Programare Orientată pe Obiecte

Design patterns - Factory, Strategy, Observer

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* și *Observer*.

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.

Abstract Factory Pattern

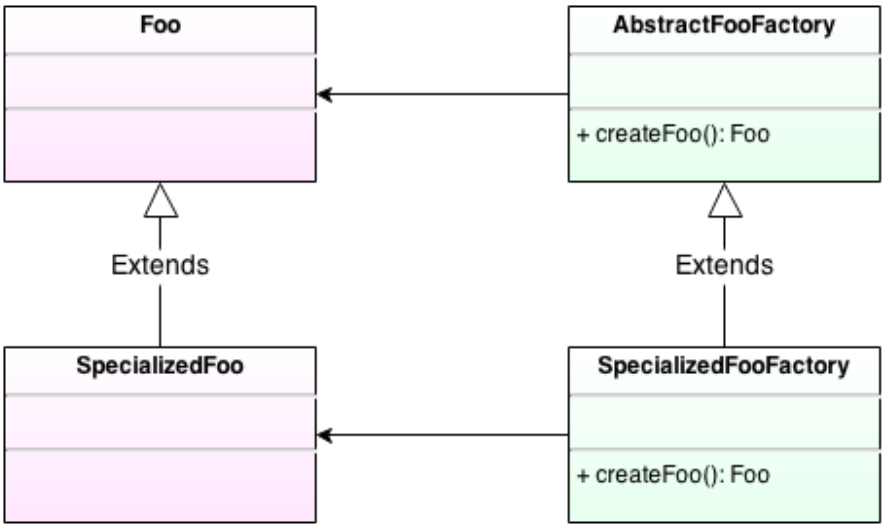

Fig. 1: Diagrama de clase pentru Abstract Factory

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.

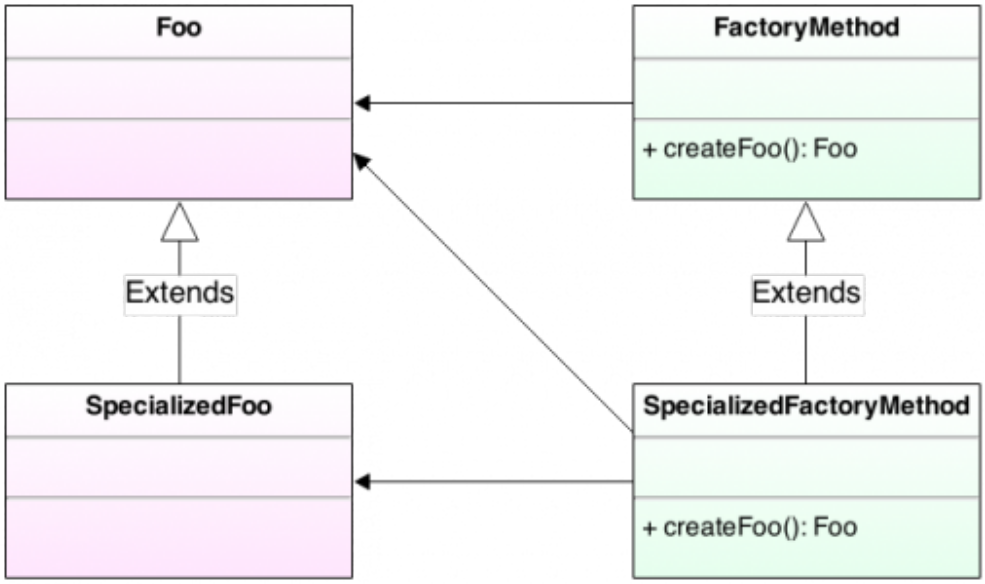

Fig. 2: Diagrama de clase pentru Factory Method

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.

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

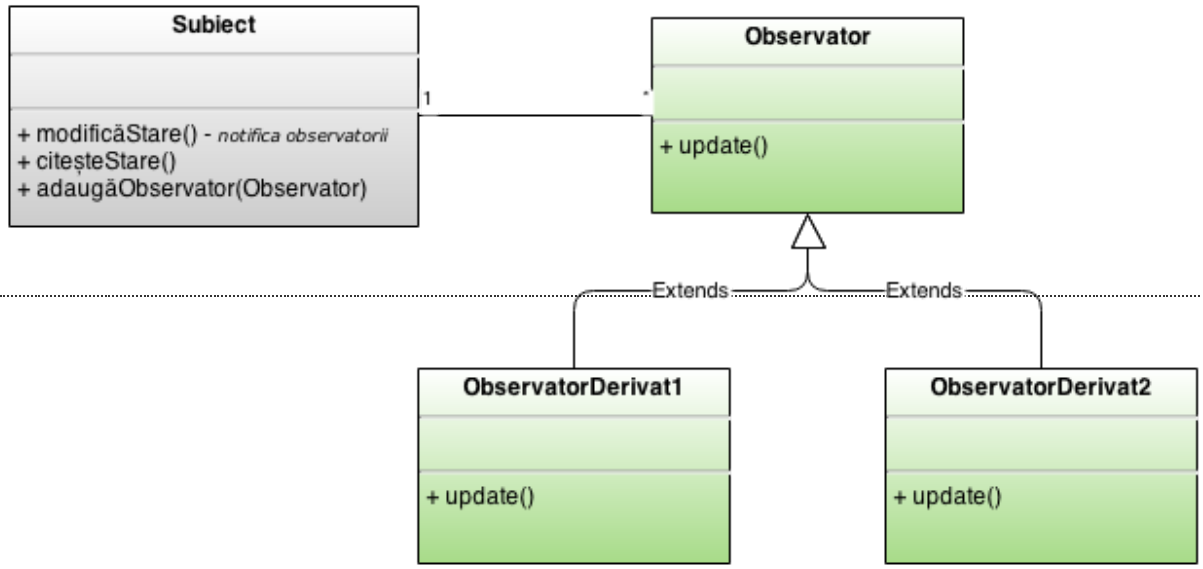

Fig. 3: Diagrama de clase pentru Observer Pattern

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: Urmariti link-ul de la referinte catre postul de pe Stack Overflow care descrie necesitatea pattern-ului Strategy. Pe langa motivul evident de incapsulare a prelucrarilor/algoritmilor (care reprezinta strategiile efective), se prefera o anumita abordare: la runtime se verifica mai multe conditii si se decide asupra strategiei. Concret, folosind mecanismul de polimorfism dinamic, se foloseste o anumita instanta a tipului de strategie (*ex. Strategy str = new CustomStrategy*), care se paseaza in toate locurile unde este nevoie de Strategy. Practic, in acest fel, utilizatorii unei anumite strategii vor deveni agnostici in raport cu strategia utilizata, ea fiind instantiata intr-un loc anterior si putand fi gata utilizata. Ganditi-va la browserele care trebuie sa detecteze daca device-ul este PC, smartphone, tableta sau altceva si in functie de acest lucru sa randeze in mod diferit. Fiecare randare poate fi implementata ca o strategie, iar instantierea strategiei se va face intr-un punct, fiind mai apoi pasata in toate locurile unde ar trebui sa se tina cont de aceasta strategie.

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

În cadrul acestui laborator veți implementa o aplicație *mock* care primește date și le procesează.

Arhitectura ei va include patternurile Observer, Strategy și Factory și este descrisă în README-ul din arhiva dată cu scheletul de cod.

Task 1 - Observer pattern (3p)

Scheletul de cod vă oferă clasele *MainApp* (entry-point pentru testare), *Utils* și *DataRepository*.

`DataRepository` este obiectul observabil, care va primi date noi de la MainApp. Când acesta primește date noi va notifica observatorii săi: `ConsoleLogger`, `ServerCommunicationController` și `DataAggregator`.

- 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.
- *Hint:* 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.

Task 2 - Strategy pattern (3p)

Scheletul vă oferă interfața `StepCountStrategy` ce va fi implementată de către "algoritmii" de prelucrare a datelor: `BasicStepCountStrategy` și `FilteredStepCountStrategy`. Prima adună toate valorile primite, iar a doua le adună doar pe cele ce îndeplinesc niște condiții (să fie număr pozitiv și să nu fie o valoare prea mare (mai mult de 1000 de pași) venită prea curând de la ultimul update primit (în mai puțin de 1 minut).

- strategiile vor folosi datele stocate în DataRepository
- pentru strategia Filtered puteți folosi următoarele constante: `private static final int MAX_DIFF_STEPS_CONSECUTIVE_RECORDS = 1000;` și `private static final long TIME_FOR_MAX_DIFF = TimeUnit.SECONDS.toMillis(1);`

Task 3 - Factory pattern (2p)

Creați clasa `StepCountStrategyFactory` care construiește instanțe de subclase ale StepCountStrategy.

Clasa factory va conține o metodă care să întoarcă o strategie. De exemplu: `public StepCountStrategy createStrategy(String strategyType, DataRepository dataRepository)`. StrategyType poate fi un string care să indice tipul strategiei, vedeți în `Utils` stringurile definite deja.

Task 4 - Putting all together (2p)

Realizați TODO-urile din codul de test din MainApp și rulați. Puteți să vă adăugați propriile exemple de date pentru a verfica corectitudinea programului.

Resurse

- Schelet
- Soluție
- 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]
