Programare orientată pe obiecte

Tema - We Are Hiring

Deadline: 21.01.2021
Ora 23:55

Responsabili temă: Daniel Dincă, Cătălin Icleanu, Doina Chiroiu

Facultatea de Automatică şi Calculatoare
Universitatea Politehnica din Bucureşti
Anul universitar 2020 - 2021
Seria CC

1 Obiective

În urma realizării acestei teme, studentul va fi capabil:

- să aplice corect principiile programării orientate pe obiecte studiate în cadrul cursului;
- să construiască o ierarhie de clase, pornind de la un scenariu propus;
- să utilizeze un design orientat-obiect;
- să trateze excepțiile ce pot interveni în timpul rulării unei aplicații;
- să transpună o problemă din viața reală într-o aplicație ce se poate realiza folosind noțiunile dobândite în cadrul cursului.

Scopul acestei teme este de a implementa o simulare a unei aplicații de găsit locuri de muncă, aplicând noțiunile studiate în cadrul acestui curs.

2 Cerința 1 – Arhitectura aplicației (100 de puncte)

2.1 Application

Clasa care va reprezenta aplicația noastră. Aceasta va conține o listă de companii, o listă de utilizatori. Pe lângă aceste date, aplicația va permite efectuarea următoarelor operații:

- Determinarea companiilor care au fost înscrise în aplicație;

```
public ArrayList<Company> getCompanies();
```

- Determinarea unei anumite companii în funcție de numele furnizat;

```
public Company getCompany(String name)
```

- Adăugarea unei companii;

```
public void add(Company company);
```

- Adăugarea unui utilizator;

```
public void add(User user);
```

- Ștergerea unei companii – va întoarce `false` dacă compania nu există;

```
public boolean remove(Company company);
```

- Ștergerea unui utilizator – va întoarce `false` dacă utilizatorul nu există;

```
public boolean remove(User user);
```

- Determinarea joburile disponibile de la companiile pe care le preferă utilizatorul.

```
public ArrayList<Job> getJobs(List<String> companies);
```

2.2 Consumer

Clasa aceasta este o clasă abstractă care reprezinta utilizatorul uman al aplicației noastre. Un obiect de tip `Consumer` este caracterizat de un `Resume` și o listă de cunoscuți (rețeaua sociala a sa – listă cu obiecte de tip `Consumer`). Operațiile puse la dispoziție de această clasă sunt următoarele:

- Editarea informațiilor personale ale utilizatorului (urmează să fie detaliate în continuare). Pentru aceste tipuri de operații de editare, vă puteți alege cum să realizați implementarea (câte și ce fel de metode doriți să folosiți pentru acestea).
- Adăugarea unor studii;

```
public void add(Education education);
```

- Adăugarea unei experiențe profesionale;

```
public void add(Experience experience);
```

- Adăugarea unui nou cunoscut;

```
public void add(Consumer consumer);
```

Determinarea gradului de prietenie cu un alt utilizator – se realizează o parcurgere în lățime în rețeaua socială a utilizatorului;

```
public int getDegreeInFriendship(Consumer consumer);
```

- Eliminarea unei persoane din rețeaua socială;

```
public void remove(Consumer consumer);
```

- Determinarea anului absolvirii;

```
public Integer getGraduationYear();
```

- Determinarea mediei studiilor absolvite;

```
public Double meanGPA();
```

2.3 Resume

`Resume` este o clasă internă clasei `Consumer` și reține următoarele informații: un obiect de tipul `Information`, care conține datele personale ale utilizatorului, o colecție sortată care descrie educația utilizatorului, o colecție sortată care descrie experiența profesională.

2.4 Information

`Information` este clasa care va reține informațiile personale ale unui utilizator, acestea sunt: nume, prenume, email, telefon, data de naștere, sex și lista limbilor cunoscute (pentru fiecare limbă se va specifica unul dintre nivelurile **Beginner**, **Advanced**, **Experienced**). Această clasă va trebui să fie implementată respectând principiul încapsulării.

2.5 Education

`Education` este clasa care va reține informațiile referitoare la un ciclu de educație al unui utilizator. Clasa va conține următoarele informații: data de început și de sfârșit, numele instituției, nivelul de educație (liceu, licență, master, doctorat), media de finalizare. Clasa `Education` va implementa interfața `Comparable` și va folosi drept criteriu de comparație unul bazat pe data de absolvire (descrescător). În cazul în care data de sfârșit este egală, atunci se va face comparația în funcție de media de finalizare (descrescător). Dacă educația este încă în curs, atunci data de final și media de finalizare vor fi setate la **`null`**, iar comparația poate să fie făcută după data de început (crescător). Se va verifica dacă data de start și de început sunt corecte din punct de vedere cronologic la crearea unui obiect de tipul `Education`. Dacă datele sunt invalide, se va arunca o excepție de tipul `InvalidDatesException`.

2.6 Experience

`Experience` va reține informații referitoare la o perioadă de muncă anterioară sau prezentă. Clasa va conține următoarele informații: data de start și de sfârșit, poziția și compania. Clasa `Experience` va implementa interfața `Comparable` și va folosi drept criteriu de comparație tot unul bazat pe data de sfârșit (descrescător). Dacă nu se poate face diferențierea între două obiecte `Experience` pe baza acestei informații, atunci se vor compara după numele companiei (crescător). Dacă perioada de angajare este cea prezentă atunci data de final va fi setata la **`null`**. Se va verifica dacă data de start și de început sunt corecte din punct de vedere cronologic la crearea unui obiect de tipul `Experience`. Dacă datele sunt invalide, se va arunca o excepție de tipul `InvalidDatesException`.

2.7 User

Clasa `User` moștenește clasa `Consumer` și reprezintă utilizatorul care își caută de muncă.

În această clasă, o să existe o metodă care se va apela atunci când un utilizator se va angaja. Această metoda va întoarce un nou obiect de tip `Employee` care va conține toate datele pe care le avea obiectul `User`. Metoda va avea următorul antet:

```
public Employee convert();
```

Pe lângă această metodă, va mai exista și o metodă care determină scorul unui utilizator pe baza următoarei formule:

$$număr\_ani\_experiență * 1.5 + medie\_academică$$

Numărul de ani o să fie determinat aproximând prin adaos (pentru 3 luni vom considera că utilizatorul are 1 an experiență). Această metodă va avea următorul antet:

```
public Double getTotalScore();
```

2.8 Employee

Clasa `Employee` moștenește clasa `Consumer` și reprezintă un utilizator care este deja angajat într-o companie. Trebuie să aibă un câmp pentru numele companiei la care lucrează și un câmp dedicat pentru salariu. Aceste două câmpuri se vor inițializa

2.9 Recruiter

Această clasa va moșteni clasa `Employee` și va conține un câmp ce reprezintă rating-ul acestuia, valoare ce va ajuta atunci când va evalua un `User` dacă este potrivit pentru `Job`. La început, acest atribut pentru rating va avea valoarea 5 și o să fie actualizat constant cu valoarea +0.1 pentru fiecare utilizator analizat.

În această clasă va exista o metodă cu antetul:

```
public int evaluate(Job job, User user);
```

După ce va evalua un `User`, un `Recruiter` va trimite un `Request` (o cerere de angajare) către cel care manageriază compania la care este angajat.

O cerere de angajare este modelată prin intermediul următoarei clase, unde `key` – job-ul, `value1` – utilizatorul care trebuie să fie angajat pentru acel job, `value2` – recruiter-ul care s-a ocupat de evaluarea utilizatorului, `score` – scorul atribuit de recruiter:

```
public class Request<K, V> {
    private K key;
    private V value1, value2;
    private Double score;

    public Request(K key, V value1, V value2, Double score) {
        this.key = key;
        this.value1 = value1;
        this.value2 = value2;
        this.score = score;
    }

    public K getKey() {
        return key;
    }

    public V getValue1() {
        return value1;
    }

    public V getValue2() {
        return value2;
    }

    public Double getScore() {
        return score;
    }

    public String toString() {
        return "Key: " + key + " ; Value1: " + value1 + " ; Value2: " + value2 +
        ↪  " ; Score: " + score;
```

```
    }
}
```

2.10 Manager

Managerul extinde clasa `Employee`. Clasa `Manager` va conține o colecție de cereri de angajare. În această clasa va exista următoarea metodă:

```
public void process(Job job);
```

Când o să fie apelată această metodă, se va itera prin colecția cu cererile de angajare, se vor alege cele corespunzătoarea jobului indicat prin parametru și se vor procesa. Pentru fiecare job se cunoaște numărul de poziții disponibile (`noPositions`). Astfel, managerul va selecta primii `noPositions` candidați, în ordinea descrescătoare a scorului obținut, și îi va angaja în companie. Atunci când un candidat este angajat, trebuie să se realizeze o verificare (dacă a fost sau nu angajat între timp la o altă companie). Acest lucru se poate determina prin verificarea listei de utilizatori din clasa `Application`, deoarece atunci când un candidat este angajat el este șters din lista respectivă, este convertit într-un `Employee` și adăugat în departamentul corespunzător jobului. După apelul acestei metode, nu vor mai exista cereri pentru jobul indicat în colecția cu obiecte de tip `Request`. De asemenea, un job este închis după ce au fost procesate toate cererile de către manager.

2.11 Job

Clasa `Job` va conține numele job-ului, numele companiei, un flag care indică dacă jobul mai este deschis sau nu, trei obiecte de tip `Constraint` (constrângere pentru anul absolvirii, constrângere pentru numărul de ani de expereință și constrângere pentru media academică), lista de candidați, numărul de candidați de care are nevoie compania pentru respectivul job și salariul pe care îl vor primi cei care vor fi angajați.

Metodele obligatorii puse la dispoziție de această clasă sunt următoarele:

- O metodă prin care un utilizator aplică la un job în companie. Se va selecta un `Recruiter` din lista de recruiteri din compania mama, iar pe baza experienței acestuia se va stabili scorul (se va apela metoda `evaluate`). Pentru a putea determina recruiter-ul, o să se folosească o parcurgere în lățime pe rețeaua socială pentru a determina care este cel mai îndepărtat. Dacă nu se poate alege în funcție de acest criteriu, deoarece rămân mai mulți candidați, atunci se va selecta dintre aceștia cel cu rating-ul cel mai mare.

  ```
  public void apply(User user);
  ```

  Observație

  Un utilizator poate aplica doar pentru job-urile care nu au fost deja închise.

- O metodă care iterează prin lista de constrângeri și verifică dacă sunt îndeplinite pentru aplicantul primit ca parametru.

  ```
  public boolean meetsRequirments(User user);
  ```

2.12 Constraint

Fiecare `Job` va avea câte 3 constrângeri: una pentru anul absolvirii acceptat, una pentru numărul de ani de experiență, una pentru media academică. Un obiect de tip `Constraint` va conține o limită inferioară și una superioară. De exemplu, dacă obiectul `Constraint` folosit pentru numărul de ani de experiență are limita inferioară 5 și limita superioară 10, atunci toți candidații care au mai puțin de 5 ani experiență sau mai mult de 10 ani vor fi respinși.

2.13 Company

Aceasta este clasa care modelează o companie în cadrul aplicației. O companie este caracterizată prin nume, manager, departamente și recruiter-i (angajați care fac parte și din alte departamente dar sunt și recruiter-i). Pe lângă acestea, compania va permite și efectuarea următoarelor operații:

- Adăugarea unui nou departament în companie;

  ```
  public void add(Department department);
  ```

- Adăugarea unui nou recruiter;

```
public void add(Recruiter recruiter);
```

- Adăugarea unui angajat într-un departament;

```
public void add(Employee employee, Department department);
```

- Eliminarea unui angajat din companie;

```
public void remove(Employee employee);
```

- Eliminarea unui departament din companie și a tuturor angajaților care fac parte din departamentul respectiv;

```
public void remove(Department department);
```

- Eliminarea unui recruiter;

```
public void remove(Recruiter recruiter);
```

- Mutarea unui departament în alt departament și transferarea tuturor angajaților;

```
public void move(Department source, Department destination);
```

- Mutarea unui angajat dintr-un departament în alt departament;

```
public void move(Employee employee, Department newDepartment);
```

- Verificarea existenței unui departament în companie;

```
public boolean contains(Department department);
```

- Verificare existenței unui angajat în companie;

```
public boolean contains(Employee employee);
```

- Verificarea existenței unui recruiter în companie;

```
public boolean contains(Recruiter recruiter);
```

- Determinarea recruiter-ului potrivit pentru un utilizator;

```
public Recruiter getRecruiter(User user);
```

- Determinarea job-urilor disponibile dintr-o companie (cele care nu au fost deja închise);

```
public ArrayList<Job> getJobs();
```

2.14 Department

Un departament este reprezentat în aplicația noastră printr-o clasa abstractă care va conține lista angajaților din acel departament, o lista cu job-urile disponibile din acel departament. Pe lângă lista angajaților, un departament va conține următoarele metode:

- Metodă abstractă care va returna bugetul total de salarii, după aplicarea taxelor;

```
public abstract double getTotalSalaryBudget();
```

- Metodă care întoarce toate joburile deschise din departament;

```
public ArrayList<Job> getJobs();
```

- Metodă care adăuga un angajat în departament;

```
public void add(Employee employee);
```

- Metodă care șterge un angajat din departament;

```
public void remove(Employee employee)
```

- Metodă care adaugă un job în departament;

```
public void add(Job job);
```

- Metodă care întoarce angajații dintr-un departament;

```
public ArrayList<Employee> getEmployees();
```

2.15 IT

Această clasă va extinde clasa `Department`. Toți cei din departamentul IT sunt scutiți de taxe.

2.16 Management

Clasa `Management` extinde clasa `Department`. Toți cei din departamentul de `Management` vor avea un impozit egal cu 16%.

2.17 Marketing

Această clasă va extinde clasa `Department`. Cei din departamentul de `Marketing` care au un salariu mai mare de 5000 de lei vor avea un impozit egal cu 10%, iar toți cei care au un salariu mai mic de 3000 de lei vor fi scutiți de taxe.

2.18 Finance

Va extinde clasa `Department`. Toți cei din departamentul de `Finance` care au vechime mai puțin de un an vor avea un impozit egal cu 10%, iar pentru toți ceilalți impozitul va fi de 16%.

3 Cerința 2 – Testarea aplicației (30 de puncte)

Pentru a putea realiza o testare a aplicației, trebuie sa implementați o clasă `Test` ce conține o metoda `main` care parsează o serie de fișiere de intrare și realizează testarea aplicație pe baza unor scenarii care vor fi oferite.

4 Cerința 3 – Șabloane de proiectare (70 de puncte)

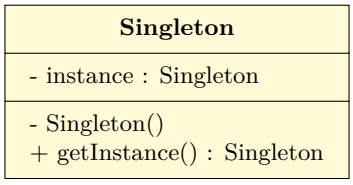
Un șablon de proiectare descrie o problemă care se întâlnește în mod repetat în proiectarea programelor și soluția generală pentru problema respectivă, astfel încât să poată fi utilizată oricând, dar nu în același mod de fiecare dată. Soluția este exprimată folosind clase și obiecte. Atât descrierea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

4.1 Identificarea șabloanelor de proiectare

4.1.1 Singleton pattern

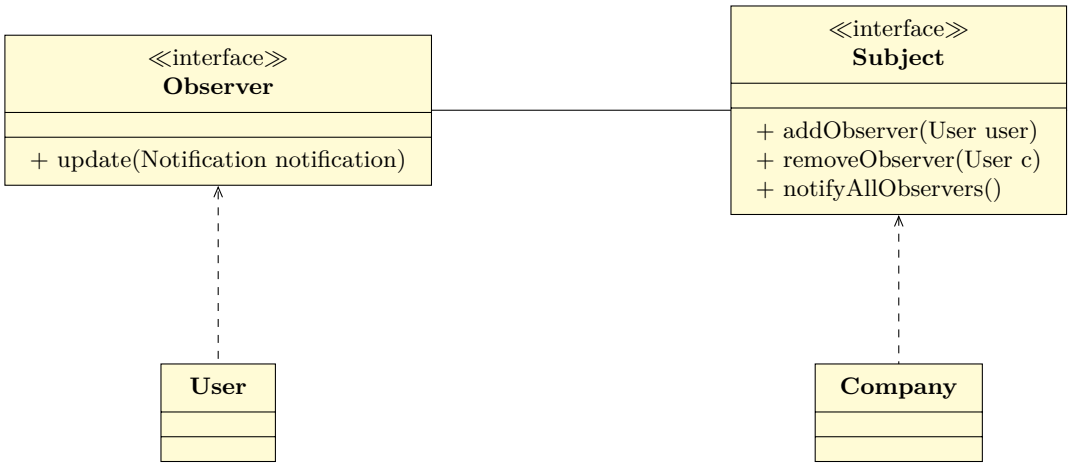
Pentru a ne asigura că fiecare utilizator are acces la informațiile aceluiași obiect de tip `Application`, avem nevoie să menținem o **unică** referință către un obiect de tip **Application**, în mai multe clase. Astfel, vom utiliza șablonul **Singleton** pentru a restricționa numărul de instanțieri ale clasei `Application` la un singur obiect. Pentru a nu consuma inutil resursele sistemului, în implementarea voastră, veți folosi **instanțierea întârziată**.



4.1.2 Observer pattern

Design pattern-ul **Observer** definește o relație de dependență unul la mai mulți între obiecte astfel încât un obiect își schimbă starea toți dependenții lui sunt notificați și actualizați automat. Acest pattern implică existența unui obiect denumit subiect care are asociată o listă de obiecte dependente, numite observatori, pe care le apelează automat de fiecare dată când se întâmplă o acțiune. În cadrul acestei aplicații vom folosi

pattern-ul **Observer** pentru a notifica candidații (instanțele clasei `User`) de fiecare dată când o companie pe care o urmăresc adaugă un job nou, când este închis un job sau când au fost respinși de la un job.

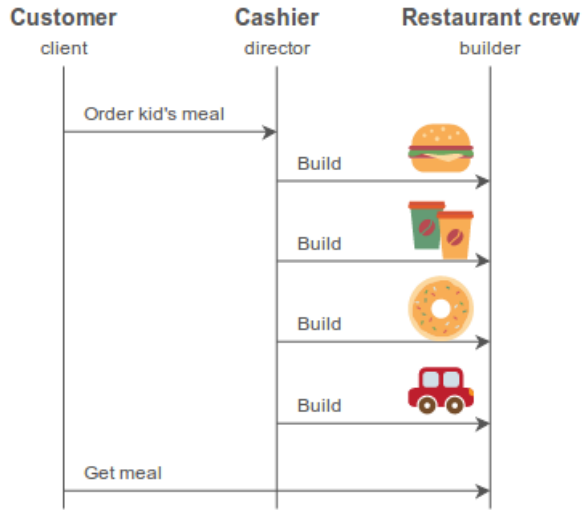


Observații

1. Un utilizator este adăugat drept observator pentru o companie în momentul în care acesta aplică la un job din compania respectivă.
2. Un utilizator este șters din colecția de observatori ai unei companii atunci când el este angajat în cadrul unei companii. Mai exact, dacă utilizatorul `UserA` este adăugat drept observator în `CompanyA`, `CompanyB` și `CompanyC`, atunci când o să fie angajat în `CompanyB` el o să dispară din colecțiile de observatori disponibile în companiile `CompanyA`, `CompanyB` și `CompanyC`.

4.1.3 Builder pattern

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. În cadrul acestei aplicații, pattern-ul este folosit pentru a instanția un `Resume`. Pentru a înțelege cum ar trebui folosit acest pattern, puteți urmări exemplul de mai jos.

⚠IMPORTANT!

Trebuie să vă asigurați că veți putea instanția, folosind mecanismul implementat pe baza pattern-ului **Builder**, orice tip de `Resume` (cu oricâte categorii de studii, cu oricâte instanțe pentru experiența profesională).

Dacă se încearcă realizarea unui `Resume` fără `Information` sau fără cel puțin o instanță în colecția cu studiile, se va arunca excepția `ResumeIncompleteException`.

Cod Java

```java
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+" "+this.phone+" "+
            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);
        }
    }

```

```
60      public static void main(String[] args) {
61          User user1 = new User.UserBuilder("Lokesh", "Gupta")
62                  .age(30)
63                  .phone("1234567")
64                  .address("Fake address 1234")
65                  .build();
66          User user2 = new User.UserBuilder("Jack", "Reacher")
67                  .age(40)
68                  .phone("5655")
69                  //no address
70                  .build();
71      }
72  }
```

4.2 Factory pattern

Factory pattern este unul dintre cele mai utilizate tipuri de șabloane de proiectare din Java. Acesta face parte din categoria celor *creational*, oferind una dintre cele mai bune modalități de instanțiere pentru obiecte ce au tipuri care provin din aceeași ierarhie de clase.

Pentru fiecare departament se va crea un obiect corespunzător de tip **IT**, **Management**, **Marketing**, **Finance**, în funcție de tipul departamentului, folosind **Factory Pattern**.

5 Cerința 4 – Interfața grafică (100 de puncte)

Va trebuie să realizați o interfață grafică folosind componenta **Swing**. Această interfața va trebui să cuprindă următoarele pagini:

5.1 Admin Page

Să se realizeze o pagină de administrator în care să se poată vedea lista cu utilizatori, companii, să se poată selecta o companie, afișându-se departamentele acesteia, angajații din fiecare departament, fiecare job din departament și să se poată calcula și afișa salariile pentru fiecare departament în parte.

5.2 Manager Page

Să se realizeze o pagină destinată managerului, în care se vor afișa cererile de angajare ale utilizatorilor. Acesta va avea posibilitatea să accepte sau să respingă o astfel de cerere. După ce va fi acceptată o cerere, se poate verifica în pagina cu angajații unei companii selectate că s-a adăugat un nou angajat.

5.3 Profile Page

Să se realizeze o pagină de profil pentru utilizatori, în care să se afișeze detaliile unui utilizator căutat după nume. În această pagină ar trebui să fie vizibile detaliile incluse în Resume ordonate după criteriile specificate.

Observație

Sunteți liberi să adaugați orice funcționalități suplimentate doriți sau le considerați utile din punct de vedere al interfeței grafice.

Se va acorda bonus pentru o interfață grafică intuitivă și complexă, frumos realizată, care pune la dispoziție toate operațiile implementate de arhitectură.

6 Bonusuri – (50 de puncte)

6.1 Interfață grafică

În ceea ce privește interfața grafică, vi se propun următoarele bonusuri:

- Un sistem de autentificare ce permite afișarea unui conținut personalizat în funcție de tipul utilizatorului (**Manager**, **Recruiter**, **User**, **Employee**).
- Pagina în care un utilizator își poate accesa lista de notificări primite.
- O pagină în care să fie afișate toate informațiile ce țin de o companie, dar în care să se poată face și modificări asupra acestora (să se mute angajați dintr-un departament în altul, să se adauge noi angajați, să se adauge recruiteri noi, să fie afișat un clasament al recruiterilor în funcție de rating etc.).

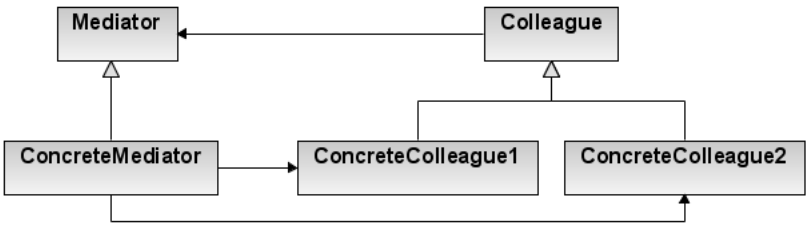
6.2 Mediator pattern

În general, într-o aplicație complexă, logica este distribuită în diferite clase. Astfel, pe măsură ce numărul acestora crește, comunicarea devine tot mai complexă, ajungându-se la o structură încâlcită de clase, având drept consecință directă un cod greu de citit și de modificat. De asemenea, capacitatea de reutilizare a codului este considerabil diminuată, deoarece obiectele devin, din punct de vedere comportamental, puternic legate de celelalte.

Design pattern-ul **Mediator** este conceput pentru a rezolvă problemele descrise anterior, promovând un cuplaj redus între componente. Acest deziderat se obține prin introducerea unei noi entități, numită mediator, aceasta fiind singura care posedă cunoștințe despre toate celelalte componente, în timp ce acestea nu cunosc decât mediatorul. Folosind acest mecanism, componentele nu se mai referă explicit, iar interacțiunile dintre componente pot fi modelate independent.

În cazul interfeței grafice, atunci când se realizează o schimbare într-o parte dintr-un **JDialog**, **JButton**, **JPanel**, **JFrame** sau o altă componentă grafică, de cele mai multe ori, trebuie să fie actualizate și alte componente ale interfeței. Exista multe moduri de a actualiza componentele interfeței grafice, dar abordarea în care acestea sunt strâns cuplate, fiecare componentă deținând cunoștințe despre toate celelalte, nu este recomandată. O modalitate mult mai bună de a obține efectul dorit, promovând un cuplaj redus între componentele grafice, este aplicarea șablonului de proiectare **Mediator**. De exemplu, pattern-urile **Command** și **Mediator** pot fi ușor îmbinate pentru a executa comenzile date de diferitele butoane de pe interfața grafică. Constructorii butoanelor înregistrează **Mediatorul** ca și **ActionListener** al event-urilor generate la apăsarea oricărui buton. Aceste evenimente pot fi captate și tratate în metoda ***actionPerformed()***. Interfața **Command**, care oferă metoda ***execute()***, va fi implementată de fiecare dintre butoane, iar, la execuție, comenzile specifice vor fi redirectate către **Mediator**, pentru a realiza decuplarea dorita. Astfel, în metoda ***actionPerformed()*** se obține comanda captată din eveniment și se execută.

Pornind de la aceste informații și, eventual, de la exemplul oferit, integrați șablonul de proiectare **Mediator** în implementarea claselor ce compun interfața grafică.



6.3 Suplimentar

Observație

Pe lângă cele specificate în această secțiune, mai puteți alege să alegeți alte design pattern-uri pe care să le integrați în dezvoltarea acestei aplicații. În această situație, veți detalia în **README** motivul pentru care ați ales aceste tipuri și cum le-ați folosit.
Punctajul o să fie acordat în funcție de dificultatea integrării / implementării șabloanelor propuse!

7 Punctaj

Cerința	Punctaj
Cerința 1 (Implementarea integrală a claselor propuse)	**100 de puncte**
Cerința 2 (Implementarea scenariului de testare propus)	**30 de puncte**
Cerința 3 (Integrarea design pattern-urilor propuse în implementare)	**70 de puncte**
Cerința 4 (Realizarea paginilor propuse pentru interfața grafică)	**100 de puncte**
Bonusuri (Implementarea unor bonusuri din cele propuse)	**50 de puncte**

Atenție!

Tema va valora 3 puncte din nota finală a disciplinei POO!

Soluțiile de genul folosirii de variabile interne pentru a stoca informații ce țin de tipul obiectului vor fi depunctate.

Tipizați orice colecție folosită în cadrul implementării.

Respectați specificațiile detaliate în enunțul temei și folosiți indicațiile menționate.

Pe lângă clasele, atributele și metodele specificate în enunț, puteți adăuga altele dacă acest lucru îl considerați util și potrivit, în raport cu principiile programării orientate pe obiecte.

Atenție!

Tema este individuală! Toate soluțiile trimise vor fi verificate, folosind o unealtă pentru detectarea plagiatului.

Tema se va prezenta la ultimul laborator din semestru.

Tema se va încărca pe site-ul de cursuri până la termenul specificat în pagina de titlu. Se va trimite o arhivă ***.zip*** ce va avea un nume de forma **grupa_Nume_Prenume.zip** (ex. ***326CC_Popescu_Andreea.zip***) și care va conține următoarele:

- un folder **SURSE** ce conține doar sursele Java și fisierele de test;
- un folder **PROIECT** ce conține proiectul ***NetBeans*** sau ***Eclipse***;
- un fișier **README.pdf** în care veți specifica numele, grupa, gradul de dificultate al temei, timpul alocat rezolvării și veți detalia modul de implementare, specificând și alte obervații, dacă este cazul. Lipsa acestui fișier sau nerespectarea formatului impus pentru arhivă duc la o depunctare de ***10 puncte***.