiektów zamknięty jest w oddzielnych obiektach, tzw. konkretnych budowniczych.



Fabryka Abstrakcji - przykład

```
abstract class GUIFactory {  
    public static GUIFactory getFactory() {  
        int sys = readFromConfigFile("OS_TYPE");  
        if (sys == 0) {  
            return new WinFactory();  
        } else {  
            return new OSXFactory();  
        }  
    }  
  
    public abstract Button createButton();  
}
```

```
abstract class Button {  
    public abstract void paint();  
}  
class WinButton extends Button {  
    public void paint() {  
        System.out.println("Przycisk WinButton");  
    }  
}  
class OSXButton extends Button {  
    public void paint() {  
        System.out.println("Przycisk OSXButton");  
    }  
}  
  
class WinFactory extends GUIFactory {  
    public Button createButton() {  
        return new WinButton();  
    }  
}  
class OSXFactory extends GUIFactory {  
    public Button createButton() {  
        return new OSXButton();  
    }  
}
```



Fabryka Abstrakcji - przykład

```
public class App {  
    public static void main(String[] args) {  
        GUIFactory factory = GUIFactory.getFactory();  
        Button button = factory.createButton();  
        button.paint();  
    }  
    // Wyświetlony zostanie tekst:  
    //   "Przycisk WinButton"  
    // lub:  
    //   "Przycisk OSXButton"  
}
```



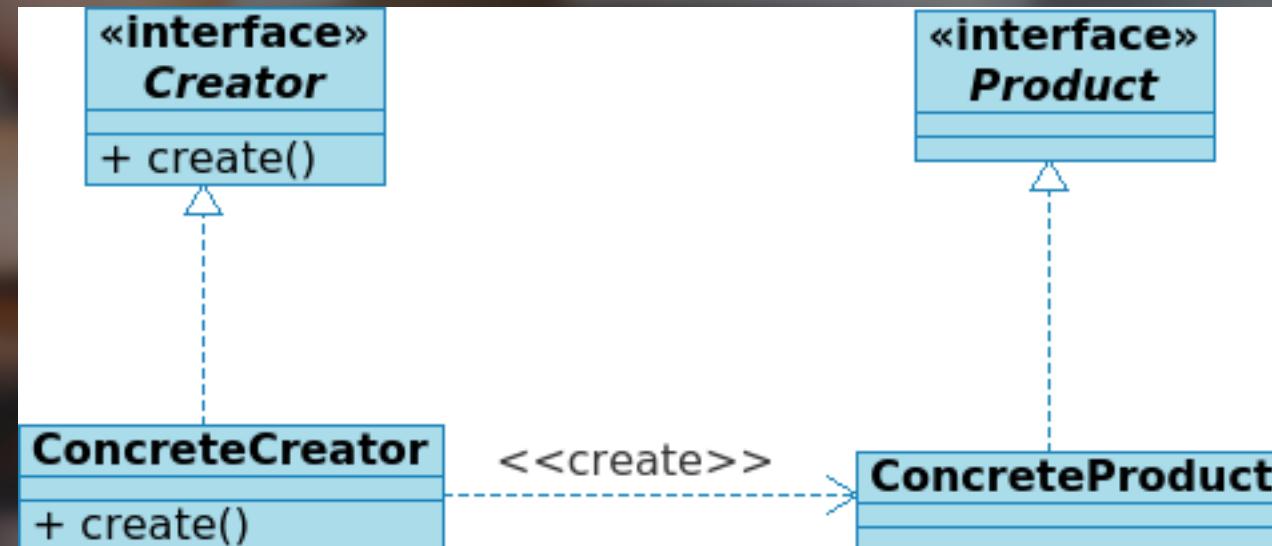
Factory Method

Wzorzec konstrukcyjny



Factory Method

Wzorzec **metody wytwórczej** dostarcza abstrakcji do tworzenia obiektów nieokreślonych, ale powiązanych typów. Umożliwia także dziedziczącym klasom decydowanie jakiego typu ma to być obiekt.





Factory Method

```
abstract class Pizza {  
    public abstract double getPrice();  
}  
  
class HamAndMushroomPizza extends Pizza {  
    private double price = 8.5;  
  
    public double getPrice() {  
        return price;  
    }  
}  
  
class DeluxePizza extends Pizza {  
    private double price = 10.5;  
  
    public double getPrice() {  
        return price;  
    }  
}  
  
class HawaiianPizza extends Pizza {  
    private double price = 11.5;  
  
    public double getPrice() {  
        return price;  
    }  
}
```

```
class PizzaFactory {  
    public enum PizzaType {  
        HamMushroom,  
        Deluxe,  
        Hawaiian  
    }  
  
    public static Pizza createPizza(PizzaType pizzaType) {  
        switch (pizzaType) {  
            case HamMushroom:  
                return new HamAndMushroomPizza();  
            case Deluxe:  
                return new DeluxePizza();  
            case Hawaiian:  
                return new HawaiianPizza();  
        }  
        throw new IllegalArgumentException("The pizza type " +  
            pizzaType + " is not recognized.");  
    }  
}
```



Factory Method

```
class PizzaLover {  
    /**  
     * Create all available pizzas and print their prices  
     */  
    public static void main (String args[]) {  
        for (PizzaFactory.PizzaType pizzaType :  
            PizzaFactory.PizzaType.values()) {  
            System.out.println("Price of " + pizzaType + " is " +  
                PizzaFactory.createPizza(pizzaType).getPrice());  
        }  
    }  
}
```



Factory Method- zadanie

Utwórz interface Car z metodą getFuel oraz dodaj 3 jego implementacje klasyfikując je według zużywanego paliwa - diesel, benzyna i elektryczny (np. DieselCar itd). Implementacja metody powinna zwracać String (lub enumerator) jakiego rodzaju paliwo zużywa pojazd.

Dodaj klasę FactoryCar z metodą getCar której jako parametr przekazany zostanie string (bądź enumerator) jakiego rodzaju samochód potrzebujesz. Metoda ta zwróci odpowiedni obiekt, np:

```
Car dieselCar = CarFactory.getCar(„disel”);
```

Dodaj testy które zweryfikują poprawność działania fabryki.



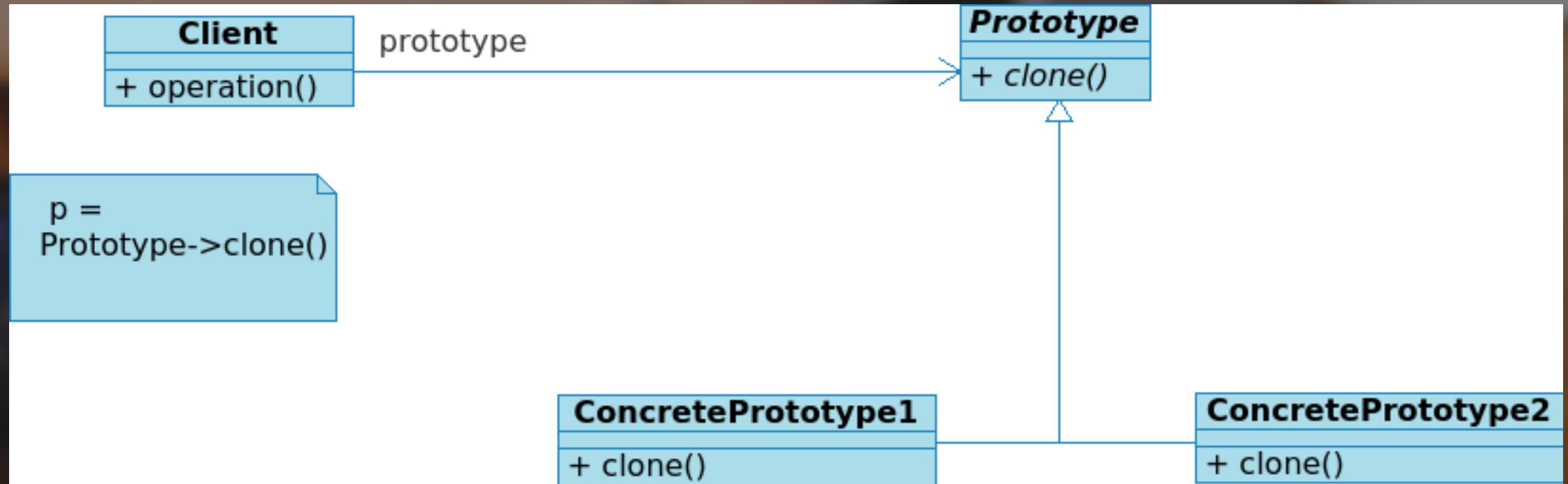
Prototyp

Wzorzec konstrukcyjny



Prototyp

Prototyp jest wzorcem, opisującym mechanizm tworzenia nowych obiektów poprzez klonowanie jednego obiektu macierzystego. Mechanizm klonowania wykorzystywany jest wówczas, gdy należy wykreować dużą liczbę obiektów tego samego typu lub istnieje potrzeba tworzenia zbioru obiektów o bardzo podobnych właściwościach.





Prototyp

```
public class Employees implements Cloneable{
    private List<String> empList;

    public Employees(){
        empList = new ArrayList<String>();
    }

    public Employees(List<String> list){
        this.empList=list;
    }
    public void loadData(){
        //read all employees from database and put into the list
        empList.add("Pankaj");
        empList.add("Raj");
        empList.add("David");
        empList.add("Lisa");
    }

    public List<String> getEmpList() {
        return empList;
    }

    @Override
    public Object clone() {
        List<String> temp = new ArrayList<String>();
        for(String s : this.getEmpList()){
            temp.add(s);
        }
        return new Employees(temp);
    }
}
```



Prototyp

```
public static void main(String[] args) throws CloneNotSupportedException {
    Employees emps = new Employees();
    emps.loadData();

    Employees empsNew = (Employees) emps.clone();
    Employees empsNew1 = (Employees) emps.clone();
    List<String> list = empsNew.getEmpList();
    list.add("John");
    List<String> list1 = empsNew1.getEmpList();
    list1.remove("Pankaj");

    System.out.println("emps List: "+emps.getEmpList());
    System.out.println("empsNew List: "+list);
    System.out.println("empsNew1 List: "+list1);
}
```



Prototyp

1. Utwórz klasę User (implementacja nie ma znaczenia ale możesz dodać 2 pola - login i password).
2. Klasa User powinna implementować interface Cloneable i nadpisywać metodę clone.
- 3. Nie nadpisuj metody `toString()`;**
4. Utwórz test w którym sprawdzisz różnicę między:

```
User user1 = new User("login", "psw");  
  
user2 = user1;  
User user3 = (User) user1.clone();
```



Singleton

Wzorzec konstrukcyjny



Singleton

Singleton jest jednym z najprostszych wzorców projektowych. Jego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji oraz zapewnienie globalnego dostępu do stworzonego obiektu – jest to obiektowa alternatywa dla zmiennych globalnych.

Singleton implementuje się poprzez stworzenie klasy, która posiada statyczną metodę **getInstance()**. Metoda ta sprawdza, czy istnieje już instancja tej klasy, jeżeli nie – tworzy ją i przechowuje jej referencję w prywatnym polu. Aby uniemożliwić tworzenie dodatkowych instancji, konstruktor klasy deklaruje się jako prywatny lub chroniony.

Singleton	
-	instance : Singleton
-	Singleton()
+	<u>getInstance() : Singleton</u>



Singleton - zalety/wady

Zalety:

- Pobieranie instancji klasy jest niewidoczne dla użytkownika. Nie musi on wiedzieć, czy w chwili wywołania metody instancja istnieje czy dopiero jest tworzona.
- Tworzenie nowej instancji zachodzi dopiero przy pierwszej próbie użycia.
- Klasa zaimplementowana z użyciem wzorca singleton może samodzielnie kontrolować liczbę swoich instancji istniejących w aplikacji.

Wady:

- Brak elastyczności, ponieważ już na poziomie kodu, na „sztywno” określana jest liczba instancji klasy.
- Utrudnia testowanie i usuwanie błędów w aplikacji.



Singleton - Implementacja

```
public class Config {  
    private static Config config;  
    private String language = "PL";  
  
    //Uniemożliwienie tworzenie obiektu przez  
domyślny konstruktor  
    private Config() {  
    }  
  
    //Jeśli instancja nieistnieje to ją utworzy a  
później za każdym razem będzie zwracał tą samą.  
    public static Config getInstance(){  
        if (config == null) {  
            config = new Config();  
        }  
  
        return config;  
    }  
  
    public void setLanguage(String language) {  
        this.language = language;  
    }  
  
    public String getLanguage() {  
        return language;  
    }  
}
```

```
@Test  
public void testSingleton() {  
    Config config = Config.getInstance();  
  
    assertEquals("PL", config.getLanguage());  
  
    config.setLanguage("DE");  
  
    Config newConfig = Config.getInstance();  
  
    assertEquals("DE", newConfig.getLanguage());  
}
```



Singleton - Zadanie

Zaimplementuj klasę Session która będzie sesją, służącą do przechowywania zalogowanego użytkownika. Dodaj 2 metody:

- loginUser(user) - zapisze zalogowanego użytkownika (np. w postaci jakiejś zmiennej loggedUser)
- logoutUser(user) - usunie zapisanego użytkownika.

Wykonaj test sprawdzający poprawność działania klasy i zaimplementowanych metod.



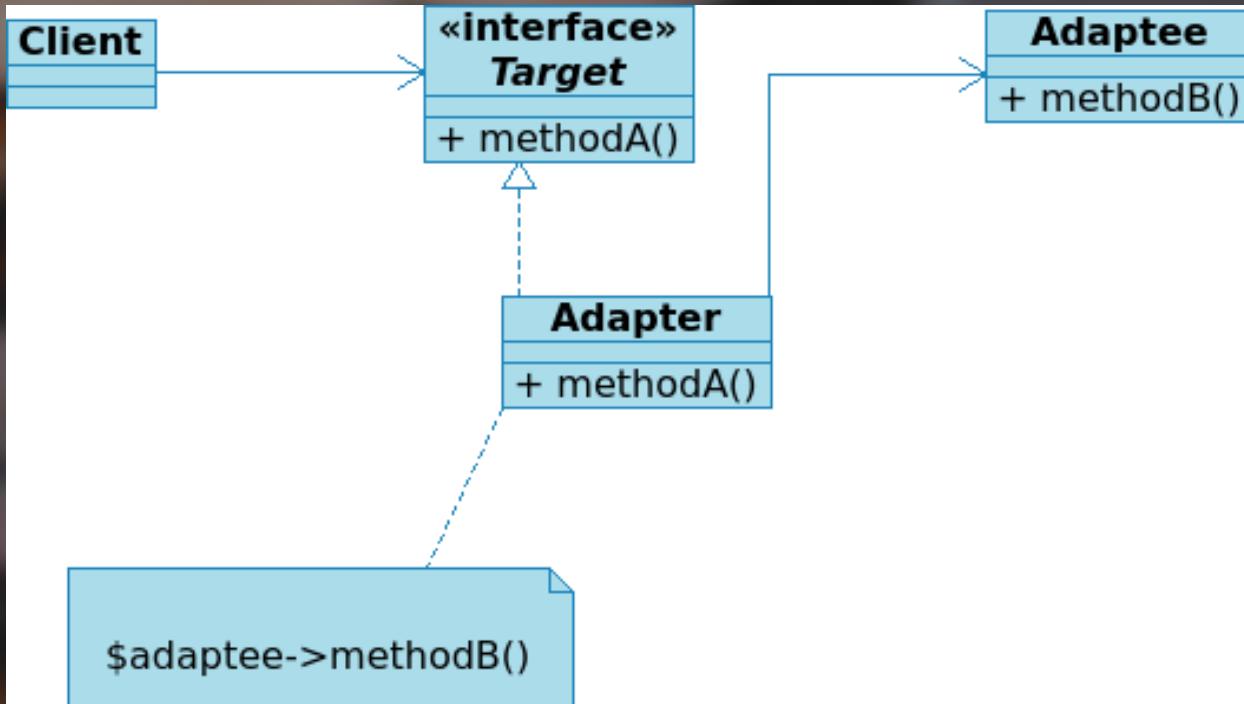
Adapter

Wzorzec strukturalny



Adapter

Wzorzec **adapter** (znany także pod nazwą **wrapper**) służy do przystosowania interfejsów obiektowych, tak aby możliwa była współpraca obiektów o niezgodnych interfejsach. Szczególnie przydaje się przypadku wykorzystania gotowych bibliotek o interfejsach niezgodnych ze stosowanymi w aplikacji. W świecie rzeczywistym adapter to przejściówka, np. przejściówka do wtyczki gniazdka angielskiego na polskie.





Adapter

```
class NewWriter{  
    public void save(String data, int color){  
        System.out.println("Save data: " + data +  
" color: " + color);  
    }  
}
```

```
interface Writer{  
    public void save(String data);  
}
```

```
class DataWriter implements Writer{  
  
    @Override  
    public void save(String data){  
  
        NewWriter adapter = new NewWriter();  
        adapter.save(data, 0);  
    }  
}
```

```
public class DesignPatternsAdapter{  
  
    public static void main(String[] args){  
  
        Writer writer = new DataWriter();  
        writer.save("Super value");  
    }  
}
```



Dekorator

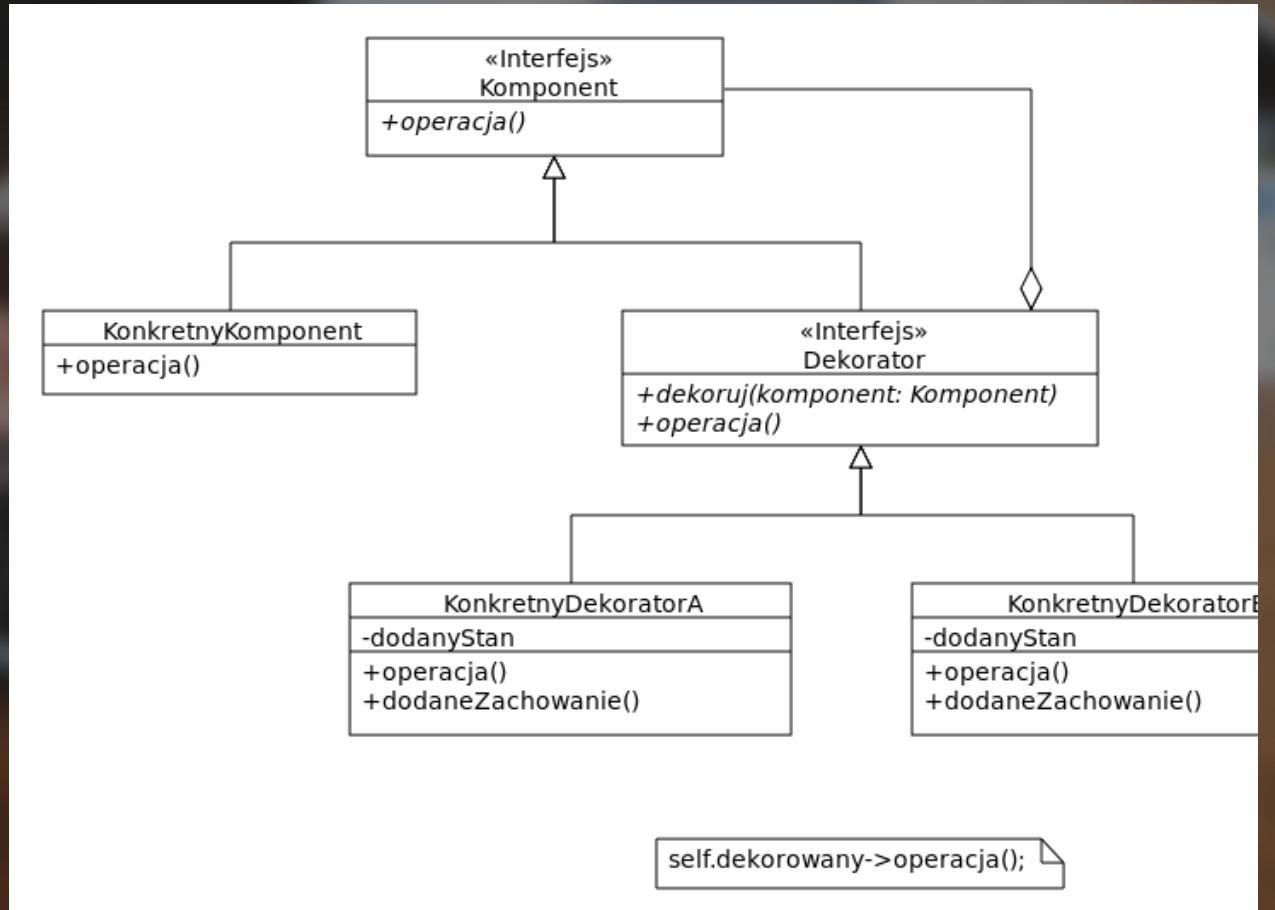
Wzorzec strukturalny



Dekorator

Jego zadaniem jest rozszerzenie funkcjonalności obiektu poprzez dynamiczne dołączenie dodatkowych zachowań.

- Wzorzec ten pozwala na "dekorowanie" zachowania klasy, czyli zmianę jej funkcjonalności bez potrzeby dziedziczenia.
- Jego działanie można upodobnić do dziedziczenia z tą różnicą, że dziedziczenie rozszerza zachowanie klasy w trakcie komplikacji a dekoratory rozszerzają klasy w czasie działania programu.





Dekorator

```
/**  
 * Interfejs wzorca dekorator  
 */  
interface Decorator{  
    String getText();  
}  
  
/**  
 * Główna klasa, która nic nie wie że jest dekorowana  
 */  
class RawText implements Decorator{  
  
    public String getText(){  
        return "Super duper text ...";  
    }  
}
```

```
// Formatowanie tekstu przy pomocy tagu HTML  
class HtmlFormatText implements Decorator{  
    private Decorator decoratorObject;  
  
    public HtmlFormatText(Decorator decoratorObject){  
        this.decoratorObject = decoratorObject;  
    }  
  
    public String getText(){  
        return "&lt; div&gt;" + decoratorObject.getText() +  
        "&lt; /div&gt;";  
    }  
}  
  
// Zmiana formatowania tekstu, symulacja konsoli linuxowej  
class ConsoleFormatText implements Decorator{  
    private Decorator decoratorObject;  
  
    public ConsoleFormatText(Decorator decoratorObject){  
        this.decoratorObject = decoratorObject;  
    }  
  
    public String getText(){  
        return "~linux@login: "+decoratorObject.getText();  
    }  
}
```



Dekorator - przykład

```
public class DesignPatternsDecorator{  
    public static void main(String[] args){  
        RawText rawText=new RawText();  
  
        ConsoleFormatText consoleFormatText=new ConsoleFormatText(rawText);  
        HtmlFormatText htmlFormatText=new HtmlFormatText(rawText);  
  
        System.out.println("\nRaw text:");  
        System.out.println(rawText.getText());  
        System.out.println("\nConsole format text:");  
        System.out.println(consoleFormatText.getText());  
        System.out.println("\nHtml format text:");  
        System.out.println(htmlFormatText.getText());  
    }  
}
```



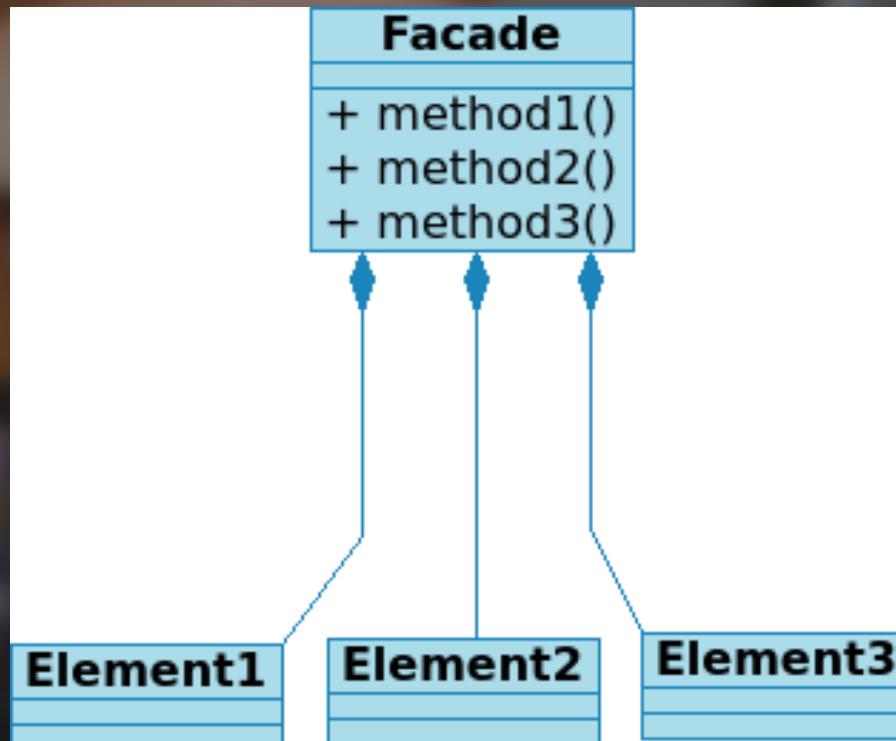
Fasada

Wzorzec strukturalny



Fasada

Fasada służy do ujednolicenia dostępu do złożonego systemu poprzez udostępnienie uproszczonego i uporządkowanego interfejsu programistycznego. Fasada zwykle implementowana jest w bardzo prosty sposób – w postaci jednej klasy powiązanej z klasami reprezentującymi system, do którego klient chce uzyskać dostęp.





Fasada - przykład

```
class CPU {  
    public void freeze() { ... }  
    public void jump(long position  
... }  
    public void execute() { ... }  
}
```

```
class HardDrive {  
    public byte[] read(long lba, int  
size) { ... }  
}
```

```
class Memory {  
    public void load(long position,  
byte[] data) { ... }  
}
```

```
class ComputerFacade {  
    private CPU processor;  
    private Memory ram;  
    private HardDrive hd;  
  
    public ComputerFacade() {  
        this.processor = new CPU();  
        this.ram = new Memory();  
        this.hd = new HardDrive();  
    }  
  
    public void start() {  
        processor.freeze();  
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR,  
SECTOR_SIZE));  
        processor.jump(BOOT_ADDRESS);  
        processor.execute();  
    }  
}
```



Fasada - przykład

```
class ComputerExample {  
    public static void main(String[] args) {  
        ComputerFacade computer = new ComputerFacade();  
        computer.start();  
        computer.anotherMethod();  
    }  
}
```



Fasada - przykład 2

```
ublic interface Shape {  
    void draw();  
}  
  
public class Rectangle implements Shape {  
  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}  
  
public class Square implements Shape {  
  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}  
  
public class Circle implements Shape {  
  
    public void draw() {  
        System.out.println("Circle::draw()");  
    }  
}
```

```
public class ShapeMaker {  
    private Shape circle;  
    private Shape rectangle;  
    private Shape square;  
  
    public ShapeMaker() {  
        circle = new Circle();  
        rectangle = new Rectangle();  
        square = new Square();  
    }  
  
    public void drawCircle(){  
        circle.draw();  
    }  
    public void drawRectangle(){  
        rectangle.draw();  
    }  
    public void drawSquare(){  
        square.draw();  
    }  
}
```



Fasada - przykład 2

```
public class FacadePatternDemo {  
    public static void main(String[] args) {  
        ShapeMaker shapeMaker = new ShapeMaker();  
  
        shapeMaker.drawCircle();  
        shapeMaker.drawRectangle();  
        shapeMaker.drawSquare();  
    }  
}
```



Kompozyt

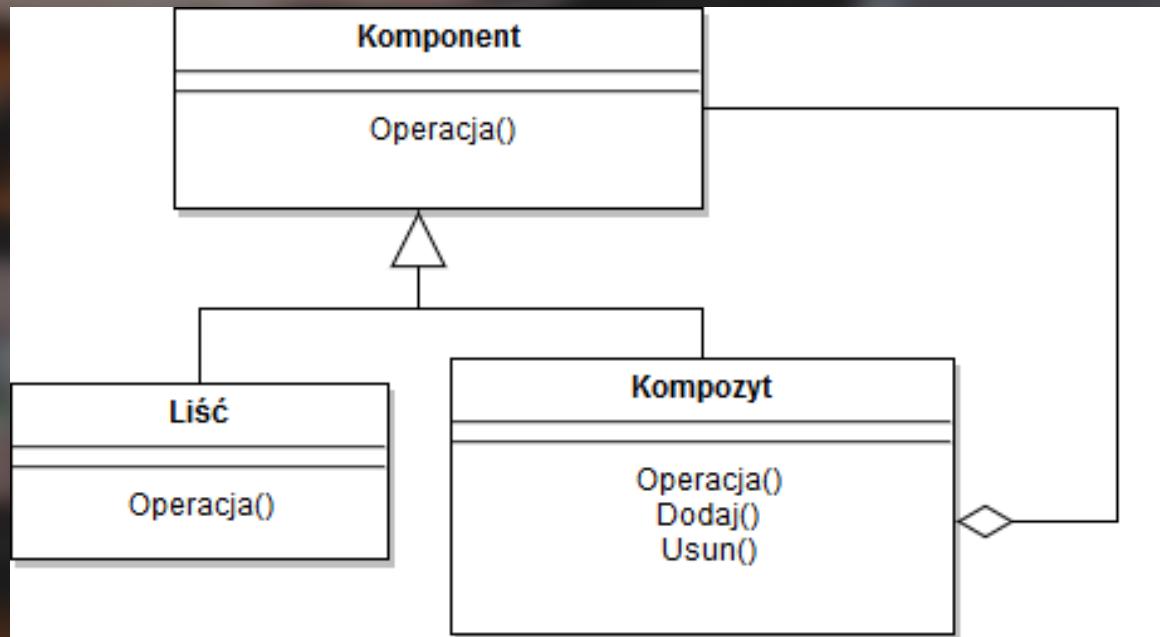
Wzorzec strukturalny



Kompozyt

Kompozyt to grupa obiektów, z których każdy może zawierać inne obiekty. Zatem każdy obiekt, może być grupą obiektów lub pojedynczym obiektem czyli liściem. Wzorzec kompozyt oparty jest o strukturę drzewiastą.

Kompozyt jak i **Liść** dziedziczy po tym samym interfejsie **Komponent** co pozwala na dostęp do obiektów tak samo jak do grupy tych obiektów. Możemy wykonywać operacje na pojedynczym obiekcie, jak i na grupie obiektów stosując ten wzorzec.





Kompozyt

```
interface Component {  
    void traverse();  
}  
  
class Primitive implements Component {  
    private int value;  
  
    public Primitive(int val) {  
        value = val;  
    }  
  
    public void traverse() {  
        System.out.print(value + " ");  
    }  
}
```

```
abstract class Composite implements Component {  
  
    private Component[] children = new Component[9];  
    private int total = 0;  
    private int value;  
  
    public Composite(int val) {  
        value = val;  
    }  
  
    public void add(Component c) {  
        children[total++] = c;  
    }  
  
    public void traverse() {  
        System.out.print(value + " ");  
        for (int i=0; i < total; i++) {  
            children[i].traverse();  
        }  
    }  
}
```



Kompozyt

```
class Row extends Composite {  
    public Row(int val) {  
        super(val);  
    }  
  
    public void traverse() {  
        System.out.print("Row");  
        super.traverse();  
    }  
  
}  
  
class Column extends Composite {  
    public Column(int val) {  
        super(val);  
    }  
  
    public void traverse() {  
        System.out.print("Col");  
        super.traverse();  
    }  
  
}
```

```
public class CompositeDemo {  
    public static void main( String[] args ) {  
        Composite row1 = new Row( 1 );  
        Composite column1 = new Column( 2 );  
        Composite column2 = new Column( 3 );  
        Composite row2 = new Row( 4 );  
        Composite row3 = new Row( 5 );  
        row1.add(column1);  
        row1.add(column2);  
        column2.add(row2);  
        column2.add(row3);  
        row1.add(new Primitive(6));  
        column1.add(new Primitive(7));  
        column2.add(new Primitive(8));  
        row2.add(new Primitive(9));  
        row3.add(new Primitive(10));  
        row1.traverse();  
    }  
}
```

Output

```
Row1 Col2 7 Col3 Row4 9 Row5 10 8 6
```



Kompozyt - zadanie

Przygotuj strukturę pozwalającą w sposób hierarchiczny przechowywać informację na temat plików. Spróbuj odwzorować poniższą strukturę:

```
-Katalog1:  
  --Katalog2:  
    ---Plik1  
    ---Plik2  
    ---Plik3  
  --Katalog3:  
    ---Plik4  
    ---Plik5  
    ---Plik6  
-Katalog3:  
  --Plik7  
  --Plik8
```

Dodaj Component z metodą getName oraz Composit przechowujący strukturę drzewiastą komponentów.



Pełnomocnik

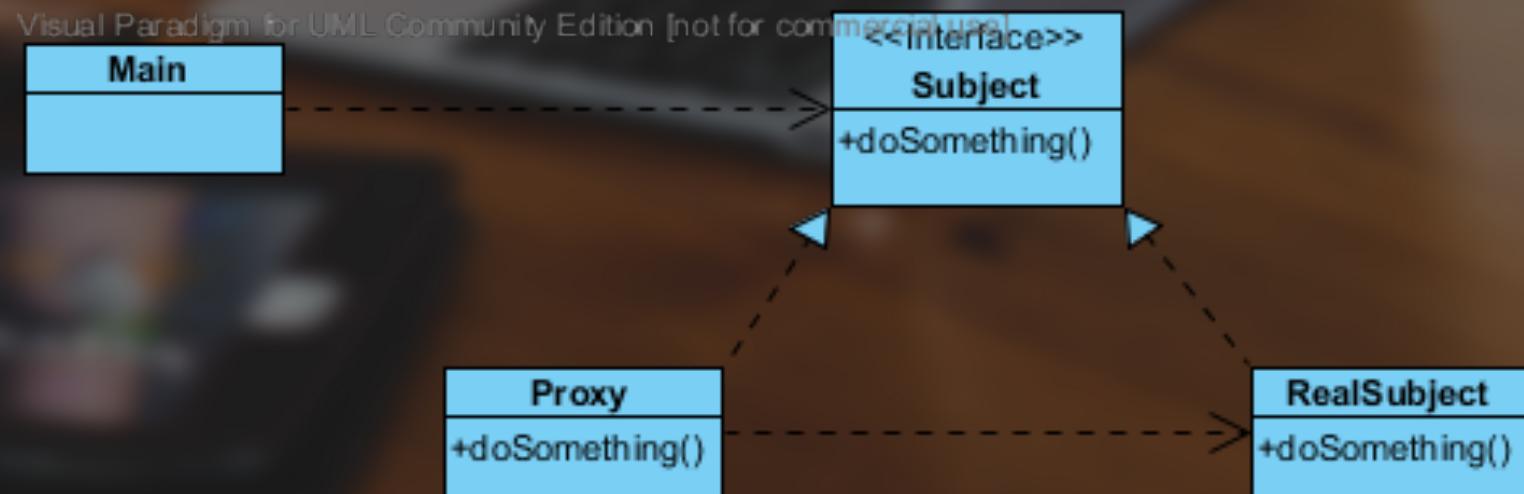
Wzorzec strukturalny



Pełnomocnik

Pełnomocnik jest wzorcem projektowym, którego celem jest utworzenie obiektu zastępującego inny obiekt. Stosowany jest on w celu kontrolowanego tworzenia na żądanie kosztownych obiektów oraz kontroli dostępu do nich.

Interfejs **Subject** definiuje pewną metodę. Klasy **Proxy** i **RealSubject** realizują powyższy interfejs na dwa różne sposoby. W klasie **Proxy** zostaje wywołana metoda **doSmething()** obiektu klasy **RealSubject**. Przed wywołaniem tej metody można oczywiście wykonać inne operacje.



Pełnomocnik



```
//Interfejs dostępu do danych
interface ImageInterface{
    String getImage();
}

//Główna klasa generująca dane
class Image implements ImageInterface{

    public String getImage(){
        return "HiRes image data ...";
    }
}

// Klasa proxy, przy pierwszym wywołaniu daje dane poglądowe
class ProxyImage implements ImageInterface{
    private Image image;
    private int counter = 0;

    public String getImage(){
        counter++;
        if (counter > 1){
            if (image == null){
                image = new Image();
            }
            return image.getImage();
        }
        return "LoRes proxy thumbnail ...";
    }
}
```



Pełnomocnik

```
public class DesignPatternsProxy{  
  
    public static void main(String[] args){  
        // Odwołania do głównego obiektu bez użycia proxy  
        ImageInterface image = new Image();  
        System.out.println("Object without proxy:");  
        System.out.println("First access: " + image.getImage());  
        System.out.println("Second access: " + image.getImage());  
  
        // Odwołania do głównego obiektu z użyciem obiektu proxy  
        image = new ProxyImage();  
        System.out.println("Object with proxy:");  
        System.out.println("First access: " + image.getImage());  
        System.out.println("Second access: " + image.getImage());  
    }  
}
```



Iterator

Wzorzec operacyjny



Iterator

Wzorzec operacyjny którego celem jest udostępnienie jednolitego interfejsu umożliwiającego iterowanie po elementach znajdujących się wewnątrz danej klasy. W językach udostępniających pętle **foreach** istnieje możliwość zaimplementowania w swojej własnej klasie iteracji obsługiwanej przez tą pętlę poprzez dziedziczenie i obsłużenie odpowiedniego interfejsu.



Iterator

```
//Interfejs wzorca iterator
interface Iterator< E >{
    boolean hasNext();
    E next();
}

// Klasa obsługująca dostęp do danych zawartych w tablicy
class ArrayIterator< E > implements Iterator{
    int posistion = 0;
    E[] items;

    public ArrayIterator(E[] items){
        this.items = items;
    }

    public boolean hasNext(){
        if (posistion >= items.length || items[posistion] == null){
            return false;
        }
        return true;
    }

    public E next(){
        E item = items[posistion];
        posistion++;
        return item;
    }
}
```



Iterator

```
//Klasa obsługująca dostęp do danych zawartych w liście
class ListIterator< E > implements Iterator{

    int posistion = 0;
    List< E > items;

    public ListIterator(List< E > items){
        this.items = items;
    }

    public boolean hasNext(){
        if (posistion >= items.size() || items.get(posistion) == null){
            return false;
        }
        return true;
    }

    public E next(){
        E item = items.get(posistion);
        posistion++;
        return item;
    }
}
```



Iterator

```
public class DesignPatternsIterator{  
    private static Integer[] intArray = new Integer[10];  
    private static ArrayList< Integer > intList = new ArrayList< Integer >();  
  
    public static void main(String[] args){  
        intArray[0] = 1;  
        intArray[1] = 11;  
        intArray[2] = 21;  
        intArray[3] = 31;  
        intArray[4] = 41;  
        intArray[5] = 51;  
        intArray[6] = 61;  
        intArray[7] = 71;  
        intArray[8] = 81;  
        intArray[9] = 91;  
  
        intList.add(1);  
        intList.add(11);  
        intList.add(21);  
        intList.add(31);  
        intList.add(41);  
        intList.add(51);  
        intList.add(61);  
        intList.add(71);  
        intList.add(81);  
  
        // Wyświetlanie zawartości przy pomocy iteratora  
        // metoda wymaga klas implementujących interfejs iterator,  
        // który zapewnia usystematyzowany dostęp do danych  
        printData(new ArrayIterator(intArray), "Tablica");  
        printData(new ListIterator(intList), "Lista");  
    }  
  
    private static void printData(Iterator iterator, String info){  
        if (iterator != null){  
            System.out.println("Dane: " + info);  
            while (iterator.hasNext()){  
                System.out.print(iterator.next() + ", ");  
            }  
            System.out.println("");  
            System.out.println("");  
        }  
    }  
}
```



Iterator - zadanie

Utwórz klasę ListWithId która będzie strukturą przechowującą obiekty dowolnej klasy. Obiekty powinny być przechowywane na HashMapie jako wartość a klucz powinien być generowany automatycznie. Dodaj 3 metody - dodawanie, usuwanie oraz pobieranie obiektu po ID. Zaimplementuj także iterator dla nowej kolekcji.

Przykład implementacji:

```
class ListWithId<V> extends HashMap<Integer, V> {  
  
    public void add(V obj) {  
        put(someId, obj);  
    }  
}
```

Przykład użycia:

```
ListWithId<User> users = new ListWithId<User>();  
users.add(new User());
```



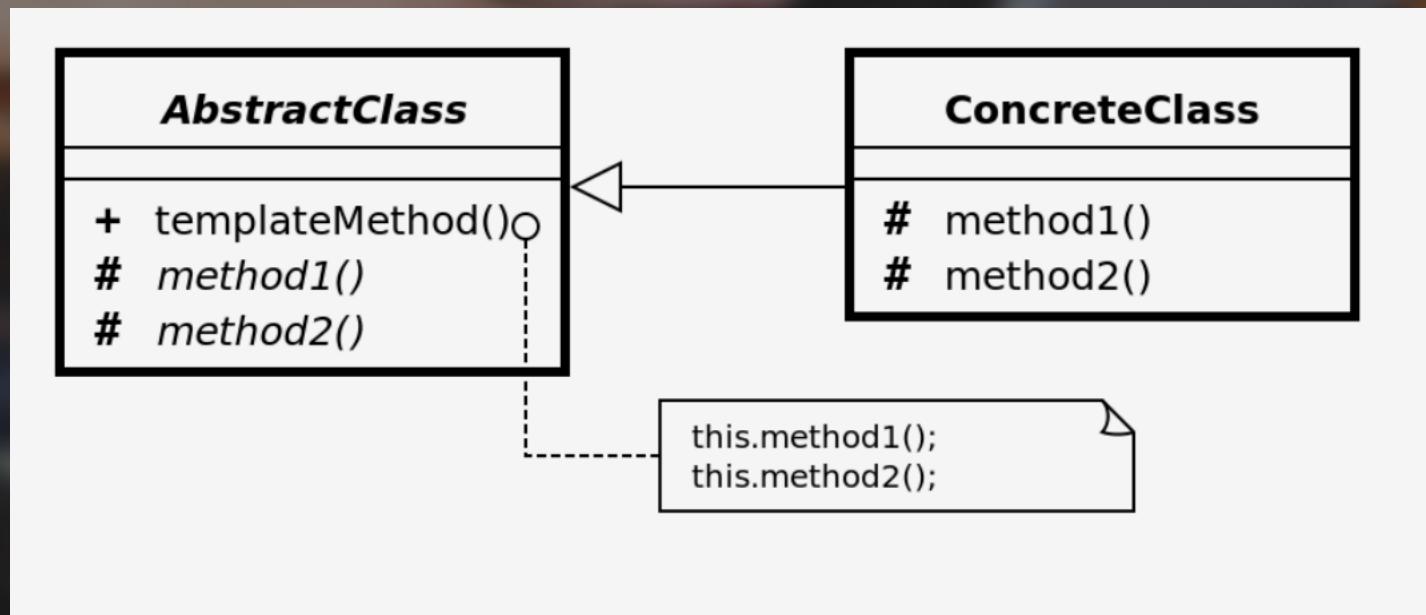
Metoda Szablonowa

Wzorzec operacyjny



Metoda Szablonowa

Jego zadaniem jest zdefiniowanie metody, będącej szkieletem algorytmu. Algorytm ten może być następnie dokładnie definiowany w klasach pochodnych. Niezmienna część algorytmu zostaje opisana w metodzie szablonowej, której klient nie może nadpisać. W metodzie szablonowej wywoływane są inne metody, reprezentujące zmienne kroki algorytmu.





Metoda Szablonowa

```
public abstract class HouseTemplate {  
  
    //metoda szablonowa – poprzez final nie może być nadpisana  
    public final void buildHouse(){  
        buildFoundation();  
        buildPillars();  
        buildWalls();  
        buildWindows();  
        System.out.println("House is built.");  
    }  
  
    //domyślna implementacja  
    private void buildWindows() {  
        System.out.println("Building Glass Windows");  
    }  
  
    //Metody które muszą być zaimplementowane  
    public abstract void buildWalls();  
    public abstract void buildPillars();  
  
    private void buildFoundation() {  
        System.out.println("Building foundation with cement,iron rods and sand");  
    }  
}
```



Metoda Szablonowa

```
public class WoodenHouse extends HouseTemplate {
```

```
    @Override  
    public void buildWalls() {  
        System.out.println("Building Wooden Walls");  
    }
```

```
    @Override  
    public void buildPillars() {  
        System.out.println("Building Pillars with Wood coating");  
    }
```

```
public class GlassHouse extends HouseTemplate {
```

```
    @Override  
    public void buildWalls() {  
        System.out.println("Building Glass Walls");  
    }
```

```
    @Override  
    public void buildPillars() {  
        System.out.println("Building Pillars with glass coating");  
    }
```



Metoda Szablonowa

```
public class HousingClient {  
    public static void main(String[] args) {  
        HouseTemplate houseType = new WoodenHouse();  
        houseType.buildHouse();  
  
        System.out.println("*****");  
  
        houseType = new GlassHouse();  
        houseType.buildHouse();  
    }  
}
```



Strategia

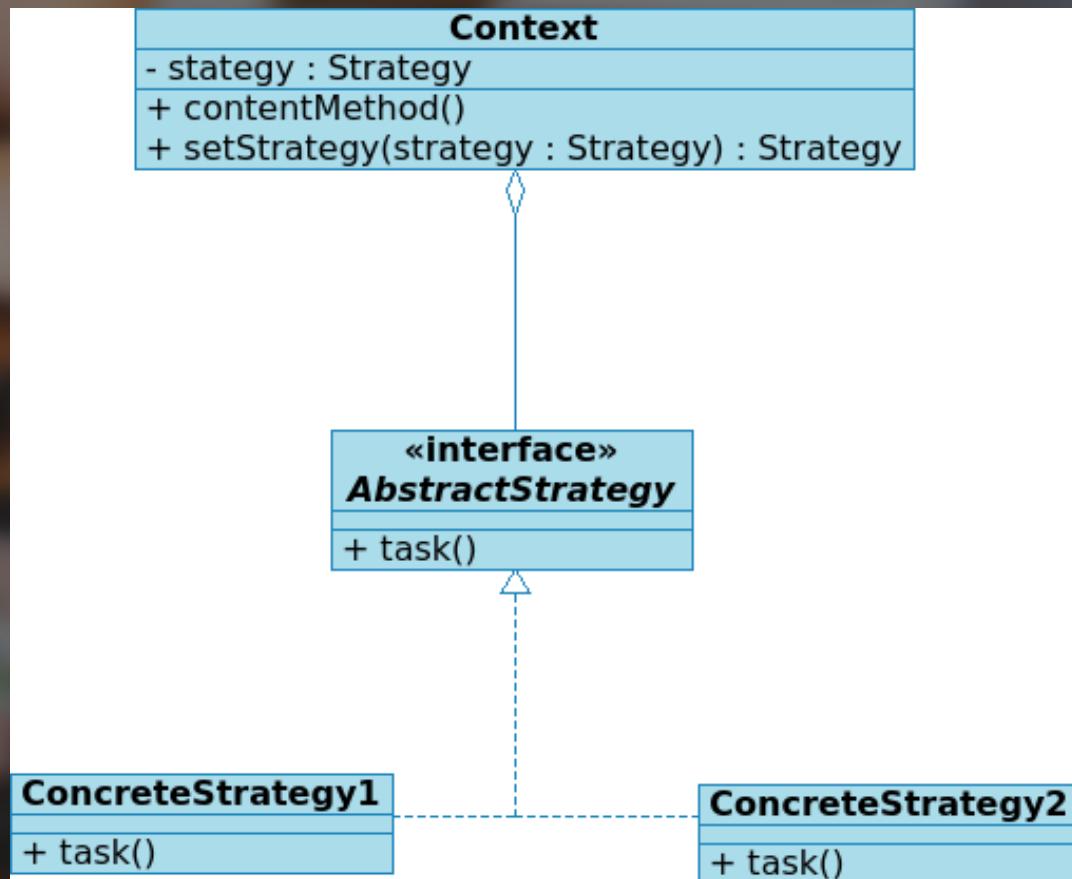
Wzorzec operacyjny





Strategia

Strategia jest wzorcem projektowym, który definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas. Dzięki temu umożliwia wymienne stosowanie każdego z nich w trakcie działania programu.





Strategia

```
public interface CompressionStrategy {  
    public void compressFiles(ArrayList<File> files);  
}
```

```
public class ZipCompressionStrategy implements CompressionStrategy {  
    public void compressFiles(ArrayList<File> files) {  
        //using ZIP approach  
    }  
}
```

```
public class RarCompressionStrategy implements CompressionStrategy {  
    public void compressFiles(ArrayList<File> files) {  
        //using RAR approach  
    }  
}
```



Strategia

```
public class CompressionContext {  
    private CompressionStrategy strategy;  
  
    //Strategie można ustawić w dolnym momencie dzialania programu  
    public void setCompressionStrategy(CompressionStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    //wykorzystuje strategie i wykonuje operacje  
    public void createArchive(ArrayList<File> files) {  
        strategy.compressFiles(files);  
    }  
}
```



Strategia

```
public class Client {  
  
    public static void main(String[] args) {  
        CompressionContext ctx = new CompressionContext();  
        ctx.setCompressionStrategy(new ZipCompressionStrategy());  
        //przygotowujemy jakiś listę plików i przekazujemy do kontekstu  
        ctx.createArchive(fileList);  
    }  
}
```



Strategia - zadanie

Przygotuj prosty kalkulator do liczenia podatków w danym kraju. Utwórz Interface TaxStrategy oraz metodę calculate(BigDecimal value);

Utwórz 3 implementację interface'u: TaxPL, TaxDE, TaxEN w którym z przekazanej kwoty wyliczysz podatek: 23%, 30%, 15%.

Utwórz klasę TaxContex która przyjmie dowolną strategię i po wywołaniu metody getTax(BigDecimal value) zwróci ile będzie wynosił podatek.

Dodaj testy które zweryfikują poprawność działania twojej implementacji.



Obserwator

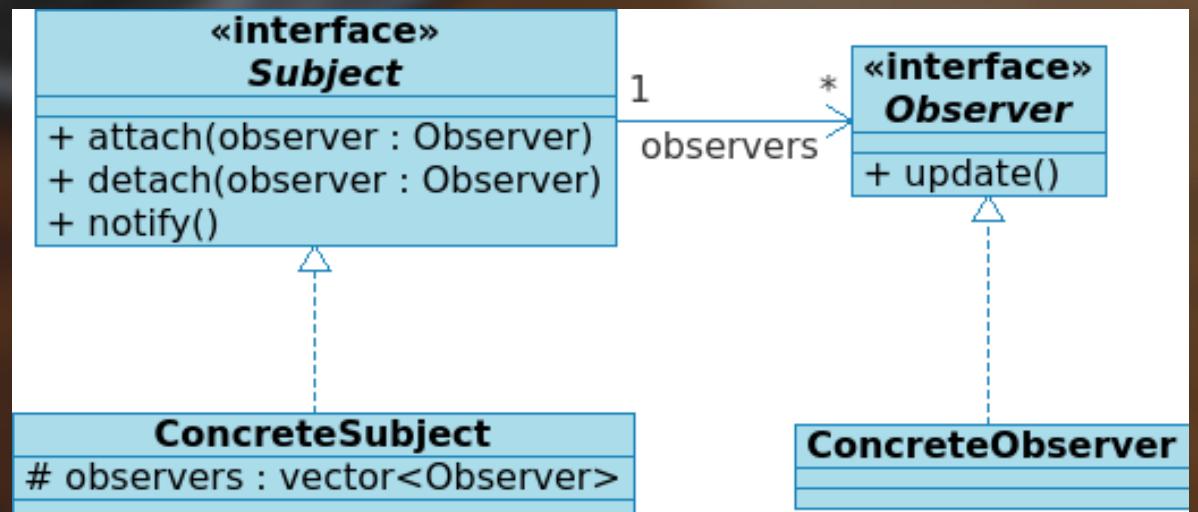
Wzorzec operacyjny



Obserwator

Głównym obszarem wykorzystania wzorca **Obserwator** jest stworzenie relacji typu jeden-do-wielu łączącej grupę obiektów. Dzięki zastosowaniu wzorca zmiana stanu (czyli zmiana aktualnych wartości pól) obiektu obserwowanego umożliwi automatyczne powiadomienie o niej wszystkich innych dołączanych elementów (obserwatorów).

Interfejs **Subject** definiuje operacje **attach()** i **detach()** pozwalające odpowiednio na dołączanie i odłączanie obserwatorów. Ponadto zdefiniowana jest też operacja **notify()**, służąca do powiadamiania wszystkich zarejestrowanych obserwatorów o zmianie stanu obiekt obserwowanego poprzez wywołanie w pętli metody **update()**, która jest zadeklarowana w interfejsie **Observer**. Operacja ta jest implementowana w klasie realizującej ten interfejs i służy do powiadamiania konkretnego obserwatora o zmianie stanu obiektu obserwowanego.





Obserwator

```
abstract class Observer {  
    protected Subject subject;  
    public abstract void update();  
}  
  
class Subject {  
    private List<Observer> observers = new ArrayList<>();  
    private int state;  
  
    public void add(Observer o) {  
        observers.add(o);  
    }  
  
    public int getState() {  
        return state;  
    }  
  
    public void setState(int value) {  
        this.state = value;  
        execute();  
    }  
  
    private void execute() {  
        for (Observer observer : observers) {  
            observer.update();  
        }  
    }  
}
```



Obserwator

```
class HexObserver extends Observer {  
    public HexObserver(Subject subject) {  
        this.subject = subject;  
        this.subject.add(this);  
    }  
    public void update() {  
        System.out.print(" " + Integer.toHexString(subject.getState()));  
    }  
}  
  
class OctObserver extends Observer {  
    public OctObserver(Subject subject) {  
        this.subject = subject;  
        this.subject.add(this);  
    }  
    public void update() {  
        System.out.print(" " + Integer.toOctalString(subject.getState()));  
    }  
}  
  
class BinObserver extends Observer {  
    public BinObserver(Subject subject) {  
        this.subject = subject;  
        this.subject.add(this);  
    }  
    public void update() {  
        System.out.print(" " + Integer.toBinaryString(subject.getState()));  
    }  
}
```



Obserwator

```
public class ObserverDemo {  
    public static void main( String[] args ) {  
        Subject sub = new Subject();  
        new HexObserver(sub);  
        new OctObserver(sub);  
        new BinObserver(sub);  
        Scanner scan = new Scanner(System.in);  
        for (int i = 0; i < 5; i++) {  
            System.out.print("\nEnter a number: ");  
            sub.setState(scan.nextInt());  
        }  
    }  
}
```

Output

```
Enter a number: 55  
37 67 110111  
Enter a number: 12  
c 14 1100  
Enter a number: -10  
fffffff6 3777777766 111111111111111111111111110110  
Enter a number: 112  
70 160 1110000  
Enter a number: 5  
5 5 101
```



Model Widok Kontroler

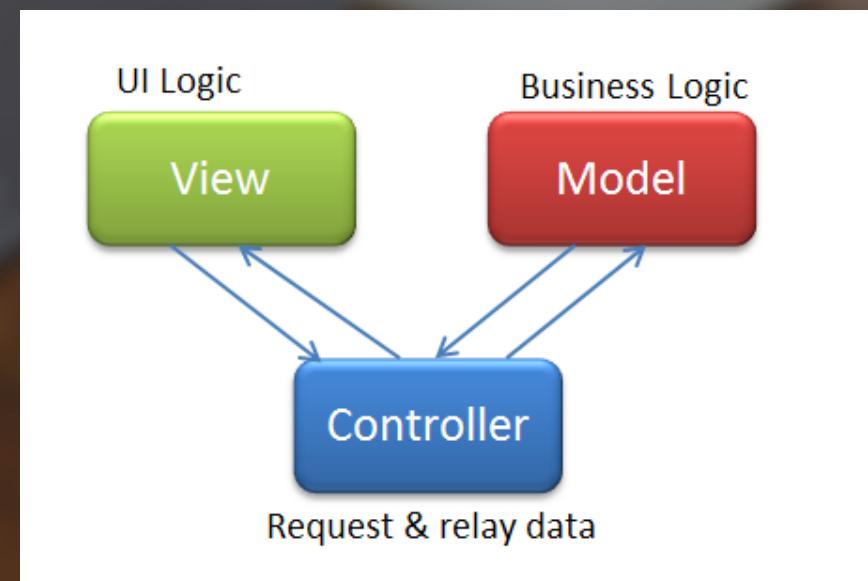
Wzorzec architektoniczny

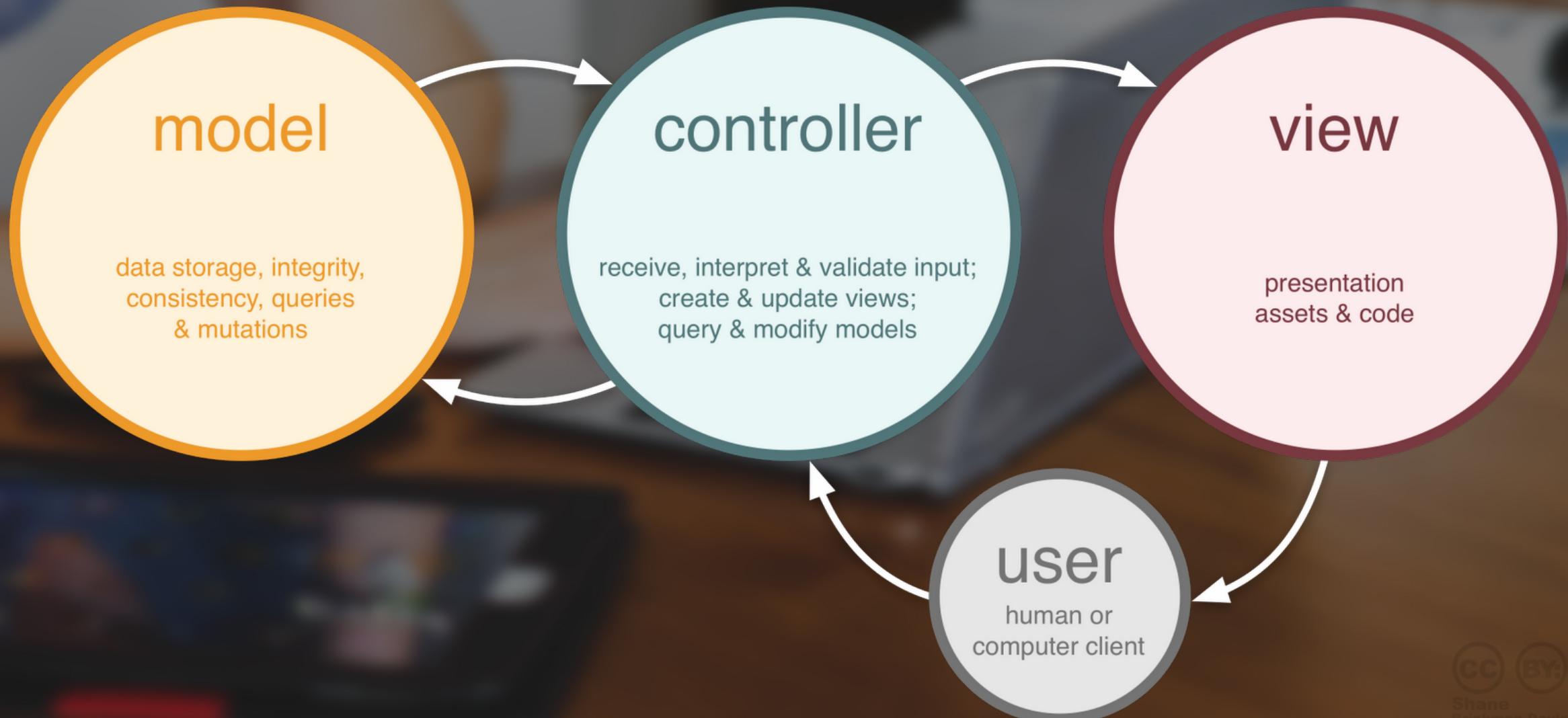


Model-View-Controller (pol. Model-Widok-Kontroler) – wzorzec architektoniczny służący do organizowania struktury aplikacji posiadających graficzne interfejsy użytkownika. Wiele prac traktuje go jako pojedynczy wzorzec, lecz może on być także traktowany jako złożony wzorzec wykorzystujący idee wzorców prostych, takich jak Obserwator, Strategia czy Kompozyt. Oba te podejścia nie wykluczają się. Model – jest pewną reprezentacją problemu bądź logiki aplikacji.

Widok – opisuje, jak wyświetlić pewną część modelu w ramach interfejsu użytkownika. Może składać się z podwidoków odpowiedzialnych za mniejsze części interfejsu.

Kontroler – przyjmuje dane wejściowe od użytkownika i reaguje na jego poczynania, zarządzając aktualizacje modelu oraz odświeżenie widoków.







MVC - Model

```
public class UserService {  
    private List<User> users = new ArrayList<>();  
  
    public void addUser(User user){  
        users.add(user);  
    }  
  
    public List<User> getAllUsers(){  
        return users;  
    }  
  
    public User getUserByLogin(String login){  
        for (User user : users) {  
            if (user.getLogin().equalsIgnoreCase(login)){  
                return user;  
            }  
        }  
        return null;  
    }  
}
```



MVC - Widok

```
public class UserView {  
    private UserController controller;  
    public void setController(UserController controller) {  
        this.controller = controller;  
    }  
  
    public void loadPage(){  
        Scanner scanner = new Scanner(System.in);  
        int choice;  
        while(true){  
            displayMenu();  
            choice = scanner.nextInt();  
            switch (choice) {  
                case 1:  
                    createUserMenu(); break;  
                case 2:  
                    showUsersMenu(); break;  
                case 3:  
                    showUserByLoginMenu(); break;  
                default:  
            }  
        }  
    }  
  
    private void displayMenu(){  
        System.out.println("===== Menu =====");  
        System.out.println("1. Create user.");  
        System.out.println("2. Get users.");  
        System.out.println("3. Remove user by login.");  
        System.out.println("=====");  
        System.out.println();  
        System.out.print("Your choice: ");  
    }  
}
```

```
private void createUserMenu() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.println("Login: ");  
    String login = scanner.next();  
    System.out.println("Password: ");  
    String password = scanner.next();  
    controller.addUser(new User(login, password));  
}  
  
private void showUsersMenu() {  
    System.out.println("Stored users: ");  
    for(User user : controller.getAllUsers()){  
        System.out.println(user);  
    }  
    System.out.println();  
}  
  
private void showUserByLoginMenu() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.println("Get user id: ");  
    User user = controller.getUserByLogin(scanner.next());  
    if (user != null) {  
        System.out.println(user);  
    } else {  
        System.out.println("Not found");  
    }  
    return;  
}
```

Wywołanie metod na kontrolerze



MVC - Controller

```
public class UserController {  
  
    private UserService userService;  
    private UserView userView;  
  
    public UserController(UserService userService, UserView userView) {  
        this.userService = userService;  
        this.userView = userView;  
        this.userView.setController(this);  
    }  
  
    public void addUser(User user){  
        boolean isValidUser = validUser(user);  
        if (isValidUser){  
            userService.addUser(user);  
        }  
    }  
  
    private boolean validUser(User user) {  
        return !user.getLogin().equals("") && !user.getPassword().equals("");  
    }  
  
    public List<User> getAllUsers(){  
        return userService.getAllUsers();  
    }  
  
    public User getUserByLogin(String login){  
        User user = userService.getUserByLogin(login);  
  
        return user;  
    }  
}
```

Połączenie kontrolera z widokiem oraz modelem

Przekazanie usera do modelu który ma zostać zapisany. Zanim zostanie przekazany można wykonać validację

MVC - Integracja



```
public class App {  
  
    public static void main(String[] args){  
        UserService userService = new UserService();  
        UserView userView = new UserView();  
        new UserController(userService, userView);  
  
        userView.loadPage();  
    }  
}
```

Utworzenie obiektów MVC i połączenie ich ze sobą.

Imitacja ładowania strony www.



MVC - Zadanie

1. Wykonaj prostą implementację wzorca MVC na podstawie zaprezentowanego przykładu. Zaimplementuj ponownie część Hotel serwisu z funkcjonalnością:

- Rezerwacji hotelu
- Pobranie wszystkich hoteli
- Pobranie hotelu wolnych

2. Zaimplementuj prosty system bankowy. Dodaj klasę Account która będzie przechowywała informację na temat loginu oraz pinu jak również zgromadzonych środków. Dodaj klasę BankService która będzie imitacją bazy. Dodaj funkcjonalność:

- Wypłacić pieniądze
- Wpłacić pieniądze
- Przelej pieniądze z konta o loginie A na konto B
- Zmiana pinu

Pamiętaj o zastosowaniu MVC.