


3 Iunie, 2016

Proiect Calitate și Testare Software

ANDREEA MARIA IONESCU

GRUPA 1069, SERIA B



CUPRINS

1. Introducere	2
2. Pattern-uri Implementate	3
2.1. Creațional	3
2.1.1. Singleton	3
2.1.2. Factory Method	4
2.1.3. Builder	5
2.2. Structurale	7
2.2.1. Adapter	7
2.2.2. Decorator	8
2.2.3. Composite	10
2.3. Comportamentale	13
2.3.1. Strategy	13
2.3.2. Observer	15
3. Metode Testate prin Unit Testing	18
4. Test Case	20
5. Test Suite	21
6. Bibliografie	22

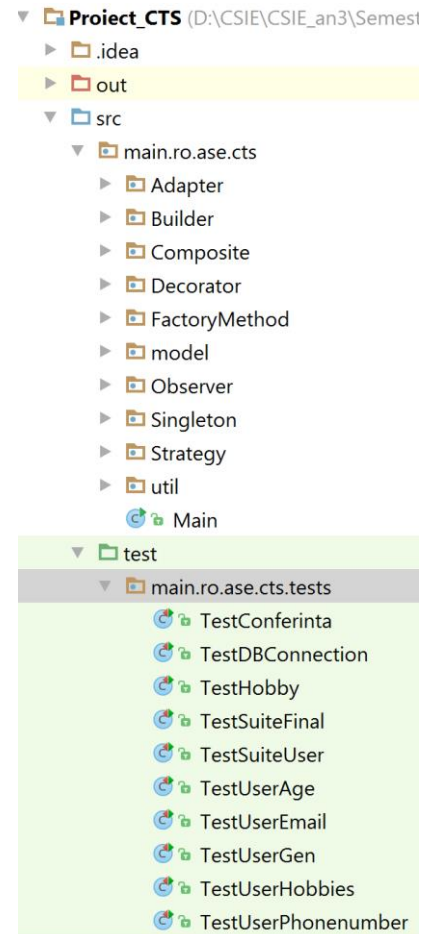
1. INTRODUCERE

Scopul acestui proiect este pe de-o parte integrarea design pattern-urilor studiate la disciplina Calitate și Testare Software, iar pe de altă parte se concentrează pe testarea calității codului scris.

Principală temă pe care o abordează este gestiunea Hobby-urilor și Conferințelor de Dezvoltare Personală la care participă un Utilizator. Astfel, principalele clase care se folosesc atât în cadrul design-pattern-urilor, cât și în testare sunt: *User*, *Hobby* și *Conferinta*. Datele sunt preluate din baza de date, drept urmare s-a realizat o clasă separată pentru conexiune intitulată *SingletonDBConnection*, acesta fiind de tip singleton. Un user se identifică prin următoarele atribute: idUser, name, email, phonenumber, age, gen, location și lista de Hobby-uri. Dintre toate aceste atribute, doar idUser și numele sunt importante, restul atributelor sunt opționale. Astfel, se poate observa că se dorește a fi cât mai flexibilă construirea de utilizatori, tocmai de aceea s-a recurs la un *UserBuilder*. Un hobby are doar trei atribute: idHobby, nameHobby și tipHobby, în timp ce o conferință se caracterizează prin atributele denumire, pret, data, confCategorie.

În cadrul proiectului, utilizatorul a fost pus în diferite scenarii. În primul rând, a fost nevoie de un factory method ce face diferența între tipurile de Hobby-uri aferente utilizatorului. Astfel, în funcție de id-ul utilizatorului, sunt preluate informații din baza de date ce au legătură fie cu Hobby-urile Indoor, fie cu cele Outdoor. De asemenea, denumirea hobby-urilor și impactul asupra utilizatorului este subiectiv, tocmai de aceea, pornind de la un hobby standard, s-a creat un decorator ce permite utilizatorului să descrie hobby-ul respectiv din perspectivă proprie.

În al doilea rând, participarea la conferințe este un hobby destul de amplu ce merită tratat separat. Pentru a putea fi la curent cu ultimele noutăți în materie de conferințe, utilizatorul nostru urmărește o aplicație mobilă ce organizează conferințele sub forma unei structuri ierarhice. Astfel, s-a folosit Composite, pattern-ul ce permite reprezentarea categoriilor de conferințe, dar și conferințele care nu fac parte din nicio categorie, sub formă de arbore. Uneori, utilizatorul are nevoie atât de vizualizarea conferințelor care îi aparțin, dar și de anumite analize asupra acestora. Tocmai de aceea, s-a folosit Strategy cu scopul de a prelucra în mod diferit datele cu privire la conferințe, în funcție de preferințele utilizatorului. Folosind aplicația mobilă, utilizatorul se abonează la anumite grupuri ce organizează conferințe și, implicit, primește



notificări din partea acestor grupuri. Notificările sunt personalizate în funcție de tipul utilizatorului – pentru Programatori sunt afișate sub forma unui Frame, iar pentru Antreprenori la consolă. Pentru realizarea întregului tool de notificare, s-a folosit pattern-ul Observer.

2. PATTERN-URI IMPLEMENTATE

2.1. CREAȚIONALE

Pattern-urile creaționale se concentrează asupra procesului de creare a obiectelor folosind una sau mai multe clase. Acestea tind să separe sistemul de modul cum sunt create, compuse sau reprezentate obiectele.

2.1.1. SINGLETON

Singleton este unul din cele mai populare design patterns, folosindu-se de cele mai multe ori pentru crearea unei singure instanțe pentru o anumite clasă. Astfel, scopul principal al acestui pattern este de a controla crearea de obiecte, limitand numărul de obiecte create la unul singur.

În cazul de față, clasa pe care s-a implementat Singleton-ul este intitulată *SingletonDBConnection* și are ca scop gestiunea conexiunii la baza de date MySQL. Clasa are o referință globală către singura instanță a Singleton-ului și returnează referința respectivă apelând metoda statică *getInstance()*. Tehnica ce se folosește pentru acest tip de Singleton este *lazy instantiation*. Ca rezultat, instanța nu este creată decât în momentul în care metoda *getInstance()* este apelată pentru prima dată. Această tehnică asigură faptul că instanța conexiunii la baza de date este creată doar atunci când este nevoie. Astfel, cu toate că se creează obiectul o singură dată, acesta poate fi refolosit de câte ori este cazul prin referire la acesta, evitându-se în acest mod crearea mai multor obiecte de același tip ce ocupă inutil memoria.

```
18 //referinta catre instanta unica globala
19 private static SingletonDBConnection db = null;
20
21 private SingletonDBConnection() {
22     try {
23         Class.forName(driver);
24     } catch (ClassNotFoundException e) {
25         e.printStackTrace();
26     }
27 }
28
29 private Connection createConnection(){
30     Connection connection = null;
31     try {
32         connection = DriverManager.getConnection(url+dbName, userName, password);
33     } catch (SQLException e) {
34         e.printStackTrace();
35         System.out.println("ERROR: Unable to Connect to Database.");
36     }
37     return connection;
38 }
39
40 public static Connection getInstance()
41 {
42     if(db==null){
43         db = new SingletonDBConnection();
44     }
45     return db.createConnection();
46 }
```

2.1.2. FACTORY METHOD

Pattern-ul Factory Method definește o metodă pentru crearea de obiecte ce fac parte din aceeași familie (interfață). Așadar, se bazează pe moștenire, iar crearea de obiecte este realizată de subclase ce implementează metoda *factory*. Practic, Factory Method are scopul de a delega crearea obiectelor către subclase.

În cazul de față, familia de obiecte este cea din sfera *hobby*-urilor. La rândul lor, hobby-urile pot fi de două tipuri: indoor sau outdoor (în spații închise sau în spații deschise). De aceea, s-a recurs la crearea celor două clase *HobbyINDOOR* și *HobbyOUTDOOR* ce moștenesc clasa abstractă *Hobby* și suprascriu metodele acesteia, adică *getDescription()* și *getWeather()*.

```
6 public abstract class Hobby {
7     private int idHobby;
8     private String nameHobby;
9     private TipHobby tipHobby;
10
11     public Hobby(int idHobby, String nameHobby, TipHobby tipHobby) {...}
16
17     public int getIdHobby() { return idHobby; }
20
21     public void setIdHobby(int idHobby) { this.idHobby = idHobby; }
24
25     public String getNameHobby() { return nameHobby; }
28
29     public void setNameHobby(String nameHobby) { this.nameHobby = nameHobby; }
32
33     public TipHobby getTipHobby() { return tipHobby; }
36
37     public void setTipHobby(TipHobby tipHobby) { this.tipHobby = tipHobby; }
40
41     @Override
42     public String toString() {...}
49
50     //metode abstracte
51     public abstract String getDescription();
52     public abstract String getWeather();
53 }
```

Mai jos, este prezentată metoda factory pentru familia de obiecte Hobby. Această metodă este realizată cu scopul de a ascunde modalitatea de creare a obiectelor. Astfel, atunci când un client (User) își cunoaște id-ul și dorește să vizualizeze toate hobby-urile sale, să poată consulta baza de date doar prin transmiterea parametrilor idUser și tipul hobby-ului (indoor sau outdoor) către metoda factory.

```

32 //FactoryMethod
33 FactoryHobby factoryHobby = new FactoryHobby();
34
35 ArrayList<Hobby> hobbiesIndoor = factoryHobby.getHobbies(1,TipHobby.INDOOR);
36
37 System.out.println("Hobby-urile mele INDOOR sunt: ");
38 for(Hobby hobbyINDOOR : hobbiesIndoor){
39     System.out.println(hobbyINDOOR.toString());
40     System.out.println(hobbyINDOOR.getDescription());
41     System.out.println(hobbyINDOOR.getWeather());
42
43 }

```

```

18 public class FactoryHobby {
19
20     Connection connection = null;
21     PreparedStatement preparedStatement = null;
22     ResultSet resultSet = null;
23
24     public ArrayList<Hobby> getHobbies(int idUser, TipHobby tipHobby){
25         ArrayList<Hobby> hobbies = new ArrayList<Hobby>();
26
27         try {
28             connection = SingletonDBConnection.getInstance();
29             String query = "select * from HOBBY where idUser = ? and tipHobby = ? ";
30             preparedStatement = connection.prepareStatement(query);
31             preparedStatement.setInt(1,idUser);
32             preparedStatement.setString(2,tipHobby.toString());
33
34             switch (tipHobby){
35                 case INDOOR: {
36
37                     resultSet = preparedStatement.executeQuery();
38                     while(resultSet.next()) {
39                         hobbies.add(new HobbyINDOOR(
40                             resultSet.getInt("idHobby"),
41                             resultSet.getString("nameHobby"),
42                             TipHobby.valueOf(resultSet.getString("tipHobby"))));
43                     }
44                     break;
45                 }
46                 case OUTDOOR: {
47                     resultSet = preparedStatement.executeQuery();
48                     while(resultSet.next()) {

```

2.1.3. BUILDER

Pattern-ul Builder este în general folosit pentru crearea de obiecte complexe, care conțin un număr foarte mare de atribute. În clasa *User* de exemplu, atributele *email*, *gen*, *location* și *hobbies* sunt opționale, prioritare fiind doar atributele *id* și *nume*. Această clasă se dorește a fi *immutable*, cu scopul ca atunci când este creată, să nu poată suferi modificări ulterioare. Cu toate că toate atributele sunt declarate *final*, și trebuie setate în constructorul clasei, se dorește și implicarea clientului în construirea obiectului, ca acesta să poată ignora o parte sau toate atributele opționale dacă dorește.

Constructorul din *UserBuilder* primește doar atributele esențiale și, astfel, acestea sunt singurele atribute care sunt declarate *final* pentru a asigura faptul că valorile sunt setate în constructorul din *User*.

Acest pattern este destul de flexibil. Un singur builder poate fi folosit pentru crearea mai multe obiecte ce variază între ele prin metodele din *UserBuilder*, apelând ulterior metoda *build()*.

```
66 | private User(UserBuilder builder){
67 |     this.idUser = builder.idUser;
68 |     this.name = builder.name;
69 |     this.email = builder.email;
70 |     this.gen = builder.gen;
71 |     this.location = builder.location;
72 |     for(Hobby h : builder.hobbies){
73 |         this.hobbies.add(h);
74 |     }
75 | }
76 | public static class UserBuilder{
77 |     private final int idUser;
78 |     private final String name;
79 |     private String email;
80 |     private Gen gen;
81 |     private String location;
82 |     private ArrayList<Hobby> hobbies = new ArrayList<>();
83 |
84 |     public UserBuilder(int idUser, String name){...}
88 |
89 |     public UserBuilder email(String email){...}
93 |
94 |     public UserBuilder gen(Gen gen){...}
98 |
99 |     public UserBuilder location(String location){...}
103 |
104 |     public UserBuilder hobbies(ArrayList<Hobby> hobbies){...}
110 |
111 |     public User build(){
112 |         return new User(this);
113 |     }
```

2.2. STRUCTURALE

Design Pattern-urile structurale se concentrează pe modalitatea prin care clasele și obiectele pot fi compuse cu scopul de a forma structuri mai mari. Practic, aceasta categorie de pattern-uri simplifică structura identificând relații, facilitează obținerea rezultatului dorit prin refolosirea codului scris în prealabil.

2.2.1. ADAPTER

Pattern-ul *Adapter* funcționează ca o punte de legătură între două interfețe care sunt incompatibile. Acest tip de pattern face parte din categoria de pattern-uri structurale și este capabil să combine două interfețe independente una de alta. Astfel, în componența acestuia se regăsește o clasă care este responsabilă să reunească funcționalitățile unor interfețe independente sau incompatibile.

În situația de față, se consideră un *User* care are ca hobby muzica la *MediaPlayer*. Recent, printre hobby-urile lui se numără și vizionarea de clipuri pe youtube. Având în vedere ca *MediaPlayer*-ul personal poate deschide doar fișiere de tip mp3, pentru a putea viziona clipurile în format mp4 sau vlc, este necesar un dispozitiv mai avansat, adică un *AdvancedMediaPlayer*.

Cu privire la implementare, există două interfețe ce se doresc a fi puse în legătură: *MediaPlayer* și *AdvancedMediaPlayer*. Clasa concretă *AudioPlayer* implementează interfața *MediaPlayer*, în timp ce clasele concrete *Mp4Player* și *VlcPlayer* implementează *AdvancedMediaPlayer*. Clasa *AudioPlayer* poate să deschidă implicit doar fișiere cu formatul mp3, în timp ce clasele celelalte (*Mp4Player* și *VlcPlayer*) pot deschide fișiere cu formatele mp4 și, respectiv, vlc.

Scopul final este ca *AudioPlayer* să aibă posibilitatea să deschidă și fișiere media cu alte formate, în afară de formatul implicit (mp3). Pentru asta, a fost creat un adaptor, adică o clasă *MediaAdapter* ce implementează interfața *MediaPlayer* și folosește obiecte de tipul *AdvancedMediaPlayer* pentru a reda melodii cu alte formate.


```

4  * Created by Andreea-Ionescu on 6/1/2016.
5  */
6  public class MediaAdapter implements MediaPlayer {
7
8      AdvancedMediaPlayer advancedMediaPlayer;
9
10     public MediaAdapter(String audiotype) {
11         if(audiotype.equalsIgnoreCase("vlc")){
12             advancedMediaPlayer = new VlcPlayer();
13         }
14         else if(audiotype.equalsIgnoreCase("mp4")){
15             advancedMediaPlayer = new Mp4Player();
16         }
17     }
18
19     @Override
20     public void play(String audiotype, String filename) {
21         if(audiotype.equalsIgnoreCase("vlc")){
22             advancedMediaPlayer.playVlc(filename);
23         }
24         else if(audiotype.equalsIgnoreCase("mp4")){
25             advancedMediaPlayer.playMp4(filename);
26         }
27     }
28 }

```

Clasa AudioPlayer folosește clasa MediaAdapter și îi pasează tipul audio dorit fără a cunoaște cu exactitate ce clasă a redat acel format.

```

4  * Created by Andreea-Ionescu on 6/1/2016.
5  */
6  public class AudioPlayer implements MediaPlayer {
7
8      MediaAdapter mediaAdapter;
9
10     @Override
11     public void play(String audiotype, String filename) {
12
13         if(audiotype.equalsIgnoreCase("mp3")){
14             System.out.println("Play mp3-ul cu numele " + filename);
15         }
16         //mediaAdapter ofera suport astfel incat sa se poata canta si celelalte formate
17         else if (audiotype.equalsIgnoreCase("vlc") || audiotype.equalsIgnoreCase("mp4")){
18             mediaAdapter = new MediaAdapter(audiotype);
19             mediaAdapter.play(audiotype, filename);
20         }
21         else{
22             System.out.println("Suport media invalid " + audiotype + " pentru acest AudioPlayer.");
23         }
24     }
25 }

```

2.2.2. DECORATOR

Pattern-ul Decorator permite utilizatorului adaugarea unei noi funcționalități la un obiect existent, fără a schimba structura acestuia. Acest tip de design pattern face parte din categoria de pattern-uri structurale și acționează ca un wrapper asupra unei clase existente. Practic, acest pattern conține o clasă decorator ce "învelește" clasa originală și oferă o funcționalitate suplimentară, menținând în același timp semnătura metodelor din clasa, intacte.

Un avantaj considerabil în implementarea acestui pattern este faptul că extinderea funcționalității obiectului se face dinamic, adică la run-time. De asemenea, decorarea se face pe mai multe niveluri, însă transparent pentru utilizator. S-a ales acest pattern pentru situația în

care utilizatorul dorește să adauge funcționalități hobby-urilor sale, iar prin decorator le împachetează pentru a le oferi o tentă personală.

La nivel de implementare, s-a preluat clasa abstractă *Hobby*, ce definește interfața obiectelor de tip hobby ce pot fi decorate cu noi funcții. Ulterior, a fost creată o clasă concretă *HobbyStandard* ce moștenește clasa de bază *Hobby* și definește obiectele de bază de la care se pornește decorarea propriu-zisă.

```
9 public class HobbyStandard extends Hobby {
10
11     public HobbyStandard(String nameHobby) {
12         this.setNameHobby(nameHobby);
13     }
14
15     @Override
16     public String getDescription() {
17         return getNameHobby() + " - un hobby ce e bine sa il ai mereu in vedere";
18     }
19
20     @Override
21     public String getWeather() {
22         return null;
23     }
24 }
```

Ulterior, a fost creată clasa decorator abstractă *DecoratorHobby* ce extinde clasa de bază *Hobby*. Această clasă reține instanța din obiectul *Hobby*.

```
7  * Created by Andreea-Ionescu on 6/1/2016.
8  */
9  public abstract class DecoratorHobby extends Hobby {
10
11     protected final Hobby hobbyAbstract;
12
13     public DecoratorHobby(Hobby hobby) {
14         this.hobbyAbstract = hobby;
15     }
16
17     @Override
18     public String getDescription() {
19         return hobbyAbstract.getDescription();
20     }
21
22     @Override
23     public String getWeather() {
24         return hobbyAbstract.getWeather();
25     }
26 }
```

HobbyCalatorit este una din clasele concrete create ulterior ce folosește *DecoratorHobby* pentru a decora obiectele de tip *Hobby*. Printre aceste clase concrete ce se folosesc de decorator, se numără și *HobbyVoluntariat* și *HobbyConferințe*.

```

8 public class HobbyCalatorit extends DecoratorHobby {
9
10 public HobbyCalatorit(Hobby hobby) {
11     super(hobby);
12 }
13
14 @Override
15 public String getDescription() {
16     return super.getDescription() + ", mai ales pentru ca vizitezi locuri noi";
17 }
18 }
19

```

2.2.3. COMPOSITE

Pattern-ul Composite este folosit atunci cand este nevoie tratarea unui grup de obiecte în mod similar ca un singur obiect. Acest pattern este compus din obiectele ce aparțin unei forme aborescente: conține atât reprezentarea frunzelor, cât și reprezentarea întregii structuri ierarhice. Practic, nodurile intermediare și cele frunză sunt tratate unitar.

În cadrul acestui pattern, se pot aplica operații identice atât la nivelul nodurilor intermediare, cât și la nivelul frunzelor. În cadrul structurii arobrescente, un nod intermediar este o clasă ce poate avea copii, în timp ce o frunză reprezintă o clasă “primitivă” ce nu are copii.

Pentru situația curentă, s-a ales aplicarea acestui pattern pentru cazul în care unul din hobby-urile unui utilizator este acela de a participa la numeroase Conferințe. Aceste conferințe pot avea topicuri diferite ce aparțin unei anume categorii sau pot avea subiecte care nu fac parte din nicio categorie. De exemplu, utilizatorul poate participa la conferinte din categoria IT, iar pentru implementare este necesară o clasă composite. În altă situație, utilizatorul poate participa la conferințe ce nu fac parte dintr-o categorie anume (Conferința Pescarilor), iar pentru asta este nevoie de o clasă frunză.

Clasa frunză (*Conferință*) și cea composite (*CategorieConferință*) au un punct comun de reper și anume interfața „Componentă” (*ElementCatalog*) care definește toate operațiile ce pot fi suprascrise de clasele frunză și cele composite. Când se efectuează o operație asupra unui composite, operația se efectuează pe toți copii acelui composite, chiar daca acești copii sunt la rândul lor fie frunze, fie composite.

```

8 public abstract class ElementCatalog {
9
10     //interfata pentru frunze - conferintele din catalogul aplicatiei
11     public abstract String getDescriere();
12     public abstract double getPret();
13     public abstract Date getData();
14
15     //interfata pentru nodurile intermediare - exista mai multe categorii de conferinte
16     // astfel sunt mai multe cataloage de unde poate alege un client
17     //gestiunea colectiei de noduri
18     public abstract void adaugaNod(ElementCatalog element);
19     public abstract void stergeNod(ElementCatalog element);
20     public abstract ElementCatalog getElement(int index);
21     public abstract int getSize();
22 }

```

The Gang of Four (Erich Gamma, 1995) au descris pattern-ul Composite ca fiind interacțiunea unui client cu o structură arborescentă prin intermediul unei interfețe Component. În primul rând, această interfață include operațiile comune pentru clasele composite și frunză, iar pentru cazul de față sunt *getDescriere()*, *getPret()* și *getData()*. În al doilea rând, interfața conține operațiile *adaugaNod()*, *stergeNod()*, *getElement()* folosite pentru elementele composite ale structurii arobrescente.

Clasa frunză *Conferinta* conține atribute proprii și suprascrie metodele *getDescriere()*, *getPret()* și *getData()* specifice interfeței Component (*ElementCatalog*).

```

8 public class Conferinta extends ElementCatalog {
9     String denumire;
10     double pret;
11     Date data;
12     String confCategorie;
13
14     public Conferinta(String denumire, double pret, Date data, String confCategorie) {
15         this.denumire = denumire;
16         this.pret = pret;
17         this.data = data;
18         this.confCategorie = confCategorie;
19     }
20
21     @Override
22     public String getDescriere() { return this.denumire; }
23
24
25
26     @Override
27     public double getPret() { return this.pret; }
28
29
30
31     @Override
32     public Date getData() { return this.data; }
33
34
35

```

Clasa composite, *CategorieConferinte*, preia de la clasa Component metodele specifice și le prelucrează corespunzător, acestea fiind aplicate asupra tuturor copiilor. Acești copii pot fi la

rândul lor considerați Componente, din moment ce există posibilitatea să fie ori obiecte de tip frunză, ori obiecte composite ce și amândouă preiau interfața Component.

```
10 public class CategorieConferinte extends ElementCatalog {
11
12     //leaga structura ierarhica
13     AbstractList<ElementCatalog> elemente = new ArrayList<>();
14     String denumireCategorie;
15
16     public CategorieConferinte(String denumireCategorie) {
17         this.denumireCategorie = denumireCategorie;
18     }
19
20     @Override
21     public String getDescriere() {...}
22
23     @Override
24     public double getPret() { throw new UnsupportedOperationException(); }
25
26
27     @Override
28     public Date getData() { throw new UnsupportedOperationException(); }
29
30
31     @Override
32     public void adaugaNod(ElementCatalog element) { elemente.add(element); }
33
34
35     @Override
36     public void stergeNod(ElementCatalog element) { elemente.remove(element); }
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
```

2.3. COMPORTAMENTALE

Pattern-urile comportamentale se concentrează pe modalitatea prin care obiectele colaborează între ele sau prin modul cum sunt delegate responsabilitățile între acestea. Această categorie de design patterns caracterizează comunicarea unui obiect și simplifică controlul dinamic al comportamentului obiectului respectiv. Spre deosebire de categoriile de pattern-uri creaționale și structurale, care se preocupă cu procesul de instanțiere și the marcare a obiectelor și a claselor, ideea centrală aici este să se concentreze pe modalitatea cum obiectele sunt interconectate.

2.3.1. STRATEGY

Pattern-ul Strategy oferă posibilitatea utilizatorului să selecteze comportamentul unui algoritm la run-time. Acest pattern definește o familie de algoritmi, încapsulează fiecare algoritm și permite interschimbarea algoritmilor în familia respectivă.

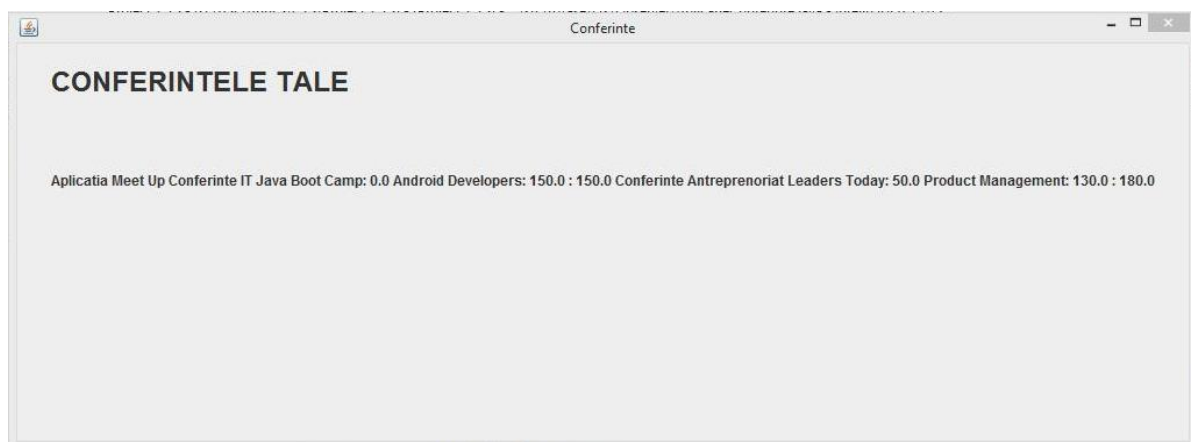
Conform acestui pattern, comportamentul unei clase nu ar trebui moștenit, ci mai degrabă încapsulat folosind interfețe. În cazul de față, se aplică Strategy pentru o aplicație de conferințe pusă la dispoziția utilizatorului și permite astfel efectuarea de diferite prelucrări și aplicarea de algoritmi în funcție de preferințele clientului. Folosind clasa composite *ElementCatalog* de la pattern-ul structural Composite, se dorește efectuarea de prelucrări asupra întregii structuri ierarhice. Astfel, se creează o interfață *StrategiePrelucrareAplicatieConferinte* unde se adaugă signaturile funcțiilor ce influențează comportamentul obiectului central, adică funcția *prelucrare()*.

```
8 public class AplicatieConferinte {
9
10     //gestioneza aplicatia
11     //flexibila la modificarea si schimbarea afisarii catalogului in functie de context
12     ElementCatalog catalog;
13
14     //referinta catre strategia de prelucrare
15     StrategiePrelucrareAplicatieConferinte refStrategie;
16
17     public AplicatieConferinte(ElementCatalog catalog) { this.catalog = catalog; }
18
19
20
21     //accesul la referinta cand vreau sa modific strategia
22     public void setRefStrategie(StrategiePrelucrareAplicatieConferinte refStrategie) {
23         this.refStrategie = refStrategie;
24     }
25
26     public void prelucrare() {
27         if (this.refStrategie != null) {
28             this.refStrategie.prelucreaza(this.catalog);
29         }
30         else
31             throw new UnsupportedOperationException();
32     }
33 }
```

Ulterior, această interfață ce definește comportamentul este implementată de subclasele *StrategieAfisareGUI* și *StrategieAnaliza*. Fiecare subclasă suprascrie diferit metoda din interfața *StrategiePrelucrareAplicatieConferinte*.

În cazul *StrategieAfisareGUI*, se deschide un frame în care sunt afișate toate conferințele pe care le conține structura ierarhică din cadrul Aplicației de Conferințe.

```
13 public class StrategieAfisareGUI implements StrategiePrelucrareAplicatieConferinte {
14
15     @Override
16     public void prelucreaza(ElementCatalog catalogConferinte) {
17         final JFrame frame = new JFrame("Conferinte");
18         final JLabel label = new JLabel();
19         frame.setSize(1040, 880);
20         JPanel panel = new JPanel();
21
22         JButton button1 = new JButton("Vezi Conferinte");
23         button1.setSize(100,50);
24
25
26         StringBuilder sb = new StringBuilder().append("<html> <h1>CONFERINTELE TALE</h1>\n")
27             .append("<div>")
28             .append(catalogConferinte.getDescriere())
29             .append("</div> </html>");
30         label.setText(sb.toString());
31
32         label.setVisible(false);
33         frame.add(panel);
34         panel.add(button1);
35         panel.add(label);
36         frame.setVisible(true);
37
38         button1.addActionListener(new ActionListener() {
39
40             public void actionPerformed(ActionEvent arg0) {
41                 label.setVisible(true);
42                 button1.setVisible(false);
43             }
44         });
45     }
```



În cazul *StrategieAnaliza*, se calculează prețul tuturor conferințelor care se regăsesc în Aplicație.

```

9 public class StrategieAnaliza implements StrategiePrelucrareAplicatieConferinte{
10
11     @Override
12     public void prelucreaza(ElementCatalog catalogConferinte) {
13         StringBuilder sb = new StringBuilder()
14             .append("Pretul total pentru toate conferintele este: ")
15             .append(pretTotal(catalogConferinte));
16         System.out.println(sb.toString());
17     }
18
19     private double pretTotal (ElementCatalog element){
20         if (element == null)
21             return 0;
22         if (element instanceof Conferinta)
23             return 1;
24         else{
25             double s=0;
26             for (int i=0;i<element.getSize();i++){
27                 s+=element.getElementMeniu(i).getPret();
28             }
29             return s;
30         }
31     }
32 }
33 }

```

Pentru pattern-ul Strategy, comportamentul este definit ca una sau mai multe interfețe separate și există clase specifice ce implementează aceste interfețe. Acest lucru permite o mai bună interacționare între comportament și clasa care folosește comportamentul respectiv. Comportamentul se poate modifica fără a afecta clasele care îl folosesc, iar clasele pot schimba strategiile comportamentale fără a interveni cu schimbări la nivel de cod.

2.3.2. OBSERVER

Pattern-ul Observer definește relații de tipul unul la mai mulți (one-to-many) între Observabil și observatori. În cazul de față, utilizatorii/clientii (observatorii) trebuie să fie notificați cu privire la schimbările ce apar la nivelul stării observabilului. Observatorii au posibilitatea să fie adăugați sau sterși de la abonarea la un anume Observabil.

Așadar, pentru situația de față s-a ales aplicarea acestui pattern la nivelul clienților ce doresc să fie notificați cu privire la conferințele din cadrul unei aplicații ce gestionează toate conferințele în funcție de fiecare categorie în parte.

Clasa Observabilă se numește *AplicatieConferinte*, iar observatorii sunt *ClientConferinta*. Pentru a-și îndeplini rolul de Observabil, clasa are nevoie de următoarele funcționalități: *adaugaClient()*, *stergeClient()* și *notificaClienti()*.


```

22
23 //interfata publica pentru gestiunea clientilor/observatorilor
24 //metode prin care un client se poate inregistra la acest observabil sau se poate dezabona
25 public void adaugaClient(ClientConferinta client) { this.clienti.add(client); }
26
27
28
29 public void stergeClient(ClientConferinta client) { this.clienti.remove(client); }
30
31
32
33 private void notificaClienti(Conferinta conferinta){
34     if(this.clienti!=null){
35         for(ClientConferinta client : this.clienti){
36             client.notificareConferinta(conferinta);
37         }
38     }
39 }
40
41 //metoda prin care se genereaza un eveniment
42 public void incarcaDateDB(String categorieConf){
43     ArrayList<Conferinta> conferinte = new ArrayList<Conferinta>();
44     Connection connection = null;
45     PreparedStatement preparedStatement = null;
46     ResultSet resultSet = null;
47     Conferinta conferinta = null;
48
49     try {
50         connection = SingletonDBConnection.getInstance();
51         String query = "select * from CONFERINTE where categorieConf = ? ";
52         preparedStatement = connection.prepareStatement(query);
53         preparedStatement.setString(1,categorieConf.toString());
54         resultSet = preparedStatement.executeQuery();
55         while (resultSet.next()){
56             conferinta = new Conferinta(

```

În plus, observabilul conține și metoda prin intermediul căreia se generează un eveniment și anume metoda *incarcaDateDB()* ce primește ca parametru categoria de conferințe și returnează din baza de date acele conferințe care corespund categoriei date. A se remarca faptul că s-a folosit Singletonul pentru preluarea conexiunii la baza de date.

De asemenea, clasa Observabilă conține o listă cu toți clienții care doresc să primească notificări. Acești clienți se abonează la rândul lor la evenimentele din cadrul Aplicației. Interfata *ClientConferinta* definește modalitatea prin care sunt notificați observatorii și poate permite gestiunea mai multor observatori.

```

6 Created by Andreea-Ionescu on 6/1/2016.
7 */
8 public interface ClientConferinta {
9
10     public void notificareConferinta(Conferinta conferinta);
11 }

```

Clasele concrete de observatori, *Antreprenor* și *Programator*, implementează interfața *ClientConferinta* și astfel suprascriu funcția *notificareConferinta()* în mod diferit. Pentru *Antreprenor*, de exemplu, afișarea notificărilor se realizează prin intermediul unei ferestre de tip pop-up, în timp ce pentru *Programator*, notificările sunt afișate la consolă.



```
22  @Override
23  public void notificareConferinta(Conferinta conferinta) {
24
25      frame.setSize(1040, 880);
26      button1.setSize(100,50);
27
28      StringBuilder sb = new StringBuilder().append("<html> <h1>CONFERINTELE TALE<h1>\n")
29          .append("<div>")
30          .append(new StringBuilder("Conferinta pe Antreprenoriat: \n"
31              + conferinta.getDescriere()
32              + " este pe data - "
33              + conferinta.getData()
34              + " si are pretul: "
35              + conferinta.getPret()
36              + " RON"))
37          .append("</div> </html>");
38      label.setText(sb.toString());
39      panel.setBackground(Color.orange);
40
41      label.setVisible(false);
42      frame.add(panel);
43      panel.add(button1);
44      panel.add(label);
45      frame.setVisible(true);
46
47      button1.addActionListener(new ActionListener() {
48
49          public void actionPerformed(ActionEvent arg0) {
50              label.setVisible(true);
51              button1.setVisible(false);
52          }
53      });
```

3. METODE TESTATE PRIN UNIT TESTING

Prin intermediul unit testing-ului, se realizează testarea codului sursă scris de programatori. O metodă de testare reprezintă o funcție ce testează bucăți de cod, fie că este vorba despre un atribut sau de o funcție ce face parte dintr-o clasă anume. Așadar, rolul principal al acestor metode este de a testa starea și comportamentul obiectului în diferite situații. Decoperind cât mai multe probleme sau buguri, se verifică astfel corectitudinea aplicației și asigură în același timp calitatea produsului software obținut. Este de preferat ca testele să fie cât mai concise.

Metodele de testare au fost realizate folosind librăria Junit3, iar acestea fac parte din *Right-BICEP* (Allan, 2010), printre care se verifică corectitudinea rezultatelor (*Right*), limitele *Boundary Conditions* definite CORRECT (**C**onformance, **O**redering, **R**ange, **R**eference, **E**xistence, **C**artinality, **T**ime), relațiile inverse (*Inverse Relationships*), verificarea rezultatului prin alte metode (*Cross-Check*), condițiile care generează erori (*Error-conditions*), performanța execuției (*Performance*).

Așadar, în cadrul proiectului s-au dezvoltat următoarele metode ce fac parte din categoriile menționate anterior:

- **Corectitudinea Rezultatelor** (Right - Sunt rezultatele corecte?)

1. testSetEmail()

Se folosește de metoda de tip `assertEquals()` pentru a identifica dacă e-mailul este cel corespunzător. Această metodă practic testează metoda `setEmail()` din clasa `User`, și face parte din `TestCase`-ul *TestUserEmail*.

- **Conformance** (Valoarea are formatul corect?)

3. testValueOfGen()

Având în vedere faptul că gen-ul unei persoane face parte dintr-o structură de tip Enumerație (`Enum`) care la rândul ei conține `String`-uri pentru a putea reprezenta genurile, acest test are rolul de a testa dacă valoarea dată are formatul corect și poate efectua conversia la `String`. Acest test face parte din `TestCase`-u *TestUserGen*.

- **Ordering** (Setul de valori trebuie sa fie ordonat sau nu?)

4. testEmailCheckSpelling()

Acest test verifică în primul rând dacă email-ul conține caracterele specifice „@” și „.” și alertează în cazul în care nu există. De asemenea, este important ca valoarea „@” să fie precedată de valoarea „.” și nu invers. Astfel, se verifică și ordinea apariției a acestor două caractere. `TestCase`-ul pentru acest test este *TestUserEmail*.

- **Range** (este valoarea între limitele maxim și minim acceptate?)

5. testVarstaInterval()

Atunci când se introduce o valoare pentru atributul vârstă, este important ca valoarea respectivă să nu fie una absurdă, adică să existe o valoare negativă sau una foarte mare. Pentru situația de față, se verifică dacă vârsta se găsește în interval 10-120 ani. Acest test aparține TestCase-ului *TestUserAge*.

6. testPhoneNumberLength()

Pentru ca un număr de telefon să fie valid, este necesar ca acesta să conțin cel puțin 10 cifre. Tocmai de aceea, se testează ca lungimea minimă a atributului de tip String phonenumber să fie de 10 caractere în cadrul TestCase-ului *TestUserPhonenumber*.

7. testIdHobbyNotNegative()

Atunci când se introduce un nou Hobby în baza de date, este necesar ca id-ul Hobby-ului respectiv să nu aibă valoarea 0 sau o valoare negativă, tocmai de aceea se aplică un test asupra id-ului acestuia și se verifică limita minimă de 1. Testul face parte din TestCase-ul *TestHobby*.

8. testPretConferinta()

O altă limită care se impune a fi respectată are legătură cu prețul unei conferințe. Această valoare trebuie să fie cel puțin egală cu 0, astfel se adaugă și acest test în cadrul TestCase-ului *TestConferinta*.

- **Reference** (Se face referire la componente externe care nu sunt controlate direct?)

9. testGetConferintaById()

Se testează dacă a fost găsită conferința în funcție de id-ul dat, iar dacă rezultatele preluate din baza de date corespund cu cele dorite. Pentru a putea compara eficient două obiecte de tip Conferinta, s-a recurs la suprascrierea metodei compareTo din cadrul interfeței Comparable. Această metodă face parte din TestCase-ul *TestConferinta*.

10. testCreateConnection()

Acest test are rolul de a testa dacă a fost creată conexiunea la baza de date. Astfel, se folosește metoda assertNotNull() ce verifică dacă există conexiunea respectivă. Acest test îi corespunde clasei *TestDBConnection*.

- **Existence** (Valoarea există?)

11. testArrayHobbiesIsNull()

Principalul scop al acestui test este de a verifica dacă lista de obiecte de tip Hobby este nulă sau nu, iar acesta face parte din TestCase-ul *TestUserHobbies*.

12. testGenNotNull()

Datorită faptului că clasa User se poate construi folosind UserBuilder, unele din atributele ce aparțin unui utilizator pot fi opționale. Drept urmare, acest test verifică dacă a fost adăugat o valoare pentru atributul gen atunci când a fost construit un utilizator și aparține TestCase-ului *TestUserGen*.

13. testIdHobbyNotNull()

Prin intermediul metodei `assertNotNull()`, se testează dacă obiectul de tip Hobby a fost instanțiat, iar metoda de test este inclusă în TestCase-ul *TestHobby*.

- **Cross-Check** (Se poate verifica rezultatul și prin alte metode?)

14. testDataConferintaValida()

Metoda este integrată în cadrul TestCase-ului *TestConferinta* și face referire la testarea metodei *dataConferintaValida()* prin intermediul altei metode din api-ul clasei Date, care compara cele două date.

- **Error-Conditions** (Se pot genera posibile erori standard la implementarea codului?)

15. testArrayHobbiesIsEmpty()

Metoda ce verifică dacă dimensiunea listei de obiecte de tip Hobby este cea potrivită și, dacă există astfel obiecte în lista respectivă, iar în caz contrar, se poate genera o excepție de tipul `NullPointerException`.

16. testArrayHobbiesGetElementByIndex()

Compară dacă este returnat elementul din listă care se regăsește la o poziție dată. Acest test este în strânsă legătură cu eroarea `IndexOutOfBoundsException`. Tocmai de aceea, atunci când este dat un index, este necesar ca acesta să fie cel puțin egal cu 0 și maxim egală cu poziția la care se găsește ultimul element din listă. Atât această metodă, cât și cea precedentă, aparțin TestCase-ului *TestUserHobbies*.

4. TEST CASE

Conceptul de Test Case se referă la o clasă ce definește setul de obiecte (fixture) și permite rularea mai multor teste. Practic, un Test Case înglobează mai mulți pași succesivi pe care un tester îi rulează cu scopul de a observa dacă funcționalitățile unui produs sunt implementate.

Fiecare Test Case realizat în cadrul proiectului implementează metodele *setUp()* și *tearDown()*. Pe de-o parte, în cadrul metodei *setUp()* sunt construite și inițializate obiectele

(*fixture*), unde aceasta se apelează înainte fiecărei metode de testare, iar pe de altă parte, metoda *tearDown()* distruge resursele alocate testului.

În cadrul JUnit3, instrumentul principal de rulare a testelor și de afișare a rezultatelor se numește Test Runner. Pentru a permite acestei componente să identifice automat clasele utilizate la testare, este important ca numele claselor de test să aibă numele sub forma „*TestMyClass*” sau „*MyClassTest*”.

Astfel, în cadrul proiectului s-au creat mai multe TestCase-uri referitoare la clasele folosite ca model în cadrul proiectului, a căror metode au fost menționate anterior. Printre aceste TestCase-uri, amintim cele referitoare la clasa *User*: *TestUserAge*, *TestUserEmail*, *TestUserGen*, *TestUserHobbies* și *TestUserPhonenumber*, cel referitor la clasa *Hobby* – *TestHobby*, cel referitor la clasa *Conferinta* – *TestConferinta*, dar și un TestCase care testează conexiunea la baza de date, *TestDBConnection*.

5. TEST SUITE

Test Suite reprezintă o colecție formată din mai multe Test Case-uri. Practic, acesta reprezintă o colecție externă unde pot fi combinate toate testele și încărcate toate metodele din clasele de Test Case sau doar metodele din suite.

În cadrul acestui proiect s-au realizat două Test Suite:

1. *TestSuiteUser* - înglobează toate TestCase-urile ce fac referire la un obiect de tip *User*
2. *TestSuiteTotal* – pe lângă *TestSuiteUser*, include și celelalte TestCase-uri care sunt independente de clasa *User*

```

7 ▶ public class TestSuiteFinal extends TestCase {
8
9     protected void setUp() throws Exception {
10         super.setUp();
11         System.out.println("***** Pregatire Testarea Totala ***** ");
12     }
13
14     protected void tearDown() throws Exception {
15         super.tearDown();
16         System.out.println("***** Terminare Testare Finala *****");
17     }
18
19     public static Test suite(){
20         TestSuite colectieTesteProiect = new TestSuite();
21
22         colectieTesteProiect.addTestSuite(TestSuiteUser.class);
23         colectieTesteProiect.addTest(new TestConferinta());
24         colectieTesteProiect.addTest(new TestDBConnection());
25         colectieTesteProiect.addTest(new TestHobby());
26
27         return colectieTesteProiect;
28     }
29
30 }

```

Pentru a putea testa toate testele care au fost efectuate în cadrul proiectul, se rulează această clasă de Test Suite.

6. BIBLIOGRAFIE

- Allan, A. (2010). *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf.
- Erich Gamma, R. H. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley.