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: Un sistem de librarie

**Executant: \_\_\_\_\_\_\_\_ / David, gr. TI-204./**

Chișinău – 2023

Contents

[**1.Introducere** 3](#_Toc136887159)

[**2.Etapele de creare a sistemului** 3](#_Toc136887160)

[**2.1. Funcționalitățile și cerințele acestuia** 3](#_Toc136887161)

[**2.2 Design patterns utilizate** 4](#_Toc136887162)

[**3.Implementarea** 5](#_Toc136887163)

# **1.Introducere**

Acest proiect de an se bazează pe conceptele și mecanismele prezentate în cadrul cursului de Teoria și Mecanisme de Proiectare a Sistemelor. Scopul acestui proiect este de a oferi o soluție simplă și eficientă, utilizând diverse pattern-uri de design, pentru a satisface cerințele sistemelor proiectate. Prin utilizarea design pattern-urilor, proiectul își propune să ofere o abordare structurată și modulară pentru implementarea sistemelor. Pattern-urile de design oferă soluții testate și validate pentru problemele comune întâlnite în dezvoltarea software-ului. Proiectul se concentrează pe implementarea unei biblioteci virtuale, în care utilizatorii pot adăuga cărți, atribui cărților categorii, șterge cărți și aplica reduceri la prețurile și categoriile de cărți. Prin utilizarea acestor pattern-uri de design, proiectul își propune să ofere o soluție modulară, ușor de extins și de întreținut, respectând principiile de proiectare a sistemelor și utilizând abordări consacrate și validate în industria software-ului.

# **2.Etapele de creare a sistemului**

Pentru a crea acest sistem, a fost necesară selecția design pattern-urilor potrivite pentru a asigura o structură coerentă și modular Principiul simplității a fost un aspect cheie în crearea acestui sistem. Fiecare design pattern și principiu solid folosit a avut în vedere simplificarea și claritatea implementării. Astfel, s-a urmărit ca utilizarea sistemului să fie cât mai intuitivă și accesibilă, fără a adăuga complexitate inutilă. Prin adoptarea unei abordări simpliste, s-a facilitat înțelegerea și utilizarea sistemului de către utilizatori și programatori.

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

Sistemul permite adăugarea și ștergerea cărților, precum și salvarea acestora. Utilizând un iterator, este posibilă parcurgerea tuturor cărților din bibliotecă. Această funcționalitate oferă flexibilitate în gestionarea cărților din sistem și facilitează efectuarea operațiilor necesare. Utilizatorii pot adăuga noi cărți în bibliotecă, șterge cărți existente sau salva starea cărților pentru a le recupera ulterior. Prin intermediul iteratorului, pot fi parcurse toate cărțile din bibliotecă într-un mod structurat și eficient. Sistemul permite adăugarea, ștergerea și salvarea cărților într-un mod simplu și eficient. Utilizând design pattern-urile potrivite, precum Singleton pentru asigurarea unei singure instanțe a bibliotecii și Factory Method pentru crearea cărților, s-a obținut o structură coerentă și ușor de utilizat. De asemenea, prin utilizarea design pattern-ului Proxy, am putut implementa funcționalitatea de preț pentru cărți, permițând setarea și obținerea prețurilor cu ajutorul obiectului BookProxy. Implementarea design pattern-ului Memento a oferit posibilitatea de a salva și restaura starea cărților, astfel încât utilizatorii să poată reveni la un anumit punct în timp. Prin intermediul iteratorului, utilizatorii pot parcurge în mod eficient și flexibil lista cărților din bibliotecă, având acces la informații despre titlu, autor și categorie. În final, sistemul a fost conceput cu scopul de a fi ușor de utilizat, punând accent pe simplitate, eficiență și utilizarea adecvată a design pattern-urilor și principiilor SOLID.

## **2.2 Design patterns utilizate**

Pentru a crea acest sistem, a fost necesară selecția design pattern-urilor potrivite pentru a asigura o structură coerentă și modulară. În implementarea proiectului, s-au utilizat următoarele design pattern-uri: S-a utilizat pattern-ul Singleton pentru a garanta că există o singură instanță a clasei `Library`, asigurând astfel accesul global la bibliotecă și evitând crearea mai multor instanțe nedorite.S-a folosit pattern-ul Factory Method pentru a oferi o abstracție și o metodă flexibilă de creare a obiectelor de tip `Book` prin intermediul clasei `BookFactory`. Aceasta permite encapsularea procesului de creare și ascunderea detaliilor de implementare.Pattern-ul Builder a fost ales pentru a permite construirea ușoară a obiectelor `Book` prin intermediul clasei `BookBuilder`. Acesta oferă posibilitatea de a seta treptat atributele cărților și de a crea obiectul final într-un mod intuitiv și flexibil.S-a implementat pattern-ul Facade prin intermediul clasei `LibraryFacade` pentru a oferi o interfață simplificată și unificată pentru utilizatori. Aceasta ascunde detaliile interne ale sistemului și oferă metode clare și ușor de utilizat pentru adăugarea și ștergerea cărților din bibliotecă.Pattern-ul Proxy a fost utilizat prin intermediul clasei `BookProxy` pentru a oferi un mecanism intermediar de acces la prețurile cărților. Acesta permite gestionarea și manipularea prețurilor cu flexibilitate, precum și obținerea prețurilor din obiectul real `Book`.S-a implementat pattern-ul Memento prin intermediul clasei `BookMemento` pentru a permite stocarea și restaurarea stării unei cărți. Aceasta oferă posibilitatea de a reveni la o stare anterioară a cărților și de a efectua operații de undo/redo asupra acestora.S-a utilizat pattern-ul Iterator prin intermediul clasei `BookIterator` pentru a parcurge lista de cărți într-un mod eficient și simplu. Acesta oferă metode pentru verificarea existenței următoarei cărți și pentru obținerea următoarei cărți din listă.

# **3.Implementarea**

Am folosit design pattern-ul Singleton pentru a asigura că există o singură instanță a clasei Library în întregul sistem. Aceasta permite o gestionare eficientă a cărților și evită crearea duplicatelor.

class Library {

    constructor() {

        if (Library.instance) {

            return Library.instance;

        }

        this.books = [];

        Library.instance = this;

        return this;

    }

    addBook(book) {

        this.books.push(book);

    }

    removeBook(book) {

        const index = this.books.indexOf(book);

        if (index > -1) {

            this.books.splice(index, 1);

        }

    }

    getBooks() {

        return this.books;

    }

}

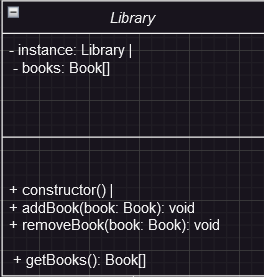


Figura 1 – Clasa Uml pentru clasa Library

Astfel, fiecare componentă care accesează biblioteca folosește aceeași instanță a clasei Library, oferind un punct centralizat de control și manipulare a cărților. Implementarea Singleton asigură că nu există multiple instanțe ale bibliotecii și facilitează un design coerent și consistent în cadrul sistemului.

Am folosit design pattern-ul Builder pentru a crea obiecte de tip Book în cadrul sistemului de bibliotecă. Acesta oferă o modalitate flexibilă și modulară de construire a obiectelor Book cu diferite proprietăți, precum titlu, autor și categorie. Utilizând Builder, putem seta sau omite anumite proprietăți ale cărților în funcție de nevoile noastre, făcând procesul de construire mai ușor și mai intuitiv.

class BookBuilder {

    constructor() {

        this.title = null;

        this.author = null;

        this.category = null;

    }

    setTitle(title) {

        this.title = title;

        return this;

    }

    setAuthor(author) {

        this.author = author;

        return this;

    }

    setCategory(category) {

        this.category = category;

        return this;

    }

    build() {

        return new Book(this.title, this.author, this.category);

    }

}

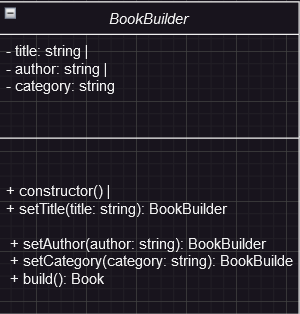


Figura 2 – Clasa UML pentru clasa BookBuilder

Putem crea obiecte Book într-un mod fluent și controlat.

În cadrul acestui sistem de bibliotecă, am utilizat și clasa `BookFactory` pentru a facilita crearea de obiecte de tip `Book`. Această abordare ne permite să creăm cărți într-un mod flexibil, oferindu-ne posibilitatea de a specifica titlul, autorul și categoria prin intermediul metodelor dedicate.

class BookFactory {

    createBook(title, author, category) {

        return new Book(title, author, category);

    }

}

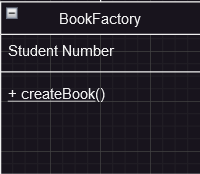


Figura 3 – Clasa Uml pentru clasa BookFactory

In acest mod putem gestiona cu ușurință procesul de creare a cărților și să ne asigurăm că acestea sunt corect configurate în momentul instantierii.

În acest sistem, am implementat un tip virtual de design pattern Proxy. Am utilizat clasa `BookProxy` pentru a oferi o abstracție suplimentară pentru obiectele de tip `Book`, permițându-ne să controlăm accesul la anumite funcționalități și să adăugăm comportamente suplimentare.

class BookProxy {

    constructor(book) {

        this.book = book;

        this.price = null;

    }

    setPrice(price) {

        this.price = price;

    }

    getPrice() {

        if (this.price !== null) {

            return this.price;

        } else {

            return this.book.price;

        }

    }

}

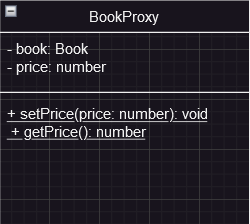


Figura 4 – Clasa uml pentru clasa BookProxy

Prin intermediul `BookProxy`, putem manipula prețurile cărților și oferi funcționalități adiționale într-un mod transparent pentru utilizatorii clasei `Book`. Astfel, am obținut un grad mai mare de flexibilitate și modularitate în gestionarea cărților din sistemul de bibliotecă.

În cadrul acestui sistem de librarie, am utilizat și un iterator pentru a permite parcurgerea și accesul la cărțile din colecție într-un mod secvențial și simplu.

class BookIterator {

    constructor(books) {

        this.books = books;

        this.index = 0;

    }

    hasNext() {

        return this.index < this.books.length;

    }

    next() {

        const book = this.books[this.index];

        this.index++;

        return book;

    }

}

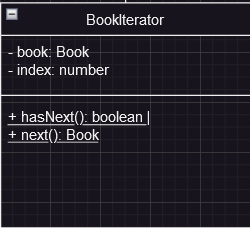


Figura 5 – Clasa Uml pentru calsa BookIterator

Acest iterator ne-a oferit posibilitatea de a itera prin cărțile existente, verificând disponibilitatea următorului element și obținând acces la fiecare carte în parte într-o manieră eficientă și convenabilă.

În plus, am implementat și pattern-ul Memento în cadrul sistemului nostru de librărie. Acest pattern ne-a permis să salvăm starea unei cărți într-un moment dat, astfel încât să o putem restaura ulterior, în cazul în care este necesar.

class BookMemento {

    constructor(book) {

        this.title = book.title;

        this.author = book.author;

        this.category = book.category;

    }

    restore(book) {

        book.title = this.title;

        book.author = this.author;

        book.category = this.category;

    }

}

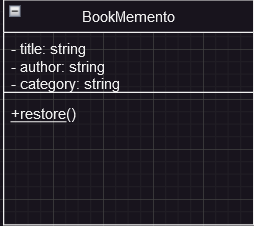


Figura 6 – Clasa uml pentru clasa BookMemento

Prin utilizarea Memento, am putut gestiona salvarea și restaurarea datelor cărților într-un mod flexibil și modular, oferind o mai mare flexibilitate și control asupra sistemului nostru de librărie.

**Conclzuie**:

În concluzie, am implementat un sistem de librărie folosind diverse design pattern-uri pentru a asigura o structură robustă și ușor de utilizat. Am folosit Singleton pentru a garanta existența unei singure instanțe a clasei Library în întregul sistem. Am utilizat și Builder pentru a crea obiecte de tip Book cu o sintaxă clară și modulară. De asemenea, am implementat Factory Method prin intermediul clasei BookFactory pentru a abstractiza procesul de creare a cărților.Pentru a gestiona accesul la prețuri, am folosit Proxy Pattern, permițând setarea și obținerea prețului prin intermediul clasei BookProxy. De asemenea, am implementat Iterator pentru a itera prin toate cărțile din bibliotecă. Pentru salvarea și restaurarea stării cărților, am utilizat Memento Pattern, folosind clasa BookMemento.În ansamblu, prin utilizarea acestor design pattern-uri, am creat un sistem de librărie bine structurat și ușor de utilizat, care facilitează adăugarea, ștergerea și restaurarea cărților, precum și iterarea prin ele.