Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Raport**

**la Proiect de an**

**Disciplina : TMPS**

**Tema: “Proiectarea unei aplicații pentru managementul unei biblioteci”**

A efectuat st.gr. TI-204 Jardan Alexandru

Chișinău – 2023

**Cuprins**

[**Introducere 2**](#_Toc137066163)

[**1. Analiza domeniului de studiu 3**](#_Toc137066164)

[**1.1 Tipuri de Design Patterns 4**](#_Toc137066165)

[**2. Descrierea aplicației 10**](#_Toc137066166)

[**2.1 Scopul aplicației 11**](#_Toc137066167)

[**2.2 Funcționalitățile aplicației 12**](#_Toc137066168)

[**2.3 Tehnologiile utilizate 13**](#_Toc137066169)

[**3. Implementarea Design Pattern-urilor 14**](#_Toc137066170)

[**3.1 Șabloane Creaționale 15**](#_Toc137066171)

[**3.2 Șabloane Structurale 17**](#_Toc137066172)

[**3.3 Șabloane Comportamentale 19**](#_Toc137066173)

[**Concluzii 22**](#_Toc137066174)

[**Bibliografie 23**](#_Toc137066175)

# Introducere

Utilizarea design pattern-urilor sau modelelor de proiectare în dezvoltarea software-ului este o practică esențială pentru a obține soluții eficiente, robuste și ușor de întreținut. Design pattern-urile reprezintă soluții testate și validate pentru probleme comune întâlnite în domeniul dezvoltării software, oferind un cadru conceptual și o abordare standardizată pentru implementarea acestora.

Beneficiile utilizării design patterns în realizarea unui proiect sunt numeroase și se reflectă în toate aspectele dezvoltării software. În primul rând, acestea permit crearea unui cod modular și structurat, cu responsabilități bine definite pentru fiecare componentă. Prin separarea logică a componentelor și prin utilizarea unor interfețe bine definite, design patterns-urile facilitează înțelegerea și extensibilitatea proiectului, reducând riscul de creare a unui cod greu de întreținut și de modificat ulterior.

Un alt beneficiu al utilizării design patterns este reutilizarea codului. Multe design pattern-uri promovează conceptul de reutilizare și modularitate, permițând dezvoltatorilor să își adapteze și să își extindă soluțiile existente pentru a se potrivi cerințelor specifice ale proiectului. Acest lucru duce la o productivitate sporită și la economisirea timpului și efortului necesare pentru a dezvolta soluții noi de la zero.

Un alt aspect important este acțiunea de abstractizare și încapsulare a logicii în design patterns. Prin definirea unor abstracții și prin separarea detaliilor implementării de interfețele publice, design patterns-urile facilitează flexibilitatea și scalabilitatea proiectului. Astfel, se pot face modificări și optimizări ulterioare fără a afecta întregul sistem sau a încălca dependențele existente între componente.

De asemenea, utilizarea design pattern-urilor îmbunătățește calitatea și testabilitatea codului. Acestea promovează utilizarea unor principii de proiectare bune, cum ar fi separarea responsabilităților, principiul deschis-închis și inversiunea dependențelor. Prin urmare, proiectele care folosesc design patterns beneficiază de un cod mai ușor de testat, cu o acoperire mai mare a testelor și cu mai puține erori de implementare.

În concluzie, utilizarea design patterns în realizarea unui proiect software aduce numeroase beneficii. Acestea includ modularitatea, reutilizarea codului, structurarea clară a componentelor, flexibilitatea și testabilitatea sporite. Prin aplicarea unor soluții testate și validate într-un cadru standardizat, dezvoltatorii pot construi proiecte mai eficiente, mai robuste și mai ușor de întreținut.

# 1. Analiza domeniului de studiu

Un design pattern (sau model de proiectare) este o soluție generală și reutilizabilă la o problemă comună de proiectare a software-ului. Acesta oferă o abordare structurată pentru rezolvarea anumitor probleme recurente în dezvoltarea software-ului, într-un mod eficient și ușor de înțeles.

Design patterns sunt importante deoarece oferă ghiduri și soluții testate în timp pentru problemele comune întâlnite în proiectarea software-ului. Utilizarea design patterns aduce o serie de beneficii:

1. Reutilizabilitate: Design patterns facilitează reutilizarea codului și a soluțiilor deja implementate. Ele oferă un set de reguli și structuri standardizate pe care le puteți aplica în diferite proiecte, economisind timp și efort în dezvoltarea software-ului.
2. Flexibilitate și scalabilitate: Design patterns ajută la crearea de software modular și flexibil. Ele permit izolarea diferitelor părți ale unei aplicații și oferă metode de interacțiune și comunicare între acestea. Aceasta facilitează extinderea și modificarea software-ului în viitor, permițându-vă să faceți față cerințelor și schimbărilor ulterioare cu ușurință.
3. Îmbunătățirea calității și a structurii codului: Design patterns promovează principiile de codare curată și organizată. Ele vă ajută să separați responsabilitățile și să creați structuri clare și ușor de înțeles în cod. Acest lucru face ca software-ul să fie mai ușor de întreținut, depanat și extins.
4. Comunicare eficientă: Utilizarea unor design patterns bine-cunoscute facilitează comunicarea între membrii echipei de dezvoltare. Ele oferă un limbaj comun și o bază de cunoștințe partajată, astfel încât toți membrii echipei să înțeleagă și să lucreze în același mod.

Design patterns sunt folosite în diverse domenii de dezvoltare software, inclusiv dezvoltarea aplicațiilor web, dezvoltarea aplicațiilor mobile, dezvoltarea jocurilor și dezvoltarea de aplicații desktop. Există o mare varietate de design patterns disponibile, împărțite în diferite categorii cum ar fi creational patterns (ex. Singleton, Factory, Builder), structural patterns (ex. Adapter, Decorator, Facade) și behavioral patterns (ex. Observer, Strategy, Template Method).

Este importanta alegerea unui design pattern potrivit pentru contextul și cerințele proiectului dvs. și să îl implementați într-un mod corespunzător. Înțelegerea și aplicarea corectă a design patterns poate duce la dezvoltarea unui software robust, extensibil și ușor de întreținut.

## **1.1 Tipuri de Design Patterns**

Există mai multe categorii generale în care putem încadra tipurile de design patterns. Aceste categorii ne ajută să înțelegem rolul și utilizarea lor în proiectarea software-ului. Iată câteva categorii importante:

1. Creational Patterns (Modele de creare): Aceste tipuri de design patterns se concentrează pe procesul de creare a obiectelor într-un mod flexibil și unificat. Acestea ascund detalii specifice ale implementării și facilitează extensibilitatea și modularitatea sistemelor. Iată câteva exemple de design patterns din această categorie:
   * Singleton: Acest pattern se asigură că o clasă are o singură instanță în întregul sistem. Astfel, se asigură că doar o instanță a clasei poate fi accesată global.
   * Factory Method: Acest pattern definește o metodă într-o clasă de bază, lăsând subclaselor să decidă tipul specific de obiect pe care îl creează. Astfel, se realizează o decuplare între codul client și clasele concrete.
   * Abstract Factory: Acest pattern furnizează o interfață pentru crearea de familii de obiecte conexe sau dependente, fără a specifica clasele concrete. Astfel, se permite schimbul ușor între diferite familii de obiecte.
   * Builder: Acest pattern este utilizat pentru a construi obiecte complexe pas cu pas. El permite crearea de obiecte variate și configurabile prin intermediul unui proces de construcție flexibil.
   * Prototype: Acest pattern permite crearea de obiecte noi prin clonarea unui obiect existent, în loc de crearea de noi instanțe. Astfel, se evită crearea repetată a obiectelor costisitoare în timp și resurse.
2. Structural Patterns (Modele structurale): Aceste tipuri de design patterns se concentrează pe organizarea și compunerea obiectelor pentru a forma structuri mai complexe. Ele rezolvă probleme legate de interacțiunea și relațiile dintre obiecte, oferind o structură clară și flexibilă. Iată câteva exemple de design patterns din această categorie:
   * Adapter: Acest pattern permite două interfețe incompatibile să funcționeze împreună, oferind un adapter între ele. Astfel, se realizează o adaptare între două componente care altfel nu ar putea comunica direct.
   * Decorator: Acest pattern permite adăugarea de funcționalități suplimentare la un obiect existent, fără a modifica structura de bază a acestuia. Astfel, se obține extensibilitatea obiectelor prin încapsularea lor în obiecte decorator.
   * Proxy: Acest pattern furnizează un substitut sau o reprezentare a unui obiect, controlând accesul la acesta. Astfel, se oferă o modalitate de control și gestionare a obiectelor la nivelul interfeței publice.
   * Composite: Acest pattern permite tratarea unui grup de obiecte similare într-un mod uniform ca și cum ar fi un singur obiect. Astfel, se permite tratamentul uniform al obiectelor individuale și al grupurilor acestora.
   * Facade: Acest pattern oferă o interfață simplificată și unificată către un subsistem complex, ascunzând detaliile și complexitatea acestuia. Astfel, se simplifică interacțiunea cu subsistemul și se reduce cuplarea între componente.
3. Behavioral Patterns (Modele de comportament): Aceste tipuri de design patterns se concentrează pe comunicarea și comportamentul între obiecte și clase. Ele oferă o modalitate de organizare a interacțiunii între obiecte, facilitând flexibilitatea și modularitatea în ceea ce privește comportamentul sistemului. Iată câteva exemple de design patterns din această categorie:
   * Observer: Acest pattern definește o relație de tip unu-la-mulți între obiecte, astfel încât atunci când un obiect se schimbă, toți obiectele dependente sunt notificate și actualizate automat.
   * Strategy: Acest pattern permite definirea unui set de algoritmi diferiți și interșanjabili și încapsularea fiecărui algoritm într-o clasă separată. Astfel, se permite schimbarea ușoară a algoritmului utilizat fără a modifica codul client.
   * Template Method: Acest pattern definește o scheletă pentru o anumită operațiune într-o clasă de bază, lăsând subclaselor să implementeze anumite părți specifice ale acestei operațiuni. Astfel, se permite personalizarea unei algoritmi de către subclase.
   * Iterator: Acest pattern oferă o modalitate de a accesa elementele unui obiect colecție într-o manieră secvențială, fără a dezvălui detaliile de implementare ale colecției. Astfel, se permite parcurgerea și manipularea elementelor colecției într-un mod flexibil și eficient.

Acestea sunt doar câteva categorii generale de design patterns. Este important de menționat că există numeroase tipuri specifice de design patterns în cadrul fiecărei categorii, fiecare cu scopul său unic și cu situațiile în care este cel mai potrivit de utilizat. Utilizarea adecvată a acestor design patterns poate îmbunătăți calitatea, modularitatea și flexibilitatea sistemelor software.

Singleton:

* + Ce este: Singleton este un design pattern creational care se asigură că o clasă are o singură instanță și oferă un punct global de acces la această instanță.
  + Obiective: Singleton este utilizat pentru a controla crearea și accesul la o singură instanță a unei clase, asigurând că nu există decât o instanță unică în întregul sistem.
  + Beneficii:
    - Acces global: Singleton oferă un punct de acces global la instanța sa, astfel încât obiectele pot comunica ușor cu aceasta în orice parte a sistemului.
    - Economisirea resurselor: Singleton evită crearea repetată a instanțelor, ceea ce poate economisi resursele sistemului.
    - Coerență: Singleton asigură că toate apelurile către instanța sa vor accesa aceeași instanță, garantând coerența datelor și comportamentului.

Builder:

* + Ce este: Builder este un design pattern creational care separă procesul de construcție a unui obiect complex de reprezentarea sa.
  + Obiective: Builder este utilizat pentru a construi obiecte complexe pas cu pas, oferind flexibilitate în alegerea și configurarea componentelor obiectului final.
  + Beneficii:
    - Simplificarea construcției: Builder permite construirea obiectelor complexe într-un mod structurat și organizat, prin intermediul unor etape clare și controlate.
    - Configurabilitate: Builder oferă posibilitatea de a configura obiectele în diferite moduri prin intermediul parametrilor și metodelor de setare a componentelor.
    - Ascunderea detalilor de construcție: Builder ascunde detaliile specifice ale construcției unui obiect, permițând astfel o mai mare abstractizare și modularitate a codului.

Abstract Factory:

* + Ce este: Abstract Factory este un design pattern creational care furnizează o interfață pentru crearea familiilor de obiecte conexe, fără a specifica clasele concrete.
  + Obiective: Abstract Factory este utilizat pentru a crea obiecte care sunt interdependente și trebuie să fie utilizate împreună într-un mod coerent.
  + Beneficii:
    - Flexibilitate în schimbarea familiei de obiecte: Abstract Factory permite schimbarea ușoară a familiei de obiecte prin înlocuirea implementării fabricii concrete cu alta, menținând în continuare interfața comună.
    - Abstractizare: Utilizând Abstract Factory, codul client poate interacționa cu obiectele prin intermediul interfeței comune, fără a depinde de clasele concrete.
    - Cuplare redusă: Abstract Factory reduce cuplarea între codul client și clasele concrete, permițând o mai mare flexibilitate în extinderea și modificarea sistemului.

Decorator:

* + Ce este: Decorator este un design pattern structural care permite adăugarea de funcționalități suplimentare la obiecte existente dinamic, fără a modifica structura lor de bază.
  + Obiective: Decorator este utilizat pentru a extinde funcționalitatea obiectelor existente în timpul executării, într-un mod flexibil și modular.
  + Beneficii:
    - Extensibilitate: Decorator permite adăugarea de comportamente suplimentare la obiecte fără a modifica codul existent, permițând astfel o extensibilitate ușoară a funcționalității.
    - Compoziție flexibilă: Prin intermediul Decorator, obiectele pot fi compuse într-un mod flexibil și modular, adăugând sau eliminând funcționalități la nivelul dorit.
    - Reutilizare: Decorator permite reutilizarea obiectelor existente, adăugând și înlăturând decoratori în funcție de necesități, fără a duplica codul.

Iterator:

* + Ce este: Iterator este un design pattern behavioral care oferă o modalitate de a parcurge secvențial elementele unui obiect colecție fără a dezvălui detaliile sale de implementare.
  + Obiective: Iterator este utilizat pentru a oferi o interfață unificată și flexibilă pentru parcurgerea și accesul la elementele unei colecții.
  + Beneficii:
    - Acces secvențial: Iterator permite parcurgerea elementelor unei colecții într-o manieră secvențială, indiferent de structura internă a colecției.
    - Ascunderea detaliilor de implementare: Iterator ascunde detaliile de implementare ale colecției, permițând modificări ale implementării fără a afecta codul client.
    - Interfață unificată: Prin intermediul Iterator, codul client poate accesa și parcurge diferite tipuri de colecții prin intermediul aceleiași interfețe unificate.

Strategy:

* + Ce este: Strategy este un design pattern behavioral care permite definirea și schimbarea ușoară a unui algoritm în timpul rulării, prin încapsularea fiecărui algoritm într-o clasă separată și interșanjabilă.
  + Obiective: Strategy este utilizat pentru a separa implementarea algoritmilor de codul client și a permite schimbarea dinamică a algoritmului utilizat.
  + Beneficii:
    - Flexibilitate: Strategy permite schimbarea ușoară a algoritmului utilizat fără a modifica codul client, permițând adaptarea comportamentului sistemului în timp real.
    - Reutilizare: Prin intermediul Strategy, algoritmul poate fi reutilizat în diferite contexte, promovând modularitatea și evitând duplicarea codului.
    - Testare și întreținere ușoară: Strategy facilitează testarea și întreținerea algoritmilor individuali, deoarece fiecare algoritm este încapsulat într-o clasă separată.

Command:

* + Ce este: Command este un design pattern behavioral care încapsulează o cerere sub formă de obiect, permițând astfel parametrizarea clienților cu diferite cereri, înregistrarea și anularea operațiilor.
  + Obiective: Command este utilizat pentru a separa solicitările de acțiune de obiectele care le execută, permițând astfel parametrizarea și gestionarea flexibilă a operațiunilor.
  + Beneficii:
    - Dezlegare cerere-execuție: Command separă cererea de execuție, permițând diferite obiecte să primească și să execute aceeași cerere în mod flexibil.
    - Undo/Redo: Command permite înregistrarea operațiilor executate, facilitând operațiile de anulare și refacere.
    - Gestionare flexibilă a comenzilor: Command facilitează gestionarea comenzilor, cum ar fi adăugarea de comenzilor într-o coadă sau compunerea lor în comenzi complexe.

Facade:

* + Ce este: Facade este un design pattern structural care oferă o interfață simplificată și unificată pentru a accesa un subsistem complex, ascunzând detaliile sale interne.
  + Obiective: Facade este utilizat pentru a oferi un punct de intrare simplu și clar într-un subsistem complex, reducând dependența și cuplarea cu subsistemul.
  + Beneficii:
    - Simplitate: Facade oferă o interfață simplificată, ușor de utilizat și de înțeles pentru a accesa un subsistem complex, ascunzând detaliile sale interne.
    - Cuplare redusă: Prin intermediul Facade, codul client depinde doar de interfața Facade, eliminând astfel dependența de detaliile interne ale subsistemului.
    - Refactorizare ușoară: Facade permite modificarea și refactizarea subsistemului fără a afecta codul client, deoarece interfața Facade rămâne neschimbată.

Flyweight:

* + Ce este: Flyweight este un design pattern structural care optimizează utilizarea memoriei prin împărțirea eficientă a obiectelor în obiecte partajate (flyweights) și obiecte ne-partajate (context).
  + Obiective: Flyweight este utilizat pentru a reduce cantitatea de memorie utilizată de obiectele multiple similare prin partajarea datelor comune și reutilizarea obiectelor.
  + Beneficii:
    - Economisirea memoriei: Flyweight reduce consumul de memorie prin partajarea datelor comune între obiectele similare.
    - Reutilizare: Flyweight permite reutilizarea obiectelor partajate în diferite contexte, eliminând duplicarea obiectelor similare.
    - Performanță îmbunătățită: Utilizarea obiectelor partajate în locul obiectelor noi poate îmbunătăți performanța aplicației prin reducerea timpului și resurselor necesare pentru crearea și distrugerea obiectelor multiple similare.

În concluzie, design patterns reprezintă soluții testate și validate pentru probleme comune de proiectare și dezvoltare a software-ului. Acestea oferă un set de principii și abordări care permit dezvoltatorilor să creeze sisteme eficiente, flexibile, ușor de întreținut și extensibile.

Tipurile de design patterns, cum ar fi creational patterns, structural patterns și behavioral patterns, acoperă diferite aspecte ale dezvoltării software-ului și oferă soluții pentru probleme specifice întâlnite în aceste domenii. Fiecare tip de design pattern are obiective și beneficii specifice, dar în general, ele urmăresc să ofere modularitate, reutilizare, extensibilitate și cuplare redusă între componente.

În final, cunoașterea și utilizarea design patterns reprezintă un instrument valoros în cadrul dezvoltării software, permițând dezvoltatorilor să construiască sisteme robuste și flexibile, care să satisfacă cerințele complexe ale aplicațiilor moderne.

# 2. Descrierea aplicației

Un sistem de gestionare a resurselor bibliotecii este extrem de important într-un mediu bibliotecar modern. Acest tip de aplicație aduce multiple beneficii și contribuie la îmbunătățirea eficienței și calității serviciilor oferite. Iată câteva aspecte relevante:

Acces rapid și ușor la informații: Un sistem de gestionare a resurselor bibliotecii permite utilizatorilor să găsească rapid informațiile despre cărți, reviste, publicații sau alte materiale disponibile în bibliotecă. Aceasta facilitează procesul de căutare și elimină nevoia de a parcurge manual rafturile bibliotecii.

Eficientizarea împrumuturilor și returnărilor: Prin intermediul unui astfel de sistem, bibliotecarii pot înregistra cu ușurință împrumuturile și returnările de cărți. Astfel, se evită confuziile și erorile în gestionarea resurselor, iar utilizatorii beneficiază de o evidență clară a termenelor de returnare.

Monitorizarea stocului și gestionarea achizițiilor: Un sistem de gestionare a resurselor bibliotecii permite bibliotecarilor să monitorizeze stocul existent și să planifice achizițiile în funcție de nevoile și preferințele utilizatorilor. Astfel, se asigură un flux constant de resurse relevante și actualizate în bibliotecă.

Notificări și comunicare eficientă: Prin intermediul aplicației, bibliotecile pot trimite notificări și alerte utilizatorilor, informându-i despre împrumuturi expirate, rezervări disponibile sau evenimente speciale. Aceasta facilitează comunicarea între bibliotecă și utilizatori și îmbunătățește experiența generală a utilizatorilor.

Un sistem de gestionare a resurselor bibliotecii poate genera rapoarte și statistici relevante referitoare la împrumuturi, popularitatea anumitor cărți sau preferințele utilizatorilor. Aceste informații sunt valoroase pentru luarea deciziilor strategice și îmbunătățirea serviciilor bibliotecii.

În concluzie, un sistem de gestionare a resurselor bibliotecii aduce multiple beneficii, cum ar fi eficientizarea accesului la informații, optimizarea procesului de împrumut și returnare, gestionarea stocului și achizițiilor, comunicarea eficientă cu utilizatorii și generarea de rapoarte și statistici relevante. Toate aceste aspecte contribuie la îmbunătățirea calității serviciilor bibliotecii și la oferirea unei experiențe superioare utilizatorilor.

## **2.1 Scopul aplicației**

Aplicația are ca scop principal crearea și gestionarea unei biblioteci virtuale. Scopul său este de a oferi utilizatorilor o platformă unde pot adăuga, organiza și vizualiza cărți și jurnale într-un mod eficient și ușor de utilizat.Unul dintre obiectivele principale ale aplicației este să permită utilizatorilor să adauge noi cărți și jurnale în bibliotecă. Prin intermediul unei interfețe intuitive, utilizatorii pot introduce informații despre titlu, autor, gen și preț pentru fiecare produs. De asemenea, aplicația oferă posibilitatea de a adăuga descrieri suplimentare pentru produsele existente.

Un alt obiectiv important al aplicației este gestionarea și organizarea colecției de cărți și jurnale. Utilizatorii pot sorta produsele în funcție de diferite criterii, cum ar fi titlul sau prețul. Astfel, pot găsi rapid și eficient produsele dorite în bibliotecă. De asemenea, aplicația oferă funcționalitatea de a afișa numărul total de produse din bibliotecă, furnizând astfel utilizatorilor o perspectivă asupra dimensiunii colecției.Aplicația are și un scop de a facilita gestionarea genurilor de cărți și jurnale. Prin intermediul unui obiect de tip GenreFactory, aplicația permite crearea și stocarea genurilor de produse. Astfel, utilizatorii pot asocia genurile corespunzătoare fiecărui produs adăugat în bibliotecă, asigurând astfel o organizare mai bună și o căutare mai eficientă a produselor.

Scopul general al aplicației este de a oferi utilizatorilor o experiență plăcută și eficientă în gestionarea unei biblioteci virtuale. Prin intermediul unei interfețe prietenoase și a funcționalităților bine definite, aplicația urmărește să ofere o soluție completă pentru organizarea și accesarea colecției de cărți și jurnale.

Un alt aspect important al aplicației este capacitatea sa de a afișa produsele într-un mod flexibil și personalizat. Utilizatorii pot selecta strategii de sortare, cum ar fi sortarea după titlu sau preț, pentru a-și adapta vizualizarea bibliotecii conform preferințelor lor. Astfel, aplicația oferă o experiență personalizată și permite utilizatorilor să găsească rapid și ușor produsele dorite.

În concluzie, scopul aplicației este de a oferi utilizatorilor o platformă eficientă și ușor de utilizat pentru gestionarea unei biblioteci virtuale. Aplicația își propune să faciliteze adăugarea, organizarea, vizualizarea și gestionarea cărților și jurnalelor, oferind o experiență personalizată și extensibilă. Prin intermediul funcționalităților sale și a pattern-urilor de proiectare implementate, aplicația urmărește să îmbunătățească eficiența și plăcerea utilizatorilor în interacțiunea cu biblioteca virtuală.

## **2.2 Funcționalitățile aplicației**

Aplicația are mai multe funcționalități care permit utilizatorilor să gestioneze și să exploreze biblioteca virtuală într-un mod eficient și plăcut. Iată câteva dintre funcționalitățile principale:

1. Adăugarea de produse: Utilizatorii pot adăuga noi cărți sau jurnale în biblioteca virtuală. Aplicația permite specificarea tipului de produs (cărți sau jurnale), titlului, autorului, genului, prețului și descrierii. Această funcționalitate oferă posibilitatea de a completa rapid și ușor detaliile produselor și de a le integra în bibliotecă.
2. Sortarea produselor: Utilizatorii pot sorta produsele din bibliotecă folosind strategii de sortare. Aplicația oferă două strategii predefinite: sortarea după titlu și sortarea după preț. Utilizatorii pot alege strategia dorită și produsele vor fi afișate în ordinea specificată. Această funcționalitate facilitează găsirea și compararea produselor în funcție de preferințele utilizatorilor.
3. Vizualizarea produselor: Aplicația afișează produsele din bibliotecă într-un mod organizat și intuitiv. Utilizatorii pot explora lista de produse și pot vedea detalii precum titlul, autorul, genul, prețul și descrierea (dacă există). Acest aspect facilitează identificarea rapidă a produselor dorite și oferă informații relevante pentru fiecare produs în parte.
4. Numărarea produselor: Aplicația furnizează informații despre numărul total de produse din bibliotecă. Utilizatorii pot accesa funcționalitatea de numărare a produselor pentru a obține o imagine de ansamblu asupra dimensiunii bibliotecii și a evoluției sale în timp.
5. Gestionarea genurilor: Aplicația utilizează un sistem de genuri pentru a categoriza produsele din bibliotecă. Utilizatorii pot adăuga noi genuri în sistem și pot asocia genuri existente produselor. Această funcționalitate facilitează organizarea și căutarea produselor în funcție de genurile lor.
6. Extensibilitate: Aplicația este proiectată pentru a fi ușor de extins cu noi tipuri de produse sau funcționalități suplimentare. Utilizând pattern-uri de proiectare precum Abstract Factory și Builder, aplicația poate fi adaptată pentru a accepta noi tipuri de produse sau pentru a adăuga caracteristici suplimentare în viitor.

Acestea sunt doar câteva dintre funcționalitățile cheie ale aplicației. În general, aplicația își propune să ofere o experiență completă și plăcută utilizatorilor în gestionarea și explorarea bibliotecii virtuale.

## **2.3 Tehnologiile utilizate**

JavaScript este un limbaj de programare de înaltă nivel care este folosit în principal pentru dezvoltarea aplicațiilor web. Acesta oferă funcționalități puternice de interacțiune cu utilizatorul și manipulare a elementelor HTML, permițând dezvoltatorilor să creeze pagini web interactive și dinamice.

Arhitectura și tehnologiile utilizate în codul dat sunt următoarele:

* Strategy Pattern: Clasa SortStrategy este clasa de bază pentru strategiile de sortare, iar clasele TitleSortStrategy și PriceSortStrategy sunt implementări specifice ale strategiilor de sortare după titlu și preț.
* Flyweight Factory: Clasa GenreFactory este o fabrică care gestionează obiecte de tipul Genre (gen) pentru a evita crearea multiplă a acelorași genuri.
* Iterator Pattern: Clasa LibraryIterator implementează un iterator pentru a itera prin produsele din bibliotecă.
* Singleton Pattern: Clasa Library este implementată ca un singleton pentru a permite o singură instanță a bibliotecii.
* Abstract Factory Pattern: Clasa ProductAbstractFactory este o fabrică abstractă care definește metoda createProductBuilder(), care este implementată în clasele concrete BookFactory și JournalFactory. Aceste fabrici concrete sunt utilizate pentru a crea instanțe ale claselor Book și Journal.
* Builder Pattern: Clasa ProductBuilder este o clasă de bază pentru construirea produselor din bibliotecă, având metode pentru setarea atributelor produsului și metoda abstractă build(), care trebuie implementată în clasele derivate. Clasele BookBuilder și JournalBuilder extind ProductBuilder și implementează metoda build() pentru a crea instanțe ale claselor Book și Journal.
* Decorator Pattern: Clasa ProductWithDescription este un decorator pentru produsele din bibliotecă, adăugând o descriere suplimentară la produsul existent.
* Facade Pattern: Clasa LibraryFacade oferă o interfață simplificată pentru interacțiunea cu biblioteca, ascunzând detalii și oferind metode conveniente pentru adăugarea și afișarea produselor.
* Command Pattern: Clasele Command, SortCommand și AddProductCommand implementează acest pattern pentru a încapsula operațiile ca obiecte și a permite invocarea lor ulterior. Astfel, se pot crea și se pot executa comenzi pentru sortarea și adăugarea de produse.

Tehnologiile utilizate în codul dat includ limbajul JavaScript, HTML și interacțiunea cu elementele DOM prin intermediul bibliotecii JavaScript. Codul pare să fie scris pentru un mediu de dezvoltare web, unde interacțiunea cu utilizatorul și afișarea rezultatelor are loc într-un browser web.

# 3. Implementarea Design Pattern-urilor

Implementarea design pattern-urilor reprezintă o etapă crucială în dezvoltarea software-ului, asigurând o structură coerentă, modulară și extensibilă a aplicației. Design pattern-urile sunt soluții recurente și testate pentru probleme comune întâlnite în dezvoltarea software, împărțite în trei categorii principale: creational, structural și behavioral.

Design pattern-urile creational se concentrează pe modul în care obiectele și clasele sunt create și inițializate într-un mod flexibil și eficient. Acestea facilitează crearea de instanțe ale claselor și obiecte într-un mod controlat, oferind posibilitatea de a ascunde detalii de implementare și de a gestiona diferitele tipuri de instanțe. Aceste pattern-uri includ Factory, Singleton, Abstract Factory, Builder și Prototype.

Design pattern-urile structural se concentrează pe organizarea și relaționarea obiectelor și claselor pentru a crea structuri mai complexe. Acestea facilitează gestionarea relațiilor între obiecte și promovează extensibilitatea și flexibilitatea sistemului. Printre design pattern-urile structurale se numără Adapter, Bridge, Composite, Decorator, Facade și Proxy.

Design pattern-urile behavioral se concentrează pe comunicarea și interacțiunea între obiecte și clase. Acestea oferă soluții pentru gestionarea comportamentului și algoritmului din cadrul aplicației. Aceste pattern-uri includ Chain of Responsibility, Command, Iterator, Mediator, Observer, State, Strategy și Template Method.

Sabloanele utilizate în implementarea proiectului nostru acoperă toate cele trei categorii de design pattern-uri: creational, structural și behavioral. Prin utilizarea acestor sabloane, am asigurat o structură coerentă și modulară a aplicației noastre, facilitând întreținerea și extensibilitatea ulterioară.Sabloanele utilizate sunt: Singleton, Builder, AbstractFactory, Decorator, Façade, Flyweight, Command, Strategy, Iterator.

Am selectat sabloanele Singleton, Builder, Abstract Factory, Decorator, Façade, Flyweight, Command, Strategy și Iterator pentru a rezolva probleme specifice din cadrul proiectului nostru de dezvoltare software. Fiecare dintre aceste sabloane aduce beneficii semnificative în ceea ce privește flexibilitatea, scalabilitatea și ușurința de întreținere a aplicației.

## **3.1 Șabloane Creaționale**

Sabloanele creationale sunt un subset al design pattern-urilor care se concentrează pe procesul de creare și inițializare a obiectelor. Acestea oferă soluții și abordări standardizate pentru crearea instanțelor de obiecte într-un mod flexibil, eficient și controlat. Sabloanele creationale ajută la separarea procesului de creare a obiectelor de utilizarea acestora, permițând o mai mare modularitate și ușurință în gestionarea dependențelor.

Am implementat următoarele 3 șabloane creaționale: Singleton, AbstractFactory, Builder.

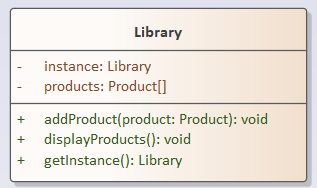


Figura 1 – Structura Singleton

Pattern-ul Singleton este implementat în clasa Library. În metoda getInstance(), se verifică dacă există deja o instanță a clasei Library. Dacă nu există, se creează o nouă instanță și se atribuie la proprietatea statică instance a clasei. Dacă există deja o instanță, se returnează instanța existentă. În acest fel, se garantează că există o singură instanță a clasei Library în întregul program și se permite accesul la această instanță prin intermediul metodei getInstance().

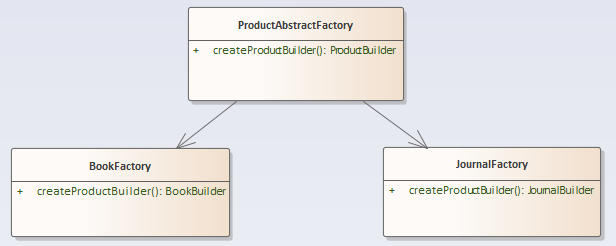


Figura 2 – Structura Abstract Factory

Există o ierarhie de clase care definesc fabrici concretele (BookFactory și JournalFactory) care implementează clasa abstractă ProductAbstractFactory. Fabricile concrete, cum ar fi BookFactory și JournalFactory, extind clasa ProductAbstractFactory și implementează metoda createProductBuilder(). Această metodă creează și returnează un obiect ProductBuilder corespunzător. De exemplu, în BookFactory, metoda createProductBuilder() returnează un obiect BookBuilder.

Pattern-ul Abstract Factory permite crearea de obiecte dintr-o familie de clase într-un mod modular și coerent. În acest caz, BookFactory și JournalFactory sunt responsabile pentru crearea și configurarea obiectelor Book și Journal, respectiv, folosind BookBuilder și JournalBuilder. Aceasta permite flexibilitate în adăugarea și gestionarea diferitelor tipuri de produse (cărți, jurnale) în cadrul bibliotecii.

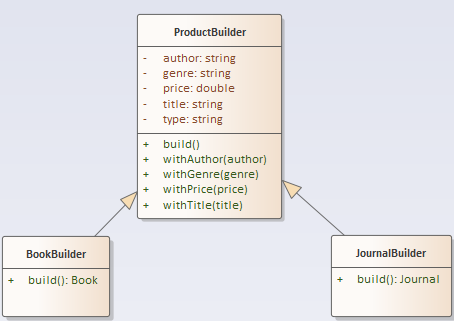


Figura 3 – Structura Builder

Există clasa ProductBuilder care este o clasă abstractă pentru construirea produselor. Aceasta are metode pentru a seta diferitele atribute ale unui produs, cum ar fi titlul, autorul, genul și prețul. Există, de asemenea, două clase concrete care extind ProductBuilder: BookBuilder și JournalBuilder, care construiesc obiecte de tip Book și Journal, respectiv.

Folosind patternul Builder, se poate construi un produs pas cu pas, setându-i atributele în mod incremental și apoi apelând metoda build() pentru a obține obiectul final. Acest pattern permite crearea de obiecte complexe cu diferite variante de inițializare, fără a complica clasa principală. În proiectul dat, se folosește patternul Builder pentru a construi obiecte de tip Book și Journal cu diverse atribute, cum ar fi titlul, autorul, genul și prețul.

## **3.2 Șabloane Structurale**

Sabloanele structurale sunt un set de design pattern-uri care se concentrează pe organizarea și relaționarea obiectelor și claselor într-un mod eficient și flexibil. Acestea facilitează construirea de structuri complexe prin intermediul interacțiunii dintre diferitele componente ale sistemului software. Prin utilizarea sabloanelor structurale, putem obține o arhitectură mai modulară, ușor de înțeles și de întreținut.

Am implementat următoarele 3 șabloane structurale: Decorator, Flyweight, Facade.

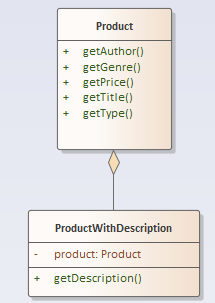


Figura 4 – Structura Decorator

Clasa ProductWithDescription primește un obiect de tip product în constructor, iar prin intermediul metodelor sale, adaugă funcționalitatea de descriere la produsul existent. Metoda getDescription() este înlocuită pentru a adăuga descrierea la descrierea existentă a produsului.



Figura 5 – Structura Facade

Clasa LibraryFacade acționează ca o fațadă peste clasa Library, oferind o interfață simplificată pentru a accesa și utiliza funcționalitățile bibliotecii.

Aceasta abstractizează logica complexă și oferă metode mai simple și mai intuitive pentru adăugarea, sortarea și afișarea produselor. Facade Pattern simplifică utilizarea și integrarea funcționalității complexe, oferind o interfață coezivă și ușor de utilizat pentru utilizatorii săi.

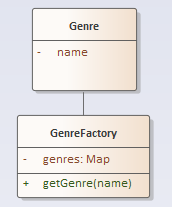


Figura 5 – Structura Flyweight

Clasa GenreFactory menține o referință către obiectele de tip Genre și le partajează în funcție de numele genului. În momentul în care se solicită un gen specific prin intermediul metodei getGenre, se verifică mai întâi dacă genul există deja în cache (obiectele genres). Dacă genul există, este returnată o referință către acel obiect, altfel este creat un obiect Genre nou și este adăugat în cache, aducând beneficii în ceea ce privește economisirea memoriei în cazul în care există mai multe obiecte Genre cu același nume în sistem.

În concluzie, sabloanele structurale reprezintă un set de design patterns care se concentrează pe organizarea și compunerea obiectelor pentru a forma structuri mai complexe în cadrul unei aplicații software. Aceste sabloane abordează problemele legate de interacțiunea și relațiile dintre obiecte, oferind o structură clară, flexibilă și ușor de înțeles.

Beneficiile utilizării sabloanelor structurale sunt multiple. Ele permit separarea responsabilităților între diferitele componente ale aplicației, reduc cuplarea strânsă între acestea și facilitează reutilizarea și extensibilitatea codului. De asemenea, sabloanele structurale contribuie la creșterea modularității și la îmbunătățirea performanței aplicației prin optimizarea interacțiunii și comunicării între obiecte.

În general, utilizarea sabloanelor structurale poate aduce beneficii semnificative în ceea ce privește structura, flexibilitatea și performanța aplicațiilor software, oferind dezvoltatorilor un set de instrumente și abordări testate și validate pentru a construi sisteme robuste și scalabile.

## **3.3 Șabloane Comportamentale**

Sabloanele comportamentale reprezintă un set de design pattern-uri care se concentrează pe gestionarea comportamentului și interacțiunii între obiecte și clase în cadrul unei aplicații software. Aceste sabloane oferă soluții și abordări standardizate pentru gestionarea algoritmilor, comunicării și manipulării evenimentelor într-un mod flexibil și modular.

Am implementat următoarele 3 șabloane comportamentale: Command, Strategy, Iterator.

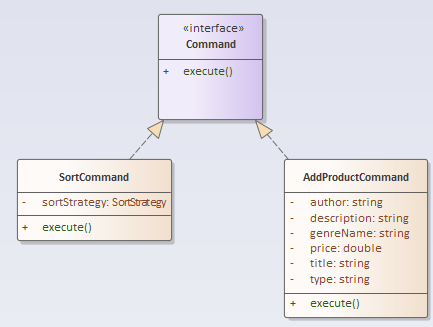


Figura 7 – Structura Command

Clasa Command reprezintă clasa abstractă pentru toate comenzile din sistem. Ea definește metoda execute() pe care toate clasele derivate trebuie să o implementeze. Această metodă este apelată pentru a executa acțiunea specifică a comenzii.

În cadrul proiectului, sunt implementate două clase concrete de comenzi:

1. SortCommand: Această clasă reprezintă o comandă de sortare a produselor din bibliotecă. Aceasta primește o instanță a clasei LibraryFacade și o strategie de sortare și le utilizează pentru a sorta produsele și a le afișa în bibliotecă.
2. AddProductCommand: Această clasă reprezintă o comandă de adăugare a unui produs în bibliotecă. Aceasta primește o instanță a clasei LibraryFacade și toate informațiile necesare pentru crearea unui produs (tip, titlu, autor, gen, preț, descriere). Comanda utilizează LibraryFacade pentru a adăuga produsul în bibliotecă.

Ambele clase de comenzi implementează metoda execute() și sunt utilizate în interacțiunea cu interfața utilizatorului (UI) pentru a efectua acțiunile specifice, cum ar fi sortarea produselor sau adăugarea unui produs nou.

Astfel, prin utilizarea pattern-ului Command, se realizează o separare între logica de comandă și execuția acesteia, oferind flexibilitate și extensibilitate în sistemul de gestionare a bibliotecii.

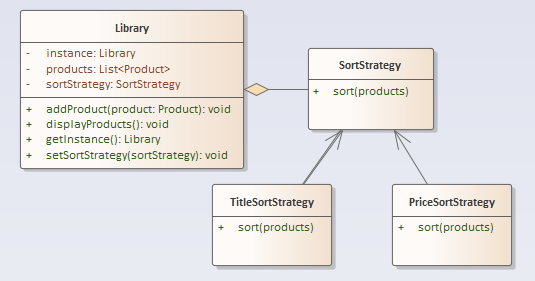


Figura 8 – Structura Strategy

Pattern-ul strategy poate fi observat prin clasele SortStrategy, TitleSortStrategy și PriceSortStrategy.

Clasa abstractă SortStrategy definește o interfață comună pentru toate strategiile de sortare. Aceasta conține o metodă sort(products) care este implementată în clasele derivate.

Clasa TitleSortStrategy și clasa PriceSortStrategy sunt implementări concrete ale interfeței SortStrategy. Acestea definesc algoritmi de sortare specifici pentru sortarea după titlu și sortarea după preț.

În clasa Library, există o instanță a clasei SortStrategy (atributul sortStrategy) care poate fi setată prin metoda setSortStrategy(sortStrategy). Astfel, se permite schimbarea comportamentului de sortare al obiectelor din bibliotecă în timpul execuției prin intermediul strategiilor.

Prin utilizarea patternului Strategy, se realizează o separare a algoritmului de sortare de clasele care le utilizează, oferind flexibilitate și posibilitatea de a adăuga și utiliza noi strategii de sortare în viitor.

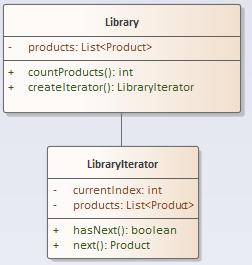


Figura 8 – Structura Iterator

Clasa LibraryIterator implementează patternul Iterator și servește ca un iterator pentru a parcurge colecția de produse din clasa Library. Ea pune la dispoziție metode precum hasNext() pentru a verifica dacă există un următor element în colecție și next() pentru a obține următorul element.

În proiect, patternul Iterator este utilizat în clasa Library pentru a itera prin colecția de produse. Metoda createIterator() din clasa Library creează un obiect LibraryIterator, care este un iterator specific pentru clasa Library. Acest iterator este returnat și poate fi utilizat pentru a itera prin colecție.

De exemplu, în metoda countProducts(), se utilizează iteratorul pentru a parcurge colecția de produse și a număra câte produse există. Metoda displayProducts() utilizează, de asemenea, iteratorul pentru a itera prin produse și a afișa detaliile acestora în interfața utilizatorului.

Astfel, prin utilizarea patternului Iterator, se abstractizează modul în care se parcurge colecția de produse din clasa Library, permițând o iterație simplă și flexibilă fără a expune detalii despre structura internă a colecției.

# Concluzii

Realizarea acestui proiect a implicat implementarea mai multor pattern-uri de design, inclusiv Singleton, Builder, Abstract Factory, Decorator, Iterator, Strategy, Command, Facade și Flyweight. Fiecare pattern a adus beneficii specifice și a contribuit la structura și funcționalitatea proiectului.

Utilizarea pattern-ului Singleton a asigurat faptul că clasa Library are o singură instanță în cadrul aplicației, permițând accesul la această instanță din orice parte a proiectului.Pattern-ul Builder a fost utilizat pentru a crea obiecte de produs (cărți și reviste) într-un mod flexibil și modular. Acesta a permis setarea opțională a diferitelor atribute ale produselor și crearea lor într-un mod coerent și ușor de înțeles.

Abstract Factory a fost utilizat pentru a crea fabrici de produse (BookFactory și JournalFactory) care au generat obiecte de produs corespunzătoare. Această abordare a permis crearea de produse într-un mod abstract, fără a specifica concret tipul de produs generat.

Decorator a fost folosit pentru a extinde funcționalitatea produselor prin adăugarea unei descrieri suplimentare. Aceasta a permis atașarea unui obiect ProductWithDescription la produsele existente pentru a le extinde comportamentul fără a modifica direct clasa de bază a produsului.Pattern-ul Iterator a fost utilizat pentru a itera prin lista de produse din clasa Library într-un mod secvențial și ușor de utilizat. Acesta a oferit o abstracție pentru parcurgerea elementelor colecției fără a expune detalii de implementare.

Pattern-ul Strategy a fost utilizat pentru a implementa diferite strategii de sortare a produselor în funcție de titlu sau preț. Aceasta a permis schimbarea comportamentului de sortare fără a modifica structura produselor sau a clasei Library. Pattern-ul Command a fost folosit pentru a encapsula operațiile pe obiectele LibraryFacade. Acesta a permis crearea de comenzi independente care pot fi executate și inversate în mod flexibil și modular.

Facade a fost utilizat pentru a oferi o interfață simplificată și unificată către funcționalitatea complexă a bibliotecii. Aceasta a ascuns detalii de implementare și a oferit o modalitate simplă de interacțiune cu biblioteca.Flyweight a fost utilizat în implementarea clasei GenreFactory pentru a optimiza utilizarea resurselor în ceea ce privește obiectele de tip Genre. Acesta a asigurat că, în cazul în care un gen există deja, se returnează aceeași instanță în loc să creeze una nouă.

În concluzie, implementarea acestor pattern-uri de design în proiectul respectiv a adus beneficii semnificative, cum ar fi modularitate, flexibilitate, ușurință în extindere și gestionarea eficientă a resurselor. Utilizarea acestor pattern-uri a contribuit la o structură coerentă a proiectului și a facilitat dezvoltarea și întreținerea acestuia într-un mod mai eficient.

# Bibliografie

1. Shalloway A. Design Patterns Explained A New Perspective on Object-Oriented Design. https://www.oreilly.com/library/view/design-patterns-explained/9780321630049/
2. Design Patterns. https://refactoring.guru/design-patterns
3. Gamma E. Design Patterns in Dynamic Programming. https://dl.acm.org/doi/abs/10.1145/21619.21636
4. Freeman E. Head First Design Patterns. https://www.oreilly.com/library/view/head-first-design/0596007124/