Laboratorul 9: Design patterns - Factory, Strategy, Observer, Command, Builder

Video introductiv: link [https://www.youtube.com/watch?v=khRqWZrz1YU] + link [https://www.youtube.com/watch?v=F2HMN4mvVYY]

Obiective

Scopul acestui laborator este familiarizarea cu folosirea unor pattern-uri des întâlnite în design-ul atât al aplicațiilor, cât și al API-urilor - *Factory*, *Strategy*, *Observer*, *Command* și *Builder*.

Introducere

Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucata de cod ce poate fi aplicata direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

Se consideră că există aproximativ 2000 de design patterns [2] [http://ptgmedia.pearsoncmg.com/images/9780321711922/samplepages/0321711920.pdf], iar principalul mod de a le clasifica este următorul:

- **“Gang of Four” patterns**:
- Concurrency patterns
- Architectural patterns - sunt folosite la un nivel mai inalt decat design patterns, stabilesc nivele și componente ale sistemelor/aplicațiilor, interacțiuni între acestea (e.g. Model View Controller şi derivatele sale). Acestea descriu structura întregului sistem, iar multe framework-uri vin cu ele deja încoporate, sau faciliteaza aplicarea lor (e.g. Java Spring). În cadrul laboratoarelor nu ne vom lega de acestea.

O carte de referință pentru design patterns este “Design Patterns: Elements of Reusable Object-Oriented Software” [1], denumită și “Gang of Four” (GoF). Aceasta definește 23 de design patterns, foarte cunoscute și utilizate în prezent. Aplicațiile pot încorpora mai multe pattern-uri pentru a reprezenta legături dintre diverse componente (clase, module). În afară de GoF, și alți autori au adus în discuție pattern-uri orientate în special pentru aplicațiile enterprise și cele distribuite.

Pattern-urile GoF sunt clasificate în felul următor:

- **Creational Patterns** - definesc mecanisme de creare a obiectelor
  - Singleton, Factory etc.
- **Structural Patterns** - definesc relații între entități
  - Decorator, Adapter, Facade, Composite, Proxy etc.
- **Behavioural Patterns** - definesc comunicarea între entități
  - Visitor, Observer, Command, Mediator, Strategy etc.

Design pattern-urile nu trebuie privite drept niște rețete care pot fi aplicate direct pentru a rezolva o problemă din design-ul aplicației, pentru că de multe ori pot complica inutil arhitectura. Trebuie întâi înțeles dacă este cazul să fie aplicat un anumit pattern, si de-abia apoi adaptat pentru situația respectivă. Este foarte probabil chiar să folosiți un pattern (sau o abordare foarte similară acestuia) fără să vă dați seama sau să îl numiți explicit. Ce e important de reținut după studierea acestor pattern-uri este un mod de a aborda o problemă de design.

În laboratoarele precedente au fost descrise patternurile Singleton și Visitor. *Singleton* este un pattern creațional, simplu, a cărui folosire este controversată (vedeți în laborator explicația cu anti-pattern). *Visitor* este un pattern comportamental, și după cum ați observat oferă avantaje în anumite situații, în timp ce pentru altele nu este potrivit. Pattern-urile comportamentale modelează interacțiunile dintre clasele și componentele unei aplicații, fiind folosite în cazurile în care vrem sa facem un design mai clar și ușor de adaptat și extins.

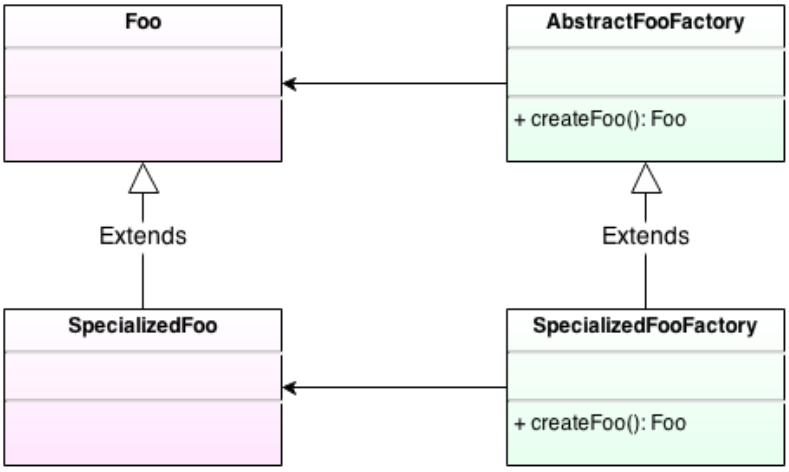
Factory

Patternurile de tip Factory sunt folosite pentru obiecte care generează instanțe de clase înrudite (implementează aceeași interfață, moștenesc aceeași clasă abstractă). Acestea sunt utilizate atunci când dorim să izolăm obiectul care are nevoie de o instanță de un anumit tip, de creearea efectivă acesteia. În plus clasa care va folosi instanța nici nu are nevoie să specifice exact subclasa obiectului ce urmează a fi creat, deci nu trebuie să cunoască toate implementările acelui tip, ci doar ce caracteristici trebuie să aibă obiectul creat. Din acest motiv, Factory face parte din categoria *Creational Patterns*, deoarece oferă o soluție legată de creearea obiectelor.

Aplicabilitate:

- În biblioteci/API-uri, utilizatorul este separat de implementarea efectivă a tipului și trebuie sa folosească metode factory pentru a obține anumite obiecte. Clase care oferă o astfel de funcționalitate puteți găsi și in core api-ul de Java, in api-ul java.nio (e.g. clasa FileSystems [http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystems.html]), în Android SDK (e.g. clasa SocketFactory [http://developer.android.com/reference/javax/net/SocketFactory.html]) etc.
- **Atunci când crearea obiectelor este mai complexă** (trebuie realizate mai multe etape etc.), este mai util să separăm logica necesară instanțierii subtipului de clasa care are nevoie de acea instanță. Asta înseamnă că puteți folosi metode factory care să vă construiască obiectul și dacă aveți doar un tip, nu mai multe.

Abstract Factory Pattern



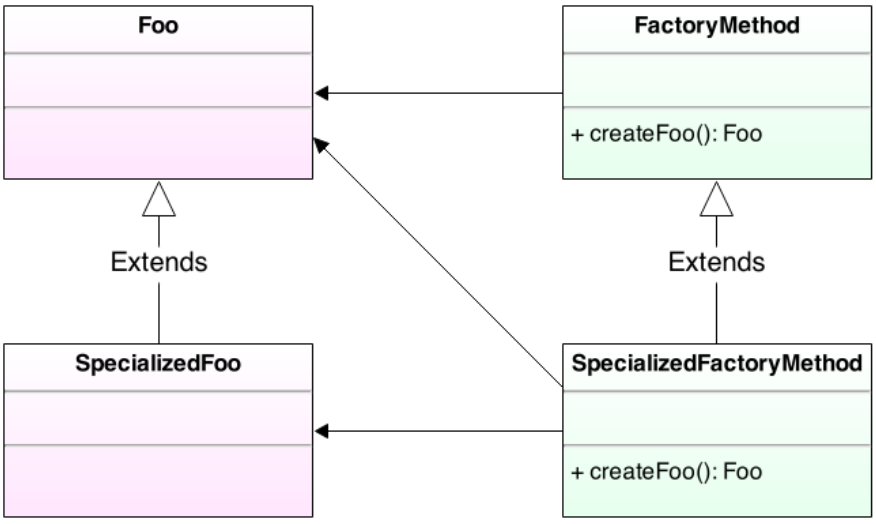
Codul următor corespunde diagramei din figure 1 . În acest caz folosim interfețe pentru factory și pentru tip, însă în alte situații putem să avem direct *SpecializedFooFactory,* fără a implementa interfața *FooFactory*.

```
public interface Foo {
    public void bar();
}
public interface FooFactory {
    public Foo createFoo();
}
public class SpecializedFoo implements Foo {
    ...
}
public class SpecializedFooFactory implements FooFactory {
    public Foo createFoo() {
        return new SpecializedFoo();
    }
}
```

Factory Method Pattern

Folosind pattern-ul Factory Method se poate defini o interfață pentru crearea unui obiect. Clientul care apelează metoda factory nu știe/nu îl interesează de ce subtip va fi la runtime instanța primită.

Spre deosebire de Abstract Factory, Factory Method ascunde construcția unui obiect, nu a unei familii de obiecte “inrudite”, care extind un anumit tip. Clasele care implementează Abstract Factory conțin de obicei mai multe metode factory.



Exemplu

Situația cea mai întâlnită în care se potrivește acest pattern este aceea când trebuie instanțiate multe clase care implementează o anumită interfață sau extind o altă clasă (eventual abstractă), ca în exemplul de mai jos. Clasa care folosește aceste subclase nu trebuie să “știe” tipul lor concret ci doar pe al părintelui. Implementarea de mai jos corespunde pattern-ului Abstract Factory pentru clasa *PizzaFactory*, și foloseste factory method pentru metoda `createPizza`.

PizzaLover.java

```
abstract class Pizza {
    public abstract double getPrice();
```

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

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.");
    }
}
public class PizzaLover {
    public static void main (String args[]) {
        for (PizzaFactory.PizzaType pizzaType : PizzaFactory.PizzaType.values()) {
            System.out.println("Price of " + pizzaType + " is " + PizzaFactory.createPizza(pizzaType).getPrice());
        }
    }
}
```

```
Output:
Price of HamMushroom is 8.5
Price of Deluxe is 10.5
Price of Hawaiian is 11.5
```

Singleton Factory

De obicei avem nevoie ca o clasă factory să fie utilizată din mai multe componente ale aplicației. Ca să economisim memorie este suficient să avem o singură instanță a factory-ului și să o folosim pe aceasta. Folosind pattern-ul Singleton putem face clasa factory un singleton, și astfel din mai multe clase putem obține instanță acesteia.

Un exemplu ar fi Java Abstract Window Toolkit (AWT [http://en.wikipedia.org/wiki/Abstract_Window_Toolkit]) ce oferă clasa abstractă java.awt.Toolkit [http://docs.oracle.com/javase/8/docs/api/java/awt/Toolkit.html] care face legătura dintre componentele AWT și implementările native din toolkit. Clasa *Toolkit* are o metodă factory `Toolkit.getDefaultToolkit()` ce întoarce subclasa de *Toolkit* specifică platformei. Obiectul *Toolkit* este un Singleton deoarece AWT are nevoie de un singur obiect pentru a efectua legăturile și deoarece un astfel de obiect este destul de costisitor de creat. Metodele trebuie implementate în interiorul obiectului și nu pot fi declarate statice deoarece implementarea specifică nu este cunoscută de componentele independente de platformă.

Observer Pattern

Design Pattern-ul *Observer* definește o relație de dependență 1 la n între obiecte astfel încât când un obiect își schimbă starea, toți dependenții lui sunt notificați și actualizați automat. Folosirea acestui pattern implică existența unui obiect cu rolul de *subiect*, care are asociată o listă de obiecte dependente, cu rolul de *observatori*, pe care le apelează automat de fiecare dată când se întâmplă o acțiune.

Acest pattern este de tip *Behavioral* (comportamental), deorece facilitează o organizare mai bună a comunicației dintre clase în funcție de rolurile/comportamentul acestora.

Aplicabilitate

Observer se folosește în cazul în care mai multe clase(*observatori*) depind de comportamentul unei alte clase(*subiect*), în situații de tipul:

- o clasă implementează/reprezintă logica, componenta de bază, iar alte clase doar folosesc rezultate ale acesteia (monitorizare).
- o clasă efectuează acțiuni care apoi pot fi reprezentate în mai multe feluri de către alte clase (view-uri ca în figură de mai jos).

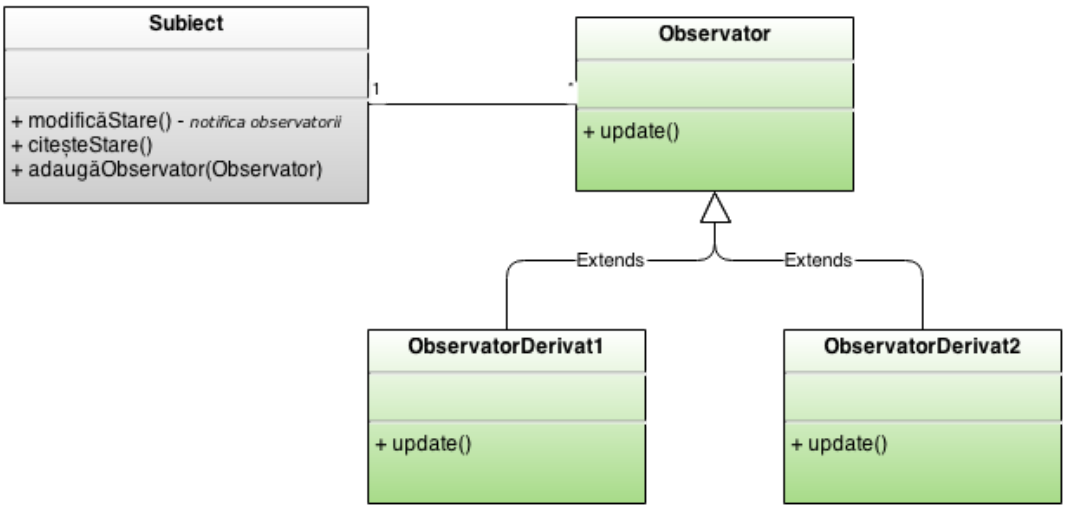
Practic în toate aceste situații clasele Observer **observă** modificările/acțiunile clasei Subject. Observarea se implementează prin **notificări inițiate din metodele clasei Subject**.

Structură

Pentru aplicarea acestui pattern, clasele aplicației trebuie să fie structurate după anumite roluri, și în funcție de acestea se stabilește comunicarea dintre ele. În exemplul din figure 3 , avem două tipuri de componente, *Subiect* și *Observator*, iar *Observator* poate fi o interfață sau o clasă abstractă ce este extinsă cu diverse implementări, pentru fiecare tip de monitorizare asupra obiectelor *Subiect*.

- observatorii folosesc datele subiectului

- observatorii sunt notificați automat de schimbări ale subiectului
- subiectul cunoaște toți observatorii
- subiectul poate adăuga noi observatori



Subiect

- nu trebuie să știe ce fac observatorii, trebuie doar să mențină referințe către obiecte de acest tip
- nu știe ce fac observatorii cu datele
- oferă o metodă de adăugare a unui *Observator*, eventual și o metodă prin care se pot deinregistra observatori
- menține o listă de referințe cu observatori
- când apar modificări (e.g. se schimbă starea sa, valorile unor variabile etc) notifică toți observatorii

Observator

- definește o interfață notificare despre schimbări în subiect
- ca implementare:
  - toți observatorii pentru un anumit subiect trebuie să implementeze această interfață
  - oferă una sau mai multe metode care să poată fi invocate de către *Subiect* pentru a notifica o schimbare. Ca argumente se poate primi chiar instanța subiectului sau obiecte speciale care reprezintă evenimentul ce a provocat schimbarea.

View/ObservatorDerivat

- implementează interfața Observator

Aceasta schemă se poate extinde, în funcție de aplicație, observatorii pot ține referințe catre subiect sau putem adauga clase speciale pentru reprezentarea evenimentelor, notificarilor. Un alt exemplu îl puteți găsi aici [http://www.research.ibm.com/designpatterns/example.htm].

Implementare

Tookit-urile GUI, cum este și Swing [http://en.wikipedia.org/wiki/Swing_%28Java%29] folosesc acest design pattern, de exemplu apăsarea unui buton generează un eveniment ce poate fi transmis mai multor *listeners* înregistrați acestuia (exemplu [http://www.programcreek.com/2009/01/the-steps-involved-in-building-a-swing-gui-application/]).

API-ul Java oferă clasele Observer [http://docs.oracle.com/javase/8/docs/api/java/util/Observer.html] și Observable [http://docs.oracle.com/javase/8/docs/api/java/util/Observable.html] care pot fi subclasate pentru a implementa propriile tipuri de obiecte ce trebuie monitorizate și observatorii acestora.

Pentru cod complex, concurent, cu evenimente asincrone, recomandăm RxJava, care folosește Observer pattern: github [https://github.com/ReactiveX/RxJava/wiki], exemplu [https://dzone.com/articles/rxjava-part-1-a-quick-introduction].

Strategy Pattern

Design pattern-ul *Strategy* încapsulează algoritmii în clase ce oferă o anumită interfață de folosire, și pot fi selecționați la runtime. Ca și Command, acest pattern este *behavioral* pentru ca permite decuplarea unor clase ce oferă un anumit comportament și folosirea lor independentă în funcție de situația de la runtime.

Acest pattern este recomandat în cazul în care avem nevoie de un tip de algoritm (strategie) cu mai multe implementări posibile si dorim să alegem dinamic care algoritm îl folosim, fără a face sistemul prea strâns cuplat.

Exemple de utilizare:

- sisteme de tip Layout Managers din API-urile pentru UI
- selectarea in mod dinamic la runtime a unor algoritmi de sortare, compresie, criptare etc.

Structură:

- trebuie să definiți o **interfață comună** pentru strategiile pe care le implementați (fie ca o «interface» sau ca o clasa abstractă)
- implementați strategiile respectând interfața comună
- clasa care are nevoie să folosească strategiile **va ști doar despre interfața lor**, nu va fi legată de implementările concrete

Denumirile uzuale în exemplele acestui pattern sunt: *Strategy* (pt interfață sau clasa abstractă), *ConcreteStrategy* pentru implementare, *Context*, clasa care folosește/execută strategiile.

Recomandare: Urmăriți link-ul de la referințe către postul de pe Stack Overflow care descrie necesitatea pattern-ului Strategy. Pe lângă motivul evident de încapsulare a prelucrărilor/algoritmilor (care reprezintă strategiile efective), se preferă o anumită abordare: la runtime se verifică mai multe condiții și se decide asupra strategiei. Concret, folosind mecanismul de polimorfism dinamic, se folosește o anumită instanță a tipului de strategie (ex. Strategy str = new CustomStrategy), care se pasează în toate locurile unde este nevoie de Strategy. Practic, în acest fel, utilizatorii unei anumite strategii vor deveni agnostici în raport cu strategia utilizată, ea fiind instanțiată într-un loc anterior și putând fi gata utilizată. Gândiți-vă la browserele care trebuie să detecteze dacă device-ul este PC, smartphone, tabletă sau altceva și în funcție de acest lucru să randeze în mod diferit. Fiecare randare poate fi implementată ca o strategie, iar instanțierea strategiei se va face într-un punct, fiind mai apoi pasată în toate locurile unde ar trebui să se țină cont de această strategie.

Command

Design pattern-ul *Command* incapsulează un apel cu tot cu parametri într-o clasă cu interfată generică. Acesta este *Behavioral* pentru că modifică interacțiunea dintre componente, mai exact felul în care se efectuează apelurile.

Acest pattern este recomandat în următoarele cazuri:

- pentru a ușura crearea de structuri de delegare, de callback, de apelare intarziată
- pentru a reține lista de comenzi efectuate asupra obiectelor
- accounting
- liste de Undo, Rollback pentru tranzacții-suport pentru operații reversibile (*undoable operations*)

Exemple de utilizare:

- sisteme de logging, accounting pentru tranzacții
- sisteme de undo (ex. editare imagini)
- mecanism ordonat pentru delegare, apel întârziat, callback

Funcționare și necesitate

În esentă, Command pattern (așa cum v-ați obișnuit și lucrând cu celelate Pattern-uri pe larg cunoscute) presupune încapsularea unei informații referitoare la acțiuni/comenzi folosind un wrapper pentru a “ține minte această informație” și pentru a o folosi ulterior. Astfel, un astfel de wrapper va deține informații referitoare la tipul acțiunii respective (în general un asemenea wrapper va expunde o metodă execute(), care va descrie comportamentul pentru acțiunea respectivă).

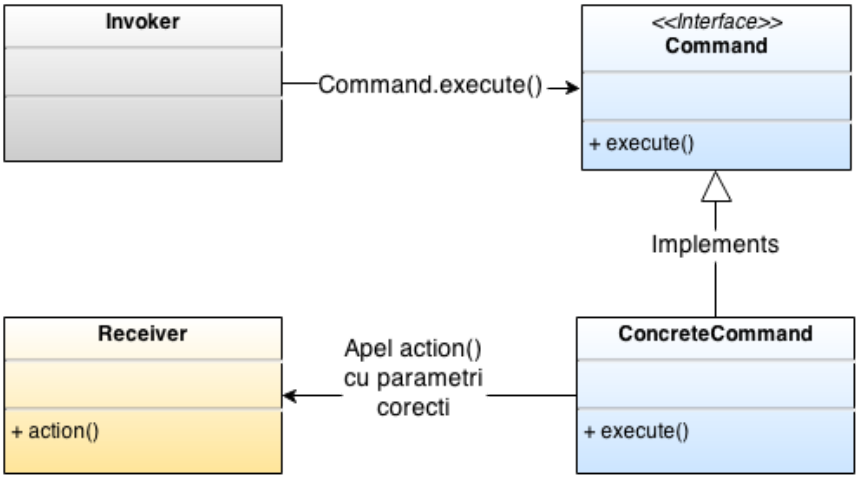
Mai mult încă, când vorbim de Command Pattern, în terminologia OOP o să întâlniți deseori și noțiunea de *Invoker*. Invoker-ul este un middleware ca funcționalitate care realizează managementul comenzilor. Practic, un *Client*, care vrea să facă anumite acțiune, va instanția clase care implementează o interfață *Command*. Ar fi incomod ca, în cazul în care aceste instanțieri de comenzi provin din mai multe locuri, acest management de comenzi să se facă local, în fiecare parte (din rațiuni de economie, nu vrem să duplicăm cod). Invoker-ul apare ca o necesitate de a centraliza acest proces și de a realiza intern management-ul comenzilor (le ține într-o listă, ține cont de eventuale dependențe între ele, totul în funcție de context).

Un client (generic spus, un loc de unde se lansează comenzi) instanțiază comenzile și le pasează Invoker-ului. Din acest motiv Invoker-ul este un middleware între client și receiver, fiindcă acesta va apela execute pe fiecare Command, în funcție de logica să internă.

Recomandare: La Referinte aveti un link catre un post pe StackOverflow, pentru a intelege mai bine de ce aveti nevoie de Pattern-ul Command si de ce nu lansati comenzi pur si simplu.

Structura

Ideea principală este de a crea un obiect de tip *Command* care va reține parametrii pentru comandă. Comandantul reține o referință la comandă și nu la componenta comandată. Comanda propriu-zisă este anunțată obiectului *Command* (de către comandant) prin execuția unei metode specificate asupra lui. Obiectul *Command* este apoi responsabil de trimiterea (*dispatch*) comenzii către obiectele care o îndeplinesc (*comandați*).



Tipuri de componente (**roluri**):

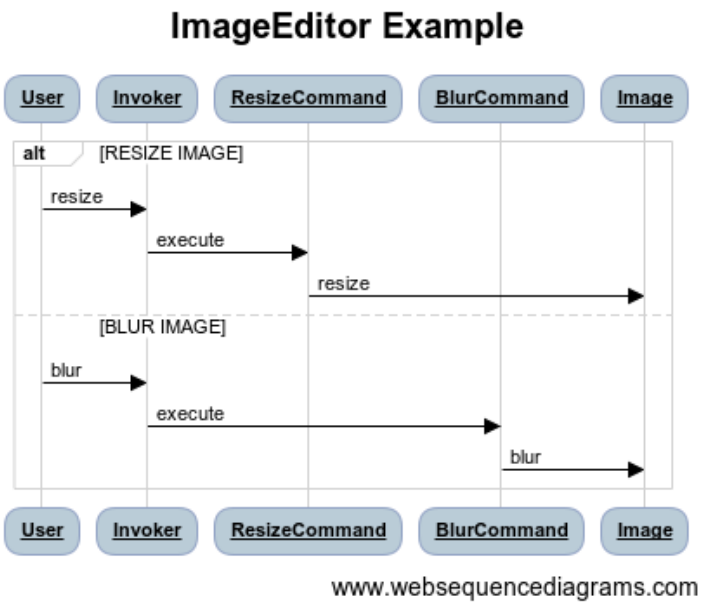
- **Invoker** - comandantul
  - apelează acțiuni pe comenzi (invocă metode oferite de obiectele de tip *Command*)
  - poate menține, dacă e cazul, o *listă a tuturor comenzilor aplicate* pe obiectul (obiectele) comandate. Este necesară reținerea acestei liste de comenzi atunci când implementăm un comportament de undo/redo al comenzilor.
  - primește clase *Command* pe care să le invoce
- **Receiver** - comandatul
  - este clasa asupra căreia se face apelul
  - conține implementarea efectivă a ceea ce se dorește executat
- **Command** - obiectele pentru reprezentarea comenzilor implementează această interfață/o extind dacă este clasă abstractă
  - *concrete command* - ne referim la implementări/subclasele acesteia
  - de obicei conțin metode cu nume sugestiv pentru executarea acțiunii comenzii (e.g. `execute()`). Implementările acestora conțin apelul către clasa *Receiver*.
  - în cazul implementării unor acțiuni *undoable* adăugăm metode pentru `undo` și/sau `redo`.
  - țin referințe către comandați (receivers) pentru a aplica/invoca acțiunea ce reprezintă acea comandă

În Java, se pot folosi atât interfețe cât și clase abstracte, pentru Command, depinzând de situație (e.g. clasă abstractă dacă știm sigur ca obiectele de tip Command nu mai au nevoie să extindă și alte clase).

În prima diagramă de mai jos, comandantul este clasa *Invoker* care conține o referință la o instanță (command) a clasei (Command). *Invoker* va apela metoda abstractă `execute()` pentru a cere îndeplinirea comenzii. *ConcreteCommand* reprezintă o implementare a interfeței *Command*, iar în metoda `execute()` va apela metoda din *Receiver* corespunzătoare acelei acțiuni/comenzi.

Exemplu

Prima diagramă de secvență prezintă apelurile în cadrul unei aplicație de editare a imaginilor, ce este structurată folosind pattern-ul Command. În cadrul acesteia, Receiver-ul este *Image*, iar comenzile *BlurCommand* și *ResizeCommand* modifică starea acesteia. Structurând aplicația în felul acesta, este foarte ușor de implementat un mecanism de undo/redo, fiind suficient să menținem în Invoker o listă cu obiectele de tip *Command* aplicate imaginii.



Pornind de la această diagramă, putem realiza o implementare a pattern-ului Command. Vom construi clasa *Image*, care va juca rolul Receiver-ului. Acesteia îi vom asocia un câmp *blurStrength*, care ne va oferi informații despre intensitatea filtrului de blur, și încă două câmpuri *length* și *width* care ne vor spune ce dimensiune are imaginea. Valorile acestor câmpuri vor fi alterate în urma aplicării comezilor de *blur* și *resize*.

```
public class Image {
   private int blurStrength;
   private int length;
   private int width;

   public Image(int length, int width) {
       this.length = length;
       this.width = width;
   }

   public int getBlurStrength() {
       return blurStrength;
   }

   public void setBlurStrength(int blurStrength) {
       this.blurStrength = blurStrength;
   }

   public int getLength() {
       return length;
   }

   public void setLength(int length) {
       this.length = length;
   }

   public int getWidth() {
       return width;
   }

   public void setWidth(int width) {
       this.width = width;
   }
}
```

Command va fi o interfață, căreia pe langă metoda de *execute()* îi vom asocia și o metodă de *undo()*.

```
interface Command {
   void execute();

   void undo();
}
```

BlurCommand și *ResizeCommand* vor implementa interfața *Command*. La apelul *execute()*, *BlurCommand* va modifica câmpul *blurStrength* din clasa *Image*, iar *ResizeCommand* va modifica dimensiunea, lungimea și înălțimea imaginii. Întrucât ne dorim să implementăm un mecanism de undo, este nevoie să reținem valoare anterioară.

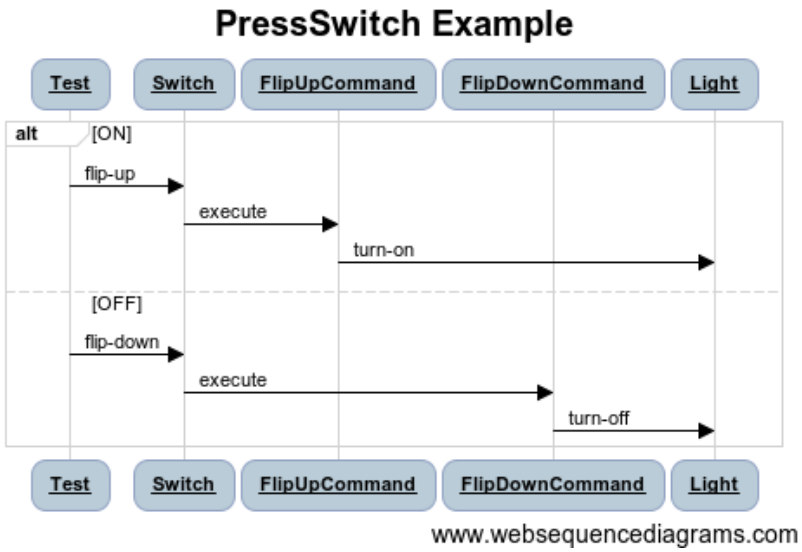
```
// Concrete command

public class BlurCommand implements Command {
   private final Image image;
   private int previousBlurStrength;
   private int nextBlurStrength;

   public BlurCommand(Image image, int blurStrength) {
       this.image = image;
       this.nextBlurStrength = blurStrength;
   }

   @Override
   public void execute() {
       previousBlurStrength = image.getBlurStrength();
       image.setBlurStrength(nextBlurStrength);
   }

   @Override
   public void undo() {
       nextBlurStrength = previousBlurStrength;
       previousBlurStrength = image.getBlurStrength();
       image.setBlurStrength(nextBlurStrength);
   }
}

public class ResizeCommand implements Command {
   private final Image image;
   private int previousWidth;
   private int previousLength;
   private int nextWidth;
   private int nextLength;

   public ResizeCommand(Image image, int width, int length) {
       this.image = image;
       nextWidth = width;
       nextLength = length;
```

```
    }

    @Override
    public void execute() {
        previousWidth = image.getWidth();
        image.setWidth(nextWidth);

        previousLength = image.getLength();
        image.setLength(nextLength);
    }

    @Override
    public void undo() {
        nextWidth = previousWidth;
        previousWidth = image.getWidth();
        image.setWidth(nextWidth);

        nextLength = previousLength;
        previousLength = image.getLength();
        image.setLength(nextLength);
    }
}
```

Invoker-ul este clasa *Editor*. Aceasta va avea două metode, edit si undo, care vor fi apelate de către user. În plus, vom păstra în această clasă și o listă a comenzilor aplicate pe imagine. Cu acestă listă vom putea implementăm un comportament de undo.
Metoda edit va primi ca parametru o referiță la o instanță command, apoi va fi inițiată o cerere de către *Editor* prin apelarea metodei *execute()*, cerând astfel execuția comenzii.

```
// Invoker

public class Editor {
    // LinkedList este folosit ca stivă în Java
    private LinkedList<Command> history = new LinkedList<>(); // păstrează comenzile aplicate pe imagine

    public void edit(Command command) {
        history.push(command);
        command.execute();
    }

    public void undo() {
        if (history.isEmpty()) return;

        Command command = history.pop();
        if (command != null) {
            command.undo();
        }
    }
}
```

Pe Wikipedia [http://en.wikipedia.org/wiki/Command_pattern] puteți analiza exemplul PressSwitch. Flow-ul pentru acesta este ilustrat în diagrama de mai jos



Builder

Design pattern-ul Builder este un design pattern creațional, cu alte cuvinte, este utilizat pentru a crea și configura obiecte. Un builder este utilizat în mod obișnuit pentru eliminarea supraincarcarilor de constructori multipli și oferă o soluție mai flexibilă la crearea obiectelor complexe.

Problema

În Programarea Orientată pe Obiecte, cel mai adesea avem clase care dețin unele date pe care le setăm și le accesăm ulterior. Crearea instanțelor unor astfel de clase ar putea fi uneori cam laborioasă. Să luăm în considerare următoarea clasă de Pizza

Pizza.java

```
public class Pizza {
    private String pizzaSize;
    private int cheeseCount;
    private int pepperoniCount;
    private int hamCount;

    // constructor, getters, setters
}
```

O clasă foarte simplă la prima vedere însă lucrurile se complică pe masură ce vom crea acest obiect. Oricare pizza va avea o dimensiune, cu toate acestea, atunci când vine vorba de topping-uri, unele sau toate pot fi prezente sau deloc, prin urmare, unele dintre proprietățile clasei noastre sunt opționale, iar altele sunt obligatorii.

Supraîncărcarea constructorilor

Crearea unei instanțe noi *new Pizza("small", 1, 0, 0)* de fiecare dată când vreau să obțin pur și simplu un obiect pizza cu brânză și nimic altceva nu mi se pare o idee bună. Și aici vine prima soluție comună - supraîncărcarea constructorului.

Pizza.java

```
public class Pizza {
    private String pizzaSize; // mandatory
    private int cheeseCount; // optional
    private int pepperoniCount; // optional
    private int hamCount; // optional

    public Pizza(String pizzaSize) {
        this(pizzaSize, 0, 0, 0);
    }

    public Pizza(String pizzaSize, int cheeseCount) {
        this(pizzaSize, cheeseCount, 0, 0);
    }

    public Pizza(String pizzaSize, int cheeseCount, int pepperoniCount) {
        this(pizzaSize, cheeseCount, pepperoniCount, 0);
    }

    public Pizza(String pizzaSize, int cheeseCount, int pepperoniCount, int hamCount) {
        this.pizzaSize = pizzaSize;
        this.cheeseCount = cheeseCount;
        this.pepperoniCount = pepperoniCount;
        this.hamCount = hamCount;
    }

    // getters
}
```

Cu toate acestea, am rezolvat problema doar parțial. Nu putem, de exemplu, să creăm o pizza cu brânză și șuncă, dar fără pepperoni ca aceasta *new Pizza("small", 1, 1)*, deoarece al treilea argument al constructorului este pepperoni. Și aici vine a doua soluție comună - și mai multă supraîncărcare de constructori.

Pizza.java

```
public class Pizza {
    private String pizzaSize; // mandatory
    private String crust; // mandatory
    private int cheeseCount; // optional
    private int pepperoniCount; // optional
    private int hamCount; // optional
    private int mushroomsCount; // optional

    public Pizza(String pizzaSize, String crust) {
        this(pizzaSize, crust, 0, 0, 0, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount) {
        this(pizzaSize, crust, cheeseCount, 0, 0, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount, int pepperoniCount) {
        this(pizzaSize, crust, cheeseCount, pepperoniCount, 0, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount, int pepperoniCount, int hamCount) {
        this(pizzaSize, crust, cheeseCount, pepperoniCount, hamCount, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount, int pepperoniCount, int hamCount, int mushroomsCount) {
        this.pizzaSize = pizzaSize;
        this.crust = crust;
        this.cheeseCount = cheeseCount;
        this.pepperoniCount = pepperoniCount;
        this.hamCount = hamCount;
        this.mushroomsCount = mushroomsCount;
    }
```

```
    // getters
}
```

Gândiți-vă ce se va întampla dacă se schimbă ordinea parametrilor. Acest lucru minor va strica funcționalitatea completă a creative unei instanțe de Pizza.

În concluzie, modelul de constructori supraincarcati funcționează, dar este greu de menținut dacă se schimbă funcționalitatea și introducem noi parametri, numărul constructorilor va crește, de asemenea.

Folosirea de getters și setters

Pizza.java

```
Pizza pizza = new Pizza();

pizza.setPizzaSize("small");
pizza.setCrust("thin");
pizza.setMushroomsCount(1);
pizza.setCheeseCount(1);

// do something with pizza
```

Această soluție nu prezintă niciunul dintre dezavantajele modelului anterior. Este ușor să scalați clasa, mai ușor de instanțiat, mai ușor de citit și mai flexibil. Modelul are însă dezavantaje grave. Construcția clasei este împărțită în apeluri multiple, prin urmare instanța poate fi într-o stare parțial construită / invalidă.

Folosirea builder pattern

Pizza.java

```
public class Pizza {
    private String pizzaSize;
    private String crust;
    private int cheeseCount;
    private int pepperoniCount;
    private int hamCount;
    private int mushroomsCount;

    public static class Builder {
        private String pizzaSize; // mandatory
        private String crust; // mandatory
        private int cheeseCount = 0; // optional
        private int pepperoniCount = 0; // optional
        private int hamCount = 0; // optional
        private int mushroomsCount = 0; // optional

        public Builder(String pizzaSize, String crust) {
            this.pizzaSize = pizzaSize;
            this.crust = crust;
        }

        public Builder cheeseCount(int cheeseCount) {
            this.cheeseCount = cheeseCount;
            return this;
        }

        public Builder pepperoniCount(int pepperoniCount) {
            this.pepperoniCount = pepperoniCount;
            return this;
        }

        public Builder hamCount(int hamCount) {
            this.hamCount = hamCount;
            return this;
        }

        public Builder mushroomsCount(int mushroomsCount) {
            this.mushroomsCount = mushroomsCount;
            return this;
        }

        public Pizza build() {
            return new Pizza(this);
        }
    }

    private Pizza(Builder builder) {
        this.pizzaSize = builder.pizzaSize;
        this.crust = builder.crust;
        this.cheeseCount = builder.cheeseCount;
        this.pepperoniCount = builder.pepperoniCount;
        this.hamCount = builder.hamCount;
        this.mushroomsCount = builder.mushroomsCount;
    }

    // getters
}
```

Am făcut constructorul privat, astfel încât clasa noastră să nu poată fi instanțiată direct. În același timp am adăugat o clasă static Builder cu un constructor care are parametrii noștri obligatori pizzaSize și crust, metode de setare a parametrilor opționali și, în final, o metodă *build()* metoda care va returna o nouă instanță a clasei Pizza. Metodele setter returnează instanța de builder în sine, oferindu-ne astfel o interfață fluentă cu metoda de înlănțuire.

Pizza.java

```
Pizza pizza = new Pizza.Builder("large", "thin")
    .cheeseCount(1)
    .pepperoniCount(1)
    .build();
```

Este mult mai ușor să scrieți și, mai important, să citiți acest cod. La fel ca în cazul constructorului, putem verifica parametrii trecuți pentru orice încălcare, cel mai adesea în cadrul metodei *build()* sau a metodei setter, și putem arunca IllegalStateException dacă există încălcări înainte de a crea o instanță a clasei.

Modelul Builder are unele dezavantaje minore. În primul rând, trebuie să creați un obiect Builder înainte de a crea obiectul clasei în sine. Aceasta ar putea fi o problemă în unele cazuri critice de performanță iar clasa devine puțin mai mare când vine vorba de liniile de cod.

În ansamblu, modelul Builder este o tehnică foarte frecvent utilizată pentru crearea obiectelor și este o alegere bună de utilizat atunci când clasele au constructori cu parametri multipli (în special opționali) și este posibil să se schimbe în viitor. Codul devine mult mai ușor de scris și de citit în comparație cu constructorii supraincarcati, iar clasa este la fel de bună ca folosirea de getters și setters, dar este mult mai sigură.

Summary

Principii de design adresate de aceste patternuri:

- Dependency Injection Principle [https://stackoverflow.com/questions/62539/what-is-the-dependency-inversion-principle-and-why-is-it-important] - componentele trebuie să depindă de tipuri abstracte, nu de implementări
    - Factory respectă acest principiu, componentele depinzând de interfața pentru un tip, nu de un subtip anume
- Separarea codului care se schimbă de cel care rămâne la fel - în cazul Strategy folosim o interfață pentru strategii, depindem de aceea, si putem schimba implementările fără a modifica codul care le folosește (exemplu [https://www.freecodecamp.org/news/the-strategy-pattern-explained-using-java-bc30542204e0/])
- Loosly coupled design [http://thephantomprogrammer.blogspot.com/2015/07/strive-for-loosely-coupled-design.html] - în cazul Observer componentele sunt slab legate între ele

Exerciții

Laboratorul trebuie rezolvat pe platforma LambdaChecker, fiind găsit aici [https://beta.lambdachecker.io/contest/20].

În cadrul acestui laborator, exercițiile valorează în total 20p. Pentru primirea punctajului maxim pe acest laborator, trebuie să acumulați 10p din rezolvarea exercițiilor, orice depășește 10p fiind contorizat ca și bonus.

Task 1 [https://beta.lambdachecker.io/problem/43/20] - Observer, Strategy, Factory (10p)

Part 1. (6p) - Observer DataRepository este obiectul nostru observabil in scenariul de fata. O instanta de DataRepository poate primi date, prezentate sub forma unui SensorData pe care acesta le stocheaza intern.

Creati doua clase, ConsoleLogger si ServerCommunicationController, care vor fi Observeri pentru DataRepository. In momentul in care metoda “addData” dintr-o instanta de DataRepository se apeleaza, comportamentul celor doi observeri va fi urmatorul:

- ConsoleLogger va afisa urmatorul mesaj: “New server data: ” + new_data.toString()
- ServerCommunicationController va afisa urmatorul mesaj: “Generated server message: ” + new_server_message.toString()

unde new_data este instanta de SensorData nou introdusa, iar new_server_message este o instanta de ServerMessage construita pe baza instantei de SensorData nou adaugata.

Pentru Id-ul din ServerMessage, folositi Id-ul din Utils. Atasati cate o instanta din fiecare observator la instanta de DataRepository.

- Pentru a avea deja mecanismul de notificare vom folosi interfața Observer [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observer.html] și clasa Observable [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observable.html] din java.util. Dacă doriți și aveți timp puteți să vă implementați propriile interfețe Oberver-Observable.
- Observer-Observable din java.util sunt deprecated din java 9 pentru că sunt prea simple, însă asta le face potrivite pentru acest laborator. Într-o aplicație reală puteți folosi alte api-uri care sunt mult mai complexe și oferă foarte multe tipuri de obiecte și mecanisme (termenul folosit este *reactive programming*).
- Citiți în README [https://github.com/oop-pub/laboratoare/blob/master/design-patterns/skel/src/README] rolul fiecărui observator.

Vedeți metodele din Observable [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observable.html] pentru notificarea observatorilor, schimbarea stării obiectului observat și adăugarea de observatori.

Part 2. (4p) - Strategy, Factory Fiecare instanta de tip SensorData contine si numarul de pasi facuti la un anumit timestamp. Vrem sa cream un alt observator care va calcula numarul total de pasi executati de aplicatie, stocati in instante de SensorData in interiorul lui DataRepository, folosind doua strategii de calcul.

Extindeti din interfata StepCountStrategy doua clase, care vor servi drept strategiile noastre:

- BasicStepCountStrategy - calculeaza numarul total de pasi dintr-un DataRepository
- FilteredStepCountStrategy - calculeaza numarul total de pasi dintr-un DataRepository, adaugand la total doar pasii care indeplinesc urmatoarele conditii: sunt pozitivi si nu sunt mai mari cu MAX_STEP_COUNT (valoare statica in Utils, egala cu 1000) pasi decat numarul de pasi precedent.

In interiorul acestor clase de strategie se va stoca si instanta de DataRepository, pentru a calcula numarul total de pasi.

Creati un StepCountStrategyFactory care, in functie de un parametru de tip String, va intoarce fie un BasicStepCountStrategy, fie un FilteredStepCountStrategy.

Aceste strategii vor fi folosite intr-un alt observator, DataAggregator. Acesta primeste o strategie, iar la fiecare operatie de tip 'addData' va afisa urmatorul mesaj:

- “Basic strategy, total step count: ” + total_steps, daca strategia este Basic
- “Filtered strategy, total step count: ” + total_steps, daca strategia este Filtered

Creati doua strategii pe baza celor doua String-uri primite la intrare, pe care le veti folosi pentru a instantia doi observatori de tip DataAggregator. Atasati cei doi observatori la instanta de DataRepository.

Task 2 [https://beta.lambdachecker.io/problem/44/20] - Builder pattern (2p)

a) Scrieți câmpuri în skeletul clasei House pentru câteva facilități obligatorii în construcția unei case, spre exemplu locația construcției, numărul de etaje, încălzire, camere dar și unele opționale pe care le poate selecta sau nu clientul, cum ar fi electrocasnice, piscină, panouri solare, securitate etc.

Completați constructorul privat, metodele de get și metoda toString.

b) În clasa de builder, completați câmpurile, constructorul și metodele de adăugare a facilităților opționale.

c) Finalizați metoda build și testați funcționalitatea într-o clasă Main creată de voi, acoperind cazuri în care se construiește o casa doar cu facilitați obligatorii și altele adăugând și pe cele opționale.

Task 3 [https://beta.lambdachecker.io/problem/41/20] - Command pattern (8p)

Implementați folosind patternul Command un editor de diagrame foarte simplificat. Scheletul de cod [https://github.com/oop-pub/oop-labs/tree/master/src/lab10] conține o parte din clase și câteva teste.

Componentele principale ale programului:

- *DiagramCanvas* - reprezintă o diagramă care conține obiecte de tip DiagramComponent
- *DrawCommand* - interfață pentru comenzile făcute asupra diagramei sau a componentelor acesteia
- *Invoker* - primește comenzile și le execută
- *Client* - entry-point-ul în program

(4p) Implementați 5 tipuri de comenzi, pentru următoarele acțiuni:

- Draw rectangle - crează o DiagramComponent și o adaugă în DiagramCanvas
- Resize - modifică width și height al unei DiagramComponent pe baza unui procent dat
- Change color - modifică culoarea unei DiagramComponent
- Change text - modifică textul unei DiagramComponent
- Connect components - conectează o DiagramComponent la alta

Implementați pe Invoker metoda execute() care va executa comanda primită ca argument.

Comenzile primesc în constructor referința către DiagramCanvas și alte argumente necesare lor. De exemplu, comanda pentru schimbarea culorii trebuie sa primească și culoarea nouă și indexul componentei.

Pentru acest task nu este nevoie să implementați și metoda *undo()*, doar *execute()*.

Comenzile implementează în afară de metodele interfeței și metoda toString() [https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html#toString()] pentru a afișa comanda. Recomandăm folosirea IDE-ului pentru a o genera.

(6p) Implementați în comenzi și în Invoker mecanismul de undo/redo al comenzilor. Recomandăm în Invoker sa folosiți două structuri de date, una care să mențină comenzile efectuate, iar una pentru comenzile făcute undo. Metoda reset() de pe Invoker va avea ca scop resetarea tuturor membrilor acestuia.

Resurse

- Exerciții din alți ani

Referințe

- Dynamic-binding vs static binding [http://geekexplains.blogspot.ro/2008/06/dynamic-binding-vs-static-binding-in.html]
- Lazy Instantiation [http://www.javaworld.com/article/2077568/learn-java/java-tip-67--lazy-instantiation.html]
- Exemple simple pattern Observer [http://sourcemaking.com/design_patterns/observer/java/1]
- Explicații pattern Observer [http://sourcemaking.com/design_patterns/observer].
- De ce avem nevoie de Strategy Pattern? [https://stackoverflow.com/questions/1710809/when-and-why-should-the-strategy-pattern-be-used]
- Command design pattern [https://sourcemaking.com/design_patterns/command]
- De ce avem nevoie de Command Pattern? [https://stackoverflow.com/questions/32597736/why-should-i-use-the-command-design-pattern-while-i-can-easily-call-required-met]

poo-ca-cd/laboratoare/design-patterns.txt · Last modified: 2022/12/12 12:19 by radu_bogdan.pavel