Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Proiect de curs**

*Disciplina: Tehnici și Mecanisme de Proiectare Software*

**Tema: Analiza și proiectarea aplicații pentru managmentul unei librării de muzică**

Au efectuat: grupa TI-204 Valciuc Andrei

A verificat: asist.univ Mihai Gaidau

Chișinău - 2023

**Cuprins**

[1. Tipuri de modele de design 5](#_Toc137070187)

[1.1 Modele de design creaționale 5](#_Toc137070188)

[1.2 Modele de design structurale 6](#_Toc137070189)

[1.3 Modele de design comportamentale 7](#_Toc137070190)

[2. Realizarea sistemului 8](#_Toc137070191)

[2.1 Analiza temei propuse 8](#_Toc137070192)

[2.2 Implementarea sistemului 9](#_Toc137070193)

[Concluzie 18](#_Toc137070194)

[Bibliografie 19](#_Toc137070195)

**Introducere**

În lumea modernă, software-ul a devenit omniprezent, jucând un rol crucial în aproape toate aspectele vieții noastre. Fie că vorbim despre aplicații mobile, platforme de socializare, sisteme de operare sau aplicații web, toate acestea necesită o proiectare solidă și bine gândită pentru a funcționa eficient și a satisface nevoile utilizatorilor.

Tehnicile și mecanismele de proiectare software reprezintă metodele și instrumentele utilizate pentru a concepe și dezvolta software-ul într-un mod structurat și eficient. Ele au evoluat în mod constant odată cu dezvoltarea tehnologiei și a cerințelor în creștere ale industriei IT. Proiectarea software este un proces complex, care implică luarea deciziilor importante legate de arhitectură, interacțiunea utilizatorului, modularitatea, performanța și multe alte aspecte cheie.Unul dintre principalele obiective ale tehnicilor și mecanismelor de proiectare software este de a crea software modular, flexibil și ușor de întreținut. Prin utilizarea unor metode precum dezvoltarea orientată pe obiecte, proiectarea bazată pe componente sau proiectarea arhitecturală, inginerii software pot construi sisteme software complexe prin împărțirea acestora în componente mai mici și mai ușor de gestionat.O altă preocupare importantă în proiectarea software este asigurarea calității și performanței. Tehnici precum testarea automată, analiza statică a codului, profilarea și optimizarea sunt utilizate pentru a identifica și elimina erorile și ineficiențele din software. Acestea asigură funcționalitatea corectă și performanța optimă a aplicațiilor, contribuind la satisfacția utilizatorilor și la creșterea productivității.În plus, interacțiunea cu utilizatorii este un aspect crucial în proiectarea software. Tehnici de proiectare a interfețelor utilizator grafice (GUI) și principiile de proiectare UX (User Experience) sunt utilizate pentru a crea aplicații prietenoase și intuitive, care să ofere o experiență plăcută și eficientă utilizatorilor.Tehnicile și mecanismele de proiectare software sunt în continuă evoluție, odată cu apariția de noi tehnologii și provocări. Cu toate acestea, ele reprezintă fundamentul dezvoltării software-ului și joacă un rol esențial în crearea de produse și servicii inovatoare.

Pe măsură ce tehnologia continuă să avanseze, este vital ca inginerii software să fie familiarizați cu aceste tehnici și mecanisme de proiectare software și să le aplice în mod corespunzător pentru a crea soluții software fiabile, robuste și scalabile. O proiectare software bine gândită nu doar că asigură funcționalitatea corectă a unei aplicații, ci contribuie și la creșterea eficienței și succesului întreprinderilor și organizațiilor care le dezvoltă.

# 1. Tipuri de modele de design

Există o varietate de tipuri de design patterns utilizate în proiectarea software, fiecare având un scop specific și oferind soluții la probleme comune întâlnite în dezvoltarea aplicațiilor. Printre cele mai cunoscute tipuri de design patterns se numără:

Design Patterns Creational (de creație): Aceste tipuri de design patterns se concentrează pe modul în care obiectele și clasele sunt create și instanțiate. Exemple de design patterns creational includ Singleton, Factory, Abstract Factory și Builder, care oferă metode pentru a crea obiecte în mod eficient și flexibil.

Design Patterns Structural (structurale): Acestea se concentrează pe relațiile și structura dintre obiecte și clase. Aceste design patterns facilitează organizarea și compunerea obiectelor în entități mai mari. Exemple de design patterns structurale includ Adapter, Composite, Decorator și Proxy, care permit interacțiunea și extinderea obiectelor într-un mod flexibil și modular.

Design Patterns Behavioral (de comportament): Aceste tipuri de design patterns se referă la comunicarea și interacțiunea între obiecte și clase. Ele oferă soluții pentru gestionarea comportamentului și fluxului de date într-un sistem software. Exemple de design patterns behaviorale includ Observer, Strategy, State și Command, care permit implementarea unor comportamente și algoritmi flexibili și extensibili.

Fiecare tip de design pattern are propriile sale avantaje și este potrivit pentru anumite situații. Combinația și utilizarea corectă a acestor design patterns pot îmbunătăți calitatea, modularitatea și flexibilitatea aplicațiilor software, facilitând dezvoltarea și întreținerea acestora pe termen lung.

### 1.1 Modele de design creaționale

Modelele de design creationale reprezintă un tip de design pattern care se concentrează pe procesul de creare și instanțiere a obiectelor. Aceste modele oferă soluții și abordări pentru a crea obiecte într-un mod flexibil și eficient, având în vedere diverse cerințe și contexte. Iată câteva exemple de modele de design creationale:

Singleton: Acest model de design are ca scop garantarea faptului că o clasă are o singură instanță în cadrul unei aplicații. Singleton permite accesul global la această instanță și este util atunci când este nevoie de un obiect care să fie partajat și utilizat în întregul sistem.

Factory Method: Acest model de design se bazează pe crearea de obiecte prin intermediul unei metode de fabrică. În loc să instantieze direct obiecte, clasa clientă utilizează metoda de fabrică pentru a crea și returna obiecte specifice, ascunzând detaliile concrete ale instanțierii. Aceasta oferă un nivel suplimentar de flexibilitate și extensibilitate în crearea obiectelor.

Abstract Factory: Modelul Abstract Factory furnizează o interfață pentru crearea unei familii de obiecte legate sau dependente. Utilizând acest model, se poate crea o familie de obiecte care să coexiste și să fie compatibile între ele. Această abordare permite crearea de obiecte în funcție de un anumit context sau variantă, fără a expune detalii specifice.

Builder: Modelul Builder se concentrează pe construirea complexă a obiectelor pas cu pas, fără a expune toate detaliile de construcție într-o singură metodă. Aceasta oferă un mod structurat de a crea obiecte complexe, permițând personalizarea și configurarea acestora într-un mod flexibil.

Prototype: Modelul Prototype se bazează pe clonarea și duplicarea unui obiect existent pentru a crea noi instanțe. Acesta oferă o modalitate de a crea obiecte complexe fără a depinde de o clasă specifică sau a efectua operațiuni costisitoare de inițializare.

Aceste modele de design creationale oferă abordări diferite pentru crearea și instanțierea obiectelor, fiecare având avantajele și utilizările sale specifice. Utilizarea adecvată a acestor modele poate facilita dezvoltarea software-ului, creând obiecte într-un mod eficient și flexibil.

### 1.2 Modele de design structurale

Modelele de design structurale reprezintă un tip de design pattern care se concentrează pe relațiile și structura între obiecte și clase. Aceste modele oferă soluții pentru organizarea și compunerea obiectelor în entități mai mari, permițând astfel dezvoltarea unui software modular și ușor de întreținut. Iată câteva exemple de modele de design structurale:

Adapter: Modelul Adapter este utilizat pentru a permite colaborarea între două clase incompatibile sau cu interfețe diferite. El oferă un adaptor care convertește interfața unei clase într-o altă interfață așteptată de client, facilitând astfel comunicarea și interacțiunea între acestea.

Composite: Modelul Composite permite tratarea obiectelor individuale și a colecțiilor acestora într-un mod uniform. Prin intermediul acestui model, se pot crea structuri de arbori în care fiecare nod poate fi tratat ca un obiect individual sau ca o colecție de noduri similare. Acest lucru facilitează tratarea și manipularea obiectelor complexe într-un mod recursiv.

Decorator: Modelul Decorator oferă posibilitatea de a adăuga funcționalitate suplimentară la un obiect existent fără a modifica structura acestuia. Acest model permite atașarea dinamică a funcționalității noi prin încapsularea obiectului original într-un decorator care îi extinde comportamentul.

Proxy: Modelul Proxy oferă un înlocuitor sau un substitut pentru un obiect real, controlând accesul și comportamentul acestuia. Proxy-ul poate oferi servicii suplimentare, cum ar fi lazy loading (încărcare amânată) a datelor sau securitate, fără a afecta interacțiunea cu obiectul real.

Facade: Modelul Facade oferă o interfață simplificată și unificată pentru a accesa un sistem complex sau un set de clase. Prin intermediul acestui model, se ascunde complexitatea și detaliile de implementare ale sistemului, permițând clientului să interacționeze cu el printr-o interfață simplificată.

Aceste modele de design structurale oferă abordări diferite pentru organizarea și structurarea obiectelor și claselor într-un sistem software. Ele facilitează dezvoltarea unui cod modular, ușor de înțeles și de întreținut, permitând extensibilitatea și flexibilitatea în implementare. Prin utilizarea adecvată a acestor modele, se poate obține o arhitectură software solidă și scalabilă.

### 1.3 Modele de design comportamentale

Modelele de design comportamentale reprezintă un tip de design pattern care se concentrează pe comunicarea și interacțiunea între obiecte și clase în cadrul unei aplicații software. Aceste modele oferă soluții pentru gestionarea comportamentului și fluxului de date într-un sistem, permitând dezvoltarea unui software flexibil și extensibil. Iată câteva exemple de modele de design comportamentale:

Observer: Modelul Observer permite notificarea automată a unui set de obiecte (observatori) atunci când starea unui alt obiect (subiect) se schimbă. Astfel, observatorii pot reacționa și lua măsuri corespunzătoare în funcție de evenimentele produse în subiect.

Strategy: Modelul Strategy oferă o modalitate de a selecta și utiliza diferite algoritme sau strategii în funcție de context sau necesitate. El permite definirea unor familii de algoritmi, încapsulând fiecare într-o clasă separată și permițând înlocuirea acestora în timpul execuției.

State: Modelul State permite obiectelor să-și schimbe comportamentul în funcție de starea internă. El facilitează implementarea unei mașini de stare, unde fiecare stare corespunde unui set specific de acțiuni și comportamente. Schimbarea stării permite obiectului să schimbe și comportamentul asociat.

Command: Modelul Command encapsulează o cerere ca un obiect, permițând astfel parametrizarea clienților cu diverse cereri și manipularea lor într-un mod flexibil. Acest model facilitează separarea clientului care inițiază cererea de obiectul care o primește și o procesează.

Iterator: Modelul Iterator oferă o modalitate de a parcurge elementele unei colecții fără a dezvălui structura internă a acesteia. El definește o interfață comună pentru a accesa elementele într-o manieră secvențială, permițând astfel parcurgerea colecției în mod abstract și uniform.

Aceste modele de design comportamentale oferă abordări diferite pentru gestionarea și organizarea comportamentului și interacțiunilor în cadrul unei aplicații software. Ele facilitează dezvoltarea unui cod modular, extensibil și ușor de întreținut, permițând implementarea de comportamente și logici variate într-un mod flexibil. Prin utilizarea adecvată a acestor modele, se poate obține o arhitectură software eficientă și ușor de gestionat.

# 2. Realizarea sistemului

### 2.1 Analiza temei propuse

Alegerea de a dezvolta o aplicație de management a unei biblioteci muzicale este justificată de numeroasele beneficii și oportunități pe care le oferă. Prin intermediul acestei aplicații, utilizatorii vor putea organiza și gestiona colecția lor de muzică într-un mod eficient și ușor de utilizat. Unul dintre avantajele principale ale unei astfel de aplicații este capacitatea de a avea o vedere de ansamblu asupra întregii biblioteci muzicale. Utilizatorii pot adăuga, căuta și vizualiza piese, albume și artiști într-un mod centralizat și organizat. Aceasta facilitează găsirea rapidă a muzicii dorite și permite utilizatorilor să-și creeze propriile liste de redare și mixuri personalizate. De asemenea, o aplicație de management a unei biblioteci muzicale poate oferi opțiuni avansate de sortare și filtrare a muzicii. Utilizatorii pot ordona piesele în funcție de diferite criterii, cum ar fi titlu, artist sau gen muzical. Aceasta facilitează navigarea și explorarea colecției muzicale într-un mod flexibil și personalizat. Un alt beneficiu al unei astfel de aplicații este posibilitatea de a adăuga și gestiona informații suplimentare despre piese, cum ar fi versurile sau datele despre artist. Aceste detalii suplimentare pot îmbogăți experiența de ascultare a muzicii și pot oferi context și înțelegere mai profundă a creațiilor muzicale. De asemenea, aplicația poate oferi funcționalități avansate, precum crearea de liste de redare personalizate, partajarea muzicii cu prietenii sau accesul la servicii de streaming muzical. Aceste caracteristici adiționale pot îmbunătăți și extinde funcționalitatea aplicației, oferind utilizatorilor o experiență mai bogată și mai captivantă în gestionarea și ascultarea muzicii.

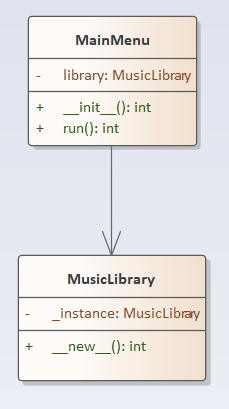
prin urmare, dezvoltarea unei aplicații de management a unei biblioteci muzicale are ca scop simplificarea și organizarea eficientă a colecției de muzică. Aceasta oferă utilizatorilor o experiență mai plăcută și mai personalizată în ascultarea și explorarea muzicii, facilitând accesul la diverse funcționalități și informații suplimentare despre piese și artiști.

### 2.2 Implementarea sistemului

În cadrul acestui proiect au fost implemntate 6 modele de design printre care sunt 2 modele creaționale: singleton și factory, structurale: decorator și composite, comportamentale: Iterator și strategy.

**Singleton:**

Singleton este un design pattern care se concentrează pe crearea unei singure instanțe a unei clase și asigurarea că această instanță este accesibilă global. Acesta se realizează prin restricționarea creării multiple a instanțelor și furnizarea unei metode statice pentru a accesa acea instanță unică.



**Figura 2.1 –** Diagrama UML de clasă pentru Singleton

Iată secvența de cod în care este implementat Singleton în clasă MusicLibrary:

class MusicLibrary:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

cls.\_instance.songs = []

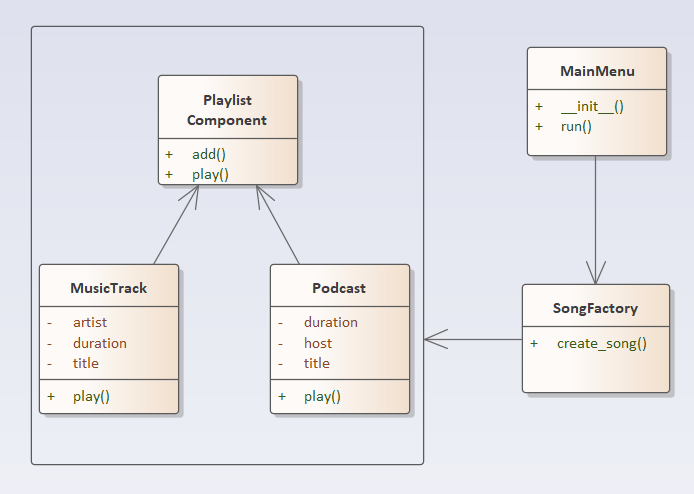
cls.\_instance.playlists = []

return cls.\_instance

Variabila de clasă \_instance este inițializată cu None. Această variabilă va stoca instanța unică a clasei MusicLibrary. Metoda \_\_new\_\_ este suprascrisă pentru a controla crearea instanțelor și a returna întotdeauna instanța unică existentă sau nou-creată. În interiorul metodei \_\_new\_\_, se verifică dacă cls.\_instance este None, ceea ce înseamnă că instanța unică nu a fost creată încă. Dacă instanța unică nu există, se creează o nouă instanță a clasei MusicLibrary utilizând metoda super().\_\_new\_\_(cls). Apoi, variabila \_instance este actualizată cu noua instanță creată. Indiferent dacă instanța unică este nou-creată sau existentă, metoda \_\_new\_\_ returnează întotdeauna cls.\_instance, asigurându-se că întotdeauna se returnează aceeași instanță unică.

**Factory:**

Patternul Factory este utilizat în codu pentru a crea obiecte de tipul MusicTrack sau Podcast în funcție de un anumit tipul ales. Aceasta oferă o modalitate de a separa logica de creare a obiectelor de restul codului.



**Figura 2.2 –** Diagrama de clasă pentru Factory

Mai jos poate fi observată secvența de cod în care este implementat Factory:

class PlaylistComponent:

def add(self, component):

raise NotImplementedError()

def play(self):

raise NotImplementedError()

class MusicTrack(PlaylistComponent):

def \_\_init\_\_(self, title, artist, duration):

self.title = title

self.artist = artist

self.duration = duration

def play(self):

print("Now playing:", self.title, "by", self.artist)

class Podcast(PlaylistComponent):

def \_\_init\_\_(self, title, host, duration):

self.title = title

self.host = host

self.duration = duration

def play(self):

print("Now playing podcast:", self.title, "hosted by", self.host)

class SongFactory:

def create\_song(self, song\_type, title, artist, duration):

if song\_type == "song":

return MusicTrack(title, artist, duration)

elif song\_type == "podcast":

return Podcast(title, artist, duration)

else:

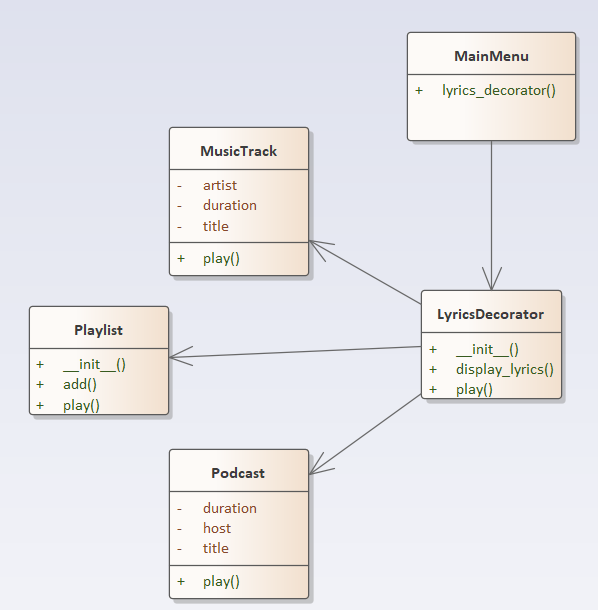
raise ValueError("Invalid song type.")

În secvența de cod de mai susclasa SongFactory acționează ca o fabrică pentru crearea obiectelor de tip MusicTrack și Podcast pe baza unui anumit song\_type. Prin intermediul metodei create\_song, se ia decizia despre tipul de obiect care trebuie creat pe baza valorii argumentului song\_type. Dacă song\_type este "song", se creează și se returnează un obiect MusicTrack, iar dacă song\_type este "podcast", se creează și se returnează un obiect Podcast. În cazul în care valoarea song\_type nu este nici "song", nici "podcast", se aruncă o excepție pentru a semnala faptul că tipul de obiect cerut nu este valid.

**Decorator:**

Decoratorul este un pattern de proiectare utilizat pentru a extinde funcționalitatea unui obiect existent fără a modifica codul original al acestuia. Acesta permite adăugarea de comportamente suplimentare înainte sau după apelul metodelor obiectului de bază.

Prin intermediul decoratorului, se încapsulează obiectul original într-un obiect decorat care conține comportamentul suplimentar dorit. Decoratorul implementează aceeași interfață ca și obiectul de bază, astfel încât poate fi folosit în locul acestuia în codul existent.



**Figura 2.3 –** Diagrama UML de clasă pentru Decorator

class MusicTrack(PlaylistComponent):

def init(self, title, artist, duration):

self.title = title

self.artist = artist

self.duration = duration

def play(self):

print("Now playing:", self.title, "by", self.artist)

class LyricsDecorator(PlaylistComponent):

def init(self, component, lyrics):

self.component = component

self.lyrics = lyrics

def play(self):

self.component.play()

self.display\_lyrics()

def display\_lyrics(self):

print("Lyrics:\n", self.lyrics)

In acest exemplu, clasa `LyricsDecorator` este un decorator pentru clasa `MusicTrack`. Decoratorul adauga comportament suplimentar la metoda `play` a obiectului `MusicTrack`. Cand apelam metoda `play` pe obiectul `decorated\_track`, mai intai se executa metoda `play` a obiectului `MusicTrack`, iar apoi se afiseaza si versurile melodiei prin apelul metodei `display\_lyrics` din decorator.

Astfel, decoratorul ne permite sa adaugam functionalitati suplimentare la un obiect fara a modifica direct clasa obiectului initial. Prin adaugarea decoratorului, putem extinde sau modifica comportamentul obiectului initial in mod dinamic, la runtime.

**Composite:**

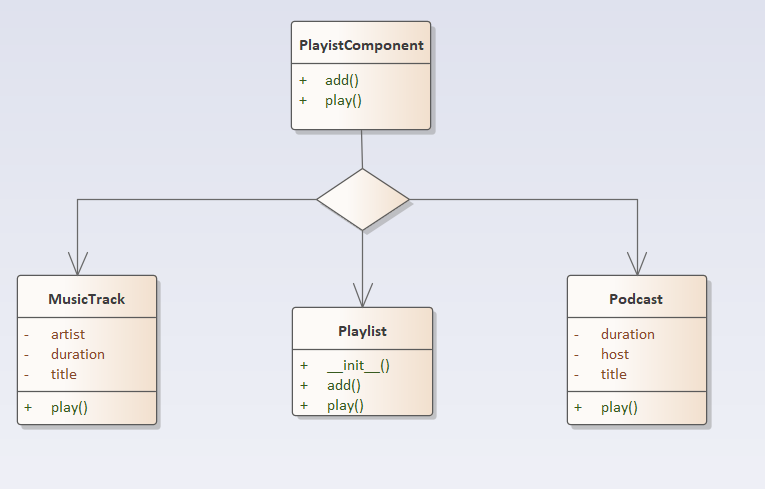
Composite pattern este utilizat în programare pentru a crea ierarhii de obiecte într-un mod recursiv, astfel încât obiectele individuale și grupurile de obiecte să fie tratate în mod uniform.

Principala utilizare a Composite pattern constă în construirea unei structuri de tip arbore, în care obiectele individuale și grupurile de obiecte pot fi tratate în același mod. Aceasta permite crearea de ierarhii complexe de obiecte, în care un obiect individual poate conține sau să fie format din alte obiecte.

Prin intermediul Composite pattern, se poate trata un obiect individual și un grup de obiecte în același fel, fără a fi necesară o logică specială pentru fiecare tip de obiect. Aceasta facilitează gestionarea și manipularea structurilor complexe de obiecte.

De asemenea, Composite pattern facilitează adăugarea și eliminarea dinamică a obiectelor în structură, deoarece atât obiectele individuale, cât și grupurile de obiecte respectă aceeași interfață comună. Astfel, se poate adăuga sau elimina obiecte într-un mod transparent pentru client.

În rezumat, Composite pattern este utilizat pentru a crea ierarhii de obiecte în care obiectele individuale și grupurile de obiecte sunt tratate în mod uniform, permițând gestionarea și manipularea structurilor complexe de obiecte într-un mod flexibil și dinamic.



**Figura 2.4 –** Diagrama UML de clasă pentru Composite

Mai jos putem observa secvența de cod în care este implemntat Composite:

class PlaylistComponent:

def add(self, component):

raise NotImplementedError()

def play(self):

raise NotImplementedError()

class Playlist(PlaylistComponent):

def \_\_init\_\_(self, name):

self.name = name

self.playlist = []

def add(self, component):

self.playlist.append(component)

def play(self):

print("Playlist:", self.name)

for item in self.playlist:

item.play()

class MusicTrack(PlaylistComponent):

def \_\_init\_\_(self, title, artist, duration):

self.title = title

self.artist = artist

self.duration = duration

def play(self):

print("Now playing:", self.title, "by", self.artist)

class Podcast(PlaylistComponent):

def \_\_init\_\_(self, title, host, duration):

self.title = title

self.host = host

self.duration = duration

def play(self):

print("Now playing podcast:", self.title, "hosted by", self.host)

În codul dat, Composite pattern este utilizat pentru a crea o structură ierarhică de componente în cadrul unui playlist muzical. Aceasta permite tratarea atât a componentelor individuale (leaf), cum ar fi piese muzicale și podcast-uri, cât și a compozițiilor (playlist-uri) ca entități uniforme. În cadrul proiectului, clasa PlaylistComponent reprezintă interfața comună pentru toate componentele din playlist. Aceasta definește două metode abstracte, add și play, care vor fi implementate de clasele concrete.

Clasa MusicTrack reprezintă o componentă individuală (leaf) care reprezintă o piesă muzicală. Aceasta implementează interfața PlaylistComponent și oferă o implementare a metodei play specifică pentru redarea pieselor muzicale.

Clasa Podcast reprezintă o altă componentă individuală (leaf) care reprezintă un podcast. Aceasta, de asemenea, implementează interfața PlaylistComponent și oferă o implementare a metodei play specifică pentru redarea podcast-urilor.

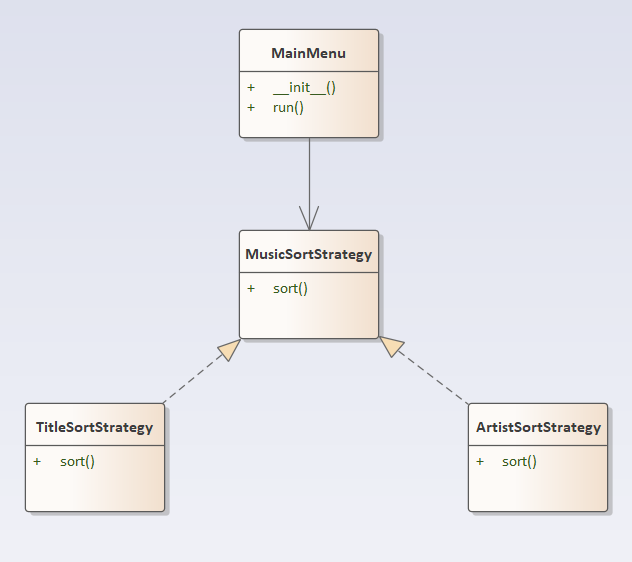
Clasa Playlist reprezintă o compoziție, adică un playlist care poate conține atât piese muzicale, cât și podcast-uri. Aceasta implementează interfața PlaylistComponent și oferă metode pentru adăugarea de componente în playlist (add) și redarea întregului playlist (play). Componentele individuale (leaf) și compozițiile (playlists) sunt stocate într-o listă în cadrul clasei Playlist.

Astfel, putem crea playlist-uri complexe prin adăugarea de componente individuale (leaf) și compoziții (playlists) într-un playlist principal. Atunci când apelăm metoda play a playlist-ului principal, acesta va reda în mod recursiv toate componentele conținute, fie ele piese muzicale sau podcast-uri.

Composite pattern ne permite să tratăm uniform atât componente individuale (leaf), cât și compoziții (playlists), permițând o structură ierarhică flexibilă și ușor de gestionat în cadrul proiectului.

**Strategy:**

Strategy pattern este utilizat pentru a defini o familie de algoritmi, împachetând fiecare algoritm într-o clasă separată și făcându-le interschimbabile. Acest pattern permite selecția și schimbarea dinamică a algoritmului utilizat în timpul executării programului, fără a afecta codul clientului. Prin utilizarea strategy pattern, putem izola diferitele algoritme în clase separate, numite strategii, și putem crea o interfață comună pentru acestea. Astfel, putem crea obiecte strategie care implementează această interfață comună și pot fi schimbate în timpul rulării programului fără a modifica codul clientului.



**Figura 2.5 –** Diagrama UML de clasă pentru Strategy

În continuare putem observa secvența de cod în care este impelmentat strategy pattern:

Mai întâi, definim o interfață numită MusicSortStrategy, care declară o metodă sort() pentru sortarea muzicii:

class MusicSortStrategy:

def sort(self, songs):

pass

În continuare, creăm clase de strategie concrete care implementează această interfață. Să luăm în considerare două strategii: SortByTitleStrategy și SortByArtistStrategy:

class SortByTitleStrategy(MusicSortStrategy):

def sort(self, songs):

return sorted(songs, key=lambda x: x.title)

class SortByArtistStrategy(MusicSortStrategy):

def sort(self, songs):

return sorted(songs, key=lambda x: x.artist)

În clasa MainMenu, introducem o metodă sort\_music() în care este selectată și aplicată strategia de sortare:

class MainMenu:

def \_\_init\_\_(self):

self.library = MusicLibrary()

self.sort\_strategy = SortByTitleStrategy()

def sort\_music(self):

print("\nSort music by:")

print("1. Title")

print("2. Artist")

choice = input("Enter your choice: ")

if choice == "1":

self.sort\_strategy = SortByTitleStrategy()

elif choice == "2":

self.sort\_strategy = SortByArtistStrategy()

else:

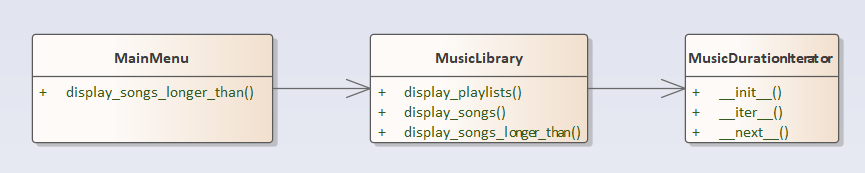
print("Invalid choice. Using default sorting strategy.")

self.library.songs = self.sort\_strategy.sort(self.library.songs)

Aici, metoda sort\_music() solicită utilizatorului criteriul de sortare preferat și atribuie obiectul strategiei corespunzător atributului sort\_strategy. Apoi, apelează metoda sort() pe strategia selectată pentru a efectua operația de sortare pe lista de melodii. Folosind modelul Strategy, decuplăm algoritmul de sortare de clasa MainMenu și permitem selecția dinamică și interschimbabilitatea strategiilor de sortare. Acest lucru promovează flexibilitatea și mentenabilitatea, deoarece noi strategii de sortare pot fi adăugate cu ușurință fără a modifica codul existent.

**Iterator:**

Iterator pattern este utilizat pentru a itera (parcurge) o colecție de obiecte într-un mod uniform, fără a expune detalii de implementare ale colecției subiacente. Acesta oferă o interfață comună pentru parcurgerea elementelor unei colecții, indiferent de structura internă a colecției.



**Figura 2.6 –** Diagrama de clasă pentru iterator.

class MusicDurationIterator:

def \_\_init\_\_(self, songs, duration\_threshold):

self.songs = songs

self.duration\_threshold = duration\_threshold

self.index = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

while self.index < len(self.songs):

song = self.songs[self.index]

self.index += 1

if float(song.duration) > self.duration\_threshold:

return song

raise StopIteration

În acest cod, clasa MusicDurationIterator reprezintă un iterator care permite parcurgerea melodiilor din biblioteca de muzică în funcție de o durată minimă specificată. Aceasta implementează interfața Iterator și conține metodele \_\_next\_\_() și \_\_iter\_\_() pentru a permite iterația prin colecție.

Iteratorul MusicDurationIterator primește ca argumente o listă de melodii și o durată minimă. La fiecare apel al metodei \_\_next\_\_(), iteratorul verifică dacă melodia curentă are o durată mai mare decât valoarea minimă specificată. Dacă da, melodia este returnată și iteratorul trece la următoarea melodie din listă. Dacă nu, iteratorul continuă cu următoarea melodie până când găsește o melodie care îndeplinește condiția sau se termină lista. Clasa MusicLibrary utilizează acest iterator pentru a returna o instanță a MusicDurationIterator prin metoda get\_songs\_longer\_than(). Aceasta primește o valoare de durată și returnează un iterator care poate fi folosit pentru a parcurge melodiile din bibliotecă care au o durată mai mare decât valoarea specificată. În clasa MainMenu, metoda display\_songs\_longer\_than() utilizează iteratorul pentru a itera prin melodiile mai lungi decât o anumită durată și pentru a le afișa prin apelul metodei play() a fiecărei melodii.

# Concluzie

În concluzie, analiza și proiectarea aplicației pentru managementul unei librării muzicale și implementarea a design pattern-uri au adus multiple beneficii în ceea ce privește organizarea și eficiența codului, flexibilitatea și extensibilitatea aplicației.

În cadrul obiectivelor noastre, am acoperit trei categorii principale de design pattern-uri: modele creationale, structurale și comportamentale. Prin utilizarea acestor pattern-uri, am obținut o arhitectură mai robustă și modulară, care a facilitat dezvoltarea și întreținerea aplicației.

Modelele creationale, precum Singleton și Factory, ne-au oferit mecanisme pentru a crea și gestiona în mod eficient instanțele de obiecte. Singleton ne-a permis să avem o singură instanță a clasei MusicLibrary, asigurând coerența și centralizarea accesului la resursele muzicale. Factory a furnizat o abordare unificată pentru crearea cântecelor și podcast-urilor, eliminând complexitatea de a alege constructorul adecvat în diverse situații.

Modelele structurale, cum ar fi Composite și Decorator, ne-au ajutat să gestionăm colecțiile de cântece și playlist-uri într-un mod eficient și flexibil. Composite a permis tratarea atât a cântecelor individuale, cât și a playlist-urilor ca și componente ale unei ierarhii unificate, oferind posibilitatea de a lucra cu ele în mod consistent. Decorator a extins funcționalitatea componentelor de bază, permițând adăugarea de comportamente suplimentare (cum ar fi afișarea versurilor) fără a modifica direct clasele existente.

Modelele comportamentale, cum ar fi Iterator și Strategy, ne-au furnizat mecanisme pentru gestionarea și manipularea comportamentului obiectelor. Iterator a facilitat iterarea și filtrarea colecțiilor de cântece, oferind o interfață simplă și consistentă pentru a accesa și prelucra elementele colecției. Strategy a permis schimbarea dinamică a strategiei de sortare a cântecelor, asigurând flexibilitatea și reutilizabilitatea procesului de sortare.

În ansamblu, implementarea acestor design pattern-uri a adus beneficii semnificative în ceea ce privește structurarea și eficiența codului, flexibilitatea și extensibilitatea aplicației de management al librăriei muzicale. Aceste pattern-uri ne-au permis să abordăm probleme specifice într-un mod elegant și modular, facilitând dezvoltarea, întreținerea și adaptarea aplicației pe termen lung.

# Bibliografie

<https://refactoring.guru/design-patterns>

<https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm>