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 MusicPlayer

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

Chișinău – 2023

Contents

[**1.Introducere** 3](#_Toc136532675)

[**2.MusicPlayer, un sistem complex** 3](#_Toc136532676)

[**2.1. Funcționalitățile și cerințele acestuia** 4](#_Toc136532677)

[**2.2 Design patterns utilizate** 4](#_Toc136532678)

[**3. Implementare** 5](#_Toc136532679)

[**Concluzie** 12](#_Toc136532680)

# **1.Introducere**

Studiul design patternurilor este deosebit de important în dezvoltarea software deoarece ne oferă un set de soluții validate pentru problemele comune. Înțelegerea și aplicarea design patternurilor ne ajută să dezvoltăm software modular, flexibil și ușor de întreținut. Ele ne permit să scriem cod mai clar și mai eficient, reducând duplicarea și complexitatea. De asemenea, studiul design patternurilor ne ajută să devenim programatori mai buni, dezvoltând abilități de proiectare. Design patternurile reprezintă soluții testate și recunoscute în domeniul dezvoltării software pentru probleme comune întâlnite în proiectarea și implementarea aplicațiilor. Acestea oferă un set de abordări și principii care ne ajută să dezvoltăm software modular, flexibil și ușor de întreținut. Design patternurile sunt împărțite în trei categorii principale: creational,structural și behavioral. Fiecare categorie se concentrează pe aspecte specifice ale proiectării software și oferă soluții adecvate pentru diferite probleme. Patternurile de creare (se concentrează pe procesul de creare a obiectelor într-un mod flexibil și adaptabil. Acestea permit crearea de obiecte într-un mod mai generic și mai independent de clasele concrete. Patternurile structurale se concentrează pe organizarea și relațiile între obiecte. Acestea abordează modul în care clasele și obiectele pot fi asamblate pentru a forma structuri mai mari și mai complexe. Exemple de design patternuri structurale includ Facade, Bridge și Adapter. Patternurile comportamentale se concentrează pe interacțiunile și comportamentul între obiecte și clase. Acestea rezolvă probleme legate de comunicarea și coordonarea între obiecte. Exemple de design patternuri comportamentale includ Observer, Strategy și Iterator. Observer permite unui obiect să notifice alte obiecte despre modificările stării sale. Strategy ne permite să definim o familie de algoritmi și să-i folosim într-o manieră flexibilă.

# **2.MusicPlayer, un sistem complex**

Sistemul de muzică player creat în JavaScript este o aplicație complexă care îmbină cele trei tipuri de design pattern-uri pentru a oferi funcționalități avansate și o experiență plăcută utilizatorilor. Acest sistem permite utilizatorilor să adauge muzică preferată, să sorteze melodiile după artist și să exploreze diverse alte opțiuni.

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

Pentru a crea o instanță unică a sistemului de muzică player, vom utiliza design pattern-ul Singleton. Acesta ne permite să avem o singură instanță a clasei music player în întreaga aplicație. Astfel, utilizatorii vor putea interacționa cu același sistem de muzică player indiferent de zona aplicației în care se află. Pentru a adăuga muzică preferată, vom folosi design pattern-ul Factory Method. Acesta ne permite să creăm obiecte de tipul unei clase într-un mod flexibil. Vom utiliza Factory Method pentru a crea obiecte de tipul "Melodie" pe baza preferințelor utilizatorilor. Astfel, ei vor putea adăuga și gestiona melodiile preferate într-un mod ușor și intuitiv.Pentru a crea o interfață simplificată și ușor de utilizat, vom folosi design pattern-ul Facade. Acesta ne permite să ascundem complexitatea și detaliile de implementare ale sistemului de muzică player în spatele unei interfețe simple și coerente. Utilizatorii vor putea naviga prin funcționalitățile sistemului fără a fi nevoiți să se preocupe de aspectele tehnice și complexe. În final, implementarea sistemului de muzică player în JavaScript utilizând cele trei tipuri de design pattern-uri: creational, structural și behavioral, va oferi o experiență avansată și plăcută utilizatorilor. Aceste design pattern-uri ne ajută să dezvoltăm o aplicație modulară, flexibilă și ușor de întreținut. De asemenea, ele facilitează reutilizarea și extensibilitatea codului, permițând dezvoltatorilor să adauge și să modifice funcționalități fără a afecta întregul sistem. Prin studiul și aplicarea design pattern-urilor, dezvoltatorii pot construi aplicații mai eficiente și mai scalabile, care să răspundă nevoilor utilizatorilor într-un mod optim.

## **2.2 Design patterns utilizate**

Proiectul de music player este un exemplu excelent în care au fost utilizate mai multe design patterns pentru a adăuga funcționalități complexe și pentru a facilita gestionarea eficientă a datelor și a operațiilor.

Unul dintre design patterns folosite în acest proiect este Singleton. Singleton este folosit pentru a asigura că există o singură instanță a clasei care gestionează music player-ul în întregul sistem. Această instanță unică asigură coerența și consistența stării music player-ului și permite accesul la funcționalități comune din orice parte a aplicației.

Alt design pattern utilizat este Factory Method. Factory Method este folosit pentru a crea diverse tipuri de playlist-uri și melodii în funcție de cerințe specifice. Acest pattern oferă flexibilitate în crearea obiectelor și permite extinderea facilă a sistemului cu noi tipuri de playlist-uri și melodii în viitor.

Un alt pattern folosit este Iterator. Iterator permite parcurgerea și accesul secvențial la diferitele melodii din playlist-uri. Acesta oferă o abordare simplă și unificată pentru a itera prin melodii, indiferent de structura internă a playlist-urilor.

Memento este un alt design pattern utilizat în proiectul de music player. Acesta este folosit pentru a salva și restaura starea music player-ului în timpul utilizării. Prin utilizarea Memento, utilizatorii pot reveni la starea anterioară a music player-ului și pot continua redarea melodiilor de unde au rămas.

Un alt pattern folosit în proiect este Facade. Facade oferă o interfață simplificată și unificată pentru a accesa diferite funcționalități ale music player-ului, precum crearea și gestionarea playlist-urilor sau adăugarea melodiilor. Acesta ascunde detaliile complexe ale implementării și oferă o experiență facilă și intuitivă utilizatorilor.

Builder este un alt pattern utilizat pentru a construi playlist-uri personalizate. Utilizând Builder, utilizatorii pot adăuga melodii într-un playlist într-un mod flexibil și modular. Aceasta permite personalizarea și configurarea ușoară a playlist-urilor în funcție de preferințele utilizatorilor.

Prin utilizarea acestor design patterns în proiectul de music player, s-au obținut multiple beneficii, precum modularitatea, flexibilitatea, extensibilitatea și ușurința în gestionarea funcționalităților complexe ale music player-ului. De asemenea, aceste design patterns au contribuit la îmbunătățirea calității și eficienței dezvoltării proiectului, facilitând dezvoltarea, testarea și mentenanța acestuia.

# **3. Implementare**

Utilizarea diverselor design pattern-uri ne-a permis să simplificăm sistemul de music player și să obținem un cod curat și modular. Prin aplicarea acestor pattern-uri, am reușit să separăm responsabilitățile și să reducem dependențele între componente, facilitând astfel înțelegerea și extinderea ulterioară a sistemului. Am obținut un cod mai clar, mai ușor de întreținut și mai ușor de testat, ceea ce contribuie la dezvoltarea și menținerea unui sistem robust și scalabil. Utilizarea design pattern-urilor ne-a permis să abordăm cerințele complexe ale proiectului într-un mod structurat și eficient.

În această secvență de cod, am implementat pattern-ul Singleton pentru a ne asigura că o singură instanță a unei clase este creată și utilizată în întregul sistem. Prin aceasta, am garantat că anumite resurse sau funcționalități sunt accesibile și partajate în mod coerent în cadrul aplicației noastre. Singleton ne-a oferit control asupra creării și utilizării obiectului, asigurând astfel o gestionare eficientă a resurselor și evitarea creării multiple a instanțelor.

class MusicPlayer {

    constructor() {

        this.playlist = [];

        this.favoriteSongs = [];

    }

    static getInstance() {

        if (!MusicPlayer.instance) {

            MusicPlayer.instance = new MusicPlayer();

        }

        return MusicPlayer.instance;

    }

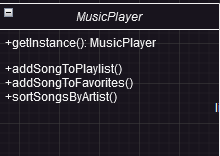


Figura 1 – clasa UML pentru clasa MusicPlayer

Singleton ne-a oferit control asupra creării și utilizării obiectului, asigurând astfel o gestionare eficientă a resurselor și evitarea creării multiple a instanțelor.

In clasa SongFactory am utilizat design patternul Factory Method, care ne permite sa cream obiecte de tip Song fara a specifica clasa concreta a obiectului creat. Prin intermediul metodei factory createSong(), putem crea obiecte Song bazate pe anumite criterii sau logica specifica. Astfel, putem abstractiza procesul de creare a obiectelor Song si putem extinde functionalitatea de creare a Song-urilor prin adaugarea de noi metode factory sau clase derivate. Prin implementarea design patternului Factory Method, am obtinut un cod mai modular, flexibil si usor de extins, deoarece am izolat logica de creare a obiectelor intr-o clasa separata.

class SongFactory {

    createSong(title, artist) {

        return new SongBuilder()

            .withTitle(title)

            .withArtist(artist)

            .build();

    }

}

În clasa SongBuilder, am utilizat design pattern-ul Builder pentru a facilita construcția obiectelor de tip Song. Acest pattern permite crearea pas cu pas a instanțelor complexe de Song prin furnizarea de metode separate pentru setarea diferitelor atribute. Fiecare metodă returnează o instanță a clasei SongBuilder, permițând înlănțuirea metodelor și simplificând procesul de personalizare.

class SongBuilder {

    constructor() {

        this.song = new Song();

    }

    withTitle(title) {

        this.song.title = title;

        return this;

    }

    withArtist(artist) {

        this.song.artist = artist;

        return this;

    }

    build() {

        return this.song;

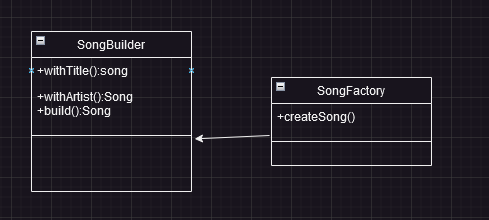


Figura 2 – Diagrama UML pentru clasele SongFactory si SongBuilder

Prin decuplarea logicii de construcție de clasa Song, pattern-ul Builder îmbunătățește lizibilitatea și întreținerea codului, permițând în același timp crearea flexibilă și intuitivă a obiectelor de tip Song.

În cadrul sistemului de music player, am implementat design pattern-ul Facade pentru a oferi o interfață simplificată și unificată pentru funcționalitățile complexe ale sistemului. Prin intermediul clasei Facade, am ascuns detaliile de implementare și am expus un set coerent de metode care permit utilizatorilor să acceseze și să gestioneze funcționalități precum redarea muzicii, adăugarea la playlist sau controlul volumului.

 class PlayerFacade {

    constructor() {

        this.musicPlayer = MusicPlayer.getInstance();

        this.songFactory = new SongFactory();

    }

    addSongToPlaylist(title, artist) {

        const song = this.songFactory.createSong(title, artist);

        this.musicPlayer.addSongToPlaylist(song);

    }

    addSongToFavorites(title, artist) {

        const song = this.songFactory.createSong(title, artist);

        this.musicPlayer.addSongToFavorites(song);

    }

    sortSongsByArtist() {

        this.musicPlayer.sortSongsByArtist();

    }

}

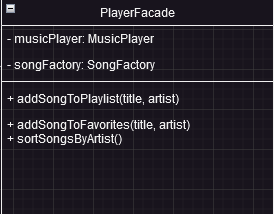


Figura 3 – Clasa UML pentru clasa PlayerFacade

Acest pattern ne-a ajutat să abstractizăm complexitatea sistemului și să oferim o experiență mai intuitivă și ușor de utilizat pentru utilizatori.

În cadrul sistemului Music Player, am implementat design pattern-ul Iterator pentru a permite iterarea prin playlist-uri. Am creat clasa `PlaylistIterator`, care primește un obiect `Playlist` și permite parcurgerea melodiilor din acesta. Astfel, am putut obține o abstracție eficientă pentru a accesa și parcurge melodiile într-un mod simplu și consistent.

class PlaylistIterator {

    constructor(playlist) {

        this.playlist = playlist;

        this.currentIndex = 0;

    }

    hasNext() {

        return this.currentIndex < this.playlist.songs.length;

    }

    next() {

        const song = this.playlist.songs[this.currentIndex];

        this.currentIndex++;

        return song;

    }

}

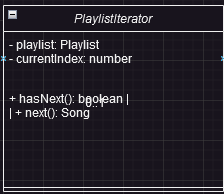


Figura 4 – Clasa UML pentru clasa PlaylistIterator

Prin utilizarea design pattern-ului Iterator, am reușit să separăm logica de parcurgere a playlist-urilor de implementarea internă a acestora, obținând astfel un cod mai modular și ușor de întreținut.

Am utilizat design pattern-ul Memento pentru a stoca și restaura starea sistemului de muzică. Acesta ne permite să salvăm instantanee ale obiectului MusicPlayer și să le recuperăm ulterior, asigurând o gestionare eficientă a stării și posibilitatea de revenire la un punct anterior în timp.

class MusicPlayerCareTaker {

    constructor() {

        this.mementos = [];

    }

    saveState(musicPlayer) {

        const memento = new MusicPlayerMemento(

            musicPlayer.playlist,

            musicPlayer.favoriteSongs

        );

        this.mementos.push(memento);

    }

    restoreState() {

        const lastMemento = this.mementos.pop();

        if (lastMemento) {

            return lastMemento;

        }

        return null;

    }

}

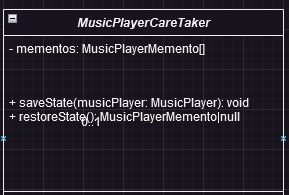


Figura 5 – UML Clasa pentru MusicPlayerCareTaker clasa

# **Concluzie**

Sistemul de muzică creat beneficiază de utilizarea unor design pattern-uri care îmbunătățesc eficiența, modularitatea și ușurința de întreținere a codului. Prin utilizarea design pattern-urilor Singleton, Factory Method, Iterator, Memento și Facade, am putut obține o structură bine organizată și modulară, care permite gestionarea playlist-ului, adăugarea de melodii, sortarea și stocarea stării sistemului. Utilizarea acestor design pattern-uri a condus la un sistem de muzică bine structurat, ușor de utilizat și de întreținut. Prin separarea responsabilităților și utilizarea principiilor de design solid, am creat un sistem scalabil și extensibil, capabil să gestioneze diverse funcționalități ale unui player de muzică.