Programare orientată pe obiecte – Seria CC
Design patterns – Laborator 12

Mihai Nan – mihai.nan@upb.ro

Anul universitar 2020 – 2021

1 Design patterns

1.1 Prezentare generală

Un șablon de proiectare descrie o problemă care se întâlnește în mod repetat în proiectarea programelor și soluția generală pentru problema respectivă, astfel încât să poată fi utilizată oricând, dar nu în același mod de fiecare dată. Soluția este exprimată folosind clase și obiecte. Atât descrierea problemei cât și a soluției sunt abstracte astfel încât să poată fi folosite în multe situații diferite.

Scopul șabloanelor de proiectare este de a asista rezolvarea unor probleme similare cu unele deja întâlnite și rezolvate anterior. Ele ajută la crearea unui limbaj comun pentru comunicarea experienței despre aceste probleme și soluțiile lor.

Cele 4 elemente cheie care definesc un șablon de proiectare sunt următoarele:

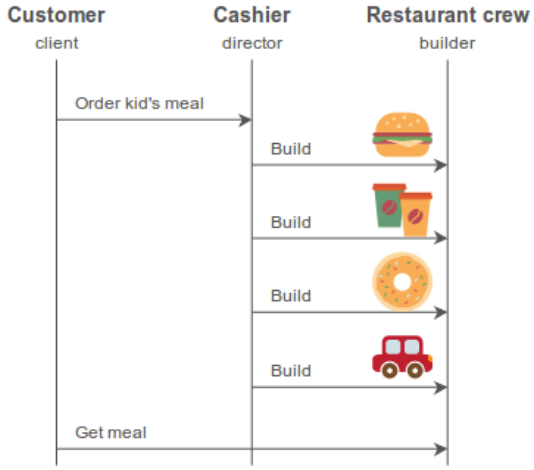
1. **Numele șablonului de proiectare** – având în vedere că există multe tipuri de șabloane de proiectare, este important ca fiecăruia să îi fie aplicat un nume sugestiv, în strânsă legătură cu problema pe care o rezolvă, care să permită identificarea rapidă a acestuia și a documentației aferente.
2. **Descrierea situației în care poate fi aplicat** – este foarte important să știm care sunt situațiile în care putem aplica un șablon de proiectare. De aceea, este nevoie să se prezinte o descriere a problemei și a contextului în care ar putea să apară. Această descriere poate fi realizată din perspective diferite:
   - Ar putea să fie o descriere axată pe aspecte specifice de proiectare, cum ar fi modul de reprezentare a diverșilor algoritmi folosind principiile programării orientate pe obiecte.
   - Poate fi realizată o descriere care să conțină o ierarhie de clase sau o structură de obiecte care sunt implicate în implementarea șablonului de proiectare.
   - Uneori, este important să specificăm o listă de condiții care trebuie să fie îndeplinite pentru a putea aplica șablonul de proiectare. În acest caz, descrierea trebuie să conțină această listă de condiții.
3. **Descrierea soluției** – descrie elementele care alcătuiesc proiectarea, relațiile, responsabilitățile și colaborările acestora. Este indicat ca soluția să nu conțină doar codul complet, ci și o

descriere formală a unei probleme și modul în care o interacțiune generală a conceptelor (clase și obiecte) poate rezolva problema.

4. **Rezultatele și consecințele utilizării** – reutilizarea codului reprezintă, adesea, un factor esențial în programarea orientată pe obiecte, motiv pentru care pentru un șablon de proiectare trebuie să fie prezentate consecințele pe care le au folosirea acestuia asupra flexibilității, extensibilității sau portabilității soluției software.

1.2 Șablonul Builder

Acest pattern este folosit în restaurantele de tip fast food care furnizează meniul pentru copii. Un meniu pentru copii constă de obicei într-un fel principal, unul secundar, o băutură și o jucărie. Pot exista variații în ceea ce privește conținutul mediului, dar procesul de creare este același. Fie că la felul principal se alege un hamburger sau un cheesburger procesul va fi același. Vânzătorul le va indica celor din spate ce să pună pentru fiecare fel de mâncare, pentru băutură și jucărie. Toate acestea vor fi puse într-o pungă și servite clienților.



Acest șablon dorește separarea construcției de obiecte complexe de reprezentarea lor astfel încât același proces să poată crea diferite reprezentări. **Builder**-ul creează părți ale obiectului complex de fiecare dată când este apelat și reține toate stările intermediare. Când meniul este terminat, clientul primește rezultatul de la **Builder**. În acest mod, se obține un control mai mare asupra procesului de construcție de noi obiecte. Spre deosebire de alte pattern-uri, din categoria *creational*, care creau produsele într-un singur pas, pattern-ul **Builder** construiește un produs pas cu pas la comanda coordonatorului.

```
public class User  {
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional
    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }
```

```
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public int getAge() {
        return age;
    }
    public String getPhone() {
        return phone;
    }
    public String getAddress() {
        return address;
    }
    public String toString() {
        return "User:"+this.firstName+" "+this.lastName+" "+this.age+" "+th
            this.address;
    }
    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private int age;
        private String phone;
        private String address;
        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }
        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }
        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }

        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }
        public User build() {
            return new User(this);
        }
    }
```

```
    public static void main(String[] args) {
        User user1 = new User.UserBuilder("Lokesh", "Gupta")
                              .age(30)
                              .phone("1234567")
                              .address("Fake address 1234")
                              .build();
        User user2 = new User.UserBuilder("Jack", "Reacher")
                              .age(40)
                              .phone("5655")
                             //no address
                              .build();
    }
}
```

2 Exerciții propuse

2.1 Singleton – Exercițiul 1

Implementați o clasă `Catalog` care conține o listă cu obiecte de tip `Course`. Va trebui să vă asigurați că pentru această clasă va putea există o singură instanță care să poată fi accesată din orice clasă a proiectului.

```
public class Catalog {
    // TODO -- Adaugati aici implementarea exercitiului
}

public class Course {
}
```

2.2 Factory – Exercițiul 2

Pornind de la clasa abstractă `User`, definiți clasele `Student`, `Parent`, `Assistant` și `Teacher` care vor moșteni clasa `User`.

```
public abstract class User {
    private String firstName, lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String toString() {
        return firstName + " " + lastName;
    }
}
```

Pentru a putea realiza o instanțiere ușoară a obiectelor pentru aceste tipuri de clase, veți implementa o clasă `UserFactory` care va avea o metodă `getUser` ce va returna un obiect de tip `User`, folosind proprietățile șablonului de proiectare **Factory**.

2.3 Builder – Exercițiul 3

Pe baza claselor definite anterior, veți completa implementarea clasei `Course`. În cadrul aplicației noastre, un obiect de tipul `Course` o să conțină: un nume (de tipul `String`), un profesor titular, o listă de asistenți, o colecție ordonată cu obiecte de tipul `Grade` și o listă de studenți. Pentru a putea seta câmpurile unui obiect de tip `Course`, veți folosi șablonul de proiectare **Builder**.

```
public class Grade {
    private Double partialScore, examScore;
    private Student student;

    public Grade() {
        partialScore = 0.0;
        examScore = 0.0;
    }
```

```
    public void setPartialScore(Double score) {
        partialScore = score;
    }

    public void setExamScore(Double score) {
        examScore = score;
    }

    public Double getTotal() {
        return partialScore + examScore;
    }
}
```

Va trebui să vă asigurați că două obiecte de tip `Grade` vor putea să fie comparate (în funcție de punctajul total). De asemenea, va trebui să adăugați în clasa `Catalog` o listă cu obiecte de tip `Course`.

2.4 Observer – Exercițiul 4

Aplicația noastră le permite părinților unui student să se aboneze la `Catalog` pentru a putea primi notificări în momentul în care copilul este notat de către un profesor sau de către un asistent.

Pentru a putea realiza acest lucru, veți folosi șablonul de proiectare **Observer** și veți implementa o clasă `Notification` (stabiliți voi care sunt atributele și metodele din această clasă – este obligatoriu să fie suprascrisă metoda `toString`).

```
public interface Observer {
    void update(Notification notification);
}

public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(Grade grade);
}
```

Important

Rămâne să stabiliți voi ce clasă va implementa interfața `Observer` și ce clasă va implementa interfața `Subject`.

2.5 Strategy – Exercițiul 5

Fiecare profesor va aplica o politică prin care la sfârșitul semestrului selectează cel mai bun student. Pentru a realiza acest lucru în cadrul implementării, va trebui să folosiți șablonul de proiectare **Strategy**. Veți defini câte o clasă pentru fiecare din următoarele strategii:

1. `BestPartialScore` – această strategie va selecta studentul care are cel mai mare punctaj în timpul semestrului;

2. `BestExamScore` – această strategie va selecta studentul care are cel mai mare punctaj în examen;
3. `BestTotalScore` – această strategie va selecta studentul care are punctajul total maxim.

Veți adăuga în clasa `Course` o metodă cu antetul:

```
// Va returna cel mai bun student, tinand cont de strategia aleasa
 ↪  de profesor pentru curs
public Student getBestStudent();
```

2.6 Visitor – Exercițiul 6

Folosind șablonul de proiectare **Visitor**, vom implementa funcționalitatea prin care fiecare asistent o să poată completa notele de pe parcurs ale studenților, iar fiecare profesor o să poată completa notele de la examen ale studenților săi. Pentru acest lucru, vom porni de la următoarele 2 interfețe: `Element` și `Visitor`.

```
public interface Element {
    void accept(Visitor visitor);
}

public interface Visitor {
    void visit(Assistant assistant);
    void visit(Teacher teacher);
}
```

Clasele `Assistant` și `Teacher` for implementa interfața `Element`, iar clasa `ScoreVisitor` va implementa interfața `Visitor`. În clasa `ScoreVisitor` vom avea două dicționare în care sunt stocate notele studenților pentru examene și pentru parcurs.

- Dicționarul `examScores` va avea cheia de tip `Teacher` și valoare de tip listă de `Tuple` (Student, Numele cursului – ca `String`, nota pe care a acordat-o studentului pentru cursul indicat – ca `Double`).
- Dicționarul `partialScores` cu semnificație similară, dar pentru notele de pe parcurs atribuite de asistenți.

În continuare, se va prezenta implementarea de la care veți porni pentru această clasă.

- ***TODO1*** – veți determina toate notele pe care le are de trecut asistentul primit ca parametru de metoda respectivă. Veți verifica dacă pentru o intrare din lista de note există sau nu un obiect de tip `Grade` pentru cursul indicat corespunzător studentului. Dacă există, atunci se va seta nota de pe parcurs pentru acel obiect, dacă nu există, se va crea un nou obiect `Grade` și se va adăuga cursului.
- ***TODO2*** – veți determina toate notele pe care le are de trecut profesorul primit ca parametru de metoda respectivă. Veți verifica dacă pentru o intrare din lista de note există sau nu un obiect de tip `Grade` pentru cursul indicat corespunzător studentului. Dacă există, atunci se va seta nota de la examen pentru acel obiect, dacă nu există, se va crea un nou obiect `Grade` și se va adăuga cursului.

De asemenea, în aceste două metode ar trebui să fie apelată și metoda de trimitere a notificărilor din clasa `Catalog` – metoda `notifyObservers`.

```
public class ScoreVisitor implements Visitor {
    private HashMap<Teacher, ArrayList<Tuple<Student, String,
     ↪  Double>>> examScores;
    private HashMap<Assistant, ArrayList<Tuple<Student, String,
     ↪  Double>>> partialScores;

    private class Tuple<K, V1, V2> {
        private K key;
        private V1 value1;
        private V2 value2;

        public Tuple(K key, V1 value1, V2 value2) {
            this.key = key;
            this.value1 = value1;
            this.value2 = value2;
        }

        public K getKey() {
            return key;
        }

        public V1 getValue1() {
            return value1;
        }

        public V2 getValue2() {
            return value2;
        }
    }

    @Override
    public void visit(Assistant assistant) {
        // TODO1
    }

    @Override
    public void visit(Teacher teacher) {
        // TODO2
    }
}
```