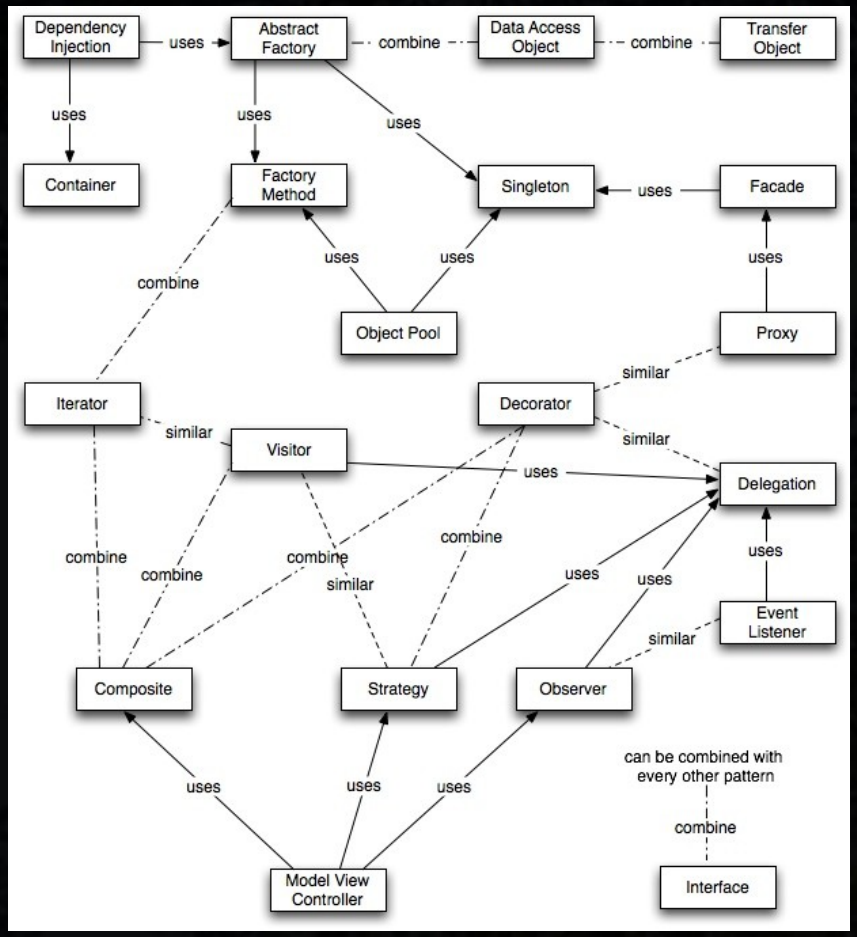
Wzorce projektowe
Michał Węgorek



Wzorce projektowe
Plan prezentacji

- Co to jest i po co to jest?
- Podział
- Najczęściej spotykane wzorce
- Bibliografia

Co to jest i po co to jest?

- **Wzorzec projektowy (ang. Design pattern)** -
  uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych.

  Termin “wzorzec projektowy” - Kenta Becka - 1987, spopularyzowany w 1995 roku w książce Design Patterns autorstwa Gammy, Helma, Johnsona i Vlissidesa (zwanych zwykle "Bandą Czworga" lub "Bandą Czterech" - ang. Gang of Four).
- Czego się spodziewałem a co zastałem
- Po co wzorce – precyzja, lepsze zrozumienie o czym mówimy, krótsza rozmowa, nie wymyślamy na nowo koła
- **Rozmowa programistów:**
  Do poradzenia sobie z problemem zmieniających się interfejsów dla zewnętrznych usług zastosujemy Adaptery generowane przez Singletonową Fabrykę

Czym są wzorce projektowe?

„Każdy wzorzec opisuje problem, który ciągle na nowo pojawia się w naszym otoczeniu i opisuje rdzeń jego rozwiązania w taki sposób, że można go używać milion razy i nigdy w ten sam sposób.”

(Christopher Alexander)

SZACHY

- Nauczyć się reguł gry. Nauczyć się, jakie ruchy wykonują poszczególne figury i jak kończy się partia gry.
- Nauczyć się techniki. Trzeba wiedzieć, jaka jest wartość poszczególnych figur, jak należy rozpoczynać partię (tzw. standardowe otwarcia) by przyniosło to jak najwięcej korzyści w przyszłości, jak się efektywnie bronić i jak przeprowadzać ataki.
- Nauczyć się wzorców gry, studiując rozegrane już partie. Nauczyć się ich używać w całkiem nowych sytuacjach powstałych podczas gry na szachownicy.

PROGRAMOWANIE

- Nauczyć się reguł programowania. Nauczyć się języka programowania, jego składni.
- Nauczyć się techniki programowania. Wiedzieć, jakie są poszczególne struktury danych, kiedy je zastosować i w jaki sposób je stworzyć w danym języku programowania. Nauczyć się popularnych algorytmów, ich zalet i wad oraz metod implementacji.
- Nauczyć się wzorców, które sugerują, w jaki sposób należy rozwiązywać daną klasę problemów i w ten sposób pokonywać problemy, które mogą się pojawić przed nami w przyszłości.

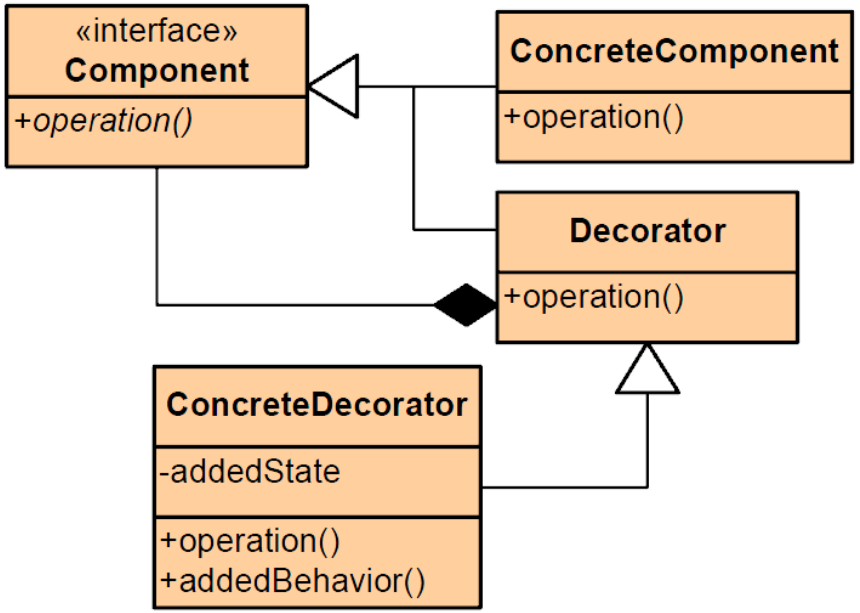
Podział wzorców

- Konstrukcyjne
  -wykorzystuje się je do pozyskania obiektów zamiast bezpośredniego tworzenia instancji klasy
  -programy zyskują na elastyczności, gdyż można decydować, jaki typ obiektu ma zostać utworzony w danym przypadku
- Strukturalne
  -pomagają łączyć obiekty w większe struktury
- Behawioralne
  -charakteryzują sposób, w jaki klasy lub obiekty oddziaływują i dzielą odpowiedzialności

Podział wzorców GoF

		PRZEZNACZENIE		
		KONSTRUKCYJNE	STRUKTURALNE	BEHAWIORALNE
		Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
		Budowniczy Fabryka abstrakcyjna Prototyp Singleton	Adapter Most Kompozyt Dekorator Fasada Pełnomocnik Pyłek	Łańcuch zobowiązań Polecenie Iterator Mediator Pamiątka Obserwator Stan Strategie Odwiedzający

Dekorator



Decorator

Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Dekorator

- opakowanie oryginalnej klasy w nową klasę "dekorującą"
- oryginalny obiekt przekazany jako parametr konstruktora dekoratora
- metody dekoratora wywołują metody oryginalnego obiektu i dodatkowo implementują nową funkcjonalność
- alternatywa dla dziedziczenia - rozszerza zachowanie klasy w trakcie działania programu

Dekorator

```
// interfejs Okno
interface Okno {
    public void rysuj(); // rysuje Okno na ekranie
    public String pobierzOpis(); // zwraca opis Okna
}

// implementacja zwykłego okna bez pasków przewijania
class ZwykłeOkno implements Okno {
    public void rysuj() {
        // rysuj okno
    }

    public String pobierzOpis() {
        return "zwykłe okno";
    }
}

// abstrakcyjna klasa dekorator - implementuje interfejs Okno
abstract class OknoDekorator implements Okno {
    protected Okno dekorowaneOkno; // dekorowane Okno

    public OknoDekorator(Okno dekorowaneOkno) {
        this.dekorowaneOkno = dekorowaneOkno;
    }
}
```

```
// pierwszy dekorator dodający pionowe paski przewijania
class PionowePrzewijanieDekorator extends OknoDekorator {
    public PionowePrzewijanieDekorator (Okno dekorowaneOkno) {
        super(dekorowaneOkno);
    }

    public void rysuj() {
        rysujPionowyPasekPrzewijania();
        dekorowaneOkno.rysuj();
    }

    private void rysujPionowyPasekPrzewijania() {
        // rysuj pionowy pasek przewijania
    }

    public String pobierzOpis() {
        return dekorowaneOkno.pobierzOpis() + ", z pionowym paskiem przewijania";
    }
}

// drugi dekorator dodający poziome paski przewijania
class PoziomePrzewijanieDekorator extends OknoDekorator {
    public PoziomePrzewijanieDekorator (Okno dekorowaneOkno) {
        super(dekorowaneOkno);
    }

    public void rysuj() {
        rysujPoziomyPasekPrzewijania();
        dekorowaneOkno.rysuj();
    }

    private void rysujPoziomyPasekPrzewijania() {
        // rysuj poziomy pasek przewijania
    }

    public String pobierzOpis() {
        return dekorowaneOkno.pobierzOpis() + ", z poziomym paskiem przewijania";
    }
}
```

Dekorator

```
public class DekorowaneOknoTest {
    public static void main(String[] args) {
        // utwórz dekorowane Okno z poziomymi i pionowymi paskami przewijania
        Window dekorowaneOkno = new PoziomePrzewijanieDekorator(
                new PionowePrzewijanieDekorator(new ZwykłeOkno()));

        // wypisz opis Okna
        System.out.println(dekorowaneOkno.pobierzOpis());
    }
}
```

Dekorator

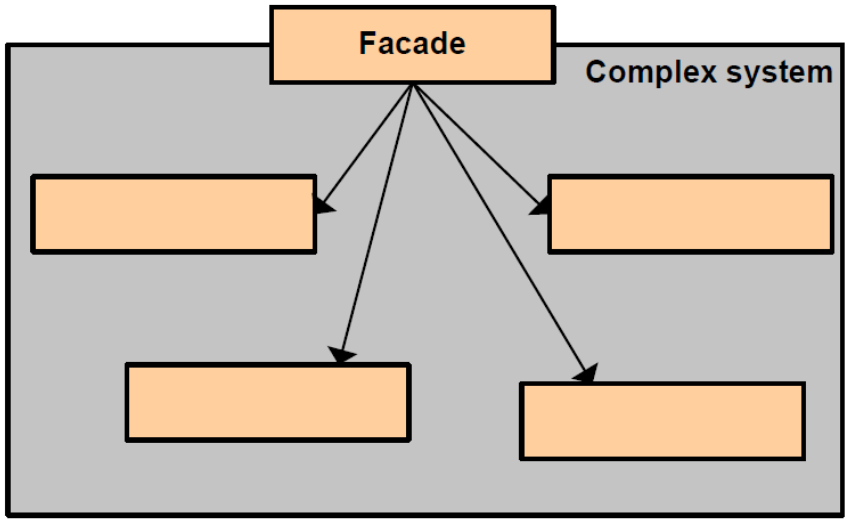
Stosowalność

-aby dynamicznie i w przezroczysty sposób dodawać zobowiązania do pojedynczych obiektów

-w wypadku zobowiązań które mogą być cofnięte

-gdy rozszerzanie przez definiowanie podklas jest niepraktyczne (dużo niezależnych rozszerzeń które przy próbie uwzględnienia każdej ich kombinacji prowadzą do ogromnej liczby podklas)

Fasada



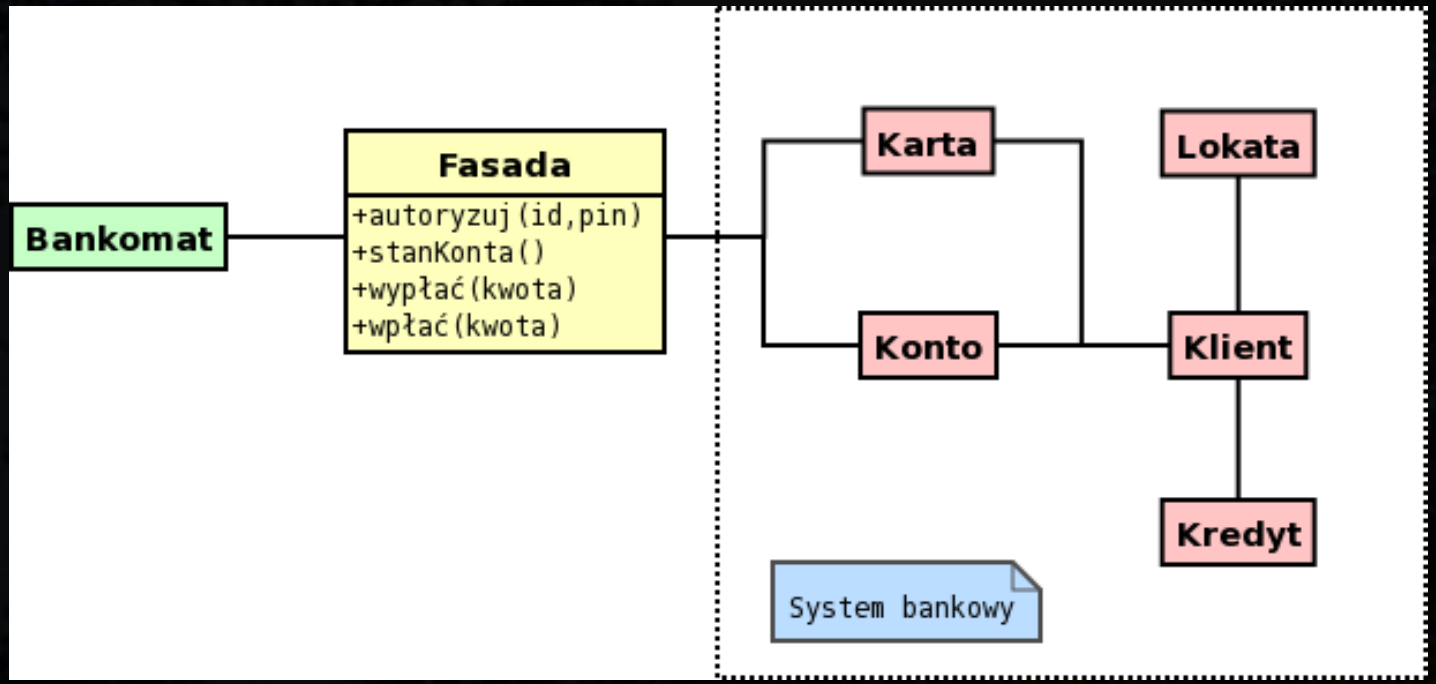
Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Fasada

- Problem: Potrzebny jest wspólny, jednorodny interfejs do zbioru implementacji lub interfejsów (np. w ramach podsystemu). Chcemy ograniczyć niepożądane sprzężenie ze składnikami podsystemu oraz zabezpieczyć się przed jego zmianami.
- Rozwiązanie: Zdefiniuj pojedynczy kontrakt dla całego podsystemu – obiekt fasadę, który opakowuje podsystem.
- Przykład



Fasada - korzyści

- ukrywanie złożoności tworzonego przez siebie systemu przez dostarczenie udokumentowanego, publicznego API. Korzyścią z zastosowania w tym przypadku fasady jest to, że programiści korzystający z systemu muszą przyswoić sobie tylko API fasady a nie wszystkich obiektów systemu.
- uproszczenie używania cudzej biblioteki programistycznej przez zdefiniowanie wygodnych i dostosowanych do konkretnego zastosowania metod pośredniczących między systemami. Ubocznym skutkiem takiego uproszczenia jest zwiększenie czytelności swojego kodu.

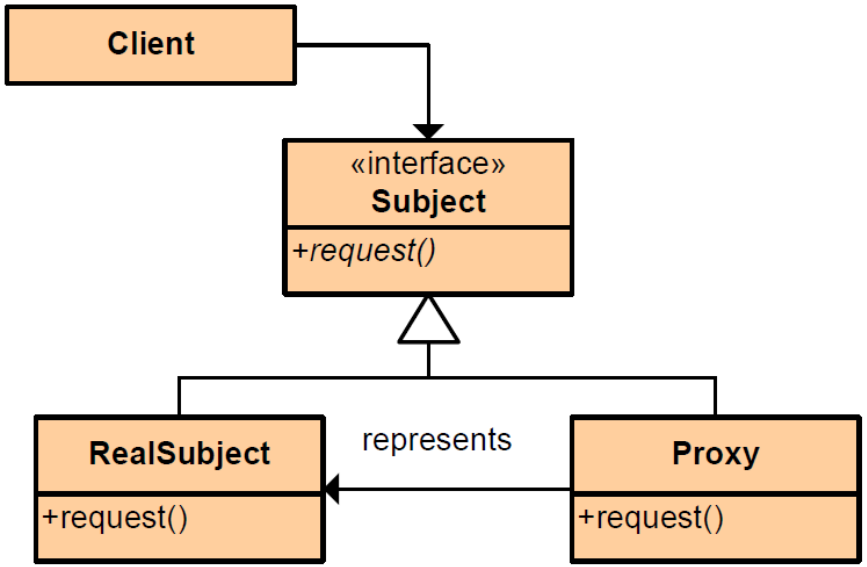
W ogólności wzorzec fasady pozwala wykorzystać podzbiór możliwości skomplikowanego systemu w prostszy, specyficzny dla danego zastosowania sposób.

Pełnomocnik

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.



```java
interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        // Potentially expensive operation
        // ...
        System.out.println("Loading    "+filename);
    }

    public void displayImage() { System.out.println("Displaying "+filename); }
}

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}

class ProxyExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");
        Image image3 = new ProxyImage("HiRes_10MB_Photo3");

        image1.displayImage(); // loading necessary
        image2.displayImage(); // loading necessary
        image1.displayImage(); // no loading necessary; already done
        // the third image will never be loaded - time saved!
    }
}
```

Pełnomocnik - korzyści

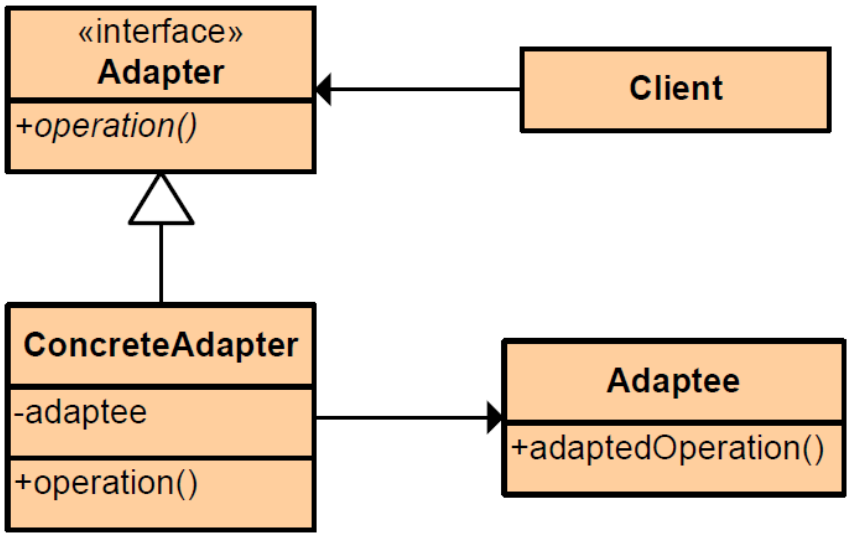
dodatkowy poziom pośredniości dostępu do obiektu, może:

-ukryć fakt, że obiekt jest zdalny

-wykonywać optymalizacje jak tworzenie obiektu na żądanie

-ukrywanie np. kopiowania-przy-zapisywaniu,odłożenie na później kopiowania obiektu, będzie on rzeczywiście skopiowany tylko, jeśli użytkownika tego zażąda

Adapter



Adapter

Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter

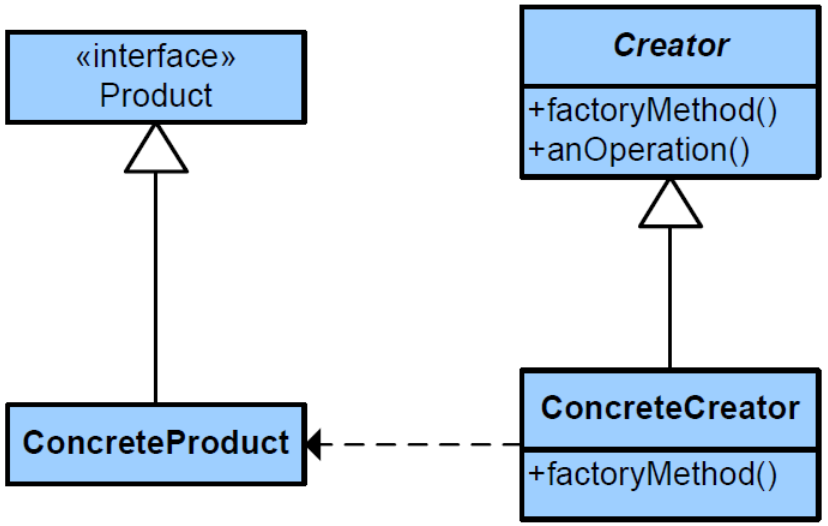
- Rozwiązuje problem niezgodnego interfejsu
- Pozwala na współprace klas, które nie mogą współpracować ze względu na interfejs
- Przykład:
  Zapisujemy liczby, a ten kto używa naszej klasy żądą true/false

Metoda fabrykująca

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Metoda fabrykująca

- Jeśli proces tworzenia obiektu jest złożony (np. zależy od ustawień konfiguracyjnych lub wejścia użytkownika), należy użyć metody fabrykującej:

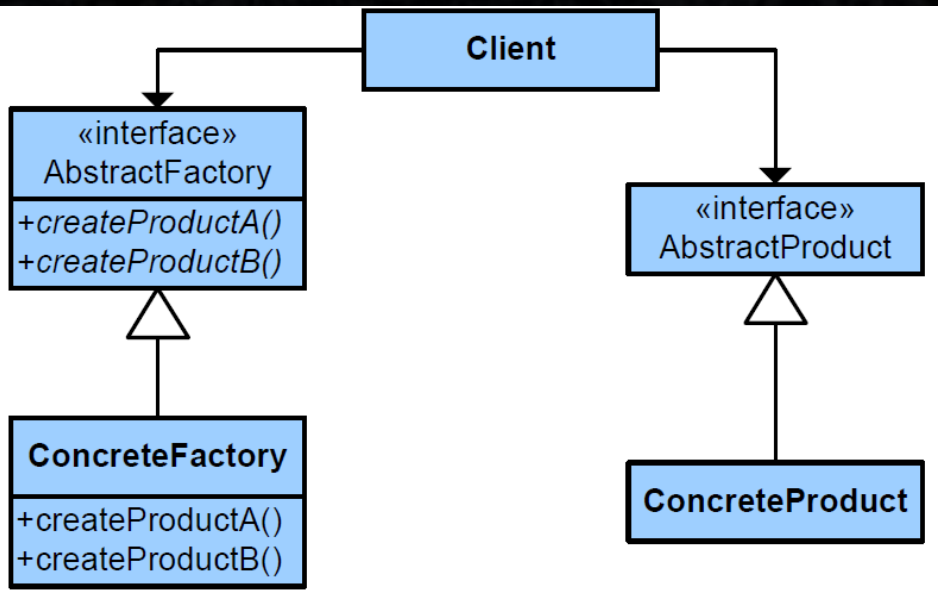
```
public class ImageReaderFactory
{
    public static ImageReader getImageReader( InputStream is )
    {
        int imageType = figureOutImageType( is );

        switch( imageType )
        {
            case ImageReaderFactory.GIF:
                return new GifReader( is );
            case ImageReaderFactory.JPEG:
                return new JpegReader( is );
            // etc.
        }
    }
}
```

Fabryka abstrakcyjna

Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Fabryka abstrakcyjna

- Enkapsulacja grupy metod fabrykujących dotyczących tego samego zagadnienia.

Fabryka abstrakcyjna

```
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();
}


class WinFactory extends GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}


class OSXFactory extends GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}
```

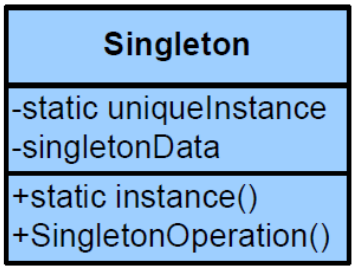
```
abstract class Button {
    public abstract void paint();
}


class WinButton extends Button {
    public void paint() {
        System.out.println("Przycisk WinButton");
    }
}


class OSXButton extends Button {
    public void paint() {
        System.out.println("Przycisk OSXButton");
    }
}


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

Singleton

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Singleton – JAVA 5

```
public class Singleton {
  private static volatile Singleton INSTANCE;

  // Protected constructor is sufficient to suppress unauthorized calls to the constructor
  protected Singleton() {}

  private static synchronized Singleton tryCreateInstance() {
    if (INSTANCE == null) {
      INSTANCE = new Singleton();
    }
    return INSTANCE;
  }

  public static Singleton getInstance() {
     // use local variable, don't issue 2 reads (memory fences) to 'INSTANCE'
     Singleton s = INSTANCE;
     if (s == null) {
       //check under lock; move creation logic to a separate method to allow inlining of getInstance()
       s = tryCreateInstance();
     }
     return s;
  }
}
```

Singleton-Bill Pugh

- Równoważne ostatniemu, ale działa we wszystkich wersjach Javy, obecnie zalecana wersja. Nie wymaga specjalnych konstrukcji (volatile/synchronized).

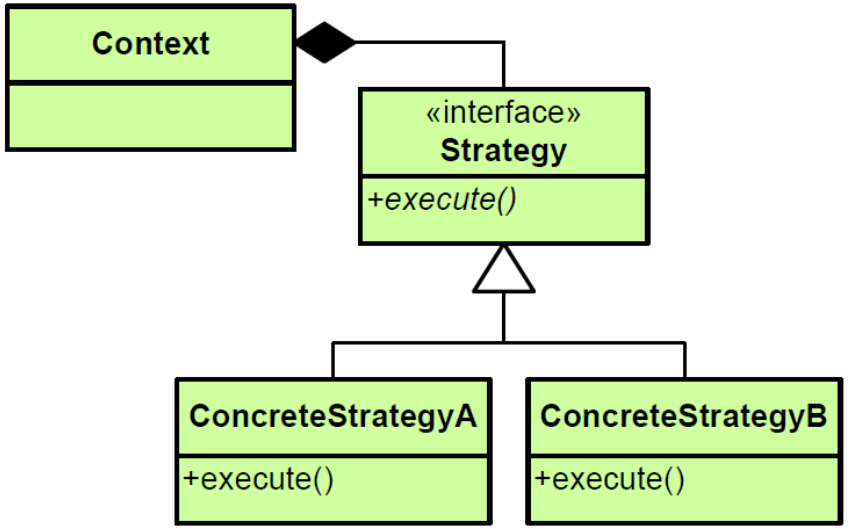
```
public class Singleton {
  // Protected constructor is sufficient to suppress unauthorized calls to the constructor
  protected Singleton() {}

  /**
   * SingletonHolder is loaded on the first execution of Singleton.getInstance()
   * or the first access to SingletonHolder.INSTANCE, not before.
   */
  private static class SingletonHolder {
    private final static Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Strategia

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Strategia

Problem: Jak w projekcie uwzględnić różnorodne, ale powiązane algorytmy lub strategie? Jak zapewnić łatwą możliwość ich zmiany?

Rozw.: Wszystkie algorytmy/strategie zdefiniuj w oddzielnych klasach o wspólnym interfejsie.

```java
// The context class uses this to call the concrete strategy
interface Strategy {

    void execute();

}

// Implements the algorithm using the strategy interface
class FirstStrategy implements Strategy {

    public void execute() {
        System.out.println("Called FirstStrategy.execute()");
    }

}

class SecondStrategy implements Strategy {

    public void execute() {
        System.out.println("Called SecondStrategy.execute()");
    }

}

class ThirdStrategy implements Strategy {

    public void execute() {
        System.out.println("Called ThirdStrategy.execute()");
    }

}

// Configured with a ConcreteStrategy object and maintains a reference to a Strategy object
class Context {

    Strategy strategy;

    // Constructor
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void execute() {
        this.strategy.execute();
    }
```

Dziękuję

Bibliografia

- http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/images/pattern_map.jpg
  - strona tytułowa
- Wikipedia, fragmenty kodu pochodza w wikipedii
  http://en.wikipedia.org/wiki/Facade_pattern
  http://en.wikipedia.org/wiki/Decorator_pattern
  http://en.wikipedia.org/wiki/Adapter_pattern
  http://en.wikipedia.org/wiki/Singleton_pattern
  http://en.wikipedia.org/wiki/Abstract_factory_pattern
  http://en.wikipedia.org/wiki/Factory_method_pattern
  http://en.wikipedia.org/wiki/Proxy_pattern
  http://en.wikipedia.org/wiki/Strategy_pattern
- www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf
  - grafiki wprowadzające opis każdego wzorca projektowego
- Inżynieria oprogramowania 2007– wykład 11 – Jacek Sroka
- Gamma, Helm, Johnson, Vlissides „Wzorce projektowe”, WNT 2005.