Design Patterns (Návrhové vzory)

Introduction (Úvod)

Juraj Petráš, Žilina, 2011

e-mail: j.petras@petrasoft.sk

Agenda

1. Motivácia

1. Definícia

1. Typy návrhových vzorov

1 Motivácia

Prečo je dobré poznať návrhové vzory?

- Ušetria čas
- Zvyšujú kvalitu
- Zlepšujú porozumenie pre developerov opravujúcich alebo rozširujúcich zdrojový kód

História

Inšpirovaný prácou Christophera Alexandera, rakúskeho Američana, ktorý sa 1977-79 zaslúžil o zavedenie Návrhových vzorov a jazyka v oblasti architektúry (nie softvérovej ale tej, čo navrhuje budovy)

Kent Beck and Ward Cunningham začali experimentovať s aplikáciu týchto myšlienok na tvorbu softvéru v roku 1987

Návrhové vzory získali popularitu po uvedení knihy „Elements of Reusable Object-Oriented Software“ v 1994 skupinou "Gang of Four" (Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides) na konferencii OOPSLA v Portlande

V tom istom roku bola aj prvá konferencia Pattern Languages of Programming Conference a založené Portland Pattern Repository

Definícia

Čo je to návrhový vzor?

Wikipedia hovorí“ „Návrhový vzorec je všeobecne znova-využiteľné riešenie bežného opakujúceho sa problému v návrhu softvéru“

Návrhový vzor je popis alebo šablóna ako riešiť nejaký typ softvérového problému bez konkrétne špecifikácie aplikačných tried a objektov.

Návrhový vzorec nie je hotový návrh, ktorý môže byť jednoducho transformovaný do kódu.

Doménou návrhových vzorcov sú abstraktné triedy, objekty a relácie medzi nimi .

Nie všetky softvérové vzory sú návrhové vzory. Napríklad, algoritmy riešia výpočtové problémy a nie návrhové problémy.

Typy návrhových vzorov

- Creational Patterns: Vytvárajúce vzory.
- Structural Pattern: Štrukturálne vzory
- Behavioral Patterns: Vzory správania.
- Concurrency Patterns: Vzory paralelného spracovania.

Creational patterns
(Vytvárajúce vzory)

Factory method: Definuje rozhranie vytvorenie objektu avšak podtriedy rozhodnú, ktorá trieda nakoniec vytvára objekt.

Abstract factory: Poskytuje rozhranie na vytvorenie skupiny vzájomne súvisiacich objektov bez špecifikácie konkrétnych tried

Builder: Oddeľuje konštrukciu komplexných objektov od ich reprezentácie umožňujúc ten istý proces konštrukcie pre rôzne reprezentácie.

Lazy initialization: Predstavuje taktiku pozdržaného vytvorenia objektu, ktorého vytvorenie je náročné, až do momentu, kedy je objekt skutočne potrebný.

Object pool: Zabraňuje náročnému získavaniu a deštrukcii objektov pomocou ich recyklácie.

Prototype: to aký objekt bude vytvorený rozhoduje prototyp objektu, nove insťancie sa vytvárajú klonovaním prototypu

Singleton: zaisťuje, že trieda má len 1 objekt, ku ktorému zabezpečuje globálny prístup

Multiton: Zaisťuje, že trieda má len menom identifikovateľné objekty. Poskytuje globálny prístup k nim.

Factory method

Cieľom je abstrakcia vytváranej konkrétnej triedy.

Často sa tento vzor kombinuje s štrukturálnym Composite vzorom

Creator

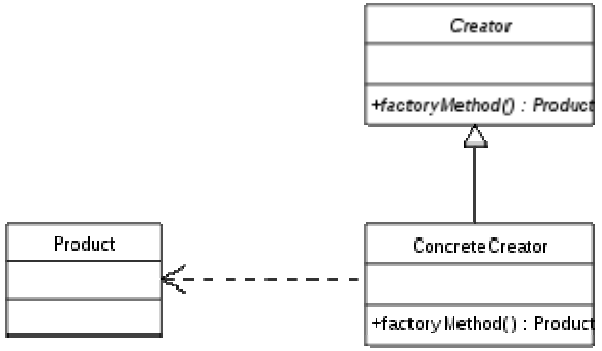
- Abstraktná supertrieda definujúca abstraktnú metódu factoryMehod() na vytvorenie objektov (Product)

Concrete Creator

- Konkrétna implementácia Creatora zodpovedného za vytvorenie produktu v konkrétnej implementácii factoryMethod().

Product

- Supertrieda produktov vytváraných v konkrétny podtriedach Creatora.



Factory method

```
public class MazeGame {
 public MazeGame() {
    Room room1 = makeRoom();
    Room room2 = makeRoom();
    room1.connect(room2);
    this.addRoom(room1);
    this.addRoom(room2);
 }

 protected Room makeRoom() {
    return new OrdinaryRoom();
 }
}
public MagicMazeGame extends MazeGame {
 protected Room makeRoom() {
     return new MagicRoom();
 }
}
```

Abstract factory

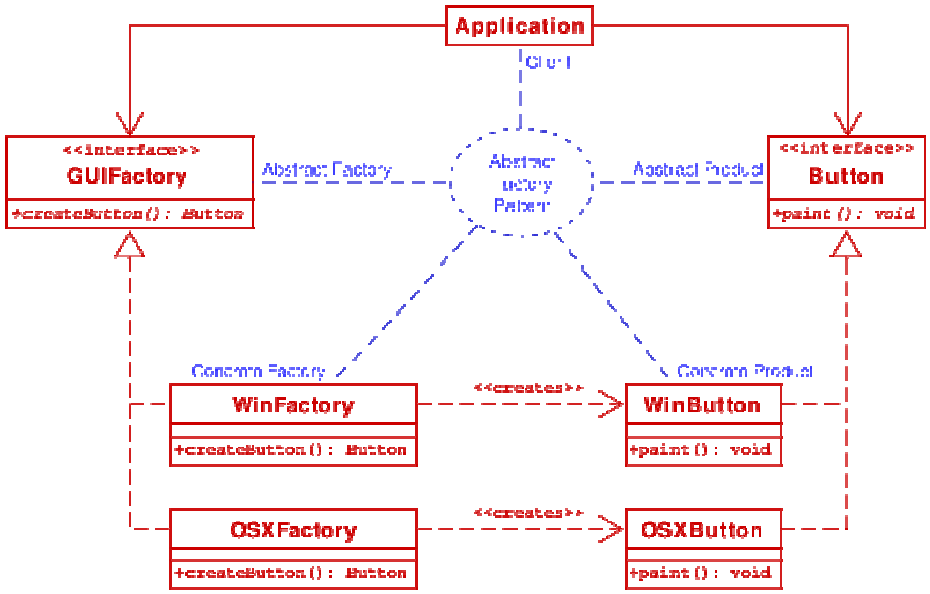
Cieľom je izolácia vytvárania objektov od ich použitia

Vzor separuje detaily implementácie skupiny objektov od ich použitia

Klient používa konkrétnu inštanciu AbstractFactory bez toho, aby vedel o akú konkrétnu triedu ide

Zvyčajne obsahuje viacero create* metód, ktoré vytvárajú skupinu objektov s rovnakom štýle

Výhody: nové podtriedy môžu byť pridané bez modifikácie klientskeho kódu



Abstract factory

```
/* GUIFactory example -- */
interface GUIFactory {
public Button createButton();
}
class WinFactory implements GUIFactory {
public Button createButton() {
return new WinButton();
}
}
class OSXFactory implements GUIFactory {
public Button createButton() {
return new OSXButton();
}
}
interface Button {
public void paint();
}
class WinButton implements Button {
public void paint() {
System.out.println("I'm a WinButton");
}
}
class OSXButton implements Button {
public void paint() {
System.out.println("I'm an OSXButton");
}
}
```

```
class Application {
public Application(GUIFactory factory) {
Button button = factory.createButton();
button.paint();
        }
}
public class ApplicationRunner {
public static void main(String[] args) {
new Application(createOsSpecificFactory());
}
public static GUIFactory createOsSpecificFactory() {
int sys = readFromConfigFile("OS_TYPE");
if (sys == 0) {
        return new WinFactory();
}
else {
        return new OSXFactory();
}
}
}
```

Builder

Cieľom je abstrakcia krokov potrebných na vytvorenie objektov tak, že rôzne implementácie môžu vytvoriť rozličné reprezentácie.

Často sa tento vzor kombinuje s štrukturálnym Composite vzorom

Builder

- Abstraktné rozhranie na vytvorenie objektov (Product)

Concrete Builder

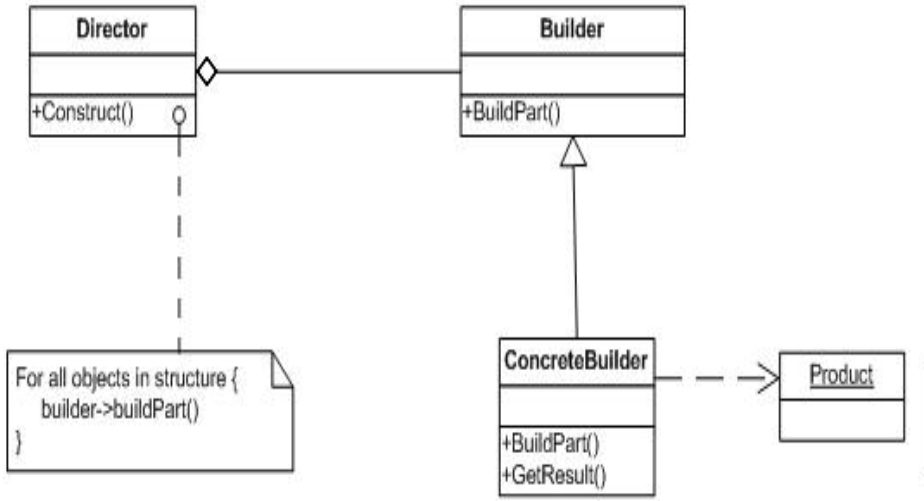
- Konkrétna implementácia Buildera zodpovedného za vytvorenie produktu a zloženie jeho častí.

Director

- Director zodpovedá za správnu sekvenciu pri tvorbe objektov tvoriacich produkt. Pri tvorbe produktu používa ako parameter Concrete Builder.

Product

- Konečný výsledok vytvorený Directorom, ktorý využíva Builder.



Builder

```
 /** "Product" */
class Pizza {
private String dough = "";
private String sauce = "";
private String topping = "";
public void setDough(String dough) {
this.dough = dough;
}
public void setSauce(String sauce) {
this.sauce = sauce;
}
public void setTopping(String topping) {
this.topping = topping;
}}
/** "Abstract Builder" */
abstract class PizzaBuilder {
protected Pizza pizza;
public Pizza getPizza() {
return pizza;
}
public void createNewPizzaProduct() {
pizza = new Pizza();
}
public abstract void buildDough();
public abstract void buildSauce();
public abstract void buildTopping();
}
```

```
/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
public void buildDough() {
pizza.setDough("cross");
}
public void buildSauce() {
pizza.setSauce("mild");
public void buildTopping() {
pizza.setTopping("ham+pineapple");
}}
/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
public void buildDough() {
pizza.setDough("pan baked");
}
public void buildSauce() {
pizza.setSauce("hot");
}
public void buildTopping() {
pizza.setTopping("pepperoni+salami");
}}
/** "Director" */
class Cook {
private PizzaBuilder pizzaBuilder;
public void setPizzaBuilder(PizzaBuilder pb) {
pizzaBuilder = pb;
}
public Pizza getPizza() {
return pizzaBuilder.getPizza();
}
public void constructPizza() {
pizzaBuilder.createNewPizzaProduct();
pizzaBuilder.buildDough();
pizzaBuilder.buildSauce();
pizzaBuilder.buildTopping();
}}
```

Builder

```
 /** A given type of pizza being constructed. */
public class BuilderExample {
public static void main(String[] args) {
Cook cook = new Cook();
PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

cook.setPizzaBuilder(hawaiianPizzaBuilder);
cook.constructPizza();

Pizza hawaiian = cook.getPizza();

cook.setPizzaBuilder(spicyPizzaBuilder);
cook.constructPizza();

Pizza spicy = cook.getPizza();
}
}
```

Lazy initialization

Cieľom je pozdržanie vytvorenia objektu alebo skupiny objektov až do okamihu ich prvého použitia.

```
public class Fruit {

   private static final Map<String,Fruit> types = new HashMap<String, Fruit>();
   private final String type;

   // using a private constructor to force use of the factory method.
   private Fruit(String type) {
      this.type = type;
   }

   /**
   * Lazy Factory method, gets the Fruit instance associated with a
   * certain type. Instantiates new ones as needed.
   * @param type Any string that describes a fruit type, e.g. "apple"
   * @return The Fruit instance associated with that type.
   */
   public static synchronized Fruit getFruit(String type) {
      if(!types.containsKey(type)) {
         types.put(type, new Fruit(type)); // Lazy initialization
      }
      return types.get(type);
   }
}
```

Object pool

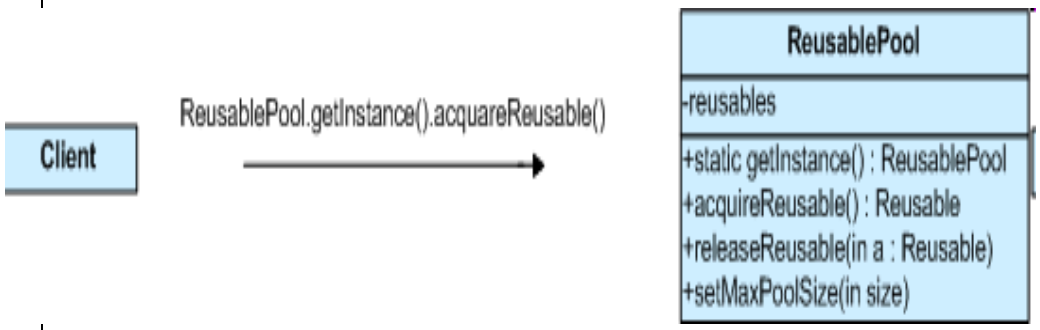
Cieľom je rýchle získanie/zrušenie initializovaného objektu

Object Pool je množina objektov initializovaná k okamžitému použitiu, ktoré sú získavané alebo vracané podľa potreby.

Object pool je zodpovedný za initializáciu novo-získaného objetu a zneplatnenie vráteného objektu

Nástrahy

- Všetky objekty boli už vydané
- Ak pool môže dynamicky rásť pri jeho zväčšovaní môže dlhšie trvať získanie nového objektu
- V multithreaded prostredí pri konštantnej veľkosti poolu, thread žiadajúci objekt môže byť blokovaný pokiaľ iný thread nevráti objekt do poolu.
- Používanie neplatného objektu. Objekt bol referencovaný 2 klientmi a jeden ho vrátil bez upovedomenia druhého používateľa.



Prototype

Objekt je vytváraný klonovaním protypu. Konkrétne subtriedy prototypu vytvárajú klony bez toho, aby klient vedel a akú konkrétnu triedu sa jedná.

```
abstract class Prototype implements Cloneable {
   public Object clone() throws CloneNotSupportedException {
      // call Object.clone()
      Prototype copy = (Prototype) super.clone();
      //In an actual implementation of this pattern you might now change references to
      //the expensive to produce parts from the copies that are held inside the prototype.
      return copy;
   }
   abstract void prototype(int x);
   abstract void printValue();
}

/**
 * Concrete Prototypes to clone
 */
class PrototypeImpl extends Prototype {
   int x;
   public PrototypeImpl(int x) {
      this.x = x;
   }
   @Override
   void prototype(int x) {
      this.x = x;
   }
   public void printValue() {
      System.out.println("Value :" + x);
   }
}
```

Singleton

Koncept ktorý garantuje vytvorenie jedinej inštancie danej triedy v systéme.
Často je kombinovaný z Abstract Factory, Builder, alebo Prototype

```
public class Singleton {
   private static final Singleton INSTANCE = new Singleton();
   // Private constructor prevents instantiation from other classes
   private Singleton() {
   }
   public static Singleton getInstance() {
      return INSTANCE;
   }
}
/* thread safe implementation */
public class Singleton {
  // Private constructor prevents instantiation from other classes
  private Singleton() {
  }
  /**
   * SingletonHolder is loaded on the first execution of Singleton.getInstance()
   * or the first access to SingletonHolder.INSTANCE, not before.
   */
  private static class SingletonHolder {
    public static final Singleton INSTANCE = new Singleton();
  }
  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Multiton

Koncept rozširuje koncept Singleton o jedinečné pomenované inštancie.
Singleton garantuje jednu inštanciu v celom systéme, Multiton garantuje jednu inštanciu s daným menom

```
public class FooMultiton {
    private static final Map<Object, FooMultiton> instances = new HashMap<Object, FooMultiton>();
    private FooMultiton() /* also acceptable: protected, {default} */ {
        /* no explicit implementation */
    }

    public static FooMultiton getInstance(Object key) {
        synchronized (instances) {

            // Our "per key" singleton
            FooMultiton instance = instances.get(key);

            if (instance == null) {

                // Lazily create instance
                instance = new FooMultiton();

                // Add it to map
                instances.put(key, instance);
            }

            return instance;
        }
    }
    // other fields and methods ...
}
```

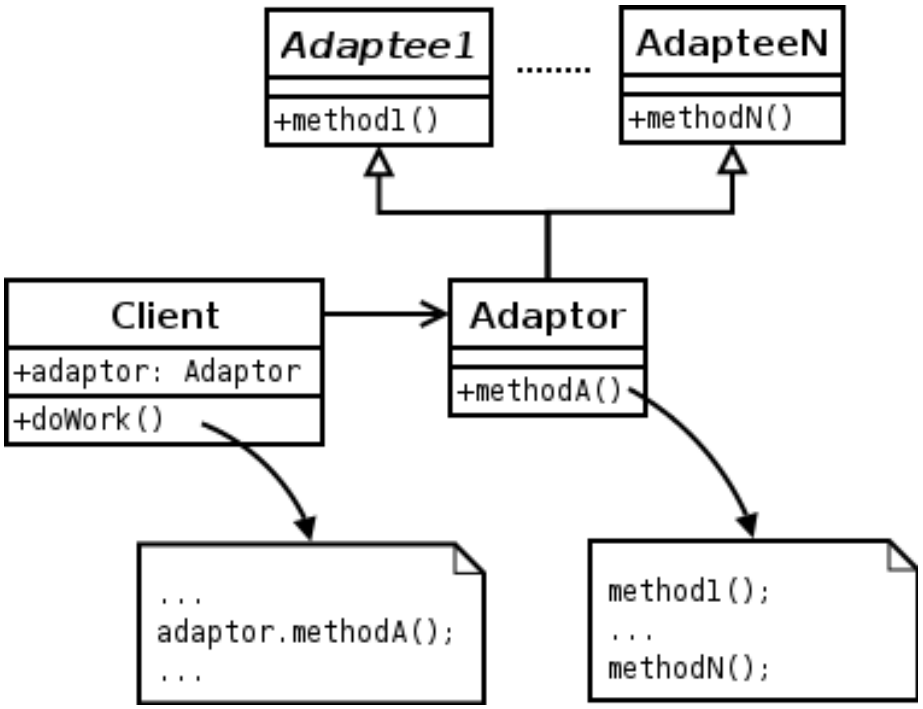
Structural Patterns: Štrukturálny vzory

- Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- Bridge decouples an abstraction from its implementation so that the two can vary independently.
- Composite composes zero-or-more similar objects so that they can be manipulated as one object.
- Decorator dynamically adds/overrides behaviour in an existing method of an object.
- Facade provides a simplified interface to a large body of code.
- Flyweight reduces the cost of creating and manipulating a large number of similar objects.
- Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Adapter/Wrapper

Je návrhový vzor, ktorý transformuje jeden interface do iného kompatibilného interfacu.

Príklady: tranformácia int2boolean alebo rôzne formáty dátumov



Bridge

Je návrhový vzor, ktorý oddeľuje abstrakciu od jej konkrétnej implementácie, čím umožňuje viaceré implementácie tejto abstrakcie.

Abstraction definuje abstraktný interface

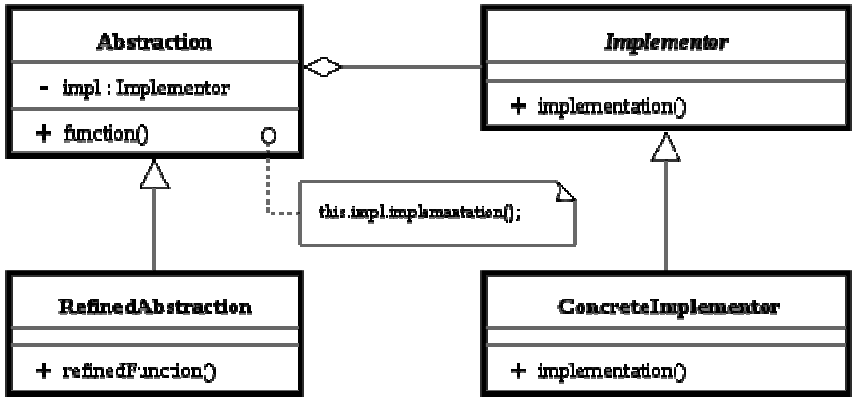
a referuje Implementora

RefinedAbstraction rozširuje interface Abstraction

Implementor defineje interface pre implementačné triedy

ConcreteImplementor implementuje Implementor interface

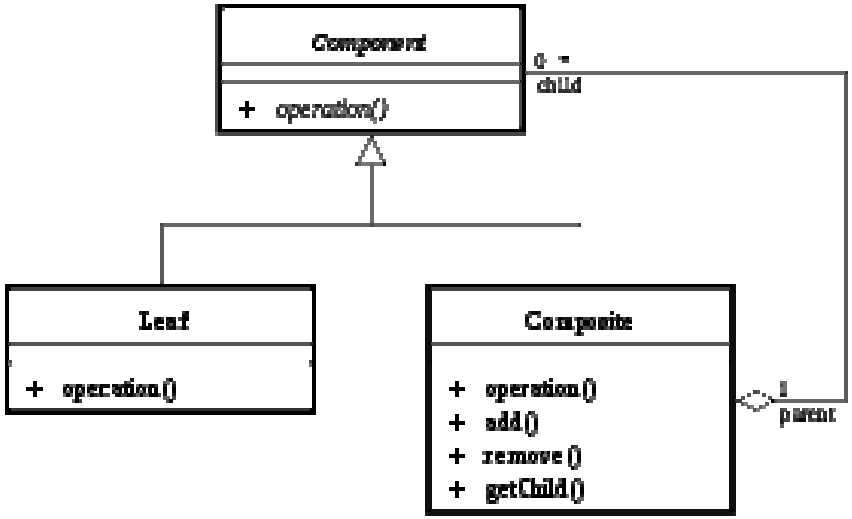
Príklad, kedy je návrhový vzor nahradený prostriedkami jazyka Java



Composite

Je návrhový vzor, ktorý umožňuje zaobchádzať so skupinou objektov, ako keby to bol jeden objekt. Zámerom vzoru je realizovať „has-a“ hierarchiu. Používa sa ak potrebujeme narábať rovnako s štrukturovaným objektom rovnako ako s primitívnym.

Príklady: resizing of shapes



Composite

```
import java.util.List;
import java.util.ArrayList;
/** "Component" */
interface Graphic {
    //Prints the graphic.
    public void print();
}
/** "Composite" */
class CompositeGraphic implements Graphic {
    //Collection of child graphics.
    private List<Graphic> mChildGraphics = new ArrayList<Graphic>();
    //Prints the graphic.
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }
    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}
```

```
/** "Leaf" */
class Ellipse implements Graphic {
    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
public class Program {
    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();
        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();
        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);
        graphic2.add(ellipse4);
        graphic.add(graphic1);
        graphic.add(graphic2);
        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();
    }
}
```

Decorator

Je návrhový vzor, ktorý umožňuje dynamicky rozšíriť (dekorovať) funkcionalitu danej triedy .

Príklady: meranie doby prístupu do databázy

Postup:

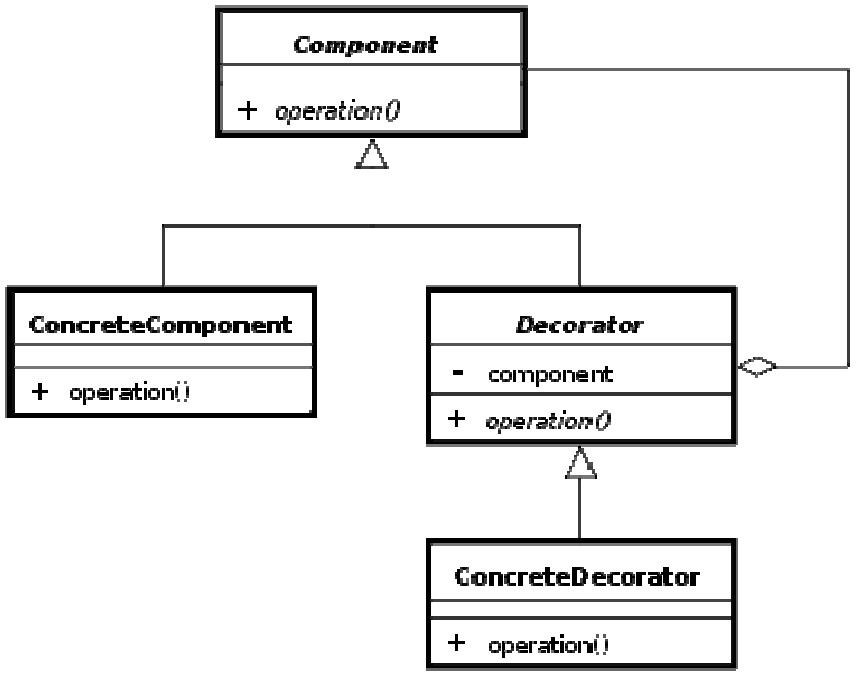
• Vytvor podtriedu "Decorator" pôvodnej triedy "Component"

• V triede Decorator pridaj referenciu na ištanciu Component

• V konštruktori Decorator pridaj parameter Component a initializuj Component referenciu;

• V triede Decorator presmeruj všetky "Component" metódy na "Component" ištanciu

• V triede ConcreteDecorator , override tie Component metódy, ktoré potrebujú rozšírenie.



Decorator

```
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the Window
}
// implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }
    public String getDescription() {
        return "simple window";
    }
}
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow; // the Window being decorated

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
    public void draw() {
        decoratedWindow.draw();
    }
}
// the first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        decoratedWindow.draw();
        drawVerticalScrollBar();
    }
    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical scrollbars";
    }
}
// the second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {...}
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

Facade

Je návrhový vzor, ktorý poskytuje zjednodušený interface ku skupine objektov alebo knižníc.

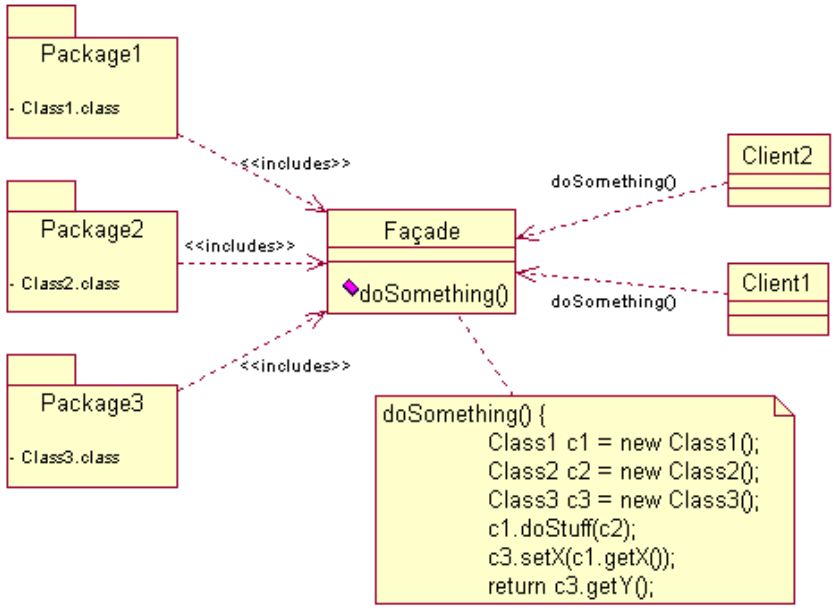
Príklady:

•zjednodušené a často sa opakujúce použitia knižníc alebo frameworkov

•Zlepšenie čitateľnosti kódu

•Zníženie závislostí od modulov

•Zlepšenie kvality



Flyweight

Je návrhový vzor, ktorý minimalizuje spotrebu zdrojov (pamäť alebo DB priestor) tým, že umožňuje zdieľanie týchto zdrojov medzi podobnými objektami.

Príklady:

- Glyph objekt reprezentujúci formátovacie vlastnosti znaku v text editore.
- Character obsahuje len nezdieľateľné atribúty (pozíciu) a referenciu na glyph
- Produkt vs. Kontrakt
- Vyzváňajúci tón v telefónnej ústredni

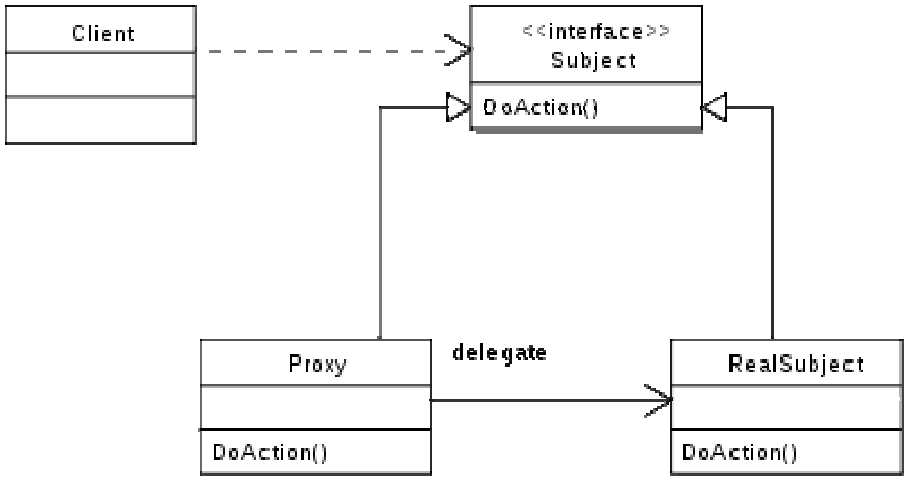
Proxy

Je návrhový vzor, ktorý je najvšeobecnejšou formou, kde jedna trieda slúži ako interface pre druhú triedu.

Príklady:

•Proxy k sieťové pripojeniu, veĽkých objektov v pamäti alebo databáze, ktoré sú nákladné alebo nemožné duplikovať

•Reference counting pointer používaných v garbage collectors



Proxy

```
interface Image
{
   void displayImage();
}
class RealImage implements Image
{
   private String filename;
   public RealImage(String filename)
   {
      this.filename = filename;
      loadImageFromDisk();
   }
   private void loadImageFromDisk()
   {
      System.out.println("Loading   " + filename);
   }
   public void displayImage()
   {
      System.out.println("Displaying " + filename);
   }
}
```

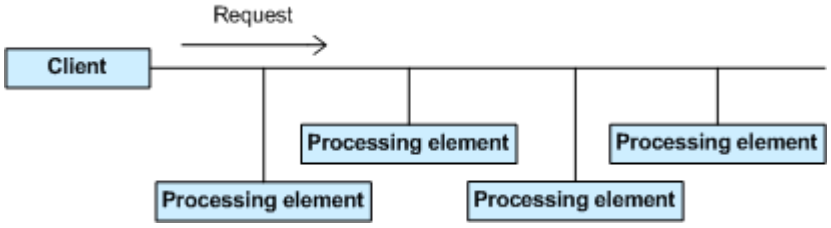
```
//on System B
class ProxyImage implements Image
{
   private String filename;
   private RealImage image;
   public ProxyImage(String filename)
   {
      this.filename = filename;
   }
   public void displayImage()
   {
      if (image == null)
      {
        image = new RealImage(filename);
      }
      image.displayImage();
   }
}
class ProxyExample
{
   public static void main(String[] args)
   {
      Image image1 = new ProxyImage("HiRes_10MB_Photo1");
      Image image2 = new ProxyImage("HiRes_10MB_Photo2");
      image1.displayImage(); // loading necessary
      image2.displayImage(); // loading necessary
      image1.displayImage(); // loading unnecessary
   }
}
```

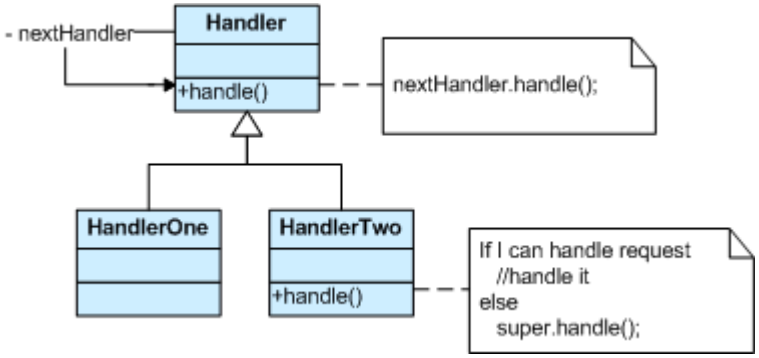
Behavioral patterns: Vzory správania

- Chain of responsibility delegates commands to a chain of processing objects.
- Command creates objects which encapsulate actions and parameters.
- Interpreter implements a specialized language.
- Iterator accesses the elements of an object sequentially without exposing its underlying representation.
- Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- Memento provides the ability to restore an object to its previous state (undo).
- Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.
- State allows an object to alter its behavior when its internal state changes.
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
- Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object

Chain of responsibility / Reťazenie zodpovedností

- Je návrhový vzor, ktorého cieľom je oddeliť klienta vysielajúceho žiadosť (request)od množiny spracujúcich objektov (handlerov) .
- Štruktúra zreťazených handlerov nie je známa klientovi, dokonca štruktúra môže byť menená dynamicky.
- Request prechádza cez chain handlerov, až pokiaľ nie je úplne spracovaný, t.j konkrétny handler sa rozhodne už ďalej neposielať request

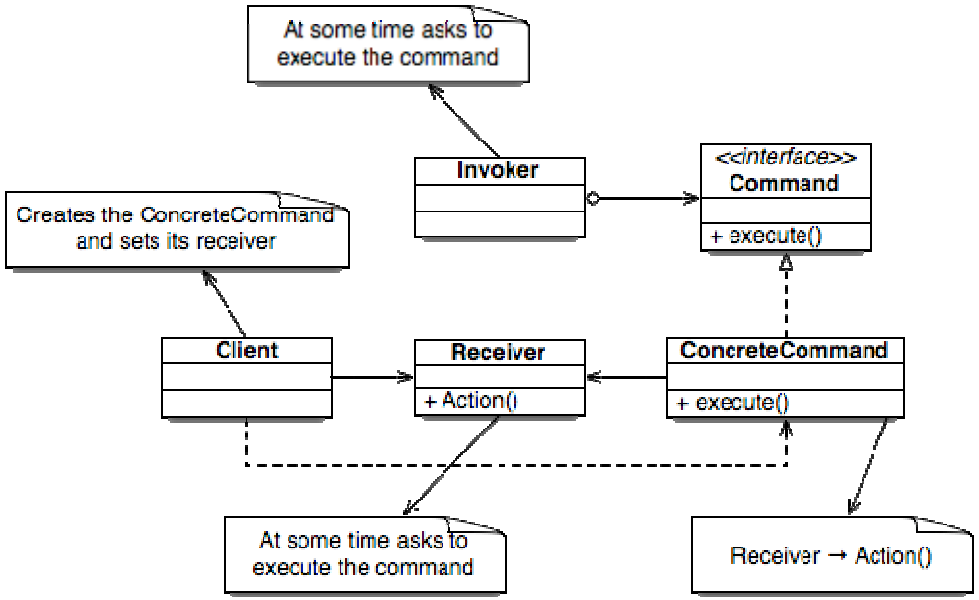




Command / Príkaz

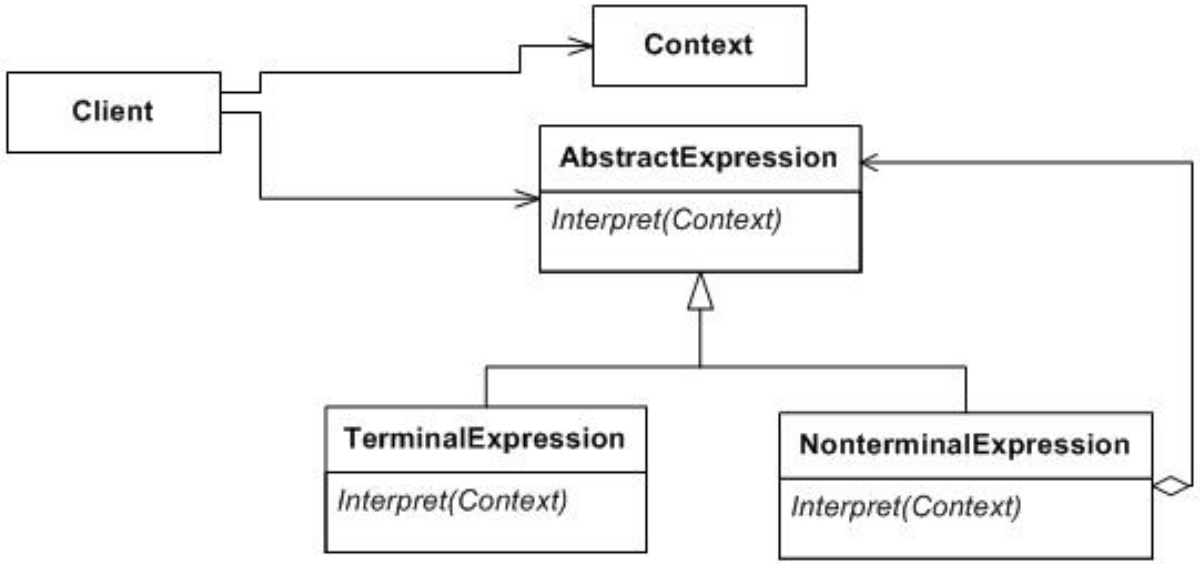
•Je návrhový vzor, ktorý zapuzdruje všetky informácie potrebné na vykonanie metódy v neskoršom čase : meno metódy, objekt prijímajúci metódu a všetky parametre metódy.

•Oddeľuje objekt, ktorý vyvoláva metódu od objektu, ktorý vie akú metódu treba vykonať



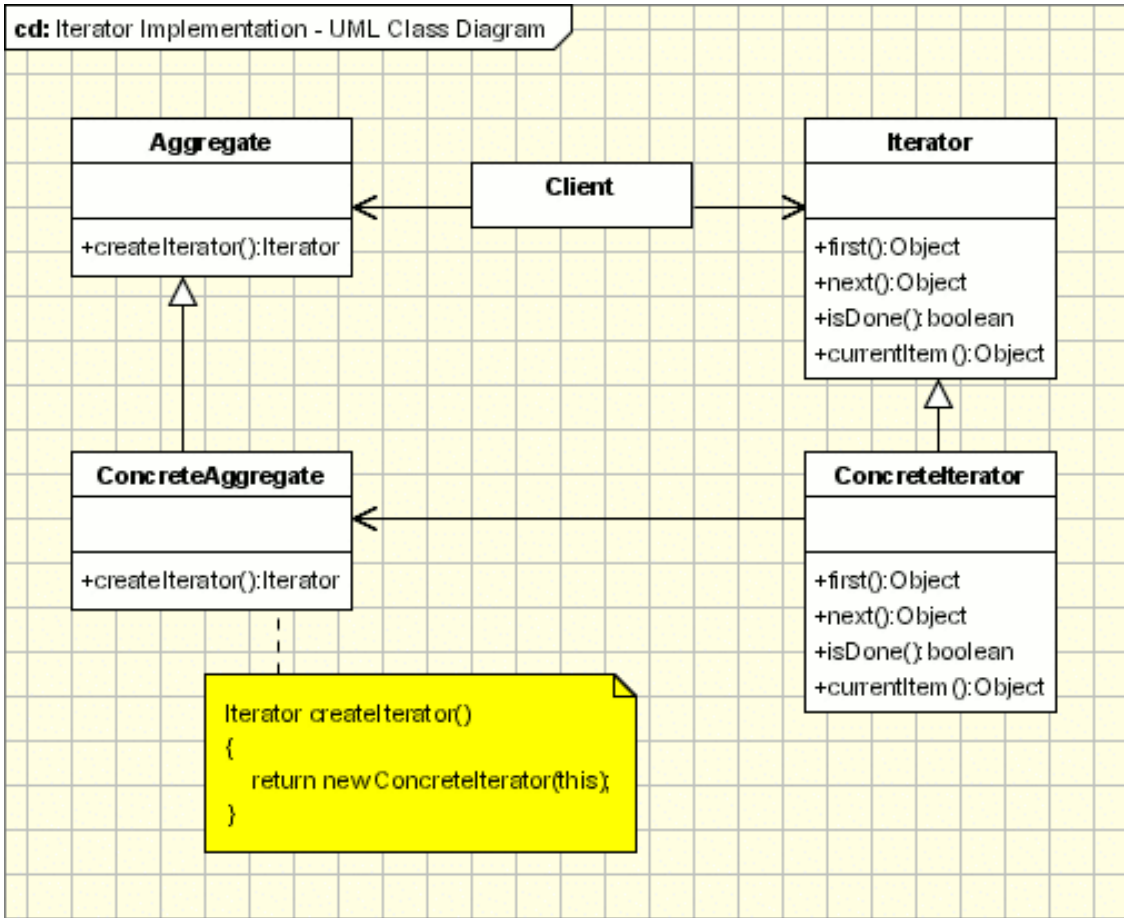
Interpreter

- Je návrhový vzor, ktorý interpretuje vety jazyka.
- Každý symbol (koncový i nekoncový) je reprezentovaný triedou
- Syntaktický strom je inštanciu Composite vzoru, ktorý interpretuje vety jazyka



Iterator

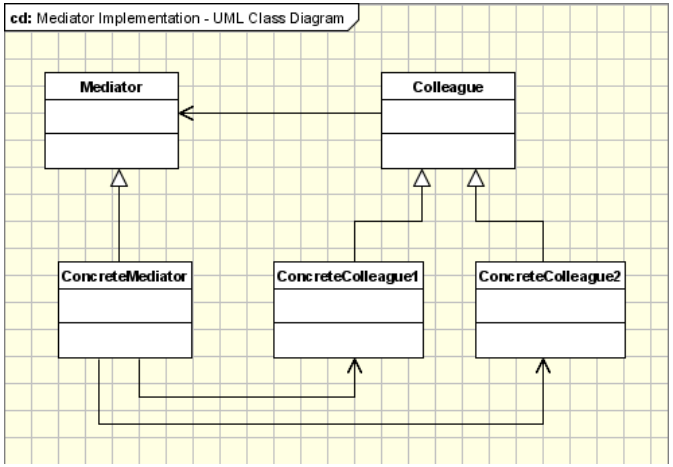
- Je návrhový vzor, ktorý prechádza a sprístupňuje jednotlivé položky množiny .
- Podporovaný už vo viacerých jazykoch



Iterator

•Je návrhový vzor, ktorý definuje objekt (mediator), ktorý zapúzdruje ako množina objektov vzájomne inter-reagujú .

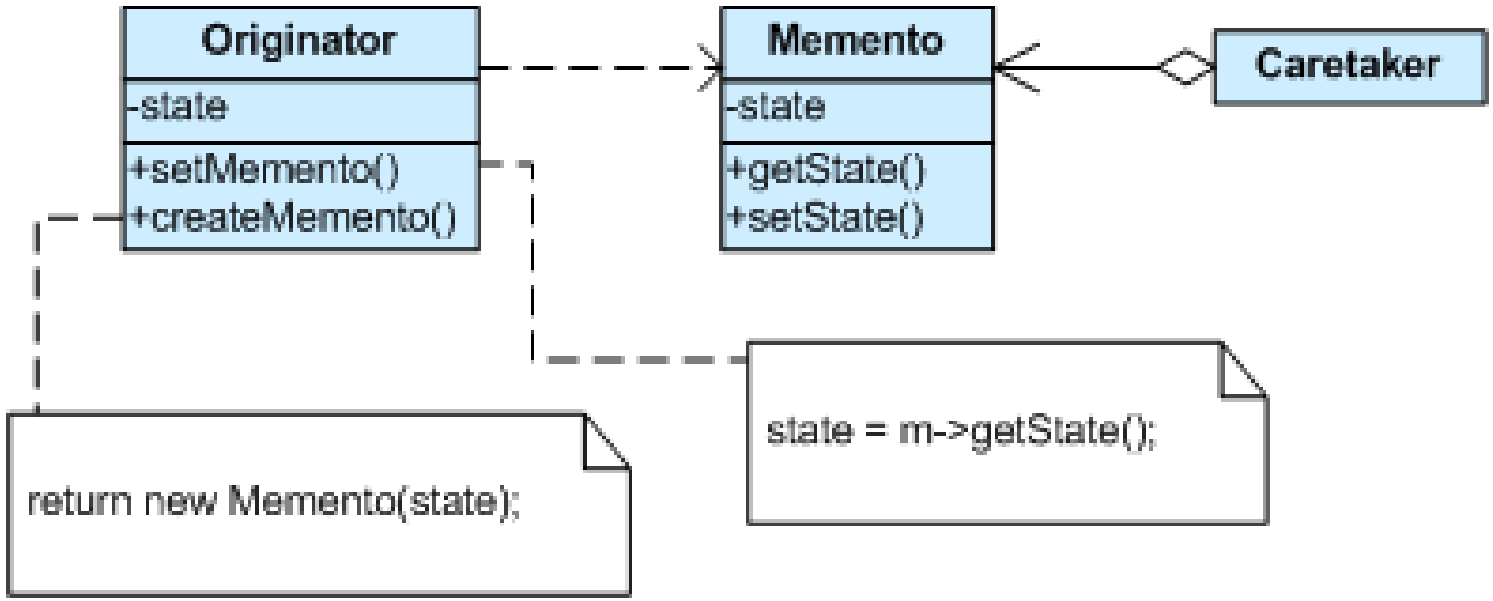
•Mediátor oddeľuje vzájomne spolupracujúce objekty a skrýva spôsob aký spolupracujú



Memento

•Je návrhový vzor, ktorý umožňuje vrátiť stav objektu do predchádzajúceho stavu (undo/rollback) bez porušenia zapúzdrenia stavu .

•Caretaker si vyžiada od Originatora Memento, ktoré zachycuje stav Originatora v čase vyžiadania. Iba Originator ma prístup k vnútornému stavu uloženému v Memente. Ak Originator treba vrátiť do pôvodného stavu tak , Caretaker mu zašle Memento s príkazom obnovenia do stavu uloženého v Memente

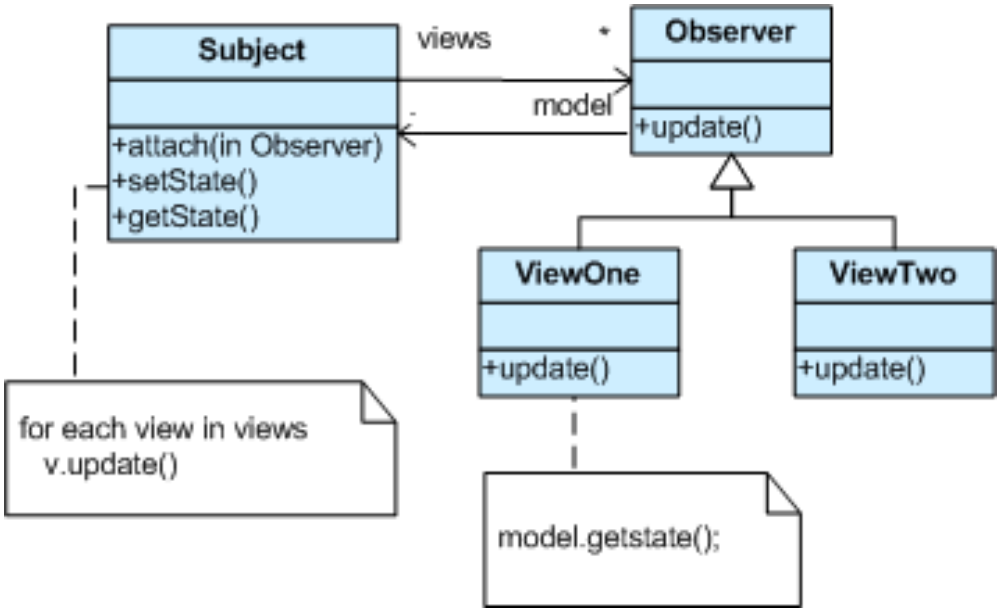


Observer

•Je návrhový vzor, ktorý rieši závislosť viacerých závislých objektov (dependants) od stavu jedného objektu.

•Spoločná časť je abstrahované do Subject. Rozdielna a variabilná časť je reprezentovaná hierarchiou Observerov.

•Subject pozná všetkých svojich registrovaných observerov, ktorých informuje pri každej svojej zmene.

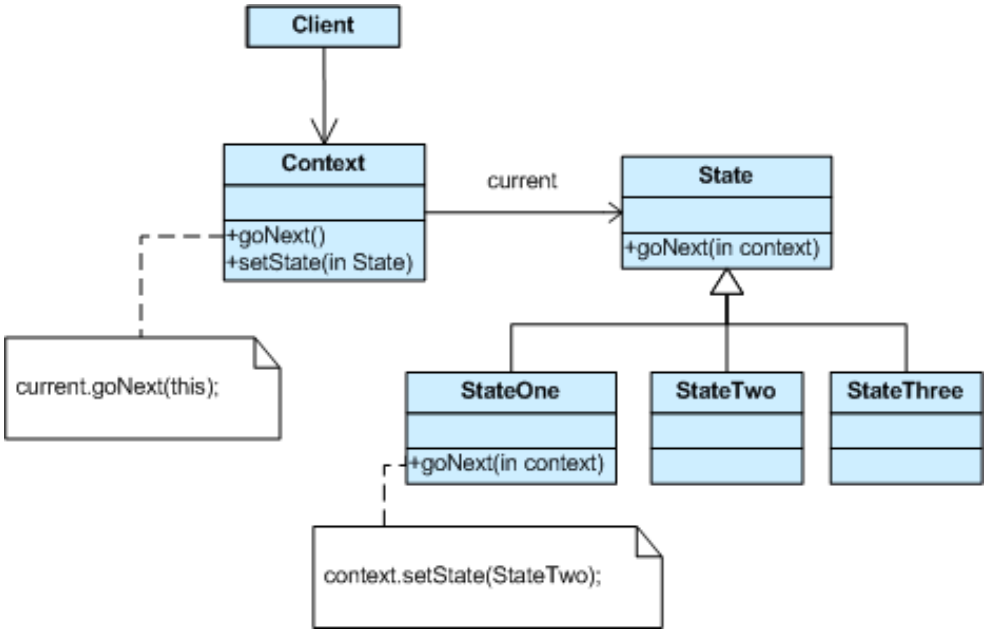


State

•Je návrhový vzor, ktorý zabezpečuje zmenu správania objektu dynamicky.

•Context je objekt, ktoré stav reprezentovaný jednou z konkrétnych inštancií State.

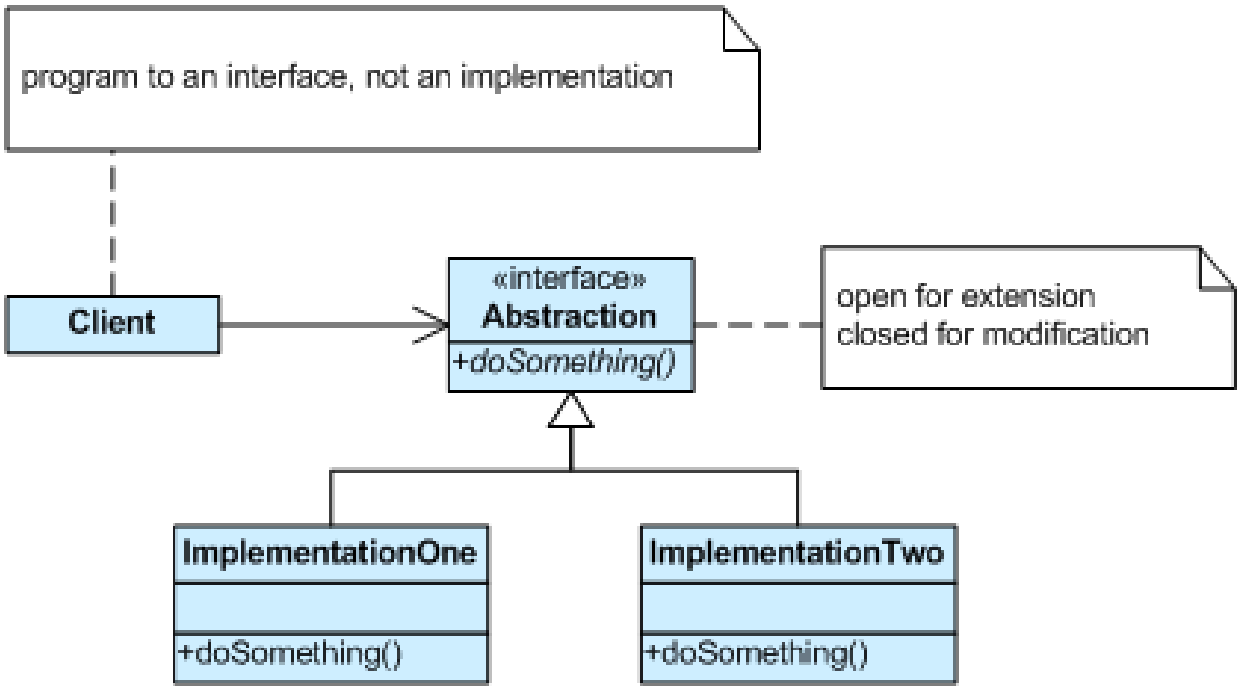
•Prechody medzi jednotlivým objektmi môžu byť centrálne definované buď v Context-e (table driven state transitions) alebo v jednotlivých sub-triedach State, kde sa jednoduchšie vykonávajú sprievodné akcie spojené so zmenou stavu.



Strategy

•Je návrhový vzor, ktorý zapúzdruje skupinu algoritmov, čím ich robí vzájomne vymeniteľnými.

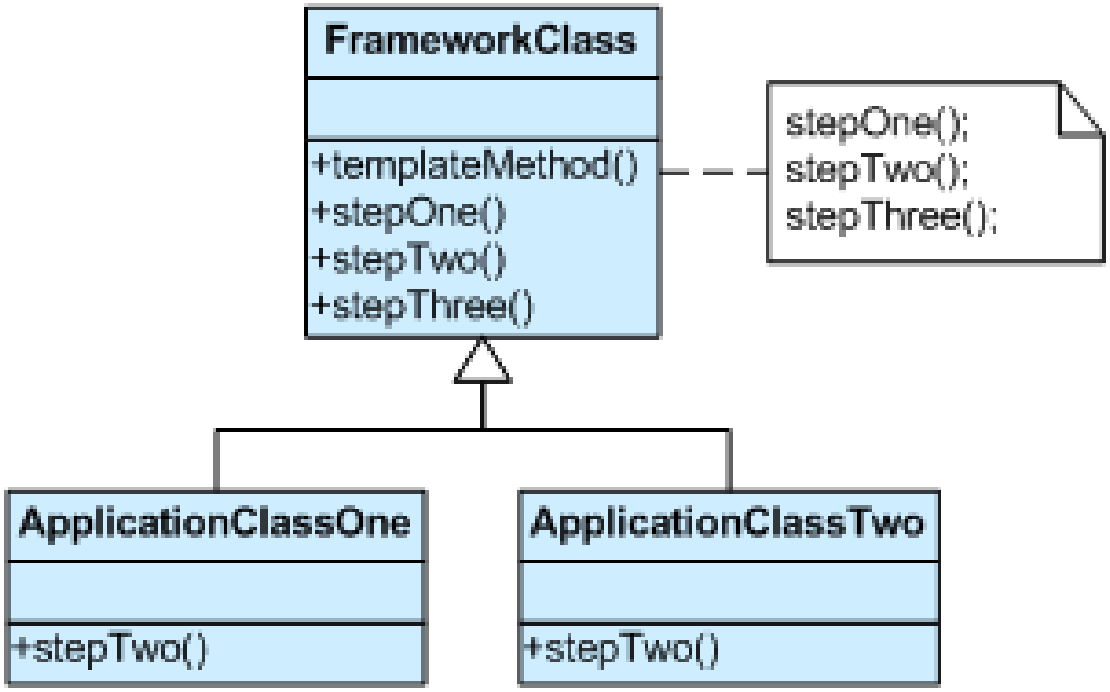
•Client sa viaže len na definovanú Abstrakciu a nezaoberá sa konkrétnymi detailmi konkrétnej Implementácie



Template method

•Je návrhový vzor, ktorý predstavuje skeleton algoritmu reprezentovaný jednotlivými krokmi

•Podtriedy môžu svojim spôsobom uskutočniť predpísaný krok.



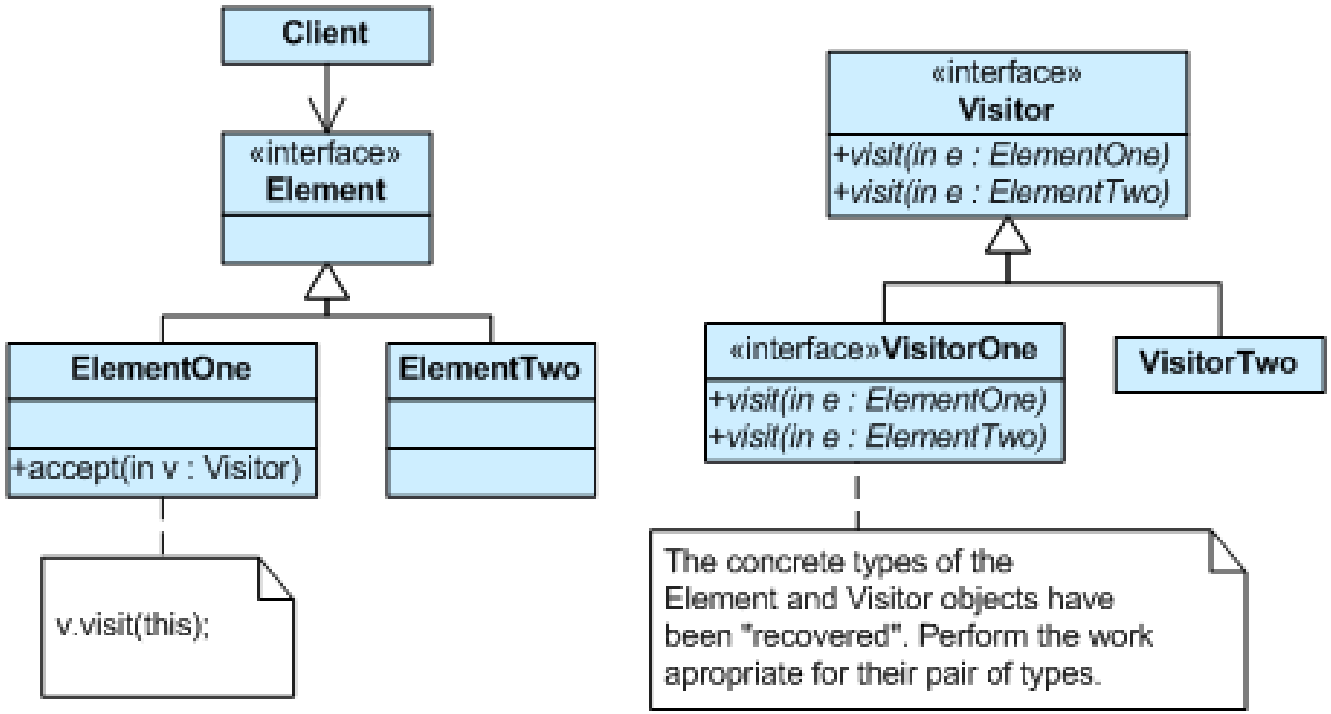
Visitor

- Je návrhový vzor, ktorý umožňuje vykonať operáciu na elementoch štruktúry.
- Visitor umožňuje definovať novú operáciu bez zmeny tried elementov, na ktorých sa táto operácia má vykonať
- Použitie, ak vykonaná akcia závisí od 2 objektov (double-dispatch).
- Hierarchia jednoduchých elementov odráža len ich štruktúru
- Operácie nad týmito elementmi sú v separátnej hierarchii Visitorov.
- Nová operácia elementom sa pridáva cez nového Visitora

Visitor

•Client volá na konkrétnom elemente operáciu accept s parametrom konkrétnou inštanciou Visitora napríklad: accept(new VisitorOne())

•Konkrétny element napr. elementOne volá visitor.visitElementOne(this)



Visitor píklad

•Napíšte Java kód, ktorý umozžní dynamicky pridať operáciu nad štruktúrou auta, ktoré sa skladá s kolies, motora a karosérie