Programare orientată pe obiecte – Seria CC
Temă – Catalog electronic

Mihai Nan – mihai.nan@upb.ro

Anul universitar 2022 – 2023
Deadline: 10.01.2023

1 Arhitectura aplicației – (140 de puncte)

1.1 Clasa `Catalog`

Implementați o clasă `Catalog` care conține o listă cu obiecte de tip `Course`.

```
public class Catalog { /* TODO Implementarea pentru clasa Catalog */ }
public abstract class Course { }
```

Această clasă trebuie să conțină următoarele metode:

```
// Adaugă un curs în catalog
public void addCourse(Course course) { /* TODO */ }
// Șterge un curs din catalog
public void removeCourse(Course course) { /* TODO */ }
```

Observație

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.
Astfel, pentru implementarea acestei clase vom utiliza șablonul de proiectare **Singleton**.

1.2 Clasele `User`, `Student`, `Parent`, `Assistant`, `Teacher`

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

1.2.1 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;
    }
}
```

1.2.2 Clasa Student

Clasa Student va conține două câmpuri private de tip Parent pentru a reține referințe către mama și tatăl studentului.

Această clasă trebuie să conțină următoarele metode:

```
public void setMother(Parent mother) { /* TODO */ }
public void setFather(Parent father) { /* TODO */ }
```

Pentru restul claselor, momentan puteți adăuga doar constructori potriviți.

Observație

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**.

1.3 Clasa Grade

Această clasă implementează interfețele Comparable și Cloneable.

Clasa trebuie să conține câmpurile:

```
private Double partialScore, examScore;
private Student student;
private String course; // Numele cursului
```

Pentru aceste câmpuri trebuie să implementați metode getter și setter.

Pe lângă metodele din cele două interfețe, getteri și setteri, clasa mai trebuie să conțină metodă pentru calculul notei :

```
public Double getTotal() { /* TODO */ }
```

Va trebui să vă asigurați că două obiecte de tip Grade vor putea să fie comparate (în funcție de punctajul total).

1.4 Clasa Group

Această clasă va modela **prin moștenire** o colecție ordonată ce poate conține doar obiecte de tip Student.

Fiecare grupă va avea asociat un obiect de tip Assistant și un ID de tip șir de caractere.

Vom considera că două grupe sunt egale dacă au același ID.

Implementarea clasei trebuie să respecte principiul încapsulării.

Pentru această clasă trebuie să existe implementare pentru următorii constructori:

```
public Group(String ID, Assistant assistant, Comparator<Student> comp) { }
public Group(String ID, Assistant assistant) { }
```

1.5 Clasa Course

Clasa Course este o clasă abstractă ce conține: un nume (de tipul String), un profesor titular, o mulțime de asistenți (colecție fără duplicate), o colecție ordonată cu obiecte de tipul Grade, un dicționar ce conține grupele (cheia este de ID-ul grupei, iar valoarea este grupa) și un număr întreg ce reprezintă numărul de puncte credit.

Această clasă va conține pe lângă metodele setter și getter următoarele metode:

```
// Setează asistentul în grupa cu ID-ul indicat
// Dacă nu există deja, adăuga asistentul și în mulțimea asistenților
public void addAssistant(String ID, Assistant assistant)
// Adaugă studentul în grupa cu ID-ul indicat
public void addStudent(String ID, Student student)
// Adaugă grupa
public void addGroup(Group group)
// Instanțiază o grupă și o adaugă
public void addGroup(String ID, Assistant assistant)
// Instanțiază o grupă și o adaugă
public void addGroup(String ID, Assistant assist, Comparator<Student> comp)
// Returnează nota unui student sau null
public Grade getGrade(Student student)
// Adaugă o notă
public void addGrade(Grade grade)
// Returnează o listă cu toți studenții
public ArrayList<Student> getAllStudents()
// Returnează un dicționar cu situația studenților
public HashMap<Student, Grade> gettAllStudentGrades()
// Metodă ce o să fie implementată pentru a determina studenții promovați
public abstract ArrayList<Student> getGraduatedStudents();
```

1.5.1 Clasa PartialCourse

Clasă care moștenește clasa Course și îi implementează metoda abstractă. Pentru acest tip de curs, considerăm promovați toți studenții pentru care $total \geq 5$.

1.5.2 Clasa FullCourse

Clasă care moștenește clasa Course și îi implementează metoda abstractă. Pentru acest tip de curs, considerăm promovați toți studenții pentru care $partial \geq 3$ și $exam \geq 2$.

Observație

Pentru a putea seta câmpurile unui obiect de tip Course, veți folosi șablonul de proiectare **Builder**.

Important

În clasa Course veți defini o clasă internă abstractă CourseBuilder pe care o veți extinde în clasele pentru cele două tipuri de cursuri (clasele interne FullCourseBuilder și PartialCourseBuilder).
Atenție la parametrizarea claselor!

1.6 Șablonul de proiectare Observer

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`.

1.7 Șablonul de proiectare Strategy

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, ținând cont de strategia aleasă
 ↪  de profesor pentru curs
public Student getBestStudent();
```

1.8 Șablonul de proiectare Visitor

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);
}
```

Trebuie să stabiliți care sunt clasele care implementează interfața `Element`. 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.

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

Clasa `Tuple` este o clasă internă privată a clasei `ScoreVisitor` ce trebuie să fie implementată generic.

1.9 Șablonul de proiectare Memento

Vrem ca pentru un curs să oferim posibilitatea de a face un back-up al notelor, iar pentru acest lucru vom folosi șablonul de proiectare **Memento**.

În clasa `Grade` vom implementa metoda:

```
public Object clone() {
    // TODO
}
```

Important

Aveți grijă să clonați corespunzător cele două referințe de tip `Double` folosite pentru a reține cele două note.

În clasa `Course`, vom implementa clasa internă privată `Snapshot` pe care o folosim pentru a stoca notele. De asemenea, vom adăuga un câmp privat de tipul `Snapshot` în clasa `Course` și vom implementa următoarele două metode:

```
public void makeBackup() { // TODO1 }
public void undo() { // TODO2 }
```

1.10 Testarea aplicației

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

Important

În vederea acordării punctajului parțial pentru primele 2 cerințe, trebuie să vă includeți în `main` o testare a **tuturor** funcționalităților implementate.

2 Interfața grafică – (60 de puncte)

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

2.1 Student Page

Să se realizeze o pagină în care să se poată vedea lista cursurilor la care este înscris un student. Pagina trebuie să ofere posibilitate de selectare a unui curs pentru care să fie afișate informații suplimentare: profesorul titlular al cursului, lista de asistenți de la acel curs, asistentul pe care îl are asociat studentul, notele pe care le-a primit studentul la acel curs.

2.2 Teacher/Assistant Page

Să se realizeze o pagină destinată profesorului/asistentului, în care se vor afișa toate cursurile la care predau aceștia și lista cu notele pe care aceștia le au de validat. De asemenea, va trebui să adăugați un buton care să valideze notele studentului/asistentului (apelarea metodelor din clasa `ScoreVisitor`).

2.3 Parent Page

Să se realizeze o pagină de profil pentru părinți, în care să se afișeze detaliile notificările pe care aceștia le-au primit.

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.

3 Bonusuri – (50 de puncte)

3.1 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 (**Student**, **Parent**, **Teacher**, **Assistant**).
- O pagină în care să fie afișate toate informațiile ce țin de un curs, dar în care să se poată aplica modificări asupra acestora (să se mute adauge un nou asistent, să se aduge un student, să se adauge o notă etc.) și să se afișeze diverse statistici legate de curs.
- O pagină în care să se poate introduce cursuri (pentru fiecare curs să fie specificate informațiile necesare – eventual pot fi preluate dintr-un fișier de tip **JSON**).

3.2 Mediator pattern

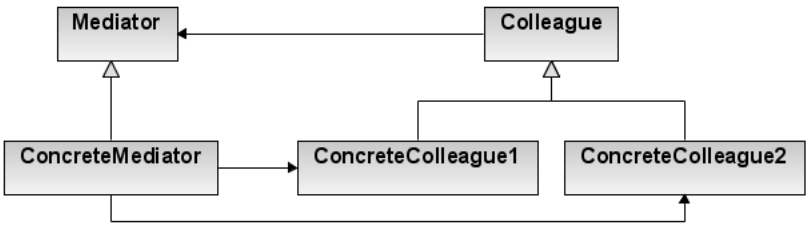
În general, într-o aplicație complexă, logica este distribuită în diferite clase. Astfel, pe măsură ce numărul acestora crește, comunicarea devine tot mai complexă, ajungându-se la o structură

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.



3.3 Suplimentar

Observație

Pe lângă cele specificate în această secțiune, mai puteți să alegeți alte șabloane de proiectare 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!

4 Punctaj

Cerința	Punctaj
Cerința 1 (Implementarea integrală a claselor propuse și **testarea lor**)	**60 de puncte**
Cerința 2 (Integrarea design pattern-urilor propuse în implementare)	**60 de puncte**
Cerința 3 (Implementarea scenariului de testare propus)	**20 de puncte**
Cerința 4 (Realizarea paginilor propuse pentru interfața grafică)	**60 de puncte**
Bonusuri (Implementarea bonusuri din cele propuse)	**50 de puncte**

Atenție

Tema va valora 2 puncte din nota finală a disciplinei POO sau 2.5 puncte cu bonus!
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 în ultima săptămână 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 în mediul ales de voi (de exemplu ***NetBeans***, ***Eclipse***, ***Intellij IDEA***);
- 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***.