Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Raport**

**la TMPS**

Tema: Sistemul de vânzare a biletelor de cinema și teatru

**Executant: \_\_\_\_\_\_\_\_ / Alhaj Ibrahim Bashar, gr. TI-204./**

Chișinău – 2023

Contents

[**1.Introducere:** 3](#_Toc136911718)

[**2.Sistemul de vanzare a ticketelor** 3](#_Toc136911719)

[**2.1. Funcționalitățile și cerințele acestuia** 4](#_Toc136911720)

[**2.2 Design patterns utilizate** 5](#_Toc136911721)

[**3. Implementare** 6](#_Toc136911722)

[**Concluzie** 12](#_Toc136911723)

# **1.Introducere:**

Design pattern-urile sunt soluții generice la probleme comune de proiectare software. Ele oferă abordări și structuri testate și validate pentru a rezolva anumite situații. Oamenii le învață pentru a îmbunătăți calitatea, eficiența și flexibilitatea dezvoltării software. Design pattern-urile oferă un limbaj comun între dezvoltatori și permit reutilizarea și extensibilitatea codului. Ele facilitează dezvoltarea software de înaltă calitate, cu o arhitectură solidă și ușor de înțeles și întreținută. Prin învățarea și aplicarea design pattern-urilor, dezvoltatorii pot crea soluții mai robuste, scalabile și ușor de modificat, conducând la o dezvoltare mai eficientă și la obținerea de rezultate superioare. Unele tipuri comune de design pattern includ: creational (crează obiecte într-un mod flexibil), structural (gestionează relațiile între obiecte), behavioral (definește comportamentul și interacțiunea între obiecte). Design patterns pot fi implementate într-o varietate de limbaje de programare, inclusiv Java, C++, C#, Python, JavaScript și altele. Aceste concepte sunt independente de limbaj și pot fi aplicate în orice limbaj care suportă programarea orientată pe obiecte și abstracțiuni de nivel înalt. Există mai multe design pattern-uri populare care sunt utilizate în mod frecvent în dezvoltarea software. Printre cele mai comune exemple se numără: Singleton, Factory, Observer, Decorator, Strategy și MVC (Model-View-Controller). Singleton asigură existența unei singure instanțe a unei clase. Factory furnizează o modalitate de creare a obiectelor în mod dinamic. Observer permite notificarea automată a unor obiecte despre schimbările într-un alt obiect. Decorator extinde funcționalitatea unei clase existente fără a o modifica direct. Strategy permite schimbarea algoritmului în timpul rulării. MVC separă modelul, vizualizarea și controlerul pentru o structură mai modulară și flexibilă.

# **2.Sistemul de vanzare a ticketelor**

Tema aleasă, un sistem de vânzare a biletelor pentru cinema și teatru, reprezintă un proiect complex care beneficiază de utilizarea a trei tipuri principale de design pattern-uri: creational, behavioural și structural. Design pattern-urile creational sunt utilizate în crearea și inițializarea obiectelor. În cazul sistemului de vânzare a biletelor, acestea pot fi aplicate în crearea biletelor pentru diverse evenimente, folosind pattern-uri precum Singleton pentru a asigura o singură instanță a claselor de gestiune a biletelor și Factory pentru a crea obiecte de tip bilet în funcție de cerințele utilizatorului. Design pattern-urile behavioural sunt concentrate pe gestionarea comportamentului și interacțiunii între obiecte. În cadrul sistemului de vânzare a biletelor, pattern-uri precum Observer pot fi utilizate pentru notificarea clienților cu privire la disponibilitatea și modificările biletelor, asigurând o comunicare eficientă și în timp real. Design pattern-urile structural se concentrează pe organizarea și structura componentelor sistemului. În cadrul sistemului de vânzare a biletelor, acestea pot fi aplicate în gestionarea relațiilor între diferitele componente, utilizând pattern-uri precum Composite pentru a crea o ierarhie a obiectelor și Decorator pentru a adăuga funcționalități suplimentare obiectelor existente.Prin aplicarea tuturor celor trei tipuri de design pattern-uri, sistemul de vânzare a biletelor devine mai flexibil, modular și ușor de extins. Utilizarea acestor pattern-uri permite o gestionare eficientă a obiectelor și interacțiunilor dintre ele, asigurând un sistem solid și scalabil.

## **2.1.** **Funcționalitățile și cerințele acestuia**

Sistemul de vânzare a biletelor pe care l-am dezvoltat vine în întâmpinarea nevoilor utilizatorilor prin oferirea unei game variate de funcționalități și caracteristici. Printre funcționalitățile cheie se numără posibilitatea de a alege locul și rândul dorit în sala de cinema sau teatru. Utilizatorii pot selecta cu ușurință locul preferat, având opțiunea de a alege între locuri standard sau locuri VIP pentru a se bucura de o experiență mai exclusivă.Pe lângă alegerea locului, sistemul nostru permite și aplicarea de reduceri pentru utilizatorii eligibili. Aceasta înseamnă că, în funcție de criterii precum vârsta, abonamentul sau alte calificări speciale, utilizatorii pot beneficia de prețuri reduse sau de promoții speciale. Astfel, dorim să oferim o experiență accesibilă și avantajoasă pentru toți clienții noștri.

Sistemul de vânzare a biletelor vine cu funcționalități avansate și caracteristici personalizate, pentru a asigura o experiență deosebită utilizatorilor noștri. Încurajăm utilizatorii să profite de toate facilitățile oferite, de la alegerea locului preferat până la beneficierea de reduceri și rezervări online, pentru a se bucura în deplinătate de evenimentele culturale și de divertisment.

## **2.2 Design patterns utilizate**

Proiectul a fost dezvoltat cu o atenție deosebită acordată utilizării design pattern-urilor, care au adus o valoare semnificativă în ceea ce privește structura, modularitatea și extensibilitatea sistemului de vânzare a biletelor. Pentru a atinge aceste obiective, au fost implementate design pattern-uri creational, structural și behavioral.

În cadrul design pattern-urilor creational, echipa a optat pentru Singleton, Builder și Prototype. Utilizarea Singleton a permis crearea unei singure instanțe a unei clase și asigurarea accesului global la aceasta. Design pattern-ul Builder a fost implementat pentru a oferi o modalitate simplă și flexibilă de construire a obiectelor complexe, în timp ce Prototype a permis clonarea obiectelor existente, reducând astfel costurile de creare a unor noi instanțe.

În ceea ce privește design pattern-urile structurale, am ales să implementăm Facade și Decorator. Design pattern-ul Facade a oferit o interfață simplificată pentru a accesa un sistem complex de subclase, făcând astfel codul mai ușor de înțeles și de utilizat. Decorator-ul a permis atașarea dinamică a funcționalităților suplimentare la obiecte existente, fără a afecta structura acestora.

Pentru a gestiona comportamentul dinamic al sistemului, am implementat design pattern-urile Observer și State. Observer-ul a facilitat comunicarea între diferite obiecte, astfel încât modificările făcute într-un obiect să fie propagate către ceilalți observatori. State-ul a permis schimbarea comportamentului unei clase în funcție de starea în care se află, oferind astfel o abordare elegantă și ușor de întreținut pentru gestionarea tranzițiilor de stare.

Utilizarea acestor design pattern-uri a adus multiple beneficii proiectului nostru, inclusiv modularitate crescută, flexibilitate în extinderea funcționalităților și ușurință în întreținerea codului. Aceste abordări bine fundamentate au permis dezvoltatorilor să lucreze eficient și să livreze un sistem robust și scalabil. Prin aplicarea design pattern-urilor, am reușit să creăm o arhitectură solidă și să îmbunătățim calitatea și performanța sistemului nostru de vânzare a biletelor.

# **3. Implementare**

În cadrul procesului de implementare al sistemului de vânzare a biletelor, am aplicat o abordare bazată pe utilizarea diverselor design pattern-uri pentru a asigura o arhitectură scalabilă, modulară și ușor de întreținut. În această secțiune a raportului, vom explora în detaliu implementarea design pattern-urilor creational, structural și behavioral și modul în care acestea au contribuit la funcționalitatea și eficiența sistemului.

Pentru a asigura că există o singură instanță a clasei Ticket în cadrul sistemului nostru de vânzare a biletelor, am implementat design pattern-ul Singleton. Astfel, am creat o clasă TicketManager care are o metodă statică de tipul getInstance(). Această metodă returnează întotdeauna aceeași instanță a clasei Ticket, indiferent de câte apeluri sunt făcute în cadrul sistemului. Prin utilizarea Singleton, ne asigurăm că toate operațiunile referitoare la gestionarea biletelor sunt realizate într-un mod coerent și că nu există duplicări sau conflict de date.

constructor() {

        if (Ticket.instance) {

            return Ticket.instance;

        }

        Ticket.instance = this;

Tot aceeasi clasa are si alte metode, de serviciu, pentru a funcționa programul.

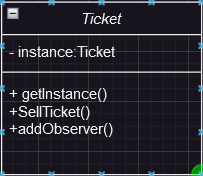


Figura 1 – Diagrama UML pentru clasa Ticket

class Ticket {

    constructor() {

        if (Ticket.instance) {

            return Ticket.instance;

        }

        Ticket.instance = this;

        this.tickets = [];

        this.observers = [];

        return this;

    }

    sellTicket(ticket) {

        this.tickets.push(ticket);

        this.notifyObservers(ticket);

    }

    addObserver(observer) {

        this.observers.push(observer);

    }

O parte din clasa Ticket scrisa pe baza Singleton.

În clasa TicketBuilder, am implementat design pattern-ul Builder pentru a crea obiecte Ticket complexe și personalizate. Builder-ul ne permite să construim treptat un obiect Ticket, adăugând în mod incremental detaliile necesare.

class TicketBuilder {

    constructor() {

        this.ticket = {};

    }

    setType(type) {

        this.ticket.type = type;

        return this;

    }

    setSeat(seat) {

        this.ticket.seat = seat;

        return this;

    }

    setIsVIP(isVIP) {

        this.ticket.isVIP = isVIP;

        return this;

    }

    setStudentDiscount(studentDiscount) {

        this.ticket.studentDiscount = studentDiscount;

        return this;

    }

    build() {

        return this.ticket;

    }

}

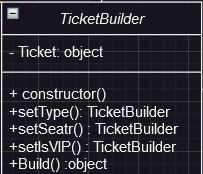


Figura 2 – Diagrama UML pentru clasa TickeBuilder

Am definit metode specifice pentru a seta diversele atribute ale unui bilet.

In clasa TicketPrototype, am folosit design pattern-ul Prototype pentru a crea copii ale unui obiect de tip Ticket existent, în loc de a crea obiecte noi de fiecare dată când avem nevoie de un ticket similar.

class TicketPrototype {

    clone() {

        return Object.assign(Object.create(Object.getPrototypeOf(this.constructor.prototype)), this);

    }

}

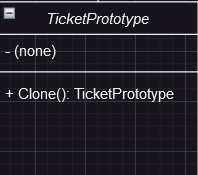


Figura 3 – Clasa UML pentru clasa TicketPrototype

Am definit clasa TicketPrototype care conține un obiect de bază, un prototip, din care putem crea copii. Când dorim să creăm un nou ticket, în loc să creăm un obiect nou, putem clona prototipul existent și să modificăm doar atributele necesare.

Clasa TicketFacade este un exemplu de utilizare a design pattern-ului Facade. Aceasta furnizează o interfață simplificată și unificată pentru utilizatorii sistemului de vânzare a biletelor.

class TicketFacade {

    constructor() {

        this.ticketSystem = new Ticket();

        this.ticketBuilder = new TicketBuilder();

    }

    sellTicket(type, seat, isVIP, studentDiscount) {

        const ticket = this.ticketBuilder

            .setType(type)

            .setSeat(seat)

            .setIsVIP(isVIP)

            .setStudentDiscount(studentDiscount)

            .build();

        this.ticketSystem.sellTicket(ticket);

        return ticket;

    }

}

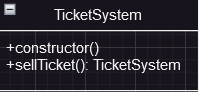


Figura 4 – Clasa uml pentru clasa TicketSystem

În clasa TicketFacade, sunt ascunse complexitățile și interdependențele dintre diversele componente ale sistemului, cum ar fi generarea biletelor, verificarea disponibilității locurilor, aplicarea reducerilor sau rezervarea biletelor.

In clasa TicketDecorator, am utilizat design pattern-ul decorator pentru a extinde functionalitatea obiectului Ticket de baza.

 class StudentDiscount {

    constructor(ticketSystem) {

        this.ticketSystem = ticketSystem;

    }

    sellTicket(ticket) {

        this.ticketSystem.sellTicket(ticket);

        console.log("Applying student discount...");

    }

}

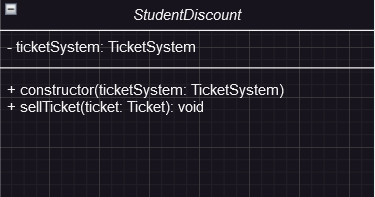


Figura 5 – Clasa UML pentru clasa StudentDiscount

Prin utilizarea pattern-ului State, am putut defini și implementa logică specifică pentru fiecare stare a biletelor, cum ar fi disponibil, vândut sau rezervat.

class TicketState {

    constructor() {

        this.state = "available";

    }

    sellTicket() {

        this.state = "sold";

        console.log("Ticket sold.");

    }

    getState() {

        return this.state;

    }

}

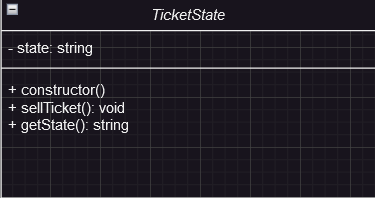


Figura 6 – Clasa uml pentru clasa TicketState

Aceasta ne-a permis să adaptăm comportamentul aplicației în funcție de starea curentă a biletelor și să oferim o experiență de vânzare a ticketelor mai dinamică și personalizată. Implementarea corectă a acestui pattern ne-a ajutat să creștem modularitatea și flexibilitatea sistemului nostru de vânzare a ticketelor.

# **Concluzie**

În concluzie, în implementarea sistemului de vânzare a biletelor, am utilizat cu succes principiile design pattern și SOLID pentru a crea un sistem robust și flexibil. Am aplicat diverse design patterns, cum ar fi Singleton pentru a asigura o instanță unică a sistemului de vânzare, Builder pentru a construi obiecte complexe de bilete, Factory Method pentru a crea obiecte de bilete de diferite tipuri, Facade pentru a oferi o interfață simplificată către sistemul de vânzare, Decorator pentru a adăuga funcționalitate suplimentară (cum ar fi reducerile pentru studenți), Memento pentru a permite salvarea și restaurarea stării biletelor, și Iterator pentru a itera prin colecții de bilete.Aceste design patterns ne-au ajutat să îmbunătățim modularitatea, extensibilitatea și înțelegerea sistemului nostru de vânzare a biletelor. Implementarea solidă a acestor principii ne-a permis să separăm responsabilitățile, să creăm clase și interfețe bine definite și să facilităm extinderea și întreținerea sistemului pe termen lung.În final, am construit un sistem de vânzare a biletelor puternic, care respectă principiile design pattern și SOLID, oferind o bază solidă pentru dezvoltarea și scalabilitatea viitoare a aplicației noastre.