Universitatea Alexandru Ioan Cuza Iași

Facultatea de Economie și Administrarea Afacerilor

Specializarea: Sisteme informaționale pentru afaceri

**Proiect Design Pattern**

**Disciplina Analiză și proiectarea orientate obiect**

Sistem de gestiunea resurselor umane

**Profesor coordonator:**

Prof. dr. Dumitriu Florin

Prof drd. David Burcovschi

## **Studenți:**

Casian Larisa SIA12

## Turcu Constantin-Eduard SIA13

1. **Definirea problemei**

Se dorește implementarea a trei funcționalități în aplicația pentru gestiunea cursurilor de formare:

1. Calcularea mediei notelor obținute de angajați la cursurile de formare, și calcularea mediei notelor pentru toate training-urile.
2. Notificarea trainerilor despre desfășurarea unui noi curs.
3. Crearea interfeței utilizator.

Pentru calcularea mediei angajaților cât și a mediei pentru training-uri avem nevoie să fie desfășurat cel puțin un training la care a participat cel puțin un angajat. Astfel se vor crea metodele **calculeazaMedieAngajati** și **calculeazaMedieTraining,** aceste metode vor fi adăugate in clasele *mediaAngajati* și *mediaTraining*.

Pentru notificarea Trainerilor este important să dispunem de Traineri. Pentru realizarea acestei implementări vom utiliza metoda **notificareTrainers**.

Interfața va conține semnătura metodelor existente în aplicație și va avea numele de Facade Interface.

Acesta fiind contextul, problema concretă este următoarea: cum vor implementa metodele de calcul, cum vor realiza notificarea și cum vom crea interfața?

1. **Soluții fără aplicarea unui șablon de proiectare**

Pentru rezolvarea acestei probleme ar putea fi imaginat 2 soluții:

*Soluție 1:*

1. Implementarea in clasa Formare a tuturor metodelor și modelelor de calcul, astfel:

* Se adaugă atributele în clasa Formare de tip *ArrayList<Angajati>* și *ArrayList<Formare>* care vor fi parametri metodelor folosite: **calculeazaMedieAngajati** și **calculeazaMedieTraining**. Atributul va stoca lista angajaților și a cursurilor de formare;
* Se adaugă cate o operație (metodă) pentru fiecare metodă de calcul, care va implementa algoritmul specific fiecărei metode;

1. Implementarea stării clasei Formare prin o clasă internă:

* Se crează o clasă nested în clasa formare de tip enum;

1. Implementarea unei interfețe simple Java care definește un set de metode:

* Interfața expune un set de metode pe care alte clase sau obiecte trebuie să le implementeze;

*Soluție 2:*

1. Se creează o clasă pentru fiecare metodă de calcul și aplicarea polimorfismului, deoarece calcularea mediei angajaților se diferențiază de calcularea mediei pentru cursurile de formare;
2. Definirea unei clase speciale pentru implementarea metodei notificareTrainers;
3. Crearea interfeței funcționale care definește un set specific de metode, astfel implementând clasa Interface principală.

**2.1 Dezavantajele soluțiilor propuse**

*Soluția 1:*

* Adăugarea atributele de tip *ArrayList<Angajati>* și *ArrayList<Formare>* în clasa Formare poate duce la o legătură strânsă între diferitele aspecte ale sistemului. Aceasta poate face ca clasa Formare să devină dependentă de detaliile specifice ale listelor de angajați și cursurilor de formare. În cazul în care structura sau reprezentarea acestor entități se schimbă, este posibil să fie necesară o modificare semnificativă a clasei Formare;
* Adăugarea a două metode suplimentare, **calculeazaMedieAngajati** și **calculeazaMedieTraining**, poate duce la creșterea complexității clasei Formare. O clasă prea complexă poate fi dificil de înțeles, de întreținut și de extins în viitor;
* Implementarea stării clasei Formare prin intermediul unei clase interne de tip enum poate duce la o limitare a extensibilității. Dacă în viitor este nevoie de adăugarea unor stări suplimentare pentru Formare, modificarea enum-ului poate necesita rescrierea codului și recompilarea acestuia, ceea ce poate fi inconvenabil și poate introduce riscuri de eroare;
* În cazul implementării stării prin enum în clasa Formare, adăugarea de noi stări poate implica modificarea directă a clasei existente. Aceasta poate duce la încălcarea principiului Open-Closed Principle, deoarece modificările directe în clasa existentă pot avea impact asupra funcționalității existente și pot genera erori neașteptate în alte părți ale sistemului;
* Adăugarea de noi metode în interfață sau modificarea semnăturilor existente poate duce la probleme de întreținere.

*Soluția 2:*

* Crearea unei clase separate pentru fiecare metodă de calcul poate duce la o creștere semnificativă a numărului total de clase în proiect. În cazul în care există mai multe metode de calcul sau funcționalități similare care necesită clase separate, acest lucru poate duce la o ierarhie complexă și la o structură de cod dificil de gestionat și de înțeles.
* Complexitatea în extinderii în cazul creării unei clase pentru implementarea unei metode simple;
* Definirea unei clase speciale pentru implementarea metodei notificareTrainers poate duce la o cuplare strânsă între această clasă și clasa sau clasele care o utilizează.
* Acest scenariu ar încărca memoria suplimentară pentru gestionarea obiectelor;

1. **Soluția de aplicare a șabloanelor Singleton, Observer și Facade:**
2. Cea mai eleganta soluție presupune aplicarea șablonului de proiectare Singleton. În acest sens se creează:

Clasa mediaAngajați și mediaTrainers are doar o instanță și oferă un punct de acces global la acea instanță. Acesta este util atunci când avem nevoie de o singură instanță a unei clase pentru a coordona acțiunile într-un sistem. În contextul creării unei clase pentru gestionarea informațiilor despre calculul mediei notelor angajați și a notelor pentru cursurile de formare (mediaAngajati și mediaTrainings). Astfel încât se gestionează în mod coerent și eficient informațiile legate de angajați și instruiri în întregul sistem.

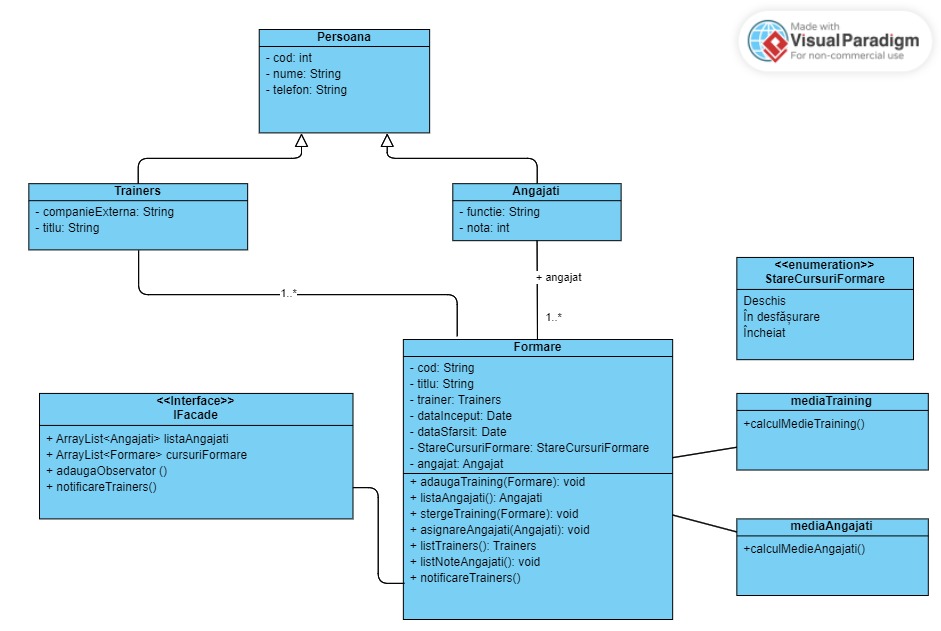
1. Cea mai eleganta soluție presupune aplicarea șablonului de proiectare Observer. În acest sens se parcurge scenariul:

Se creează metodele în clasa Formare: adaugaObservator și notificareTrainers, în cazul primei metode se primește un obiect de tip Trainers (care este un observator) și îl adaugă la o colecție de observatori stocați în cadrul obiectului. NotificareTrainers - această metodă este responsabilă pentru notificarea tuturor observatorilor adăugați anterior prin apelul metodei adaugaObservator. Prin intermediul unei iterție for, parcurgem fiecare observator din colecția observatori și apelem metoda notificareNouCurs() pe fiecare observator. Aceasta este o metodă specifică fiecărui observator care este apelat atunci când obiectul observat Formare suferă o modificare sau o actualizare a stării.

1. Cea mai eleganta soluție presupune aplicarea șablonului de proiectare Facade:

Interfața în contextul șablonului Facade servește ca un strat simplificator între un sistem complex și utilizator, ascunzând detaliile interne ale acestuia. Prin intermediul acestei interfețe, utilizatorii pot interacționa cu sistemul într-un mod simplu și intuitiv, fără a fi nevoiți să cunoască detaliile complicate ale funcționării sale interne. Aceasta oferă o modalitate eficientă de a gestiona complexitatea unui sistem prin consolidarea funcționalităților sale sub o interfață unică și coerentă. Folosind șablonul Facade, interfața devine un punct central de acces la funcționalitățile sistemului.

Diagrama de clase rezultată prin aplicarea șabloanelor Singleton, Observer și Facade este prezentată mai jos.



Codul pentru aplicarea șabloanelor este redat in continuare.

Super clasa Persoana

|  |
| --- |
| public class Persoana {  String nume;  int cod;  String telefon;   public Persoana(String nume, int cod, String telefon) {  this.nume = nume;  this.cod = cod;  this.telefon = telefon;  } } |

Clasa Angajati

|  |
| --- |
| public class Angajati extends Persoana {  int nota;  public Angajati(String nume, int cod, String telefon, int nota) {  super(nume, cod, telefon);  this.nota = nota;  } } |

Clasa Trainers

|  |
| --- |
| public class Trainers extends Persoana {  String titlu;  String companieExterna;   public Trainers(String nume, int cod, String telefon, String titlu, String companieExterna) {  super(nume, cod, telefon);  this.titlu = titlu;  this.companieExterna = companieExterna;  }  public void notificareNouCurs() {  System.*out*.println("Trainer " + nume + " a fost notificat despre un nou curs.");  } } |

În clasele de mai jos este implementat șablonul Singleton

Clasa mediaAngajati

|  |
| --- |
| public class mediaAngajati { public static Double calculeazaMedieAngajati(ArrayList<Angajati> listaAngajati) {  if (listaAngajati.isEmpty()) {  return null;   }   int sumaNote = 0;  int numarAngajati = listaAngajati.size();   for (Angajati angajat : listaAngajati) {  sumaNote += angajat.nota;  }  return (double) sumaNote / numarAngajati; } } |

Clasa mediaTraining

|  |
| --- |
| public class mediaAngajati { public static Double calculeazaMedieAngajati(ArrayList<Angajati> listaAngajati) {  if (listaAngajati.isEmpty()) {  return null;   }   int sumaNote = 0;  int numarAngajati = listaAngajati.size();   for (Angajati angajat : listaAngajati) {  sumaNote += angajat.nota;  }  return (double) sumaNote / numarAngajati;  }  } |

Clasa Formare

|  |
| --- |
| import java.util.ArrayList; import java.util.List; import java.time.LocalDate;  public class Formare {  int cod;  String titlu;  Angajati angajat;  Trainers trainer;  List<Trainers> observatori = new ArrayList<>();  StareCursuriFormare stare;  LocalDate dataInceput;  LocalDate dataSfarsit;   public Formare(int cod, String titlu, Angajati angajat, Trainers trainer,  StareCursuriFormare stare, LocalDate dataInceput, LocalDate dataSfarsit) {  this.angajat = angajat;  this.trainer = trainer;  this.stare = stare;  this.dataInceput = dataInceput;  this.dataSfarsit = dataSfarsit;  this.titlu = titlu;  this.cod = cod;  }   public void adaugaObservator(Trainers observator) {  observatori.add(observator);  }   public void notificareTrainers() {  for (Trainers observator : observatori) {  observator.notificareNouCurs();  }  }   public StareCursuriFormare getStare() {  return stare;  }   public void asignareAngajatLaCurs(Angajati angajat, Trainers trainer) {  this.angajat = angajat;  this.trainer = trainer;  System.*out*.println(angajat.nume + " a fost asignat la cursul condus de " + trainer.nume);  }  public void asignareTrainerLaCurs(Trainers trainer) {  this.trainer = trainer;  System.*out*.println(trainer.nume + " a fost asignat la cursul condus de " + this.trainer.nume);  } } |

Clasa enum StareCursuriFormare

|  |
| --- |
| public enum StareCursuriFormare {  *DESCHIS*,  *INCHEIAT*,  *IN\_DESFASURARE* } |

Interfata Facade

|  |
| --- |
| import java.util.ArrayList;  public interface Facade {  Double calculeazaMedieAngajati(ArrayList<Angajati> listaAngajati);   double calculeazaMedieTraining(ArrayList<Formare> cursuriFormare);   void adaugaObservator(Formare formare, Trainers observator);   void notificareTrainers(Formare formare);   // int getNotaAngajat(Formare formare); } |

În clasa de mai jos este reprezintat punctul de intrare (entry point) în programul Java. Aceasta este clasa în care începe executarea programului atunci când este rulat.

Clasa Main

|  |
| --- |
| import java.util.ArrayList; import java.time.LocalDate; public class Main {  public static void main(String[] args) {   LocalDate dataInceput = LocalDate.*of*(2023, 1, 1);  LocalDate dataSfarsit = LocalDate.*of*(2023, 12, 31);   Trainers trainer1 = new Trainers("Trainer1", 1000, "090929292", "Titlu1", "Amazon");  Angajati angajat1 = new Angajati("Angajat1", 1000, "09877654", 9 );  Angajati angajat3 = new Angajati("Angajat3", 3000, "982982922", 10);  Formare formare1 = new Formare(1000, "Programare Low Code", angajat1, trainer1, StareCursuriFormare.*DESCHIS*, dataInceput, dataSfarsit);   Trainers trainer2 = new Trainers("Trainer2", 2000, "090299292","Titlu2", "Google");  Angajati angajat2 = new Angajati("Angajat2", 2000, "98298289", 8);  Formare formare2 = new Formare(2000, "Power BI", angajat2, trainer2, StareCursuriFormare.*DESCHIS*, dataInceput, dataSfarsit);   ArrayList<Formare> cursuriFormare = new ArrayList<>();  cursuriFormare.add(formare1);  cursuriFormare.add(formare2);   double mediaFormare = mediaTraining.*calculeazaMedieTrainig*(cursuriFormare);  System.*out*.println("Media notelor angajatilor pentru toate cursurile de formare: " + mediaFormare);    ArrayList<Angajati> listaAngajati = new ArrayList<>();  listaAngajati.add(angajat1);  listaAngajati.add(angajat2);  listaAngajati.add(angajat3);   double MediaAngajati = mediaAngajati.*calculeazaMedieAngajati*(listaAngajati);  System.*out*.println("Media notelor tuturor angajatilor: " + MediaAngajati);   formare1.adaugaObservator(trainer1);  formare1.adaugaObservator(trainer2);   formare1.notificareTrainers();  StareCursuriFormare currentState = formare1.getStare();   System.*out*.println("Note angajați:");  for (Angajati angajat : listaAngajati) {  System.*out*.println(angajat.nume + ": " + angajat.nota);  }   formare1.asignareAngajatLaCurs(angajat1, trainer1);  formare2.asignareAngajatLaCurs(angajat2, trainer2);   Trainers trainer3 = new Trainers("Trainer3", 3000, "090299292","Titlu3", "LowCode");  formare1.asignareTrainerLaCurs(trainer3);   } } |

1. **Aplicarea șabloanelor Singleton, Observer și Facade pentru problema propusă:**

Combinarea Singleton, Facade și Observer Patterns poate fi relevantă în scenarii în care este necesară gestionarea unui sistem complex într-un mod eficient și adaptabil. Această combinație poate fi utilă în situații în care dorim să centralizăm accesul la anumite funcționalități, să observăm schimbările și să coordonăm interacțiunea între diverse componente ale sistemului.

Beneficiile combinării Singleton, Observer, Facade:

* Prin utilizarea șablonului Singleton pentru **mediaTraining** și **mediaAngajati**, se asigură că există o singură instanță centralizată pentru gestionarea acestor medii. Astfel, se evită redundanța și se asigură coerența datelor în întregul sistem.
* Implementarea șablonului Observer pentru **notificareTrainers** permite traineri-lor să fie notificați automat atunci când se produc schimbări în mediile de training. Această notificare automată poate îmbunătăți eficiența comunicării și răspunsul la modificări.
* Facade poate fi folosit pentru a ascunde complexitatea implementării mediilor și a notificărilor de către antrenori. O interfață simplificată oferită de Facade face mai ușor de înțeles și de utilizat aceste funcționalități
* Sablonul Observer permite o flexibilitate în adăugarea sau eliminarea antrenorilor. Această flexibilitate facilitează gestionarea și adaptarea sistemului la modificări în echipa de antrenori.