**Ministerul Educației și Cercetării**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**Departamentul Ingineria Software și Automatică**

**RAPORT**

**Proiect de An la Tehnici și Mecanisme de Proiectare Software**

**Tema: „Task Manager”**

|  |  |
| --- | --- |
| **Student:** | **gr. TI-204,**  **Naghiev Raghim** |
| **Coordonator:** | **Asistent universitar, Gaidău Mihai** |

**Chișinău, 2023**

**Cuprins:**

[Introducere în temă 3](#_Toc137031734)

[1.Analiza domeniului de studiu 4](#_Toc137031735)

[1.1 Importanţa temei 5](#_Toc137031736)

[1.2 Sisteme similare: 6](#_Toc137031737)

[1.3 Scopul, obiectivele și cerintețele sistemului 7](#_Toc137031738)

[1.4 Şabloanele de proiectare folosite 9](#_Toc137031739)

[2. Structura UML a sistemului 10](#_Toc137031740)

[2.1 Diagrama pentru Sigleton Pattern 10](#_Toc137031741)

[2.2 Diagrama pentru Decorator Pattern 11](#_Toc137031742)

[2.3 Diagrama pentru Adapter Pattern 13](#_Toc137031743)

[2.4 Diagrama pentru Observer Pattern 14](#_Toc137031744)

[2.5 Diagrama pentru Mediator Pattern 15](#_Toc137031745)

[2.6 Diagrama pentru Command Pattern 16](#_Toc137031746)

[3. Documentarea produsului realizat 18](#_Toc137031747)

[3.1 Implementarea Sigleton 18](#_Toc137031748)

[3.2 Implementarea Decorator 19](#_Toc137031749)

[3.3 Implementarea Adapter 20](#_Toc137031750)

[3.4 Implementarea Observer 21](#_Toc137031751)

[3.5 Implementarea Mediator 23](#_Toc137031752)

[3.6 Implementarea Command 24](#_Toc137031753)

[Concluzie: 26](#_Toc137031754)

# **Introducere în temă**

Modelele de design sunt soluții dovedite la problemele comune pe care le întâmpină dezvoltatorii în timpul proiectării și implementării sistemelor software. Acestea servesc ca principii directoare și șabloane reutilizabile, permițându-ne să construim cod robust, care poate fi întreținut și flexibil

.Într-un proiect, explorarea diverselor modele de design și a aplicațiilor lor practice evidențiază modul în care acestea îmbunătățesc arhitectura generală și calitatea sistemelor software. Modelele de design oferă un limbaj comun și un set de bune practici pentru ca dezvoltatorii să comunice și să colaboreze eficient. Ele încapsulează ani de cunoștințe colective și experiență din partea comunității de dezvoltare software, oferind perspective valoroase în rezolvarea provocărilor complexe de proiectare.

Prin utilizarea modelelor de design, dezvoltatorii pot evita să reinventeze roata și pot beneficia de soluții testate și validate în domeniul dezvoltării software. Aceste modele abordează diferite aspecte ale proiectării, cum ar fi organizarea componentelor sistemului, relațiile dintre acestea, modul în care informația este transmisă și prelucrată, gestionarea fluxului de date și multe altele.

Un exemplu de model de design este Modelul MVC (Model-View-Controller), care separă logica de prezentare a datelor (View) de logica de manipulare a datelor (Controller) și logica de stocare a datelor (Model). Prin utilizarea acestui model, dezvoltatorii pot crea sisteme mai flexibile, ușor de întreținut și extensibile.Un alt exemplu este Modelul Observer, care definește o relație de tip "unul la mulți" între obiecte, astfel încât modificările făcute într-un obiect să fie propagate și reflectate automat în toate obiectele dependente. Acest model permite o comunicare eficientă între componente și facilitează actualizarea automată a interfețelor utilizatorului în funcție de modificările aduse datelor.

Acestea sunt doar două exemple de modele de design, iar există multe altele, cum ar fi Modelul Factory, Modelul Singleton, Modelul Strategy și multe altele. Fiecare model are propriile sale avantaje și contexte de utilizare.

În concluzie, modelele de design reprezintă un instrument valoros pentru dezvoltatorii de software, oferindu-le direcție și structură în procesul de proiectare și implementare. Prin aplicarea modelelor de design, putem crea sisteme software mai robuste, ușor de întreținut și extensibile, sporind astfel calitatea produsului final.

# **1.Analiza domeniului de studiu**

Aplicația de tip Task Manager pe care doriți să o dezvoltați are un backend bazat pe Node.js, un frontend construit cu ReactJS și utilizează o bază de date PostgreSQL. Scopul acestei aplicații este de a permite utilizatorilor să gestioneze sarcinile sau task-urile lor într-un mod eficient.

Principalele funcționalități pe care aplicația ar trebui să le ofere includ posibilitatea de autentificare, adăugare și ștergere a task-urilor, modificare a acestora și afișare în funcție de un filtru.

Pentru a implementa funcționalitatea de autentificare, va trebui să permiteți utilizatorilor să se înregistreze și să se autentifice în aplicație. Aceste informații de autentificare pot fi stocate în baza de date PostgreSQL, într-un format securizat. Puteți utiliza un sistem de gestionare a autentificării bazat pe tokenuri sau sesiuni pentru a asigura securitatea și accesul controlat la aplicație.

Pentru gestionarea task-urilor, aplicația trebuie să permită utilizatorilor să adauge, să modifice și să șteargă task-uri. Un task poate fi definit prin diferite atribute, cum ar fi un titlu, o descriere, o dată limită, o prioritate sau o stare. Task-urile create de utilizatori ar trebui să fie stocate în baza de date PostgreSQL, fiecare task fiind asociat cu utilizatorul care l-a creat.

Pentru a permite utilizatorilor să filtreze și să sorteze task-urile, aplicația ar trebui să ofere opțiuni de filtrare și sortare în funcție de criterii specifice, cum ar fi dată limită, prioritate sau stare. Aceste opțiuni pot fi implementate folosind funcționalitățile de interogare și filtrare oferite de baza de date PostgreSQL.

Interfața utilizator ar trebui să fie dezvoltată cu ajutorul ReactJS și ar trebui să ofere utilizatorilor o experiență prietenoasă și intuitivă. Utilizatorii ar trebui să poată vizualiza lista de task-uri și să interacționeze cu acestea prin intermediul interfeței utilizator. Elementele interfeței ar trebui să includă opțiuni pentru crearea, editarea și ștergerea task-urilor, precum și pentru aplicarea de filtre și sortări.

Comunicarea între frontend și backend se va realiza prin intermediul unui API RESTful sau GraphQL. Frontendul dezvoltat cu ReactJS va face cereri către backend pentru a obține și a actualiza informațiile despre task-uri și utilizatori. Backendul, bazat pe Node.js, va gestiona aceste cereri și va efectua operațiile necesare în baza de date PostgreSQL pentru a asigura persistența datelor și consistența aplicației.

În final, baza de date PostgreSQL va fi utilizată pentru a stoca informațiile despre utilizatori, task-uri și alte date relevante. Schema bazei de date ar trebui să fie proiectată în mod corespunzător pentru a reflecta structura necesară pentru stocarea și gestionarea task-urilor și a informațiilor asociate acestora.

## **1.1 Importanţa temei**

Tema abordată, respectiv dezvoltarea unei aplicații de tip Task Manager, este una importantă și relevată în contextul actual al tehnologiei și al ritmului rapid al vieții cotidiene. Iată câteva aspecte care evidențiază importanța acestei teme:

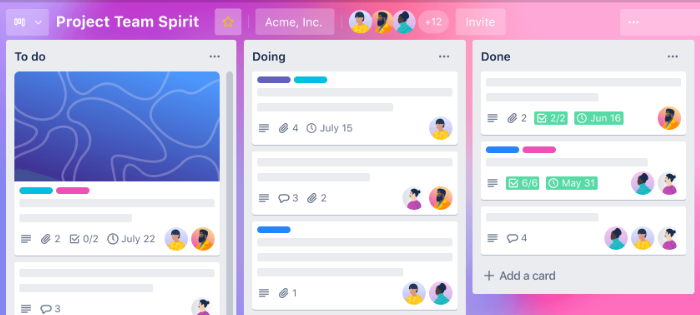
* Gestionarea eficientă a timpului: O aplicație de tip Task Manager ajută utilizatorii să-și organizeze sarcinile și să gestioneze timpul într-un mod mai eficient. Prin intermediul unei astfel de aplicații, utilizatorii pot urmări și prioritiza task-urile, seta termene limită și monitoriza progresul într-un mod structurat. Aceasta permite o planificare mai bună a activităților și o mai mare productivitate.
* Îmbunătățirea eficienței și organizării: O aplicație de tip Task Manager oferă utilizatorilor un sistem centralizat pentru a gestiona și organiza toate task-urile și sarcinile lor. Aceasta elimină nevoia de a utiliza diverse notițe, liste sau alte metode de organizare, reducând riscul de a pierde sau uita anumite sarcini. Utilizatorii pot accesa rapid și ușor lista de task-uri și pot actualiza sau completa informațiile aferente.
* Monitorizarea și urmărirea progresului: Prin intermediul unei aplicații de tip Task Manager, utilizatorii pot monitoriza progresul și evoluția sarcinilor. Aceasta permite o vizualizare clară a activităților desfășurate și a sarcinilor finalizate, oferind satisfacție și motivație utilizatorilor. De asemenea, permite identificarea și gestionarea sarcinilor în așteptare sau a celor cu prioritate ridicată.
* Colaborarea și comunicarea eficientă: O aplicație de Task Manager poate facilita colaborarea și comunicarea între membrii unui proiect sau echipe de lucru. Utilizatorii pot atribui și distribui sarcini, pot actualiza statusul lor și pot primi notificări relevante în legătură cu sarcinile asignate. Aceasta îmbunătățește coordonarea și sincronizarea activităților și permite un flux de lucru mai eficient.
* Flexibilitate și accesibilitate: Prin dezvoltarea unei aplicații de tip Task Manager cu backend pe Node.js și frontend pe ReactJS, utilizatorii pot beneficia de flexibilitate și accesibilitate sporite. Aceasta înseamnă că pot accesa și gestiona task-urile de pe diferite dispozitive, precum telefoane mobile, tablete sau computere, și pot beneficia de o experiență consistentă și sincronizată.

## **1.2 Sisteme similare:**

Există mai multe sisteme similare existente și populare în domeniul gestionării sarcinilor și a aplicațiilor de tip Task Manager. Iată câteva exemple:

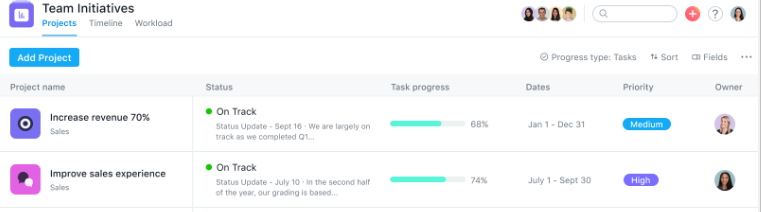
* Trello:

Trello este un sistem de gestionare a sarcinilor bazat pe panouri și liste. Utilizatorii pot crea liste de sarcini și pot organiza sarcinile în cadrul acestora. De asemenea, pot adăuga detalii, termene limită, etichete și pot colabora cu alți utilizatori în cadrul proiectelor.



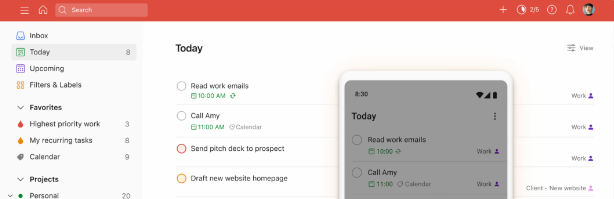
* Asana:

Asana este o platformă populară de gestionare a sarcinilor și a proiectelor. Permite utilizatorilor să creeze și să organizeze sarcini, să seteze termene limită, să atribuie sarcini altor membri ai echipei și să monitorizeze progresul într-un mod vizual.



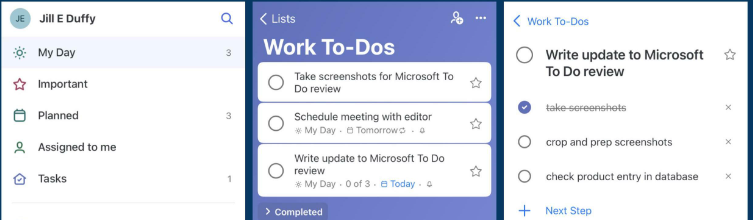
* Todoist:

Todoist este o aplicație de gestionare a sarcinilor bazată pe liste și priorități. Utilizatorii pot crea liste de sarcini, pot stabili priorități, pot adăuga termene limită și pot primi notificări pentru a fi la curent cu sarcinile lor.



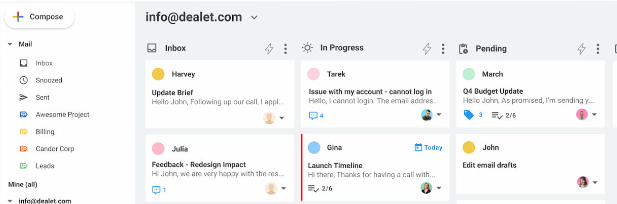
* Microsoft To Do:

Microsoft To Do este un sistem de gestionare a sarcinilor integrat cu ecosistemul Microsoft. Utilizatorii pot crea și organiza sarcini, pot stabili termene limită, pot sincroniza datele între diferite dispozitive și pot colabora în cadrul listelor de sarcini partajate.



* Google Tasks:

Google Tasks este un instrument simplu de gestionare a sarcinilor integrat în G Suite și Google Calendar. Utilizatorii pot crea sarcini, pot adăuga detalii și pot stabili termene limită. Sarcinile pot fi vizualizate și actualizate din diverse aplicații Google, precum Gmail și Google Calendar.



## **1.3 Scopul, obiectivele și cerintețele sistemului**

Scopul sistemului:

Scopul acestui sistem de Task Manager este de a permite utilizatorilor să gestioneze și să urmărească sarcinile lor într-un mod eficient și organizat. Acesta oferă o platformă centralizată pentru crearea, actualizarea și gestionarea task-urilor, precum și pentru monitorizarea progresului și prioritizarea acestora.

Obiectivele sistemului:

* Eficiență în gestionarea sarcinilor: Sistemul trebuie să faciliteze adăugarea rapidă și ușoară a sarcinilor, precum și actualizarea și ștergerea acestora. Utilizatorii trebuie să poată accesa și gestiona lista lor de sarcini într-un mod intuitiv și eficient.
* Monitorizarea progresului și urmărirea stării: Sistemul trebuie să ofere utilizatorilor posibilitatea de a actualiza și de a urmări starea fiecărui task. Acesta trebuie să permită marcarea task-urilor ca fiind "Open" (deschise), "In Progress" (în curs) sau "Closed" (închise), în funcție de etapa în care se află.
* Filtrare și sortare bazate pe statut: Sistemul trebuie să ofere utilizatorilor opțiuni de filtrare și sortare a sarcinilor în funcție de statutul acestora (Open, In Progress, Closed). Aceasta le permite utilizatorilor să vizualizeze și să se concentreze pe sarcinile relevante, în funcție de nevoile și prioritățile lor.
* Flexibilitate și personalizare: Sistemul trebuie să fie flexibil și să permită utilizatorilor să adauge detalii suplimentare la fiecare task, cum ar fi descrierea, termene limită, etichete sau altele. Aceasta le oferă utilizatorilor posibilitatea de a personaliza și adapta sistemul la nevoile lor individuale.

Cerințele sistemului:

* Autentificare și securitate: Sistemul trebuie să ofere un sistem de autentificare sigur, pentru a permite accesul doar utilizatorilor autorizați. Datele utilizatorilor și task-urile lor trebuie să fie protejate și securizate.
* Interfață intuitivă și prietenoasă pentru utilizatori: Interfața utilizator trebuie să fie intuitivă și ușor de utilizat, astfel încât utilizatorii să poată interacționa cu sistemul fără dificultate. Elementele interfeței utilizator ar trebui să fie clar definite și să permită navigarea și utilizarea facilă a funcționalităților sistemului.
* Funcționalitatea de adăugare, editare și ștergere a sarcinilor: Utilizatorii trebuie să poată adăuga rapid și ușor noi sarcini, să le editeze și să le șteargă după necesitate. Aceasta trebuie să fie realizată într-un mod simplu și eficient.
* Monitorizarea progresului și vizualizarea stării sarcinilor: Utilizatorii trebuie să poată monitoriza progresul sarcinilor și să vizualizeze clar starea acestora, în funcție de statutul lor (Open, In Progress, Closed).
* Filtrare și sortare bazate pe statut: Sistemul trebuie să ofere opțiuni de filtrare și sortare a sarcinilor în funcție de statutul acestora (Open, In Progress, Closed). Aceasta trebuie să fie realizată într-un mod eficient și rapid, pentru a permite utilizatorilor să acceseze rapid și ușor sarcinile relevante.

## **1.4 Şabloanele de proiectare folosite**

Sabloanele de proiectare sunt soluții generice și reutilizabile pentru probleme comune din dezvoltarea software-ului. Acestea oferă structuri și abordări standardizate pentru proiectarea și implementarea sistemelor software. Iată o scurtă prezentare a sabloanelor de proiectare menționate:

* **Singleton**: Şablonul Singleton se referă la crearea unei clase care poate avea o singură instanță în întregul sistem. Acesta este util în cazurile în care se dorește ca o anumită clasă să fie accesibilă global și să nu existe decât o singură instanță a acesteia.
* **Decorator**: Şablonul Decorator permite adăugarea de comportamente suplimentare la un obiect existent, fără a modifica structura acestuia. Acesta încapsulează obiectul inițial într-un decorator și oferă funcționalități extinse.
* **Adapter**: Şablonul Adapter permite adaptarea unei interfețe existente într-o altă interfață așteptată de un client. Acesta facilitează comunicarea între două componente care au interfețe diferite, transformând cerințele unei interfețe în cerințele celeilalte interfețe.
* **Observer**: Şablonul Observer permite notificarea automată a unui set de obiecte atunci când se produce o modificare în starea unui alt obiect. Aceasta facilitează comunicarea între obiecte și permite actualizarea automată a dependențelor.
* **Mediator**: Şablonul Mediator facilitează comunicarea între diferite obiecte, eliminând dependențele directe între acestea. Un mediator acționează ca un intermediar între obiectele implicate, gestionând interacțiunile și coordonând fluxul de informații.
* **Command**: Şablonul Command încapsulează o comandă într-un obiect separat, permițând astfel parametrizarea și execuția comenziilor într-un mod flexibil. Acesta separă emitentul de comandă de obiectul care o execută, oferind astfel flexibilitate și extensibilitate.

# **2. Structura UML a sistemului**

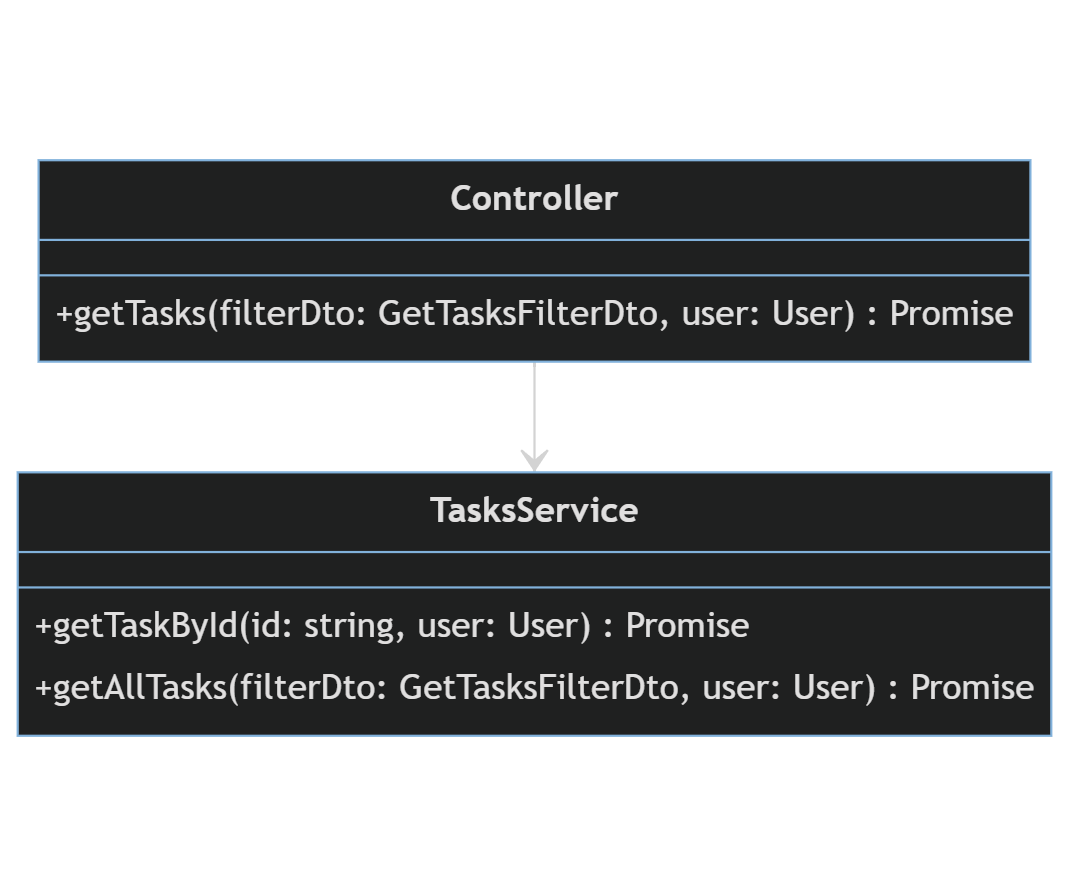
## **2.1 Diagrama pentru Sigleton Pattern**

Metoda getTasks din clasa Controller:

* Această metodă primește ca parametri un obiect GetTasksFilterDto și un obiect User.
* Apelul metodei getTasks din clasa Controller depinde de clasa TasksService pentru a obține o listă de sarcini.
* Metoda getTasks din clasa Controller apelează metoda getAllTasks din clasa TasksService, furnizând parametrii primiți în metoda getTasks.
* Rezultatul obținut din apelul metodei getAllTasks este returnat ca rezultat al metodei getTasks din clasa Controller.

Metoda getAllTasks din clasa TasksService:

* Această metodă primește ca parametri un obiect GetTasksFilterDto și un obiect User.
* Metoda getAllTasks din clasa TasksService utilizează acești parametri pentru a filtra și returna o listă de sarcini.
* Metoda getAllTasks poate utiliza și alte dependențe, cum ar fi un repository sau alte servicii, pentru a obține sarcinile din sursa de date.
* Rezultatul obținut din filtrarea sarcinilor este returnat ca rezultat al metodei getAllTasks din clasa TasksService.



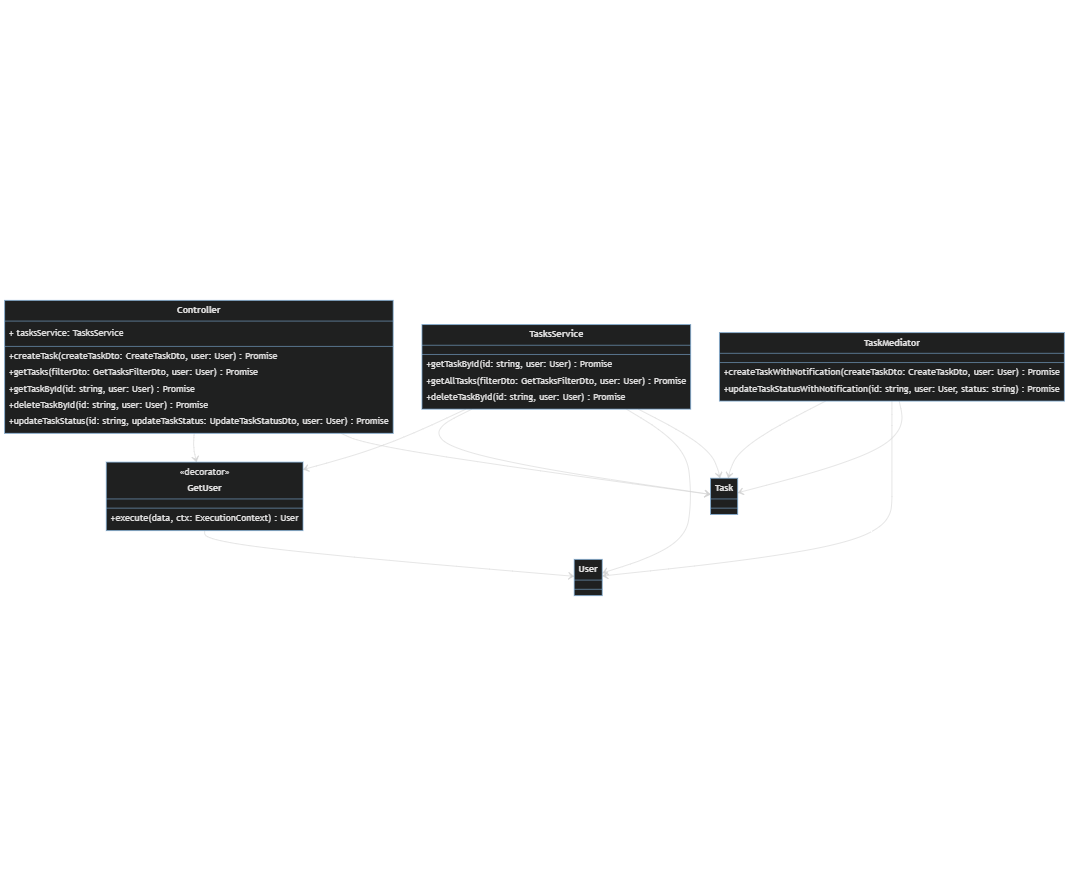
## **2.2 Diagrama pentru Decorator Pattern**

În diagrama de clase prezentată, avem trei clase principale: GetUser, Controller și TasksService. Aceste clase sunt interconectate în felul următor:

* GetUser este o clasă decoratoare care este folosită pentru a obține informații despre utilizatorul curent în cadrul contextului de execuție.
* Controller este o clasă care conține metodele pentru gestionarea cererilor HTTP și rutează cererile către servicii corespunzătoare.
* TasksService este o clasă care conține metode pentru gestionarea operațiilor legate de obiectele Task.

Interacțiunile dintre aceste clase sunt următoarele:

* GetUser este utilizată de Controller și TasksService pentru a obține informații despre utilizatorul curent.
* TasksService utilizează clasa Task pentru a reprezenta obiectele de tip task.
* TaskMediator este o clasă care intermediază operațiile legate de task-uri și implementează metodele createTaskWithNotification și updateTaskStatusWithNotification.
* Controller utilizează clasa TaskMediator pentru a crea și actualiza task-uri cu notificări asociate.



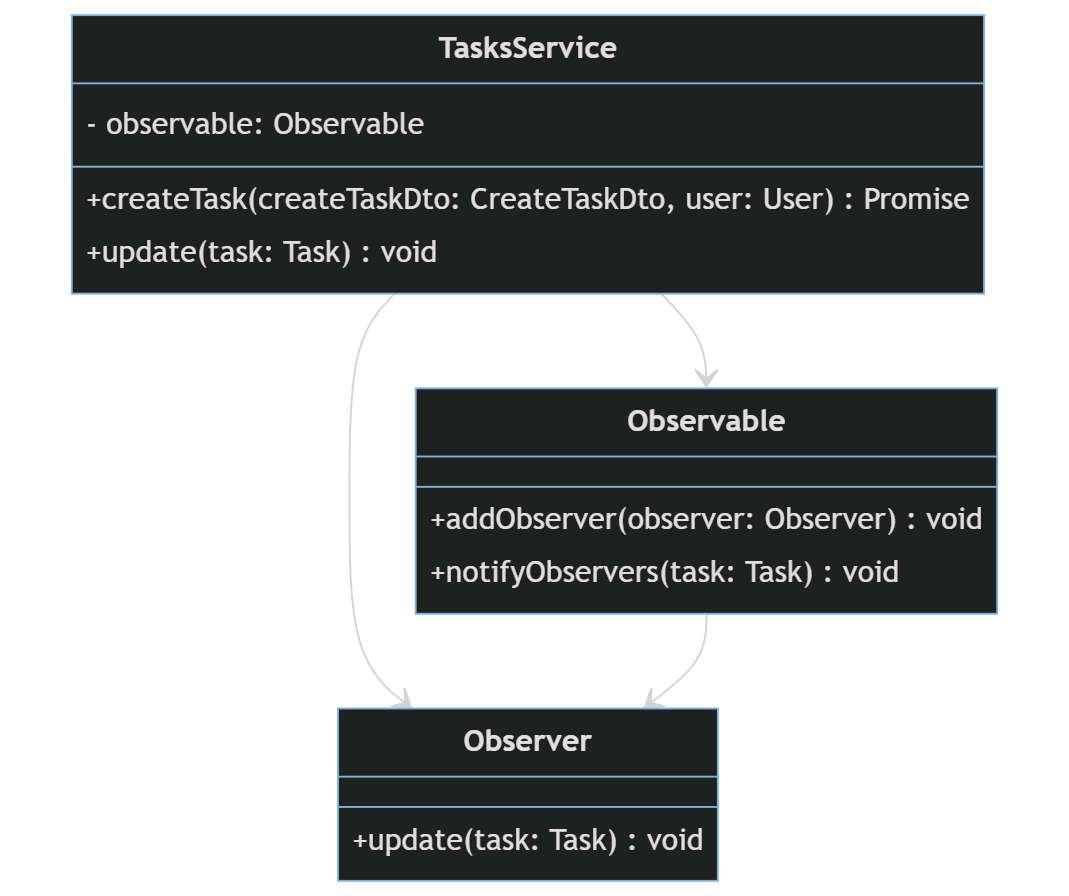
## **2.3 Diagrama pentru Adapter Pattern**

Relațiile între clase sunt reprezentate de săgeți:

* EmailService este asociat cu NotificationService, ceea ce înseamnă că EmailService utilizează NotificationService pentru a trimite e-mailuri.
* EmailServiceAdapter este, de asemenea, asociat cu NotificationService, ceea ce înseamnă că utilizează NotificationService pentru a trimite notificări (în acest caz, prin intermediul EmailService).
* TasksService este asociat cu EmailServiceAdapter, indicând că TasksService utilizează EmailServiceAdapter pentru a trimite notificări în momentul ștergerii unei sarcini.



## **2.4 Diagrama pentru Observer Pattern**



În diagrama de clase prezentată, avem trei clase principale: TasksService, Observer și Observable. Acestea reprezintă componentele principale implicate în observarea și notificarea evenimentelor în cadrul serviciilor de sarcini.

* TasksService este clasa principală care oferă funcționalități specifice serviciului de sarcini. Aceasta are o relație de utilizare cu clasa Observable, deoarece utilizează funcționalitatea acesteia pentru a notifica evenimente către observatori. Clasa TasksService implementează și interfața Observer, ceea ce înseamnă că poate fi înregistrată ca un observator și poate primi actualizări de la Observable atunci când se produce un eveniment.
* Observer este o interfață care definește metoda update(task: Task). Această interfață trebuie implementată de către clasele care doresc să acționeze ca observatori și să primească actualizări de la Observable.
* Observable este clasa care gestionează observatorii și notificările. Aceasta are două metode principale: addObserver(observer: Observer) pentru a adăuga un observator și notifyObservers(task: Task) pentru a trimite o notificare către toți observatorii înregistrați. TasksService utilizează această clasă pentru a adăuga observatorii și a trimite notificări atunci când este necesar.

## **2.5 Diagrama pentru Mediator Pattern**



În diagrama de clase dată, avem următoarele relații între clase:

* TaskMediator are o relație de dependență cu EmailService și TasksService. Aceasta înseamnă că TaskMediator utilizează serviciile oferite de aceste două clase pentru a îndeplini funcționalitatea sa. De exemplu, TaskMediator utilizează EmailService pentru a trimite notificări prin e-mail și TasksService pentru a crea și actualiza sarcini.
* TasksService are o relație de dependență cu EmailServiceAdapter. Aceasta înseamnă că TasksService utilizează serviciul de adaptare a serviciului de e-mail pentru a trimite notificări în anumite situații, cum ar fi ștergerea unei sarcini.
* EmailService și EmailServiceAdapter sunt clase separate, dar EmailServiceAdapter este utilizat de către TasksService pentru a trimite notificări. Prin urmare, există o asociere între TasksService și EmailServiceAdapter.

## **2.6 Diagrama pentru Command Pattern**

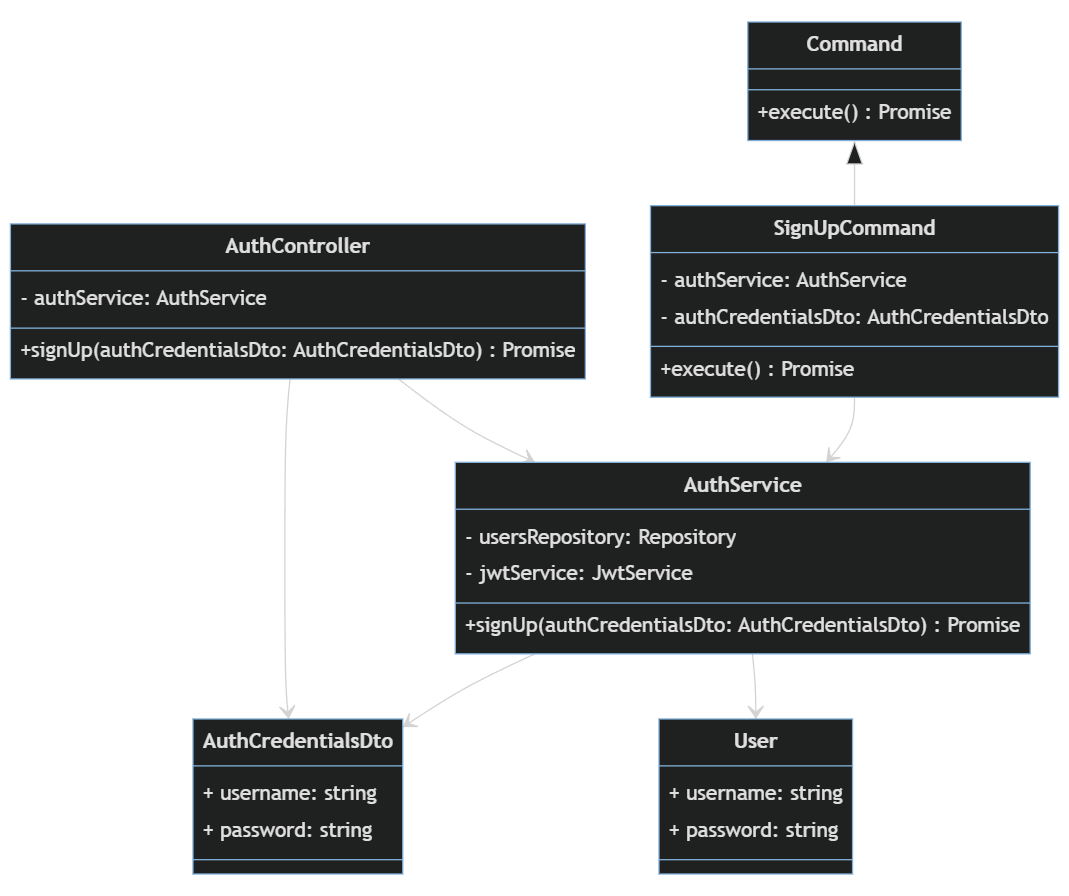


Diagrama de clase dată prezintă relațiile între clasele implicate în funcționalitatea de autentificare (auth).

* Clasa Command este o interfață care definește metoda execute(), iar clasa SignUpCommand o implementează. SignUpCommand este specializat pentru a gestiona comanda de înregistrare (sign-up).
* SignUpCommand are o dependență de tipul AuthService, deoarece utilizează acest serviciu pentru a executa operațiunea de înregistrare (signUp()).
* AuthService este responsabil pentru gestionarea logicii de autentificare. Această clasă are două dependențe: usersRepository (de tip Repository<User>) și jwtService (de tip JwtService). usersRepository este utilizat pentru a accesa și modifica entitățile User în baza de date. jwtService este utilizat pentru a genera și verifica token-uri JWT pentru autentificare.
* AuthController este un controler care expune endpoint-uri pentru operațiuni de autentificare. Acesta are o dependență de tipul AuthService pentru a gestiona logica de autentificare.
* AuthCredentialsDto este un obiect de transfer de date care conține informațiile de autentificare (username și password).
* User reprezintă o entitate care stochează informațiile despre utilizator în baza de date.

Command este o interfață pe care SignUpCommand o implementează.

* SignUpCommand depinde de AuthService pentru a executa operațiunea de înregistrare (signUp()).
* AuthController depinde de AuthService pentru a gestiona operațiunile de autentificare.
* AuthService utilizează obiecte de tip AuthCredentialsDto și User pentru a procesa și gestiona informațiile de autentificare.
* AuthController utilizează obiecte de tip AuthCredentialsDto pentru a primi informațiile de autentificare de la client.

# **3. Documentarea produsului realizat**

## **3.1 Implementarea Sigleton**

@Injectable()

export class TasksService implements Observer {

constructor(

@InjectRepository(Task)

private tasksRepository: Repository<Task>,

private readonly observable: Observable,

private readonly notificationService: EmailServiceAdapter,

) {

this.observable.addObserver(this);

}

async createTask(createTaskDto: CreateTaskDto, user: User): Promise<Task> {

const { title, description } = createTaskDto;

const task = this.tasksRepository.create({

title,

description,

status: TaskStatus.OPEN,

user,

});

await this.tasksRepository.save(task);

this.observable.notifyObservers(task);

return task;

}

În Node.js, serviciile pot fi implementate ca singleton-uri pentru a asigura că există o singură instanță a serviciului în întreaga aplicație. Există câteva motive pentru care serviciile sunt adesea implementate ca singleton-uri în Node.js:

* Eficiență: Dacă fiecare componentă a aplicației ar crea o nouă instanță a serviciului la fiecare apel, ar exista o suprapunere a resurselor și o creștere a consumului de memorie. Prin utilizarea unui singleton, se asigură că o singură instanță a serviciului este creată și utilizată de toate componentele, ceea ce reduce consumul de resurse.
* Coerență: Prin utilizarea unui singleton, se garantează că toate componentele care folosesc același serviciu vor interacționa cu aceeași instanță a acestuia. Acest lucru asigură coerența datelor și a stării serviciului în întreaga aplicație.
* Partajarea stării: Dacă un serviciu menține o stare internă sau are resurse costisitoare de inițializare, utilizarea unui singleton asigură că această stare sau resursele sunt partajate și accesibile de toate componentele care utilizează serviciul. Astfel, se evită duplicarea stării și se optimizează performanța aplicației.
* Ușurință în gestionarea dependențelor: Dacă un serviciu este implementat ca singleton, este mai ușor să se gestioneze dependențele către acel serviciu. Componentele care necesită acces la serviciu pot să-l injecteze și să-l utilizeze fără a trebui să se ocupe de crearea sau gestionarea instanțelor multiple.

## **3.2 Implementarea Decorator**

import { createParamDecorator, ExecutionContext } from "@nestjs/common";

import { User } from "./entities/user.entity";

export const GetUser = createParamDecorator(

(data, ctx: ExecutionContext): User => {

const req = ctx.switchToHttp().getRequest();

return req.user;

}

);

Decoratorul GetUser este implementat folosind funcția createParamDecorator furnizată de modulul @nestjs/common. Acest decorator este folosit pentru a obține utilizatorul curent din contextul cererii într-o rută a controlerului.

Decoratorul GetUser primește două argumente:

* data: Aceasta este o valoare opțională care poate fi folosită pentru a furniza informații suplimentare decoratorului. În exemplul dat, aceasta nu este utilizată și este setată la undefined.
* ctx: ExecutionContext: Acesta este contextul de execuție al cererii și conține informații despre modul în care cererea a fost procesată. Prin utilizarea funcției switchToHttp(), se accesează contextul specific cererii HTTP.

În implementarea decoratorului, se extrage obiectul req din contextul cererii folosind metoda switchToHttp().getRequest(). Acest obiect req reprezintă obiectul cererii HTTP și conține informații despre cerere, inclusiv utilizatorul autentificat (în cazul în care autentificarea a fost realizată anterior). În exemplul dat, se returnează proprietatea user din obiectul req, care ar trebui să fie obiectul User din entitatea user.entity.

## **3.3 Implementarea Adapter**

Clasa EmailService este o clasă care gestionează serviciul de trimitere a e-mailurilor prin intermediul pachetului nodemailer. Aceasta definește o metodă sendEmail care primește destinatarul, subiectul și corpul mesajului și trimite un e-mail către destinatar utilizând obiectul transporter creat în constructor.

async sendEmail(

recipient: string,

subject: string,

body: string,

): Promise<void> {

const mailOptions: nodemailer.SendMailOptions = {

from: 'raghimnaghiev@gmail.com',

to: recipient,

subject,

text: body,

};

await this.transporter.sendMail(mailOptions);

}}

Apoi, avem clasa EmailServiceAdapter, care implementează interfața NotificationService. Aceasta primește un obiect EmailService prin constructor și utilizează metoda sendEmail a acestuia pentru a implementa metoda sendNotification definită în interfața NotificationService.

@Injectable()

export class EmailServiceAdapter implements NotificationService {

constructor(private readonly emailService: EmailService) {}

async sendNotification(

recipient: string,

subject: string,

body: string,

): Promise<void> {

await this.emailService.sendEmail(recipient, subject, body);

}

}

În metoda deleteTaskById din clasa TasksService `this.notificationService.sendNotification` este folosit pentru a trimite o notificare de e-mail către utilizatorul `user.username` atunci când un task este şters.

## **3.4 Implementarea Observer**

Interfața Observer definește o metodă update, care primește un obiect Task și este implementată de către clasele care doresc să fie observatori și să fie notificate despre evenimente.

export interface Observer {

update(task: Task): void;

}

Clasa Observable reprezintă subiectul observabil și are două metode importante: addObserver și notifyObservers. Metoda addObserver primește un obiect de tip Observer și îl adaugă la lista de observatori. Metoda notifyObservers parcurge lista de observatori și apelează metoda update pentru fiecare observator, transmitându-i obiectul Task.

export class Observable {

private observers: Observer[] = [];

addObserver(observer: Observer): void {

this.observers.push(observer);

}

notifyObservers(task: Task): void {

this.observers.forEach((observer) => {

observer.update(task);

});

}}

În clasa TasksService, care implementează interfața Observer, se utilizează pattern-ul Observer pentru a notifica observatorii despre evenimente legate de sarcini (Task). În constructorul clasei, se adaugă obiectul TasksService ca observator în obiectul Observable:

@Injectable()

export class TasksService implements Observer {

constructor(

private readonly observable: Observable,

) {

this.observable.addObserver(this);

}

update(task: Task): void {

console.log(`Task with ID "${task.title}" has been created.`);

}}

În metoda createTask a clasei TasksService, după salvarea unei noi sarcini (task), se apelează metoda notifyObservers pentru a notifica toți observatorii despre evenimentul de creare a sarcinii:

async createTask(createTaskDto: CreateTaskDto, user: User): Promise<Task> {

await this.tasksRepository.save(task);

this.observable.notifyObservers(task);

return task;

}

## **3.5 Implementarea Mediator**

Clasa TaskMediator acționează ca un mediator între serviciul de e-mail (EmailService) și serviciul de sarcini (TasksService). Are două dependențe injectate prin constructor: EmailService și TasksService.

@Injectable()

export class TaskMediator {

private emailService: EmailService;

private tasksService: TasksService;

constructor(emailService: EmailService, tasksService: TasksService) {

this.emailService = emailService;

this.tasksService = tasksService;

}}

Clasa TaskMediator oferă două metode, createTaskWithNotification și updateTaskStatusWithNotification, care combină funcționalitatea serviciului de sarcini cu serviciul de e-mail pentru a crea sarcini și a actualiza starea lor, în timp ce trimit notificări prin e-mail către utilizatori.

Metoda createTaskWithNotification primește un obiect CreateTaskDto și un utilizator (User) și apelează metoda createTask a serviciului de sarcini (TasksService) pentru a crea o nouă sarcină. Apoi, construiește subiectul și corpul e-mailului și apelează metoda sendEmail a serviciului de e-mail (EmailService) pentru a trimite notificarea către utilizatorul specificat.

Metoda updateTaskStatusWithNotification primește un ID de sarcină, un utilizator și o nouă stare (TaskStatus) și apelează metoda updateTaskStatus a serviciului de sarcini (TasksService) pentru a actualiza starea sarcinii. Apoi, construiește subiectul și corpul e-mailului și apelează metoda sendEmail a serviciului de e-mail (EmailService) pentru a trimite notificarea către utilizatorul specificat.

## **3.6 Implementarea Command**

Interfața Command definește o metodă execute() care va fi implementată de clasele concrete ale comenzilor.

export interface Command {

execute(): Promise<void>;

}

Clasa SignUpCommand implementează interfața Command și reprezintă o comandă concretă care este responsabilă pentru înregistrarea utilizatorilor.

import { Command } from 'auth/interfaces/command.pattern';

import { AuthService } from './auth.service';

import { AuthCredentialsDto } from 'auth/dto/auth-credentials.dto';

export class SignUpCommand implements Command {

constructor(

private authService: AuthService,

private authCredentialsDto: AuthCredentialsDto,

) {}

async execute(): Promise<void> {

await this.authService.signUp(this.authCredentialsDto);

}}

Clasa SignUpCommand primește o instanță a serviciului AuthService și un obiect AuthCredentialsDto în constructor. Metoda execute() a clasei apelează metoda signUp() a serviciului AuthService, pasându-i obiectul AuthCredentialsDto, pentru a efectua înregistrarea utilizatorului.

În AuthController, în loc să apelăm direct metoda signUp() a serviciului AuthService, cream o instanță a clasei SignUpCommand și apelăm metoda execute() pentru a executa comanda.

@Controller('auth')

export class AuthController {

constructor(private authService: AuthService) {}

@Post('/signup')

async signUp(@Body() authCredentialsDto: AuthCredentialsDto): Promise<void> {

const command = new SignUpCommand(this.authService, authCredentialsDto);

return await command.execute();

}}

# **Concluzie:**

Implementarea și utilizarea multiplelor modele de design în cadrul proiectului de an au adus beneficii semnificative, cum ar fi modularitatea, extensibilitatea și flexibilitatea. Prin aplicarea conceptelor și principiilor de proiectare orientate pe obiecte și utilizarea pattern-urilor de proiectare, am reușit să obținem un cod mai clar, mai ușor de întreținut și mai ușor de extins.

Prin utilizarea pattern-urilor de proiectare precum Repository, Observer, Mediator și Command, am obținut o separare clară a responsabilităților între clasele noastre, creând astfel module independente care pot fi ușor de gestionat și testat separat. Aceasta a permis și o mai mare reutilizare a codului, deoarece am putut folosi aceleași pattern-uri în diferite părți ale aplicației noastre.

Un alt avantaj adus de implementarea acestor pattern-uri este creșterea flexibilității sistemului. Atunci când am fost nevoiți să facem modificări sau adăugări în funcționalitatea existentă, am putut face acest lucru cu un impact minim asupra celorlalte părți ale aplicației. De exemplu, atunci când am adăugat un nou serviciu de email, nu am fost nevoiți să modificăm direct serviciul existent de task-uri. În schimb, am folosit pattern-ul Mediator pentru a face interacțiunea între aceste două module, reducând astfel cuplajul și facilitând adăugarea de alte servicii în viitor.

De asemenea, am observat că utilizarea pattern-urilor de proiectare ne-a permis să respectăm principiul Single Responsibility Principle (SRP) și Open-Closed Principle (OCP). Am putut separa responsabilitățile diferitelor componente și am putut extinde comportamentul acestora prin adăugarea de noi clase și module fără a modifica codul existent.

În concluzie, prin aplicarea pattern-urilor de proiectare în cadrul proiectului nostru, am obținut un cod mai modular, mai ușor de înțeles și de întreținut, și am asigurat o arhitectură flexibilă și extensibilă. Aceste pattern-uri ne-au ajutat să construim un sistem robust și scalabil, care poate fi adaptat cu ușurință la schimbările și cerințele viitoare ale aplicației noastre.