Ministerul Educației și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică Departamentul Ingineria Software și Automatică

**Proiect de an**

**Disciplina: “Tehnici și mecanisme de proiectare software”**

**Tema: “Bibliotecă online”**

A efectuat: Cucuietu Ion, TI-202

A verificat: Prof. univ. Postaru Andrei

Conf. univ Gaidau Mihai

Chișinău - 2023

Cuprins

[Introducere 3](#_Toc136770574)

[1.Analiza domeniului de studiu 6](#_Toc136770575)

[1.1 Importanța temei 7](#_Toc136770576)

[1.2 Scopul și obiectivele sistemului 9](#_Toc136770577)

[2. Realizarea sistemului 10](#_Toc136770578)

[2.1 Singleton 15](#_Toc136770579)

[2.2 Abstract Factory 18](#_Toc136770580)

[2.3 Adapter 21](#_Toc136770581)

[2.4 Decorator 23](#_Toc136770582)

[2.5 Iterator 28](#_Toc136770583)

[2.6 State 30](#_Toc136770584)

[Concluzii: 35](#_Toc136770585)

# Introducere

În lumea tehnologiei și a dezvoltării software, programatorii au un rol esențial în crearea aplicațiilor și a sistemelor informatice care ne înconjoară. Aceștia sunt arhitecții și constructorii digitali care traduc ideile și cerințele în cod funcțional. Programatorii nu doar scriu linii de cod, ci și rezolvă probleme complexe, optimizează performanța și asigură calitatea software-ului.

Rolul programatorului:

Programatorii au responsabilitatea de a transforma concepte abstracte în soluții practice. Ei lucrează îndeaproape cu analiștii și designerii pentru a înțelege cerințele și pentru a crea aplicații și produse software care să satisfacă nevoile utilizatorilor. Programatorii sunt responsabili de scrierea codului, de testarea și de depanarea acestuia, asigurându-se că aplicația funcționează așa cum a fost proiectată și că este lipsită de erori.

Design Patterns și importanța lor:

Design Patterns, sau modele de proiectare, sunt un set de soluții recunoscute și validate pentru problemele comune în dezvoltarea software. Acestea oferă programatorilor un ghid și o structură pentru a crea aplicații eficiente, flexibile și ușor de întreținut. Importanța Design Patterns constă în faptul că acestea permit programatorilor să capitalizeze experiența și cunoștințele acumulate de alții în domeniul dezvoltării software.

Prin utilizarea Design Patterns, programatorii pot rezolva problemele recurente într-un mod standardizat și optimizat, reducând astfel erorile și îmbunătățind calitatea software-ului. Aceste modele oferă o modalitate de reutilizare a soluțiilor de succes, ceea ce duce la creșterea productivității și eficienței în dezvoltarea software.

De asemenea, Design Patterns facilitează comunicarea și colaborarea între programatori și echipele de dezvoltare. Prin utilizarea unor concepte și terminologii standardizate, programatorii pot comunica mai eficient și pot colabora în mod eficient pentru a construi aplicații complexe și robuste.

Importanţa şabloanelor (standardelor) în construirea sistemelor complexe a fost de mult recunoscută în alte discipline. În cadrul comunităţii proiectanţilor de software OOP (orientate pe obiecte) ideea de a aplica şabloane se pare că a fost inspirată de propunerea unui arhitect, Christopher Alexander, care a lansat iniţiativa folosirii unui limbaj bazat pe șabloane pentru proiectarea clădirilor şi a orașelor. Acesta afirmă că: "Fiecare şablon descrie o problemă care apare mereu în domeniul nostru de activitate şi indică esenţa soluţiei acelei probleme într-un mod care permite utilizarea soluţiei de nenumărate ori în contexte diferite".

Deşi în domeniul sistemelor OO soluţiile sunt exprimate în termeni de obiecte şi interfeţe (în loc de ziduri, uşi, grinzi etc), esenţa noţiunii de sablon este aceeaşi, adică de soluţie a unei probleme într-un context dat.

Şabloanele de proiectare sunt o memorare pentru posteritate a experienţei în domeniul proiectării sistemelor OOP.

Un şablon este descris de patru elemente:

* **Nume:** foloseşte pentru identificare; descrie sintetic problema rezolvată de şablon şi soluţia.
* **Problema:** descrie când se aplică şablonul; se descrie problema şi contextul.
* **Solutia:** descrie elementele care intră în rezolvare, relaţiile între ele, responsabilităţile lor şi colaborările între ele.
* **Consecinţe si compromisuri**: implicaţiile folosirii şablonului, costuri şi beneficii. Acestea pot privi impactul asupra flexibilităţii, extensibilităţii sau portabilităţii sistemului, după cum pot să se refere la aspecte ale implementării sau limbajului de programare utilizat. Compromisurile sunt de cele mai multe ori legate de spaţiu şi timp.

Sabloanele de proiectare rezolva multe din problemele zilnice cu care se confrunta proiectantii. Cateva din aceste probleme sunt urmatoarele:

**Gasirea obiectelor adecvate**

Asa cum se stie, un obiect, care este caramida de baza a unui sistem OO, include atat date, cat si metode (operatii) care opereaza asupra datelor. Obiectul execută o operație când primește o cerere (mesaj) de la un client.

Mesajele reprezinta singura cale prin care un obiect este determinat sa execute o operatie, in timp ce operatiile sunt singurul mod de a modifica datele interne ale obiectului. Din cauza acestor restrictii starea interna a obiectului se spune ca este incapsulata: ea nu poate fi accesata direct, iar reprezentarea ei este invizibila dinspre exteriorul obiectului.

Partea dificila in proiectarea unui sisitem OO este descompunerea sistemului in obiecte. Aceasta deoarece procesul este influentat de mai mulți factori care actioneaza adesea in mod contradictoriu: incapsularea, granularitatea, dependentele, flexibilitatea, performantele, evolutia, gradul de reutilizare etc.

**Determinarea granularitatii obiectelor**

Obiectele ce compun un sistem pot varia "ingrozitor" ca marime si numar. Ele pot reprezenta practic orice: de la componente hardware pana la aplicatii intregi. Este dificil de stabilit unde trebuie sa se "opreasca" un obiect.

Exista sabloane care acopera si acest aspect. Astfel, sablonul Facade descrie modul in care subsisteme complete pot fi reprezentate ca obiecte, iar sablonul Flyweight arata cum se poate gestiona un numar urias de obiecte la nivelele cele mai fine de granularitate. Alte sabloane descriu caile prin care un obiect poate fi descompus in obiecte mai mici. Abstract Factory si Builder reprezinta obiecte a caror unica responsabilitate este crearea de alte obiecte. Visitor si Command reprezinta obiecte a caror unica responsabilitate este implementarea unui mesaj catre alt obiect sau grup de obiecte.

**Specificarea interfetelor obiectelor**

Pentru fiecare operatie declarata intr-un obiect se precizeaza numele, obiectele pe care le ia ca parametri si valoarea returnata; aceste elemente formeaza semnatura operatiei. Multimea tuturor semnaturilor corespunzatoare operatiilor dintr-un obiect reprezinta interfata obiectului. Interfata unui obiect descrie complet setul mesajelor care pot fi trimise spre obiectul respectiv. Un tip este un nume utilizat pentru a referi o anumita interfata. Astfel, vom spune despre un obiect ca este de tipul Window daca el accepta toate mesajele corespunzatoare operatiilor definite in interfata numita Window. Ca urmare, un obiect poate avea mai multe tipuri, adica o parte a interfetei sale poate fi de un tip, iar alta parte - de alt tip. De asemenea, mai multe obiecte pot partaja un anumit tip comun, daca interfetele lor includ tipul respectiv.

În acest proiect de an, vom explora diverse tipuri de Design Patterns și vom evidenția modul în care acestea pot fi aplicate în practică. Vom examina Design Patterns de creare, structurale și comportamentale și vom prezenta exemple concrete de implementare a acestora în dezvoltarea aplicației, și anume dezvoltarea unei biblioteci online.

# 1.Analiza domeniului de studiu

Proiectul dat face parte din sfera tehnologiilor informaționale, având la bază tema descrisă în subcapitolul anterior adică dezvoltarea unei biblioteci online. Această aplicație va fi una realizată în limbjul C# fiind realizată pentru rularea pe dispozitivele desktop. Aplicația portă un caracter economic fiind compatibilă cu sfera serviciilor astfel clientul va putea intra in biblioteca online și va putea vedea ce cărți sunt în disponibile. Această aplicație va avea ca scop automatizarea procesului de informare a stocului bibliotecii.

Aplicația va fi una desktop fiind disponibilă de fapt pe toate dispozitivele care rulează pe sistemul de operare Windows, am ales acest sistem din motivul că marea majoritate dintre utilizatorii de calculator optează pentru acest sistem de operare prin urmare fiind unul foarte popular.

Lansat publicului in iunie 2000 si oficial in primavara anului 2002, C# este un limbaj de

programare care combina facilitati testate de-a lungul timpului cu inovatii de ultim moment. Creatorii acestui limbaj au fost o echipa de la firma Microsoft condusa de Anders Hejlsberg. Desi limbajul este creat de Microsoft, acesta nu este destinat doar platformelor Microsoft. Compilatoare C# exista si pentru alte sisteme precum Linux sau Macintosh. Creat ca instrument de dezvoltare pentru arhitectura .NET, limbajul ofera o modalitate facila si eficienta de a scrie programe pentru sistemul Windows, internet, componente software etc. C# deriva din doua dintre cele mai de succes limbaje de programare: C si C++.

De asemenea, limbajul este o “ruda” apropiata a limbajului Java. Pentru o mai buna intelegere a limbajului C# este interesant de remarcat care este natura relatiilor acestuia cu celelalte trei limbaje mentionate mai sus. Pentru aceasta, vom plasa mai intai limbajul C# in contextul istoric determinat de cele trei limbaje.

Pentru îndeplinirea sarcinii propuse am ales limbajul de programre C# din o serie de motive

* C# este un limbaj de programare orientat-obiect .
* Deține biblioteci grafice bogate.
* Datorită faptului că este un limbaj OOP este perfect pentru implementarea șabloanelor

studiate.

* Posibilitatea utilizării instrumentului Visual Studio, care este unul bine structurat și puternic.

Ca instrument am utilizat Visual Studio care este perfect adaptat pentru scrierea codului în C# dar cel mai improtant este un bun instrument pentru aplicarea bibliotecilor grafice.

Visual Studio include un set complet de instrumente de dezvoltare pentru generarea de aplicații ASP.NET, Servicii Web XML, aplicații desktop și aplicații mobile. Visual Basic, Visual C++, Visual C# și Visual J# toate folosesc același mediu de dezvoltare integrat (IDE) care le permite partajarea instrumentelor și facilitează crearea de soluții folosind mai multe limbaje de programare. Aceste limbaje permit să beneficieze de caracteristicile .NET Framework care oferă acces la tehnologii cheie care simplifica dezvoltarea de aplicații web ASP și XML Web Services cu Visual Web Developer.

Prezentare generală a beneficiilor IDE Visual Studio

* Codificare precisă Cu Visual Studio IDE, utilizatorilor li se oferă asistență de codare live, indiferent de limbajul de programare pe care îl utilizează.
* Debugging rapid
* Testare riguroasă
* Colaborarea echipei
* Opțiuni de personalizare

## 1.1 Importanța temei

Tehnologia informațională a avansat rapid în ultimele decenii, oferind oportunități și soluții inovatoare în diverse domenii. Bibliotecile nu fac excepție, întrucât au adoptat noile tehnologii pentru a-și îmbunătăți serviciile și a răspunde nevoilor utilizatorilor într-un mod mai eficient. Unul dintre aceste progrese tehnologice deosebit de relevante este crearea bibliotecilor online, care permit accesul facil la resursele bibliotecii, inclusiv vizualizarea și împrumutul cărților și DVD-urilor. Menționăm importanța temei alese, evidențiind avantajele și beneficiile aduse de crearea unei biblioteci online.

**Accesibilitate extinsă:**

Una dintre cele mai mari provocări cu care se confruntă utilizatorii bibliotecilor tradiționale este restricționarea accesului fizic la resursele bibliotecii. Cu toate acestea, o bibliotecă online elimină aceste restricții, oferind un acces extins și global la resursele disponibile. Utilizatorii pot vizualiza și accesa cărți și DVD-uri de la distanță, prin intermediul unui dispozitiv conectat la internet. Această accesibilitate extinsă facilitează în mod semnificativ procesul de cercetare și studiu, aducând oamenii mai aproape de cunoaștere.

**Disponibilitatea 24/7:**

O bibliotecă online funcționează non-stop, oferind utilizatorilor posibilitatea de a accesa resursele bibliotecii la orice oră din zi sau noapte. Acest aspect este deosebit de benefic pentru studenți, cercetători sau oricine altcineva care are nevoie de informații în afara orelor obișnuite de funcționare ale bibliotecii tradiționale. Prin intermediul unei biblioteci online, utilizatorii pot explora și împrumuta cărți și DVD-uri în funcție de propriul lor program și ritm de lucru.

**Gestiune eficientă a resurselor:**

O bibliotecă online oferă un sistem de gestionare eficient al resurselor. Informațiile despre cărți și DVD-uri sunt stocate într-o bază de date centrală, ușor de actualizat și de gestionat. Aceasta permite bibliotecarilor să monitorizeze și să controleze stocul de resurse în timp real, să adauge sau să elimine titluri și să ofere informații actualizate utilizatorilor. Prin automatizarea acestor procese, biblioteca online devine mai eficientă și mai flexibilă în administrarea resurselor sale.

**Personalizare și recomandări:**

Bibliotecile online oferă posibilitatea de a personaliza experiența utilizatorilor și de a recomanda resurse relevante în funcție de interesele și preferințele lor. Prin intermediul unor algoritmi avansați, biblioteca online poate oferi sugestii personalizate de cărți și DVD-uri, bazate pe istoricul de navigare și împrumut al utilizatorilor. Această funcționalitate adaugă valoare experienței utilizatorilor, facilitând descoperirea și accesul la resursele potrivite pentru nevoile lor individuale.

**Economie de timp și efort:**

O bibliotecă online reduce semnificativ timpul și efortul necesare pentru a găsi și a împrumuta cărți și DVD-uri. Utilizatorii nu mai sunt obligați să caute fizic în rafturile bibliotecii sau să aștepte la ghișeele de împrumut. Prin intermediul unei interfețe online intuitive, căutările pot fi efectuate rapid, iar procesul de împrumut poate fi finalizat cu câteva clicuri. Aceasta permite utilizatorilor să se concentreze mai mult pe explorarea și învățarea conținutului resurselor, economisind timp prețios.

**Îmbunătățirea colaborării și partajării:**

Biblioteca online facilitează colaborarea și partajarea de resurse între utilizatori. Utilizatorii pot recomanda cărți și DVD-uri unii altora, pot crea liste de lectură sau pot discuta și interacționa pe baza conținutului disponibil. Aceasta promovează schimbul de cunoștințe și sprijină comunitatea de utilizatori în procesul lor de învățare și cercetare.

Crearea unei biblioteci online pentru vizualizarea și împrumutul cărților și DVD-urilor aduce numeroase beneficii și avantaje atât utilizatorilor, cât și bibliotecilor. Accesibilitatea extinsă, disponibilitatea non-stop, gestionarea eficientă a resurselor, personalizarea și recomandările, economia de timp și efortul, precum și îmbunătățirea colaborării și partajării sunt doar câteva dintre aspectele cheie care fac din această temă o opțiune valoroasă în contextul tehnologiilor informaționale actuale. Prin adoptarea unei biblioteci online, bibliotecile pot oferi servicii mai eficiente, iar utilizatorii pot beneficia de un acces ușor și comod la cunoaștere și informație.

## 1.2 Scopul și obiectivele sistemului

**Scopul**

Scopul acestei aplicații este de a realiza o interfață grafică cât mai accesibilă utilizatorului în scopul simplificării procesului de utilizare a bibliotecii online. Partea funcțională a aplicației are ca scop principal implementarea șabloanelor studiate în cadrul disciplinei TMPS pentru gestionarea și afișarea resurselor disponibile în bibliotecă.

**Obiectivele**

* Realizarea unei aplicații care să asigure implementarea șabloanelor de programare

studiate.

* Realizarea unei interfețe accesibile utilizatorului și ușor de înțeles.
* Îndeplinirea tuturor funcționalităților de bază ale aplicației

pentru îndeplinirea comenzii.

* Realizarea unei aplicații actuale.
* Oferirea de extensibilitate și scalabilitate. Proiectul va fi dezvoltat luând în considerare posibilitatea de extindere și adăugarea de noi funcționalități pe viitor. Arhitectura aplicației va fi concepută în mod modular, permitând adăugarea ușoară de noi caracteristici și integrarea cu alte sisteme sau platforme.

# 2. Realizarea sistemului

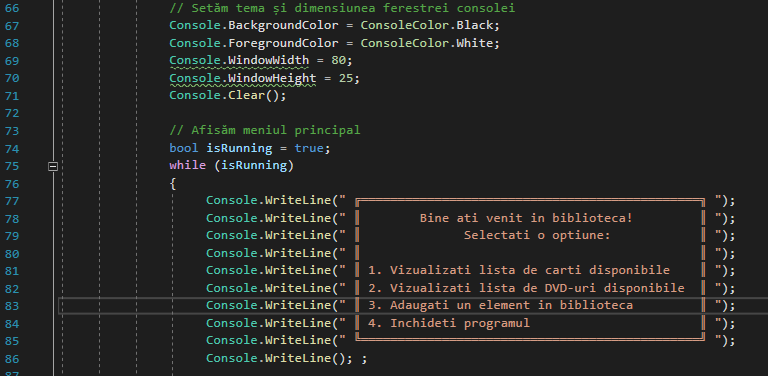
În proiectul meu se realizează implementarea unui sistem de bibliotecă online. Iată o prezentare a funcționalităților și a modului în care sistemul a fost realizat:

**Implementarea clasei principale**: Clasa Program conține metoda Main, care reprezintă punctul de intrare în aplicație. În metoda Main, sunt create instanțe ale claselor Library și User, și sunt adăugate cărți și DVD-uri în bibliotecă utilizând metoda AddItem a clasei Library.

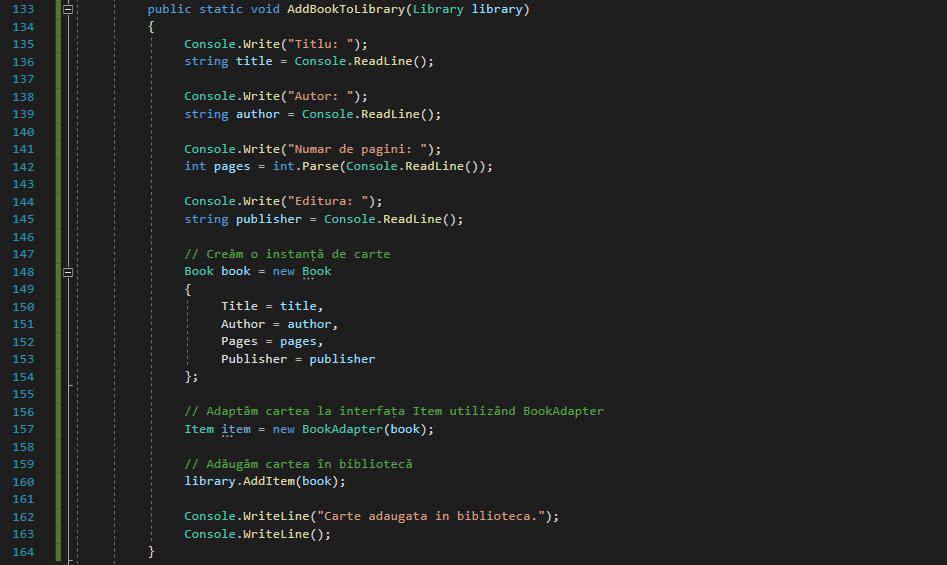


Figura 2.1 Implemenarea clasei principale

**Interacțiunea cu utilizatorul:** Utilizatorul este întâmpinat cu un meniu principal afișat în consolă. Opțiunile din meniu sunt afișate utilizând metodele Console.WriteLine și Console.ReadLine. Apoi, în funcție de opțiunea selectată, utilizatorul poate vizualiza lista de cărți sau DVD-uri disponibile, poate adăuga un element în bibliotecă sau poate închide programul.

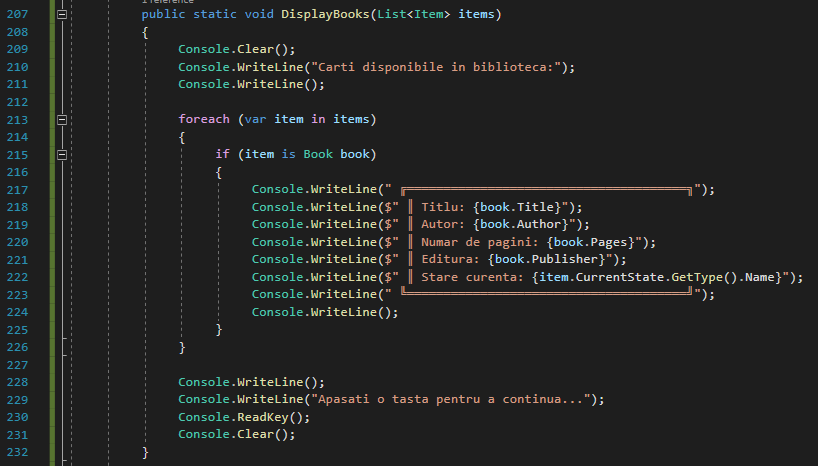


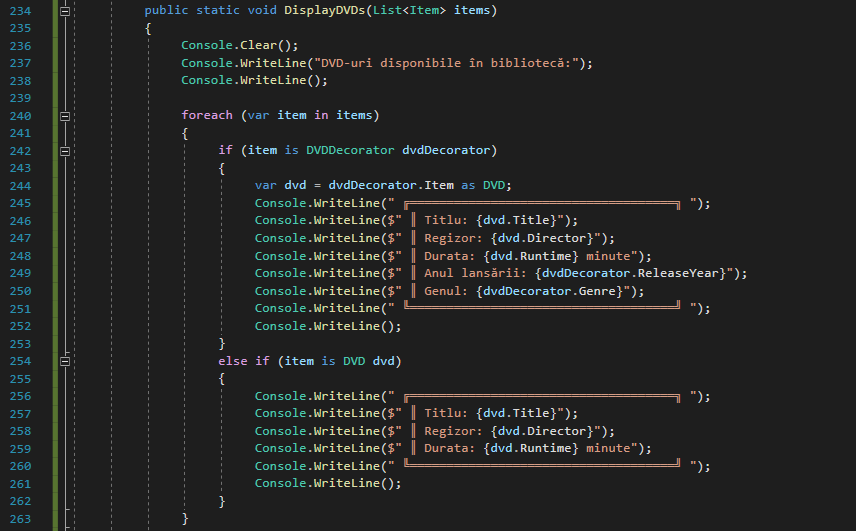
**Adăugarea unui element în bibliotecă:** Utilizatorul poate adăuga un element nou în bibliotecă selectând opțiunea corespunzătoare din meniu. Pentru fiecare tip de element (carte sau DVD), sunt implementate metodele AddBookToLibrary și AddDVDToLibrary. Aceste metode solicită utilizatorului introducerea datelor specifice (titlu, autor/regizor, număr de pagini/durată etc.) și creează o instanță a obiectului respectiv. Apoi, elementul este adăugat în bibliotecă utilizând metoda AddItem a clasei Library.



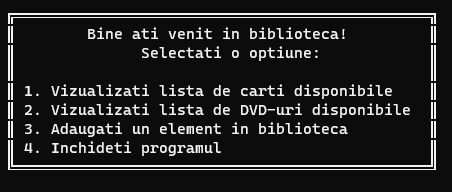


**Afișarea cărților și DVD-urilor disponibile:** Utilizatorul poate vizualiza lista de cărți și DVD-uri disponibile selectând opțiunea corespunzătoare din meniu. Metodele DisplayBooks și DisplayDVDs afișează informațiile despre cărți și DVD-uri într-un format predefinit, utilizând metodele Console.WriteLine și Console.Clear.

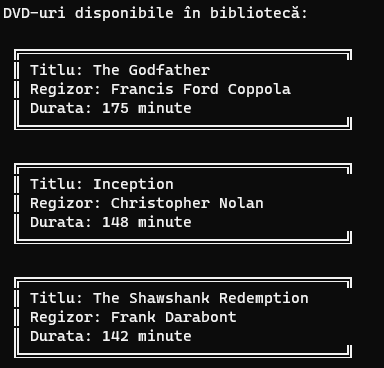




**Interfața grafică prietenoasă și simplă de utilizat**: În cadrul sistemului de bibliotecă online, se pune accent pe realizarea unei interfețe grafice cât mai accesibile și ușor de utilizat pentru utilizatori. Scopul este simplificarea procesului de interacțiune cu biblioteca și oferirea unei experiențe plăcute utilizatorilor.







**Utilizarea design pattern-urilor:** În implementarea sistemului, sunt folosite unele design pattern-uri pentru a facilita extensibilitatea și modularitatea. Aceste patternuri le vom descoperi în următorul capitol.

## 2.1 Singleton

Design pattern-ul Singleton este unul dintre cele mai cunoscute și utilizate pattern-uri de proiectare în dezvoltarea software. Acesta face parte din categoria pattern-urilor de creare (creational patterns) și are ca scop asigurarea că o clasă are o singură instanță și oferă un punct global de acces la acea instanță.

Principalele caracteristici și avantaje ale Singleton sunt:

Unicitatea instanței: Singleton garantează că o clasă are o singură instanță în întregul sistem. Aceasta este utilă în situațiile în care avem nevoie să ne asigurăm că există o singură instanță a unei clase și să avem acces global la aceasta.

Acces global: Prin intermediul Singleton, avem un punct global de acces la instanța clasei. Aceasta facilitează apelurile și interacțiunea cu această instanță în diferite părți ale sistemului.

Inițializare leneșă (lazy initialization): Singleton permite inițializarea leneșă a instanței sale, adică instanța este creată doar atunci când este nevoie de ea pentru prima dată. Aceasta optimizează resursele și performanța sistemului, deoarece nu se creează instanțe inutile.

Păstrarea stării: Deoarece Singleton are o singură instanță, orice modificare a stării acesteia este reflectată în întregul sistem. Astfel, putem utiliza Singleton pentru a păstra și accesa o stare comună sau date globale în întregul sistem.

Control asupra instanței: Singleton oferă un control strâns asupra modului în care instanța este creată și gestionată. Putem defini reguli și restricții pentru crearea și accesul la instanță, cum ar fi instanțierea lență sau gestionarea accesului concurent.

Implementarea unui Singleton poate varia în funcție de limbajul de programare și de cerințele specifice ale sistemului. O implementare clasică a Singleton implică utilizarea unei variabile statice private în cadrul clasei și definirea unui constructor privat pentru a preveni crearea de instanțe suplimentare.

În cadrul proiectului meu, Singleton este utilizat pentru a obține o singură instanță a clasei Library, care reprezintă biblioteca online. Aceasta asigură că există o singură instanță a bibliotecii în întregul sistem și permite accesul global la aceasta prin intermediul metodei statice Library.Instance.

Deci creem o clasă singleton separată numită Library, care va fi responsabilă pentru gestionarea listei de cărți și DVD-uri disponibile în bibliotecă. Clasa Library va implementa Singleton pentru a asigura că există o singură instanță a acesteia în întregul proiect. Iată cum arată implementarea:

public class Library

{

// Instanța Singleton a clasei Library

private static Library instance;

// Listă de cărți și DVD-uri disponibile în bibliotecă

private List<Item> items;

// Obține instanța Singleton a clasei Library

public static Library Instance

{

get

{

// Dacă instanța nu a fost creată, o creăm și inițializăm lista de itemi

if (instance == null)

{

instance = new Library();

instance.items = new List<Item>();

}

return instance;

}

}

private Library()

{

// Constructorul privat asigură că nu se pot crea alte instanțe

}

// Metoda pentru adăugarea unui item în bibliotecă

public void AddItem(Item item)

{

items.Add(item);

}

// Metoda pentru obținerea tuturor itemilor din bibliotecă

public List<Item> GetAllItems()

{

return items;

}

Observăm că Singleton-ul este implementat în clasa Library. Iată câteva indicii pentru a observa utilizarea singleton-ului în codul meu:

1. Constructorul clasei Library este privat, ceea ce înseamnă că nu poate fi apelat din afara clasei:

private Library()

{

// Constructorul privat asigură că nu se pot crea alte instanțe

}

1. Există o proprietate statică numită Instance în clasa Library care returnează instanța unică a clasei Library:

public static Library Instance

{

get

{

// Dacă instanța nu a fost creată, o creăm și inițializăm lista de itemi

if (instance == null)

{

instance = new Library();

instance.items = new List<Item>();

}

return instance;

}

}

Această proprietate utilizează un mecanism numit lazy initialization pentru a crea instanța singleton doar la prima accesare a proprietății Instance. Dacă instanța nu există, aceasta este creată și inițializată cu o nouă listă de itemi.

1. Metoda AddItem() a clasei Library permite adăugarea unui item în bibliotecă:

public void AddItem(Item item)

{

items.Add(item);

}

Această metodă utilizează instanța singleton a clasei Library pentru a adăuga itemul în lista de itemi.

1. Metoda GetAllItems() returnează toți itemii din bibliotecă:

// Metoda pentru obținerea tuturor itemilor din bibliotecă

public List<Item> GetAllItems()

{

return items;

}

Și această metodă utilizează instanța singleton a clasei Library pentru a accesa lista de itemi.

Observăm că în metoda Main() a clasei Program, se obține instanța singleton a clasei Library prin apelul

// Obținem instanța singleton a clasei Library

Library library = Library.Instance;

Aceasta asigură că obținem aceeași instanță a clasei Library ori de câte ori accesăm Library.Instance.

Prin utilizarea clasei Library în acest mod, avem garantarea că există o singură instanță a acesteia și putem accesa acea instanță unică din mai multe locuri ale programului. Mai jos avem și diagrama de clasă a Library

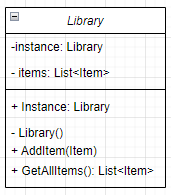


Figura 1. Singleton

## 2.2 Abstract Factory

Abstract Factory este un design pattern de tip creational care permite crearea de familii de obiecte legate sau dependente fără a specifica clasele lor concrete. Acesta furnizează o interfață comună pentru crearea diferitelor tipuri de obiecte dintr-o familie, ascunzând detaliile de implementare specifice.

Scopul principal al Abstract Factory este de a oferi o modalitate flexibilă de creare a obiectelor fără a dezvălui detalii specifice despre cum acestea sunt create sau configurate. Acest pattern promovează principiul separării responsabilităților și favorizează modularitatea, flexibilitatea și extensibilitatea în designul sistemelor software.

Componentele cheie ale pattern-ului Abstract Factory sunt:

Abstract Factory: Defineste o interfață comună pentru crearea obiectelor dintr-o familie sau o grupă de clase. Această interfață abstractă poate conține mai multe metode de creare pentru diferite tipuri de obiecte.

Concrete Factory: Implementează metodele de creare specifice din interfața Abstract Factory. Fiecare Concrete Factory este responsabilă de crearea și returnarea instanțelor concrete ale obiectelor dintr-o anumită familie.

Abstract Product: Reprezintă o interfață comună pentru un grup de obiecte asociate sau dependente. Această interfață poate defini metode comune pe care obiectele concrete le vor implementa.

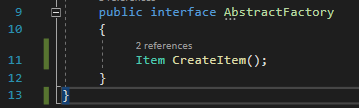
Concrete Product: Implementează interfața Abstract Product și reprezintă obiectele concrete care sunt create de Concrete Factory. Fiecare Concrete Product este parte a unei anumite familii de obiecte.

Client: Utilizează interfața Abstract Factory și Abstract Product pentru a crea și interacționa cu obiectele dintr-o anumită familie. Clientul nu este conștient de clasele concrete specifice și lucrează doar cu interfețele abstracte.

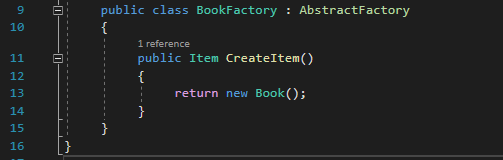
Abstract Factory este util în situațiile în care un sistem trebuie să fie independent de modul în care obiectele concrete sunt create sau configurate. Acest pattern facilitează schimbul flexibil al familiei de obiecte și promovează modularitatea și reutilizabilitatea codului.

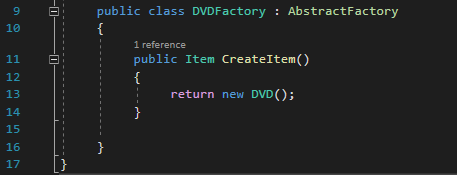
Iată cum putem integra Abstract Factory în proiectul nostru:

1. Definim o interfață AbstractFactory care va declara metode pentru crearea obiectelor dintr-o familie:

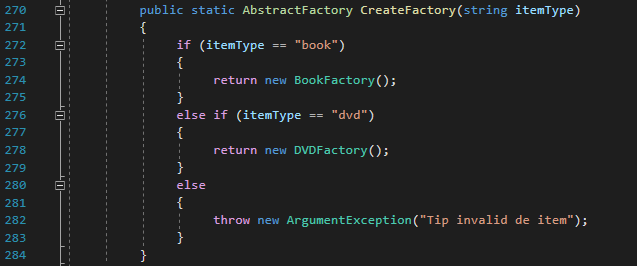


1. Implementăm două fabrici concrete care implementează interfața AbstractFactory. În cazul dat avem o fabrică pentru crearea cărților și una pentru crearea DVD-urilor:





1. În clasa Program, adăugăm o metodă CreateFactory() care va returna fabrica corespunzătoare tipului de item pe care îl dorim să îl creem:



Mai jos putem urmări diagrama UML al acestui Pattern:

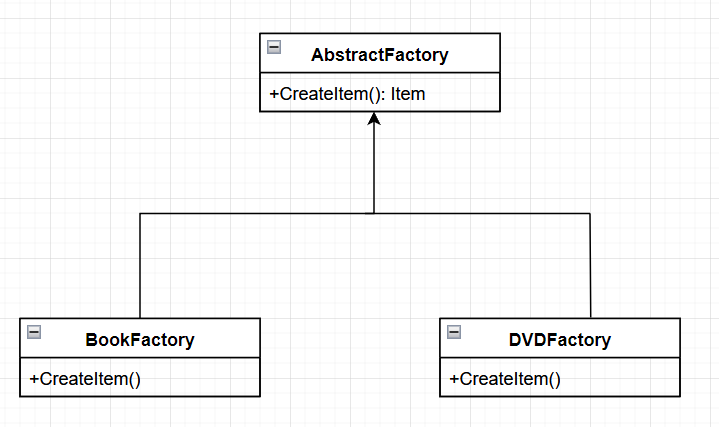


Figura 2. Abstract Factory

Prin utilizarea Abstract Factory, putem:

Să creem obiecte dintr-o familie specifică (cărți sau DVD-uri) fără a cunoaște clasele concrete implicate în creare. Acest lucru ne oferă flexibilitate în a schimba sau adăuga noi familii de obiecte în viitor.

Să gestionăm crearea obiectelor într-un mod centralizat și coerent. Fabricile concrete (BookFactory și DVDFactory) implementează metoda CreateItem() din interfața AbstractFactory, astfel încât putem utiliza metoda CreateFactory() pentru a obține fabrica potrivită în funcție de tipul de element pe care doriți să îl creați.

Să abstractizăm logica creării obiectelor și să o separăm de alte părți ale codului. Aceasta duce la un cod mai modular și mai ușor de întreținut, deoarece logica de creare a obiectelor este encapsulată în fabricile concrete.

## 2.3 Adapter

Adapterul este un design pattern structural care permite obiectelor cu interfețe incompatibile să colaboreze. Acesta rezolvă problema integrării două entități care altfel nu ar putea să lucreze împreună din cauza diferențelor în interfețe.

**Principalele componente ale pattern-ului Adapter sunt:**

Target (Ținta): Reprezintă interfața dorită de către clientul care dorește să folosească un obiect incompatibil cu acea interfață.

Adaptee (Adaptatul): Reprezintă obiectul existent cu o interfață incompatibilă cu interfața dorită de client.

Adapter: Este clasa care conectează interfața Target cu Adaptee. Adapterul implementează interfața Target și conține o referință la obiectul Adaptee, astfel încât să poată transforma apelurile la interfața Target în apeluri către metodele corespunzătoare ale Adaptee.

**Funcționarea pattern-ului Adapter este următoarea:**

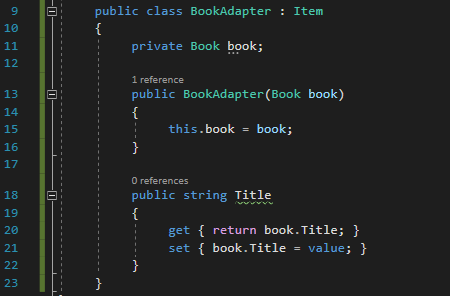
Clientul face apeluri la metodele interfeței Target pentru a interacționa cu obiectele.

Adapterul primește apelurile de la client și le transformă în apeluri către metodele corespunzătoare ale Adaptee.

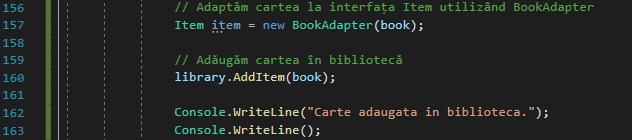
Adaptee procesează cererile și returnează rezultatele înapoi prin intermediul Adapterului către client.

Prin utilizarea pattern-ului Adapter, putem integra obiecte existente într-un sistem nou fără a le modifica structura sau interfața. Adapterul asigură comunicarea și cooperarea transparentă între componente incompatibile, fără a afecta funcționalitatea existentă.

Pentru a adăuga un adapter în proiectul meu, creem o clasă BookAdapter care va implementa interfața Item. Aceasta va adapta obiectele de tip Book la interfața Item pentru a putea fi utilizate în Library.



După ce am definit clasa BookAdapter, putem modifica metoda AddBookToLibrary din clasa Program pentru a adăuga adapterul:



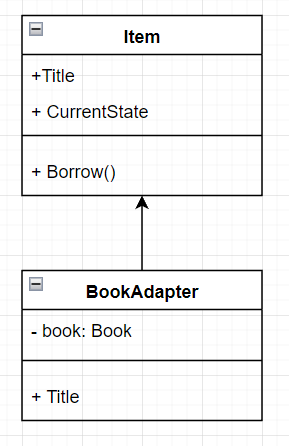


Figura 3. Adapter

Adăugarea design pattern-ului Adapter în programul meu imi permite adaptarea obiectelor de tip Book la interfața Item, astfel încât să poată fi utilizate în clasa Library.

Înainte de a adăuga Adapter-ul, clasa Library putea adăuga și returna doar obiecte de tip Item, deoarece aceasta era interfața cu care lucra. Astfel, dacă doream să adaug o carte în bibliotecă, trebuia să creez un obiect de tip Book și să îl transform într-un obiect de tip Item.

Adăugarea Adapter-ului simplifică procesul de adăugare a cărților în bibliotecă. Adapter-ul BookAdapter permite adaptarea obiectelor de tip Book la interfața Item. Acesta conține o referință către obiectul de tip Book și implementează proprietatea Title din interfața Item pentru a se conforma acesteia.

Astfel, când utilizez metoda AddBookToLibrary, creăm o instanță a clasei Book, iar apoi adaptăm această carte la interfața Item prin intermediul adapterului BookAdapter. Astfel, putem adăuga direct obiectul de tip Item în bibliotecă.

Adăugarea design pattern-ului Adapter face codul mai flexibil și extensibil. Dacă, în viitor, adăugăm și alte tipuri de obiecte (cum ar fi Journal, Magazine, etc.), vom putea adapta aceste obiecte la interfața Item prin intermediul unor adaptoare specifice. Astfel, Library poate lucra cu diverse tipuri de obiecte prin intermediul interfeței comune Item

## 2.4 Decorator

Pattern-ul Decorator este unul dintre cele mai cunoscute și utilizate pattern-uri de proiectare din categoria structurală. El permite atașarea de comportamente suplimentare la un obiect existent într-un mod flexibil, fără a modifica clasa de bază a obiectului.

Principalele componente ale pattern-ului Decorator sunt:

Component: Reprezintă clasa de bază sau interfața care definește comportamentul de bază al obiectului. Aceasta poate fi o clasă concretă sau o interfață pe care decoratorii o vor implementa.

Decorator: Este clasa abstractă care implementează aceeași interfață ca și Componentul de bază. Decoratorul conține o referință către un obiect de tip Component și poate avea și alte metode specifice.

Concrete Decorators: Sunt clasele concrete care extind Decoratorul și adaugă comportamente suplimentare la obiectul de bază. Acestea pot adăuga funcționalități noi prin intermediul metodelor suplimentare și pot apela metodele de bază ale obiectului Component.

Prin utilizarea pattern-ului Decorator, putem extinde funcționalitatea unui obiect într-un mod modular și flexibil, fără a modifica structura de bază a acestuia. Putem adăuga sau îndepărta decoratori în funcție de nevoile noastre, obținând astfel o combinație personalizată de funcționalități pentru obiectul nostru.

Un exemplu comun în care se utilizează pattern-ul Decorator este atunci când dorim să adăugăm funcționalități suplimentare la obiectele grafice, cum ar fi stilizarea sau bordura. Putem crea decoratori pentru fiecare funcționalitate suplimentară și să le aplicăm în mod dinamic la obiectele grafice, fără a modifica direct codul acestora.

Este important să menționăm că decoratorii pot fi utilizați într-o manieră recursivă, permițând adăugarea de funcționalități multiple și complexe într-un mod flexibil.

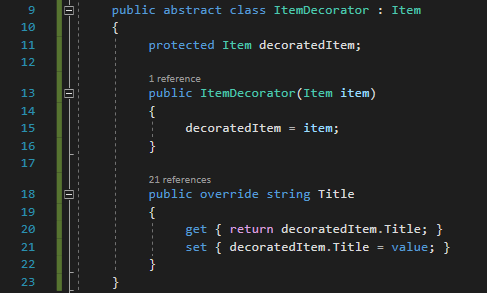
Prin utilizarea pattern-ului Decorator, obținem un design modular, extensibil și ușor de întreținut, în care comportamentul obiectelor poate fi modificat și adaptat la cerințe noi fără a afecta structura existentă..

În proiectul meu, am creat decoratorul DVDDecorator pentru a adăuga funcționalități suplimentare obiectelor de tip DVD. Decoratorul implementează aceeași interfață Item, permițându-i să fie tratat în același mod ca și obiectele DVD originale. Apoi, decoratorul conține o referință către obiectul DVD inițial, pe care îl împachetează.

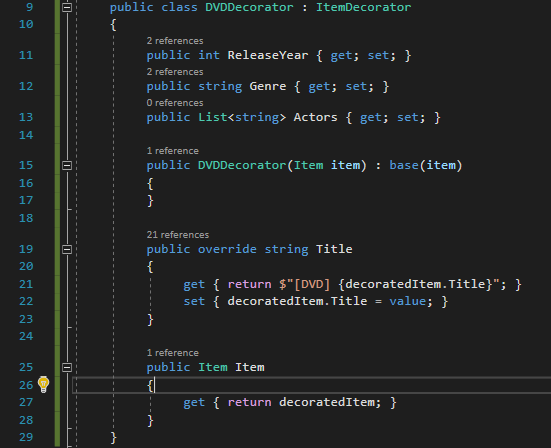
Decorarea obiectelor se realizează prin adăugarea de funcționalități suplimentare în metodele decoratorului și prin apelarea corespunzătoare a metodelor din obiectul inițial.

Un beneficiu important al utilizării design pattern-ului Decorator este că acesta permite adăugarea de comportamente suplimentare într-un mod flexibil și modular, fără a modifica structura de bază a clasei. Astfel, putem adăuga și îndepărta decoratori într-un mod dinamic, în funcție de nevoile mele.

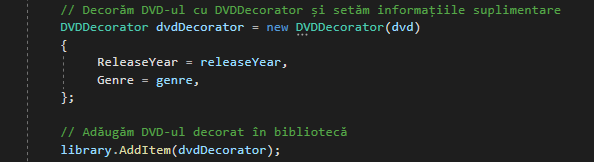
1. Creem clasa abstractă ItemDecorator care acționează ca decorator de bază și conține referința către obiectul decorat.



1. Creem clasa DVDDecorator care extinde ItemDecorator și adaugă funcționalități specifice pentru obiectele de tip DVD.

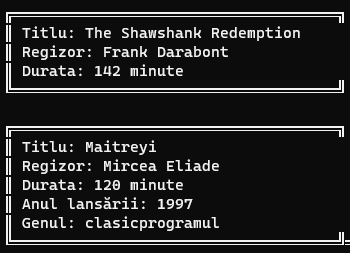


1. În metoda AddDVDToLibrary unde se creează un obiect DVD, decorăm cu DVDDecorator și setăm informațiile suplimentare. Obiectul decorat este apoi îl adăugăm în bibliotecă.



În medotda DisplayDVD unde afișăm DVD-urile, facem o verificare cu ajutorului unui if pentru a vedea dacă obiectul este decorat sau nu, pentru a printa și informațiile suplimentare.





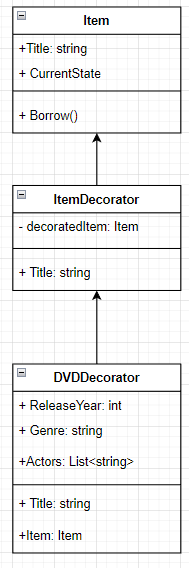


Figura 4 Decorator

Am implementat pattern-ul Decorator pentru a adăuga funcționalități suplimentare obiectelor din proiectul dvs. într-un mod flexibil și modular, fără a modifica direct clasele de bază.

## 2.5 Iterator

Pattern-ul Iterator este folosit pentru a itera prin elementele unei colecții fără a expune structura internă a colecției. Acesta oferă o modalitate standardizată de a parcurge și accesa elementele unei colecții într-un mod secvențial.

Iată câteva elemente de bază despre pattern-ul Iterator:

1. Componente principale:

* Interfața Iterator: Defineste metodele pentru navigarea și accesarea elementelor dintr-o colecție.
* Interfața Colecție: Declara metodele pentru obținerea unui iterator.
* Clasa Colecție Concretă: Implementează interfața Colecție și furnizează o implementare specifică pentru a obține un iterator.
* Clasa Iterator Concret: Implementează interfața Iterator și realizează navigarea și accesarea elementelor în cadrul colecției.

1. Funcționarea pattern-ului:

* Colecția concretă furnizează un iterator prin intermediul metodei specifice de obținere a unui iterator.
* Iteratorul permite parcurgerea secvențială a elementelor colecției și oferă metode pentru obținerea elementelor curente și trecerea la următorul element.
* Clientul utilizează iteratorul pentru a accesa elementele colecției într-un mod secvențial, fără a cunoaște structura internă a colecției.

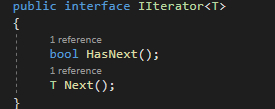
1. Beneficii ale utilizării pattern-ului Iterator:

* Separarea responsabilităților: Pattern-ul permite separarea responsabilităților de parcurgere și accesare a elementelor de structura internă a colecției.
* Flexibilitate și extensibilitate: Poți adăuga noi tipuri de iteratori sau colecții fără a modifica codul existent.
* Simplificarea codului client: Clientul poate itera prin elementele colecției utilizând aceeași interfață standardizată, indiferent de implementarea concretă a colecției.

1. Exemple de utilizare:

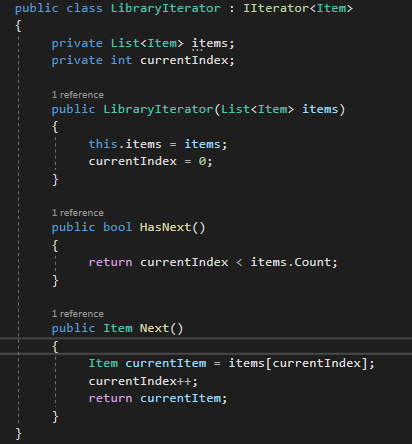
* Parcurgerea și accesarea elementelor unei liste, a unui vector sau a unei structuri de date personalizate.
* Implementarea de iterații personalizate în funcție de anumite criterii sau filtre.

1. Vom implementa design pattern-ul Iterator pentru a itera prin lista de itemi din bibliotecă. Mai întâi, vom adăuga o interfață Iterator care va defini metodele necesare pentru iterație.

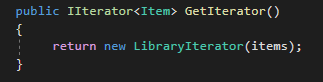


Metoda HasNext() verifică dacă mai există elemente disponibile în lista de itemi prin compararea poziției curente cu numărul total de elemente. Metoda Next() returnează elementul curent din listă și avansează poziția pentru a accesa următorul element în următoarea apelare.

1. Vom crea o clasă concretă LibraryIterator care implementează interfața IIterator și va fi responsabilă cu iterația prin lista de itemi din bibliotecă.



1. Vom actualiza clasa Library pentru a returna un iterator în metoda GetIterator(). Această metodă va crea și returna o instanță a clasei LibraryIterator.



Pattern-ul Iterator ne permite să iterăm prin elementele unei colecții (în cazul meu, o listă de obiecte Item) fără a cunoaște structura internă a colecției sau detaliile de implementare.

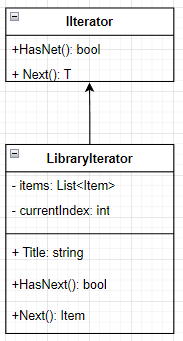


Figura 5. Iterator

Astfel, prin utilizarea pattern-ului Iterator, putem itera prin colecția de obiecte Item fără a accesa direct structura internă a acesteia. Aceasta aduce un nivel de abstractizare și flexibilitate în manipularea colecției și permite separarea responsabilităților între colecție și iterator.

## 2.6 State

Design Pattern-ul State (Stare) este un pattern comportamental care permite unui obiect să își schimbe comportamentul în funcție de starea internă în care se află. Acesta permite gestionarea eficientă a tranzacțiilor între diferite stări ale obiectului, fără a necesita utilizarea instrucțiunilor condiționale lungi sau multiple.

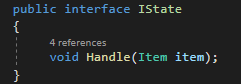
Principalele elemente ale Design Pattern-ului State sunt:

* Starea (State): Reprezintă o interfață sau o clasă abstractă care definește metodele specifice stării și acțiunile asociate acestora. Fiecare stare concretă implementează această interfață.
* Stări concrete (Concrete States): Sunt implementări specifice ale interfeței Stare. Acestea definesc comportamentul specific pentru fiecare stare a obiectului.
* Contextul (Context): Reprezintă clasa care conține obiectul ce se află în diferite stări și gestionează tranzițiile între acestea. Contextul utilizează o referință la Stare pentru a reprezenta starea curentă a obiectului și pentru a apela metodele specifice stării curente.

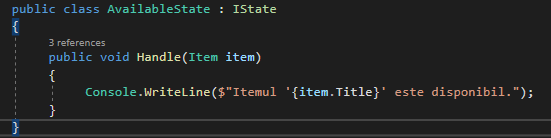
Design Pattern-ul State aduce mai multă flexibilitate și extensibilitate în gestionarea stărilor obiectelor, deoarece permite adăugarea ușoară a unor stări noi sau modificarea comportamentului stărilor existente fără a afecta codul existent. De asemenea, separă responsabilitățile specifice stărilor în clase separate, făcând codul mai modular și mai ușor de înțeles.

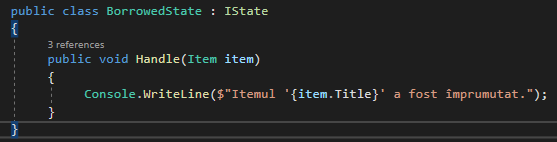
Exemplu de utilizare a Design Pattern-ului State poate fi un sistem de comenzi care poate avea diferite stări, cum ar fi "Comandă în așteptare", "Comandă în curs de procesare" și "Comandă finalizată". Fiecare stare ar putea avea metode specifice, cum ar fi "confirmareComanda()", "anulareComanda()", "finalizareComanda()" etc. Obiectul de comandă ar avea o referință la Stare și ar apela metodele specifice stării curente pentru a gestiona diferitele acțiuni.

1. Începem prin a defini interfața IState care va fi implementată de toate stările posibile:

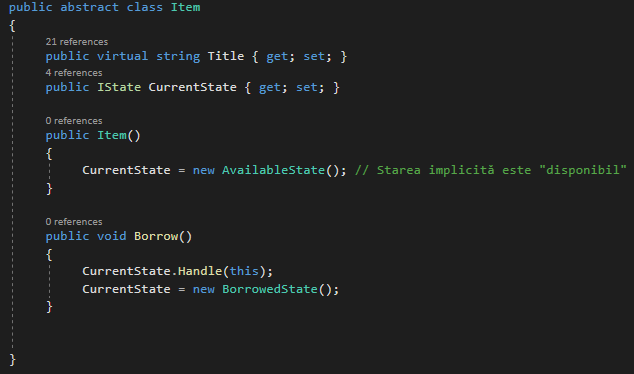


1. Creem clasele pentru fiecare stare specifică:

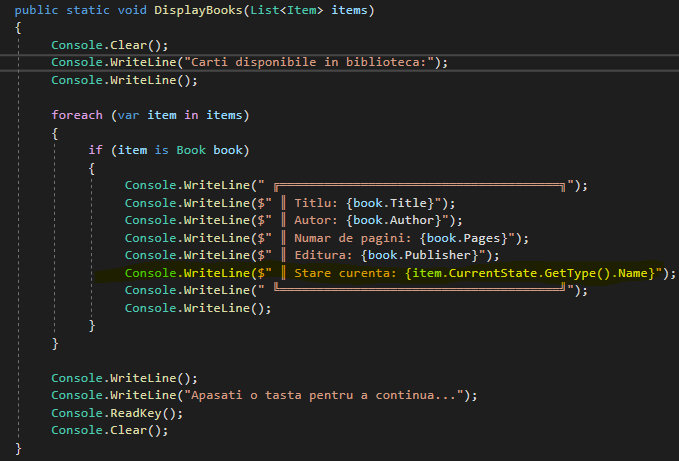


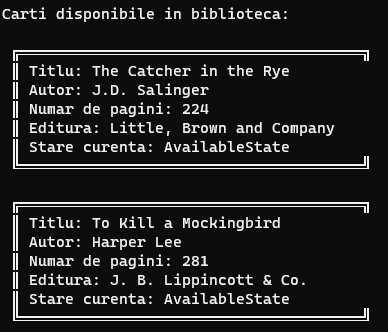


1. Vom modifica clasa Item pentru a adăuga starea curentă și pentru a utiliza starea pentru a trata comportamentul specific:



1. Vom face câteva modificări în metoda DisplayBooks pentru a afișa starea curentă a fiecărui element:





Observăm ca a apărut și starea curentă a cărților din bibliotecă.

Am implementat design pattern-ul State pentru a gestiona starea unui obiect de tip Item din bibliotecă. Fiecare element din bibliotecă poate avea diferite stări, cum ar fi "disponibil" sau "împrumutat" , iar comportamentul acestuia trebuie să se schimbe în funcție de starea curentă.

Prin utilizarea design pattern-ului State, am separat logica specifică stărilor în clasele AvailableState și BorrowedState, care implementează interfața IState. Aceste stări pot gestiona diferite acțiuni sau comportamente asociate cu starea respectivă.

De exemplu, atunci când un element este împrumutat, apelăm metoda Borrow() a obiectului Item. Aceasta va utiliza starea curentă pentru a gestiona comportamentul specific al stării "împrumutat". În acest caz, se afișează un mesaj corespunzător și starea obiectului este actualizată la BorrowedState.

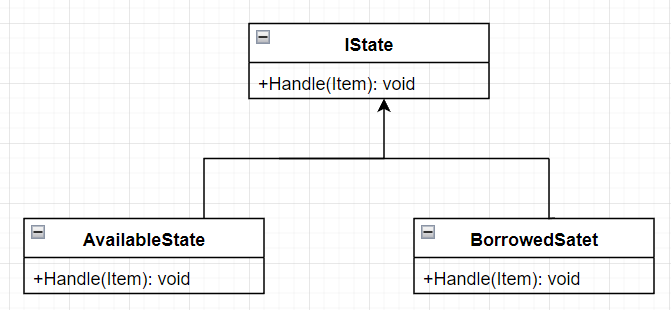


Figura 6. State

# Concluzii:

Proiectul creat a fost o implementare a unui sistem simplu de gestionare a unei biblioteci utilizând mai multe design pattern-uri. Am explorat și aplicat design pattern-uri precum Singleton, Factory Method, Adapter, Decorator, Iterator și State pentru a obține o structură modulară, flexibilă și ușor de extins.

Folosind Singleton, am asigurat că există o singură instanță a clasei Library, oferind acces global la bibliotecă. Am utilizat Factory Method pentru a delega crearea de obiecte concrete claselor specializate, precum Book și DVD. Aceasta ne-a permis să separăm logica de creare a obiectelor de logica principală a bibliotecii.

Prin intermediul pattern-ului Adapter, am adaptat obiectele de tip Book la interfața Item, permițându-le să fie utilizate în bibliotecă. Am folosit Decorator pentru a adăuga funcționalități suplimentare obiectelor DVD prin împachetarea lor în decoratori specifici. Acest lucru ne-a oferit flexibilitatea de a extinde comportamentul obiectelor fără a le modifica direct.

Cu ajutorul pattern-ului Iterator, am putut itera prin lista de itemi din bibliotecă fără a cunoaște structura internă a colecției. Acest lucru ne-a permis să separăm responsabilitățile de iterație de structura de bază a bibliotecii.

Prin utilizarea pattern-ului State, am gestionat starea obiectelor din bibliotecă și am tratat comportamentul specific în funcție de starea curentă. Acest lucru ne-a oferit o abordare modulară și ușor de extins pentru gestionarea stărilor și a comportamentelor asociate acestora.

În concluzie, prin aplicarea acestor design pattern-uri în proiectul nostru, am obținut un sistem bine organizat, modular și flexibil, care poate fi extins și adaptat ușor în viitor. Utilizarea acestor pattern-uri ne-a permis să separăm responsabilitățile, să abstractizăm anumite aspecte și să facilităm adăugarea de noi funcționalități fără a afecta structura de bază a bibliotecii.