Design patterns - Singleton, Factory, 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 - *Singleton*, *Factory* 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], 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 laboratorul Visitor Pattern au fost introduse design pattern-urile și aplicabilitatea Visitor-ului. Acesta 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.

Singleton Pattern

Pattern-ul Singleton este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect, deci reprezintă o metodă de a folosi o singură instanță a unui obiect în aplicație.

Utilizari

Pattern-ul Singleton este util în următoarele cazuri:

- ca un subansamblu al altor pattern-uri:
  - impreună cu pattern-urile Abstract Factory, Builder, Prototype etc. De exemplu, în aplicație dorim un singur obiect factory pentru a crea obiecte de un anumit tip.
- în locul variabilelor globale. Singleton este preferat variabilelor globale deoarece, printre altele, nu poluează namespace-ul global cu variabile care nu sunt necesare.

Singleton este utilizat des în situații în care avem obiecte care trebuie accesate din mai multe locuri ale aplicației:

- obiecte de tip logger
- obiecte care reprezintă resurse partajate (conexiuni, sockets etc.)
- obiecte ce conțin configurații pentru aplicație
- pentru obiecte de tip *Factory*.

Exemple din API-ul Java:

- java.lang.Runtime
- java.awt.Toolkit

Din punct de vedere al design-ului și testarii unei aplicații de multe ori se evită folosirea acestui pattern, în test-driven development fiind considerat un ***anti-pattern***. A avea un obiect Singleton a carei referință o folosim peste tot prin aplicație introduce multe dependențe între clase și îngreunează testarea individuală a acestora.

In general, codul care folosește stări globale este mai dificil de testat pentru că implică o cuplare mai strânsă a claselor, și împiedică izolarea unei componente și testarea ei individuală. Dacă o clasă testată folosește un obiect singleton, atunci trebuie testat și singleton-ul. Soluția este simularea *mock-up* a singleton-ului în teste. Încă o problemă a acestei cuplări mai strânse apare atunci când

două teste depind unul de celălalt prin modificarea singleton-ului, deci trebuie impusă o anumită ordine a rulării testelor.

Încercați să nu folosiți în exces metode statice (să le utilizați mai mult pt funcții “utility”) și componente Singleton.

Implementare

Aplicarea pattern-ului Singleton constă în implementarea unei metode ce permite crearea unei noi instanțe a clasei dacă aceasta nu există, și întoarcerea unei referințe către aceasta dacă există deja. În Java, pentru a asigura o singură instanțiere a clasei, constructorul trebuie să fie *private*, iar instanța să fie oferită printr-o metodă statică, publică.

În cazul unei implementări Singleton, clasa respectivă va fi instanțiată **lazy** (*lazy instantiation*), utilizând memoria doar în momentul în care acest lucru este necesar deoarece instanța se creează atunci când se apelează `getInstance()`, acest lucru putând fi un avantaj în unele cazuri, față de clasele non-singleton, pentru care se face *eager instantiation*, deci se alocă memorie încă de la început, chiar dacă instanța nu va fi folosită (mai multe detalii și exemplu în acest articol)

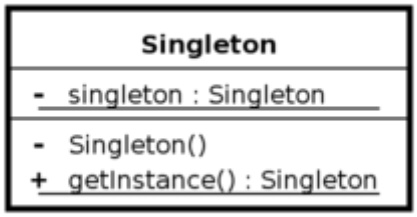

Fig. 1: Diagrama de clase pentru Singleton

Respectând cerințele pentru un singleton enunțate mai sus, în Java, putem implementa o componentă de acest tip în mai multe feluri, inclusiv folosind `enum`-uri în loc de clase. Atunci când îl implementâm trebuie avut în vedere contextul în care îl folosim, astfel încât să alegem o soluție care să funcționeze corect în toate situațiile ce pot apărea în aplicație (unele implementări au probleme atunci când sunt accesate din mai multe thread-uri sau când trebuie serializate).

```
public class Singleton {

  private static Singleton instance = null;

  private Singleton() {}

  public static Singleton getInstance() {
     if(instance == null) {
         instance = new Singleton();
     }
     return instance;
  }
  ...
```

```
}
```

- Instanța `instance` este *private*
- Constructorul este privat ca sa nu poata fi apelat decat din clasa respectivă
- Instanța este inițial nulă
- Instanța este creată la prima rulare a *getInstance()*

De ce Singleton și nu clase cu membri statici?

O clasă de tip Singleton poate fi extinsă, iar metodele ei suprascrise, însă într-o clasă cu metode statice acestea nu pot fi suprascrise (*overriden*) (o discuție pe aceasta temă puteți gasi aici, și o comparatie între static și dynamic binding aici).

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), în Android SDK (e.g. clasa SocketFactory) 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

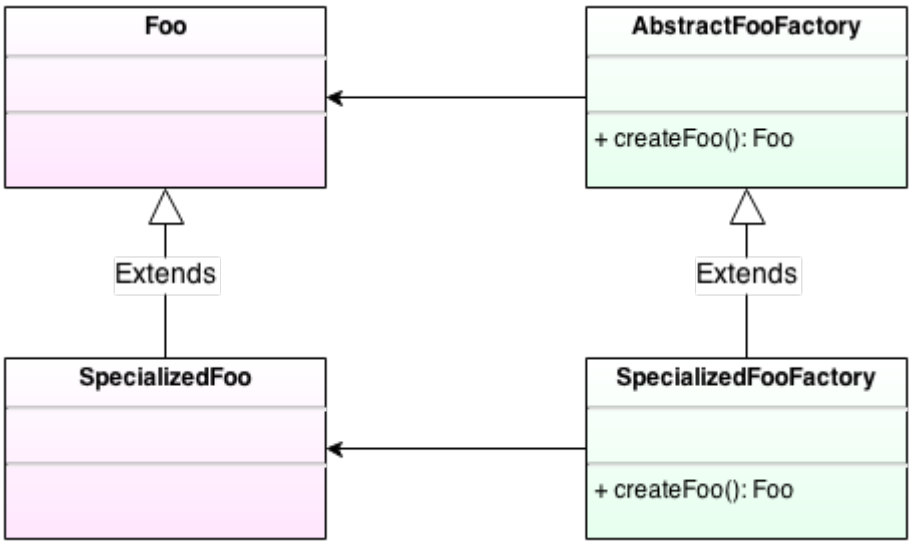

Fig. 2: Diagrama de clase pentru Abstract Factory

Codul următor corespunde diagramei din figure 2. 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 “înrudite”, care extind un anumit tip. Clasele care implementează Abstract Factory conțin de obicei mai multe metode factory.

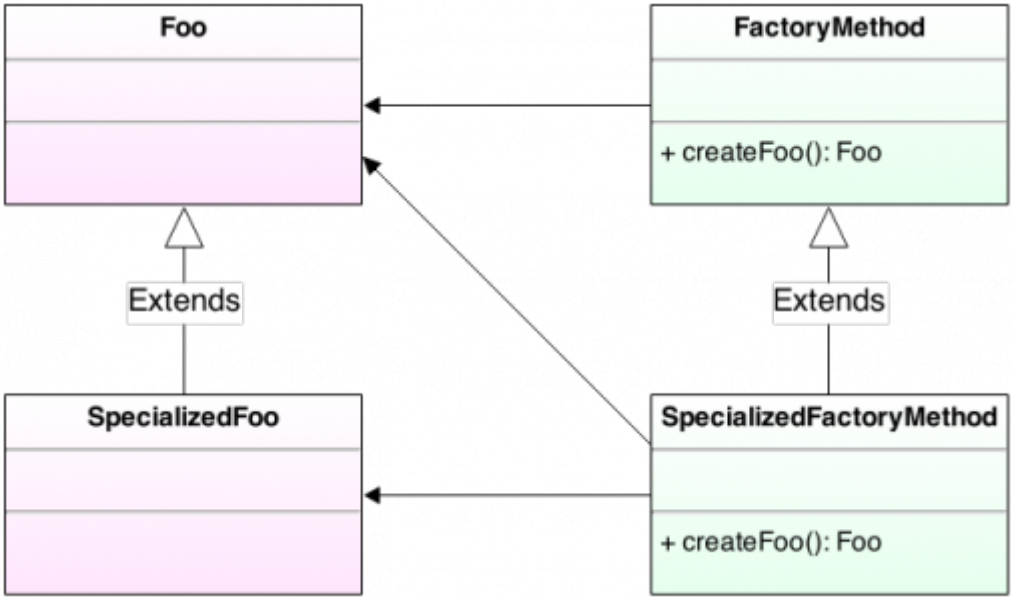

Fig. 3: Diagrama de clase pentru Factory Method

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) ce oferă clasa abstractă java.awt.Toolkit 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 4, 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

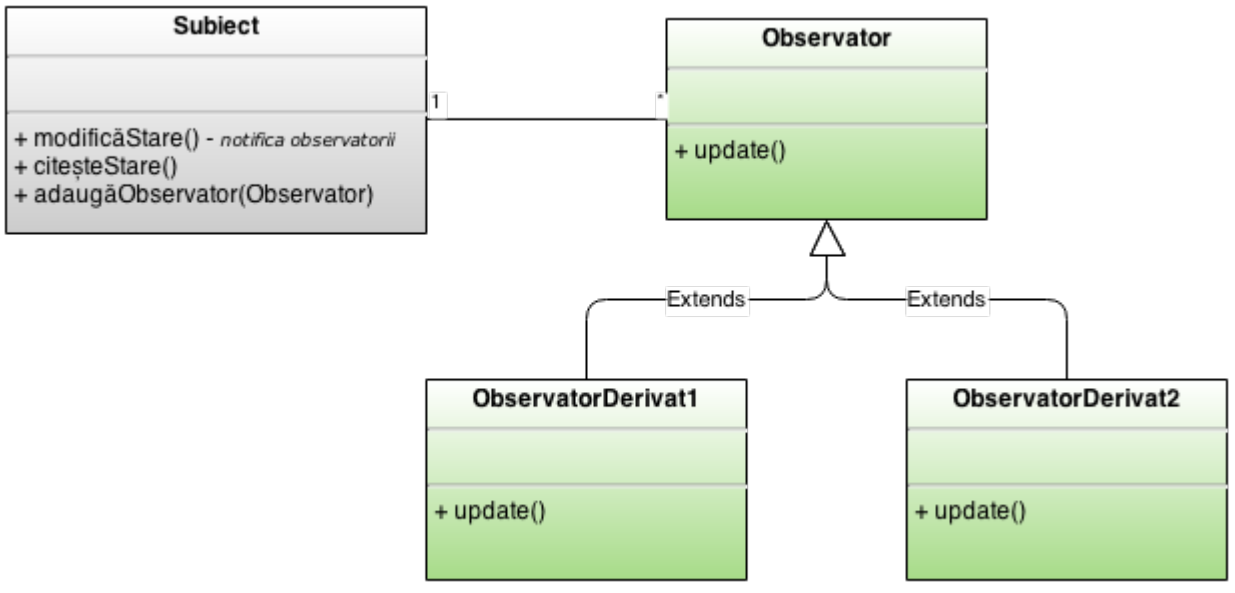

Fig. 4: Diagrama de clase pentru Observer Pattern

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.

Implementare

Un exemplu de implementare este exercițiul 2 de la laboratorul 5 (Clase interne). Observați diagrama de clase asociată acestuia.

Tookit-urile GUI, cum este și Swing folosesc acest design pattern, de exemplu apăsarea unui buton generează un eveniment ce poate fi transmis mai multor *listeners* înregistrați acestuia (exemplu).

API-ul Java oferă clasele Observer și Observable 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, exemplu.

Exerciții

Acest laborator și următorul au ca temă comună a exercițiilor realizarea unui joc controlat din consolă. Jocul constă dintr-o lume (aka hartă) în care se plimbă eroi de trei tipuri, colectează comori și se bat cu monștri. În acestă săptămână trebuie să implementați o parte din funcționalitățile jocului folosind patternurile *Singleton*, *Factory* și *Observer*, urmând ca la laboratorul următor să terminați implementarea folosind pattern-urile studiate atunci.

Detalii joc:

- *Harta*
  - reprezentată printr-o matrice. Fiecare element din matrice reprezintă o zonă care poate fi

liberă, poate conține obstacole sau poate conține o comoară (în laboratorul următor poate conține și monștrii).
    - este menținută în clasa `World`.
- *Eroii*
    - sunt reprezentați prin clase de tip `Hero` și sunt de trei tipuri: *Mage*, *Warrior*, *Priest*.
    - puteți adăuga oricâți eroi doriți pe hartă (cât vă permite memoria :))
    - într-o zonă pot fi mai mulți eroi
    - acțiunile pe care le pot face:
        - `move` - se mută într-o zonă învecinată
        - `attack` (de implementat în laboratorul următor)
        - `collect` - eroul ia comoara găsită în zona în care se află
- Entry-point-ul în joc îl consitituie clasa `Main`.
- **(4p)** Folosiți design pattern-ul *Factory* pentru crearea obiectelor.
    1. Creati clase care mostenesc `Hero` pentru fiecare tip de erou.
        - suprascrieti metoda `toString` din `Object` pentru fiecare erou
        - metoda `attack` - deocamdată nu va omorî pe nimeni - puteți afișa ceva la consolă
    2. Uitați-vă la clasele `TreasureFactory` și `HeroFactory`. Trebuie să implementăm două metode: `createTreasure` în `TreasureFactory` și o metodă de creare de eroi în `HeroFactory`, fie ea `createHero`.
        1. Puteți pune orice date doriți în comori, respectiv eroi.
        2. La `HeroFactory.createHero`, pasați ca parametru un `Hero.Type` și un `String` cu numele eroului și întoarceți un subtip de `Hero` potrivit pentru tipul de erou.
    3. După ce ați creat factory-urile, folosiți-le:
        1. Completați metoda `populateTreasures` din `World`. Folosiți-vă de membrii `map` și `treasures` din `World`. Trebuie să marcați pe hartă că aveți o comoară și să adăugați obiectul-comoară în lista de comori.
        2. Uitați-vă apoi la cazul `add` din metoda `main`. Trebuie să adăugați eroi acolo. Folosiți `HeroFactory.createHero`.
- **(2p)** Folosiți design pattern-ul *Singleton* pentru elementele din joc care trebuie să aibă doar o instanță.
    - Ce clase vor avea doar o singură instanță?
- **(4p)** Folosiți design pattern-ul *Observer* pentru a monitoriza ceea ce se întâmplă în joc. Scheletul de cod vă sugerează două tipuri de observatori, pentru bonus puteți adăuga și alții.
    1. veți folosi interfața Observer și clasa Observable din java.util.
        1. Înainte să vă apucați să scrieți, citiți comentariile din cod (e.g. TreasureDiscoverer) să vă faceți o idee despre ce vrem să facem în clasele observatoare.
        2. Asigurati-va ca implementati corect functionalitatea din **Observable**. Mare atentie la faptul ca **metoda notifyObservers nu va face nimic daca nu este apelata mai intai metoda setChanged**.
    2. Care sunt elementele observatoare și care sunt observabile? Uitați-vă și în comentariile din cod.
    3. Implementați interfețele `Observer` și `Observable` în clasele potrivite.
    4. Înregistrați observatori la `World`. Cazul `start` din metoda `main`.
    5. Notificați observatorii lui `World` când eroii execută o acțiune. Aveți două `TODO`-uri în clasa `Hero`.
- Începeți rezolvarea prin implementarea claselor pentru eroi și implementarea design pattern-ului factory pentru crearea lor. Pentru a putea vizualiza harta trebuie să implementați partea de observare a stării jocului. `World` trebuie să fie observabilă și să notifice pe observatorii săi atunci când a început jocul și când se schimbă ceva (e.g. s-a mutat un erou).
    - Urmăriți todo-urile din scheletul de cod (pentru a le vizualiza mai ușor pe toate puteți să folosiți view-ul pt ele din IDE, de exemplu în eclipse aveți *Window* → *Show View* → *Tasks*)

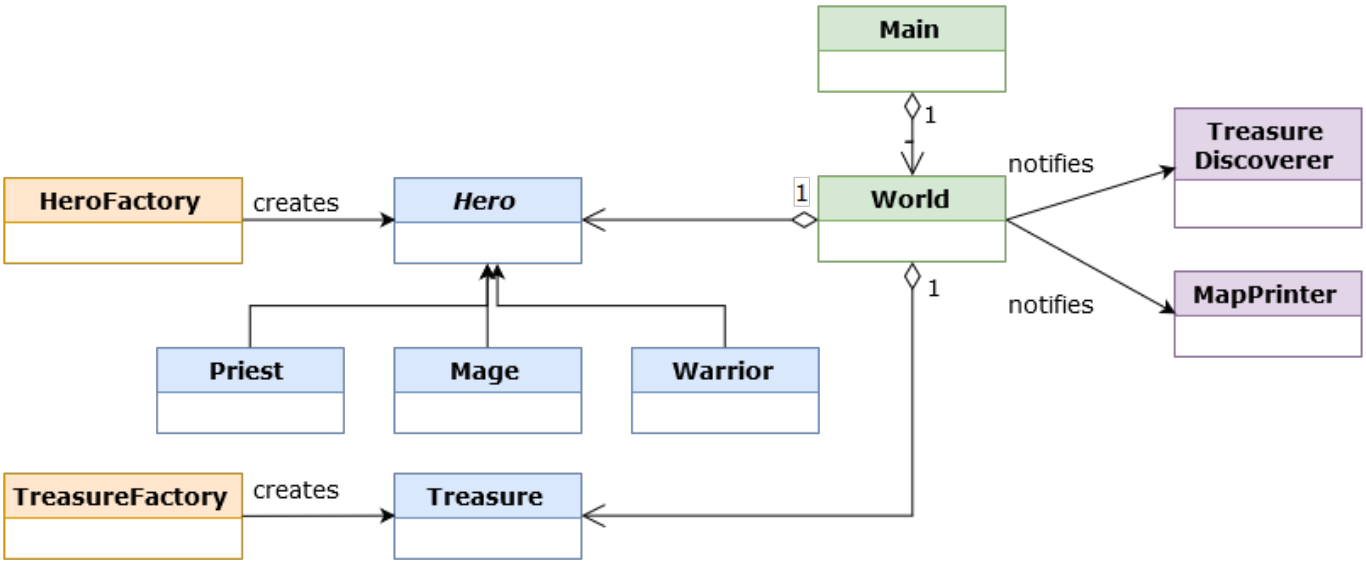

Fig. 5

Resurse

- Schelet
- Soluție
- PDF laborator

Referințe

- Dynamic-binding vs static binding
- Lazy Instantiation
- Exemple simple pattern Observer
- Explicații pattern Observer.