**MINISTERUL EDUCAȚIEI și CERCETĂRII**

**UNIVERSITATEA TEHNICĂ a MOLDOVEI**

**FACULTATEA CALCULATOARE, INFORMATICĂ și MICROELECTRONICĂ**

**DEPARTAMENTUL INGINERIA SOFTWARE și AUTOMATICĂ**

**Lucrare de curs**

Disciplina: Tehnici și Mecanizme de Proiectare Software

Tema: „Analiza și modelarea unei aplicații care va gestiona necesitățile unui Biblioteci online”

„Analysis and modeling of an application that will manage the needs of an online Library”

A efectuat st.gr. TI-202 Bunescu Gabriel.

A verificat lect.univ. Postaru Andrei.

asis.univ. Gaidau Mihai.

Chișinău 2023

Cuprins:

[Introducere 3](#_Toc136594699)

[1. Modele de design 4](#_Toc136594700)

[1.1 Utilizări ale modelelor de design 4](#_Toc136594701)

[1.2 Modele de design creațional 5](#_Toc136594702)

[1.2.1 Abstract Factory Design Pattern 5](#_Toc136594703)

[1.2.2 Builder Design Pattern 6](#_Toc136594704)

[1.2.3 Singleton Design Pattern 7](#_Toc136594705)

[1.3 Modele de proiectare structurală 8](#_Toc136594706)

[1.3.1 Decorator Design Pattern 9](#_Toc136594707)

[1.3.2 Proxy Design Pattern 10](#_Toc136594708)

[1.4 Behavioral design patterns 11](#_Toc136594709)

[1.4.1 Chain of Responsibility 11](#_Toc136594710)

[1.4.2 Command Design Pattern 13](#_Toc136594711)

[1.4.3 Observer Design Pattern 14](#_Toc136594712)

[2 Prezentarea Aplicației 15](#_Toc136594713)

[2.1 Modelele de design creațional 15](#_Toc136594714)

[2.1.1 Metoda Factory 16](#_Toc136594715)

[2.1.2 Builder 17](#_Toc136594716)

[2.1.3 Singleton 18](#_Toc136594717)

[2.2 Modelele de proiectare structurală 19](#_Toc136594718)

[2.2.1 Decorator 20](#_Toc136594719)

[2.2.2 Proxy 21](#_Toc136594720)

[2.3 Behavioral design patterns 22](#_Toc136594721)

[2.3.1 Command 23](#_Toc136594722)

[2.3.2 Chain of Responsibility 24](#_Toc136594723)

[2.3.3 Observator 26](#_Toc136594724)

[3 Diagrama UML Generală: 27](#_Toc136594725)

[Concluzie 29](#_Toc136594726)

[Bibliografie: 30](#_Toc136594727)

# Introducere

*"Designul este arta de a transforma problemele în soluții elegante."* - Christopher Alexander

În dezvoltarea aplicațiilor software, pattern-urile de design reprezintă un set de soluții recurente și testate, care ne ajută să abordăm problemele comune cu eficiență și eleganță. Acestea oferă structură, organizare și flexibilitate, facilitând crearea și întreținerea unui cod de calitate.

În cadrul acestui raport, vom explora aplicarea unor pattern-uri de design în dezvoltarea unei aplicații Python pentru o bibliotecă online. Scopul bibliotecii este de a oferi utilizatorilor săi o varietate de produse, precum cărți, filme și albume muzicale, într-un mod accesibil și ușor de utilizat.

Fiecare pattern de design reprezintă o piesă distinctă a unui puzzle complex, contribuind la întregul tablou al aplicației noastre. Vom aduce în discuție câteva dintre aceste pattern-uri și modul în care se integrează în cadrul bibliotecii noastre online.

Vom începe prin a explora Method Factory, un pattern care ne permite să abstractizăm procesul de creare a diferitelor tipuri de produse, cum ar fi cărți, filme și albume muzicale. Prin intermediul acestui pattern, putem defini o interfață comună pentru crearea acestor produse, în timp ce lăsăm subclaselor să decidă clasa concretă a obiectului care urmează să fie creat.

Un alt pattern de design pe care îl vom analiza este Singleton, care ne permite să avem o singură instanță a unei clase și să oferim un punct global de acces la aceasta.

*"Un design bun este un design care permite schimbare fără a necesita o reconstrucție majoră."* - Fred Brooks

De asemenea, vom explora pattern-ul Builder, care ne permite să separăm procesul de construcție a unui obiect complex de reprezentarea sa finală. Acest pattern ne ajută să creăm și să configurăm obiecte precum cărți, filme și albume muzicale într-un mod modular și flexibil.

Decoratori și Proxy sunt alte două pattern-uri pe care le vom analiza. Decoratorul ne permite să adăugăm funcționalități suplimentare la obiectele existente, în timp ce Proxy-ul ne permite să creăm obiecte intermediare care controlează accesul la alte obiecte.

În final, vom explora pattern-urile Command, Observator și Chain of Responsibility. Pattern-ul Command transformă solicitările în obiecte independente, permițându-ne să le parametrizăm și să le gestionăm în mod eficient. Observatorul ne permite să notificăm automat obiectele dependente de modificările de stare ale altor obiecte, iar Chain of Responsibility ne ajută să gestionăm și să direcționăm solicitările utilizatorilor într-un mod flexibil și modular.

Prin aplicarea și analiza acestor pattern-uri de design în cadrul aplicației noastre pentru biblioteca online, vom evidenția beneficiile pe care le oferă în termeni de modularitate, extensibilitate și ușurință de întreținere a codului. Astfel, vom crea o aplicație robustă și scalabilă, care răspunde nevoilor și cerințelor utilizatorilor noștri.

# Modele de design

În ingineria software, un model de proiectare este o soluție generală repetabilă la o problemă frecventă în proiectarea software. Un model de design nu este un design finit care poate fi transformat direct în cod. Este o descriere sau un șablon pentru rezolvarea unei probleme care poate fi folosită în multe situații diferite.[1]

**Motivația alegerii temei:**

* Relevanța și utilitatea în industria software: Pattern-urile de design sunt soluții practice și recunoscute la probleme comune întâlnite în dezvoltarea software. Alegerea acestei teme demonstrează dorința ta de a înțelege și de a aplica practici eficiente în proiectarea software.
* Creșterea nivelului de abstracție: Pattern-urile de design furnizează o modalitate structurată de a gândi și de a comunica despre design-ul software. Prin învățarea și aplicarea acestor pattern-uri, vei putea exprima concepte complexe într-un mod clar și concis.
* Îmbunătățirea calității și flexibilității software-ului: Pattern-urile de design sunt dezvoltate și rafinate de-a lungul timpului pentru a rezolva probleme specifice și a promova principii de design solid. Prin utilizarea lor în proiectele tale, vei putea crea sisteme mai modulare, extensibile și ușor de întreținut.
* Îmbogățirea bagajului de cunoștințe și dezvoltarea abilităților profesionale: Cunoașterea și aplicarea pattern-urilor de design reprezintă o competență valoroasă în industria IT. Alegând această temă, vei putea obține o înțelegere mai profundă a conceptelor de bază în designul software și vei putea dezvolta abilități utile în cariera ta.
* Explorarea unui domeniu vast și divers: Există o varietate largă de pattern-uri de design, fiecare adresându-se unor probleme și contexte diferite. Prin explorarea acestor pattern-uri, vei putea înțelege diversitatea abordărilor și vei putea selecta și aplica cele mai potrivite soluții în proiectele tale.

## Utilizări ale modelelor de design

Modelele de proiectare pot accelera procesul de dezvoltare prin furnizarea de paradigme de dezvoltare testate și dovedite. Proiectarea eficientă a software-ului necesită luarea în considerare a problemelor care pot să nu devină vizibile decât mai târziu în implementare. Reutilizarea modelelor de proiectare ajută la prevenirea problemelor subtile care pot cauza probleme majore și îmbunătățește lizibilitatea codului pentru programatori și arhitecții familiarizați cu modelele.

Adesea, oamenii înțeleg doar cum să aplice anumite tehnici de proiectare software la anumite probleme. Aceste tehnici sunt dificil de aplicat la o gamă mai largă de probleme. Modelele de design oferă soluții generale, documentate într-un format care nu necesită specificații legate de o anumită problemă.

În plus, modelele permit dezvoltatorilor să comunice folosind nume binecunoscute și bine înțelese pentru interacțiunile software. Modelele comune de design pot fi îmbunătățite în timp, făcându-le mai robuste decât modelele ad-hoc.[2]

## Modele de design creațional

Modelele de design creațional sunt un set de pattern-uri care se concentrează pe crearea și inițializarea obiectelor într-un sistem software. Acestea oferă soluții pentru gestionarea procesului de creare a obiectelor, ascunzând detalii de implementare și oferind flexibilitate și extensibilitate în crearea instanțelor de obiecte.

Pattern-urile de design creațional abordează aspecte precum instanțierea obiectelor, gestionarea dependențelor și configurarea acestora. Ele oferă un ghid și o structură pentru implementarea procesului de creare a obiectelor într-un mod eficient și reutilizabil.

Prin utilizarea modelelor de design creațional, dezvoltatorii pot obține o abordare mai flexibilă și modulară în crearea obiectelor. Acestea promovează separarea responsabilităților, encapsularea logicii de creare și furnizează un nivel de abstractizare în procesul de instanțiere a obiectelor.

În general, modelele de design creațional sunt utile în dezvoltarea software pentru a rezolva probleme legate de crearea și inițializarea obiectelor într-un mod controlat și eficient. Ele contribuie la crearea unui design scalabil, ușor de extins și de întreținut, și facilitează reutilizarea și configurarea flexibilă a obiectelor în cadrul unui sistem.[3]

### Abstract Factory Design Pattern

Abstract Factory este un pattern de design care oferă o modalitate de a crea familii de obiecte interdependente fără a specifica clasele concrete ale acestor obiecte. Acesta se încadrează în categoria pattern-urilor creationale și urmărește principiul de proiectare "Programați la o interfață, nu la o implementare".

Definiția abstractă a acestui pattern implică utilizarea unei clase abstracte denumite "Abstract Factory", care servește drept interfață pentru crearea obiectelor. Această interfață declară metode pentru crearea fiecărui tip de obiect în familie. Implementarea concretă a acestei clase abstracte, numită "Concrete Factory", furnizează implementările specifice pentru crearea obiectelor în funcție de familia lor.

În cadrul Abstract Factory, există o asociere strânsă între produsele create și familiile de obiecte. Fiecare familie de obiecte reprezintă un set de clase conexe care sunt create în mod coerent și pot interacționa între ele. Aceste clase conexe sunt adesea proiectate pentru a lucra împreună și a rezolva o anumită problemă sau a oferi o funcționalitate specifică.

Prin utilizarea Abstract Factory Design Pattern, se obține o abstracție suplimentară și se izolează codul client de clasele concrete ale obiectelor. Astfel, clientul utilizează doar interfața Abstract Factory și nu are cunoștință directă despre clasele concrete care sunt create. Acest lucru oferă o mai mare flexibilitate în schimbul familiei de obiecte utilizate, permițând schimbări ale implementării fără a afecta codul client.

Prin urmare, Abstract Factory Design Pattern facilitează crearea și utilizarea eficientă a familii de obiecte interdependente, oferind o abordare flexibilă și modulară în proiectarea sistemelor software.

**Problemă**

Dacă o aplicație trebuie să fie portabilă, trebuie să încapsuleze dependențele platformei. Aceste „platforme” ar putea include: sistem de ferestre, sistem de operare, bază de date etc. De prea multe ori, această încapsulare nu este proiectată în avans și o mulțime de #ifdefdeclarații de caz cu opțiuni pentru toate platformele suportate în prezent încep să se procreeze ca iepurii pe tot parcursul codului.

**Structura**

Fabrica abstractă definește o metodă de fabrică per produs. Fiecare Metodă de fabrică încapsulează newoperatorul și clasele de produse concrete, specifice platformei. Fiecare „platformă” este apoi modelată cu o clasă derivată din Factory.[6]

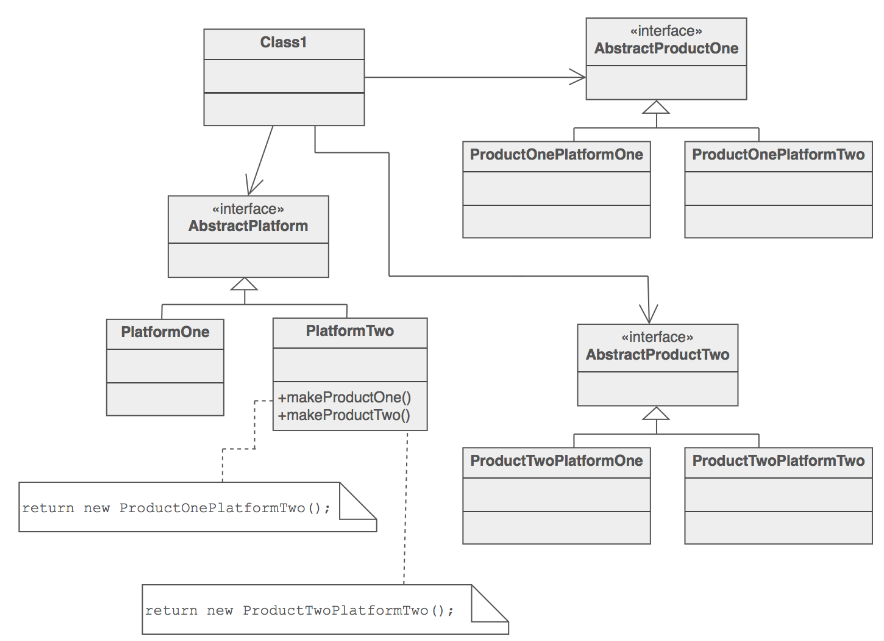


Figura 2.1 – Structura Abstract Factory Design Pattern.

### Builder Design Pattern

Builder Design Pattern este un pattern de design care permite construirea obiectelor complexe pas cu pas, separând procesul de construcție de obiectul final. Acesta aparține categoriei pattern-urilor creationale și este util atunci când există nevoia de a crea obiecte complexe cu diferite configurații sau variante, dar fără a complica clasa constructorului.

Definiția sa implică utilizarea unei clase separate numite "Builder", care este responsabilă de construirea obiectului. Builderul oferă metode de construcție pentru fiecare parte a obiectului și permite setarea opțională a proprietăților acestuia într-o manieră structurată.

Beneficiile pattern-ului Builder includ crearea de cod mai clar și mai ușor de întreținut, posibilitatea de a construi obiecte complexe etapizat și posibilitatea de a gestiona diferite configurații sau variante ale obiectului într-un mod coerent.

Astfel, Builder Design Pattern este folosit în situații în care construirea obiectelor complexe necesită un proces structurat și modular, facilitând crearea și personalizarea obiectelor într-un mod flexibil și extensibil.

**Problemă**

O aplicație trebuie să creeze elementele unui agregat complex. Specificația pentru agregat există pe stocarea secundară și una dintre multele reprezentări trebuie să fie construită în stocarea primară.

**Structura**

Reader-ul încapsulează analiza intrării comune. Ierarhia Builder face posibilă crearea polimorfă a multor reprezentări sau ținte deosebite.[7]

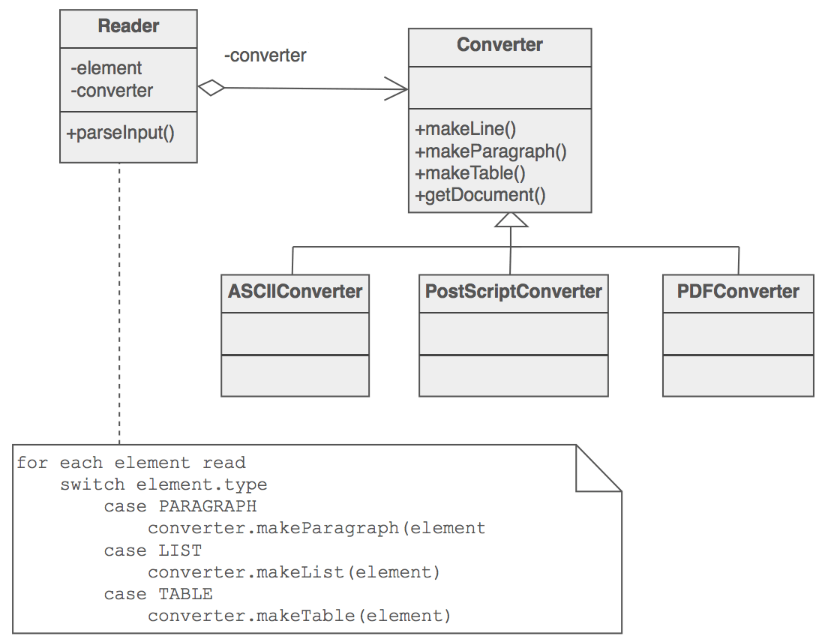


Figura 2.2 – Structura Builder Design Pattern.

### Singleton Design Pattern

Singleton Design Pattern este un pattern de design care permite crearea unei singure instanțe a unei clase și asigură că această instanță este accesibilă global. Acesta aparține categoriei pattern-urilor creationale și este utilizat atunci când există nevoia de a avea o singură instanță a unei clase în întregul sistem și de a oferi un punct centralizat de acces la acea instanță.

Definiția sa implică utilizarea unei clase speciale denumite "Singleton", care conține o metodă statică pentru a returna instanța unică a clasei. Constructorul clasei este privat sau protejat, astfel încât să nu poată fi creată o instanță nouă din exteriorul clasei. În schimb, instanța unică este creată în interiorul clasei și este păstrată într-un câmp static.

Prin utilizarea Singleton Design Pattern, se obține garantarea faptului că există o singură instanță a clasei în întregul sistem și că aceasta poate fi accesată de la orice parte a codului. Aceasta oferă un mod simplu și sigur de a partaja o resursă sau de a furniza o funcționalitate globală într-un mod eficient.

Beneficiile pattern-ului Singleton includ economisirea resurselor, evitarea creării multiple a instanțelor și asigurarea consistenței și integrității datelor. De asemenea, oferă un punct centralizat pentru gestionarea și accesarea unei resurse comune sau a unei funcționalități globale.

Cu toate acestea, trebuie luate în considerare și anumite aspecte, cum ar fi gestionarea concurenței în cazul utilizării Singleton în medii multi-threaded sau posibilitatea ca Singleton-ul să devină un punct de atac pentru cuplarea strânsă și dependența între componente.

Astfel, Singleton Design Pattern este folosit în situații în care este necesară o singură instanță globală a unei clase și se dorește accesul controlat și centralizat la această instanță în întregul sistem.

**Problemă**

Aplicația are nevoie de o singură instanță a unui obiect. În plus, sunt necesare inițializarea leneșă și accesul global.

**Structura**

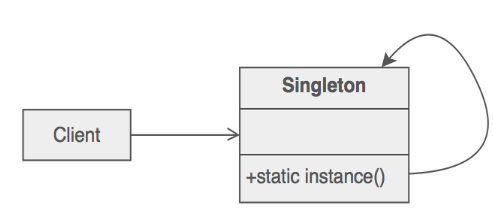


Figura 2.3 – Structura Singleton Design Pattern.

Faceți clasa instanței unice responsabile de acces și „inițializare la prima utilizare”. Instanța unică este un atribut static privat. Funcția de accesare este o metodă publică statică.[8]

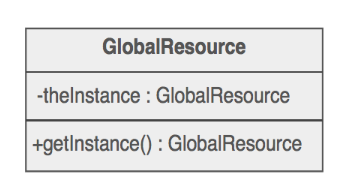


Figura 2.4 – instanța unice responsabile.

## Modele de proiectare structurală

Modelele de proiectare structurală sunt un set de pattern-uri care se concentrează pe organizarea și structurarea obiectelor într-un sistem software. Acestea oferă soluții pentru crearea și gestionarea relațiilor între obiecte, compunerea acestora în structuri mai complexe și asigurarea flexibilității și modularității în proiectare.

Pattern-urile de proiectare structurală abordează aspecte precum compunerea obiectelor, relațiile între ele, modul în care acestea sunt organizate și modul în care interacționează. Ele oferă un ghid și o structură pentru implementarea și organizarea structurilor de date și a relațiilor dintre obiecte într-un mod eficient și reutilizabil.

Prin utilizarea modelelor de proiectare structurală, dezvoltatorii pot obține o arhitectură mai flexibilă, modulară și ușor de întreținut. Acestea promovează separarea responsabilităților, reutilizarea componentelor și extensibilitatea sistemului.

În general, modelele de proiectare structurală sunt utile în dezvoltarea software pentru a rezolva probleme legate de organizarea și structurarea obiectelor într-un mod eficient și coerent. Ele ajută la crearea unui design solid și bine structurat, care să permită o dezvoltare și o întreținere mai ușoară a sistemului.[4]

### Decorator Design Pattern

Decorator Design Pattern este un pattern de design care permite adăugarea de comportamente suplimentare sau modificarea dinamică a funcționalității unei clase, fără a modifica direct codul acesteia. Acesta aparține categoriei pattern-urilor structurale și este utilizat atunci când se dorește extinderea flexibilă a funcționalității unei clase fără a afecta alte obiecte din același tip.

Prin utilizarea Decorator Design Pattern, se obține flexibilitatea de a adăuga sau de a modifica comportamentul unei clase în timpul execuției, fără a afecta alte obiecte din același tip. Aceasta permite combinarea și compunerea diferitelor decoratori pentru a obține combinații complexe de funcționalități.

Beneficiile pattern-ului Decorator includ extensibilitatea și modularitatea, deoarece adăugarea de funcționalități suplimentare se face prin adăugarea de decoratori noi, evitând modificarea directă a clasei de bază. De asemenea, oferă posibilitatea de a combina și de a compune diferite funcționalități într-un mod flexibil și dinamic.

Este important de menționat că decoratorii trebuie să implementeze aceeași interfață ca și componenta de bază, astfel încât aceștia pot fi utilizați în același mod în cod. De asemenea, trebuie avută grijă la ordinea în care sunt adăugați decoratorii, deoarece aceasta poate influența comportamentul final al obiectului decorat.

Astfel, Decorator Design Pattern este folosit în situații în care se dorește extinderea dinamică și flexibilă a funcționalității unei clase, fără a afecta alte obiecte similare și fără a utiliza moștenirea clasică.

**Problemă**

Doriți să adăugați comportament sau stare obiectelor individuale în timpul execuției. Moștenirea nu este fezabilă deoarece este statică și se aplică unei întregi clase.

**Structura**

Clientul este mereu interesat de CoreFunctionality.doThis(). Clientul poate, sau nu, să fie interesat de OptionalOne.doThis() și OptionalTwo.doThis(). Fiecare dintre aceste clase delegă întotdeauna la clasa de bază Decorator, iar acea clasă delegă întotdeauna obiectul „wrappee” conținut.[9]

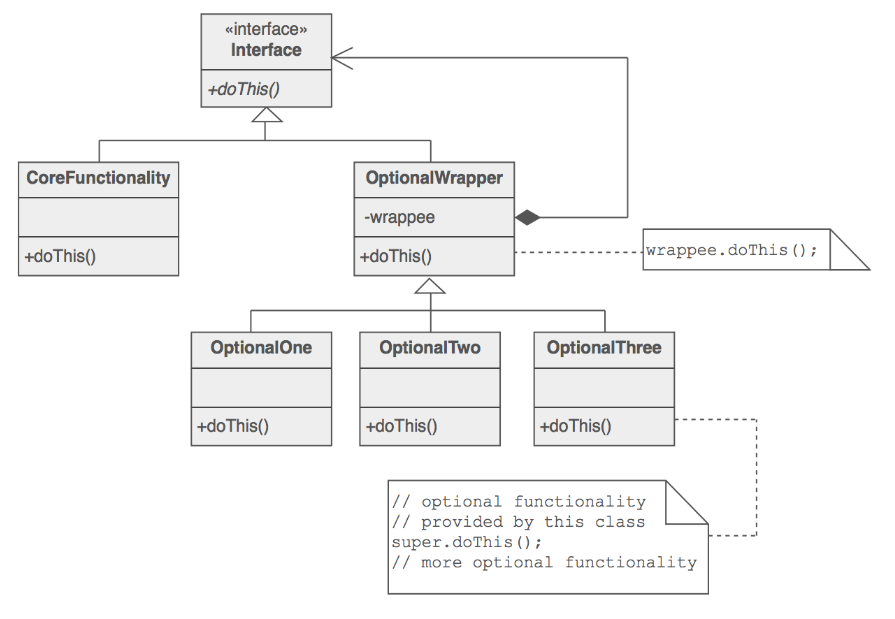


Figura 3.1 – Structura Decorator Design Pattern.

### Proxy Design Pattern

Proxy Design Pattern este un pattern de design care furnizează un obiect de înlocuire sau un substitut pentru un alt obiect, controlând accesul și interacțiunea cu acesta. Acesta aparține categoriei pattern-urilor structurale și este utilizat atunci când se dorește controlul asupra accesului la un obiect sau adăugarea de funcționalități suplimentare înainte, după sau în locul apelurilor către obiectul real.

Prin utilizarea Proxy Design Pattern, se obține un nivel suplimentar de control și gestionare a accesului la un obiect, permițând realizarea de acțiuni suplimentare înainte sau după apelurile către obiectul real. Proxy-ul poate fi utilizat pentru a implementa caching, lazy loading, securitate, gestionarea resurselor sau alte funcționalități adiționale.

Beneficiile pattern-ului Proxy includ abstractizarea și protejarea obiectului real, reducerea accesului direct și minimizarea cuplării între componentele sistemului. De asemenea, oferă posibilitatea de a adăuga funcționalități suplimentare fără a modifica obiectul real și permite controlul asupra accesului și gestionarea eficientă a resurselor.

Este important de menționat că Proxy-ul trebuie să implementeze aceeași interfață ca și subiectul, astfel încât să poată fi utilizat în locul acestuia în codul client. De asemenea, trebuie avută grijă la gestionarea corectă a ciclurilor de apeluri între proxy și subiect și să se asigure că proxy-ul nu adaugă o complexitate excesivă în sistem.

Astfel, Proxy Design Pattern este folosit în situații în care se dorește controlul și gestionarea accesului la un obiect sau adăugarea de funcționalități suplimentare înainte, după sau în locul apelurilor către obiectul real.

**Problemă**

Trebuie să susțineți obiecte care necesită resurse și nu doriți să instanțiați astfel de obiecte decât dacă și până când sunt solicitate efectiv de client.

**Structura**

Prin definirea unei interfețe Subject, prezența obiectului Proxy în locul RealSubject este transparentă pentru client.[10]

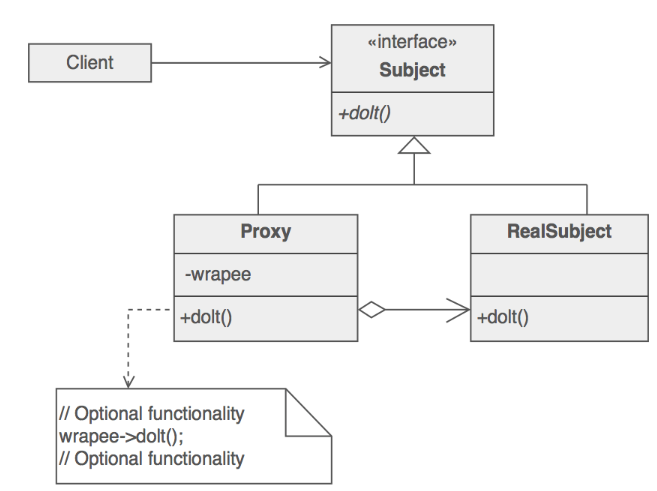


Figura 3.2 – Structura Proxy Design Pattern.

## Behavioral design patterns

Pattern-urile de design comportamentale sunt un set de pattern-uri care se concentrează pe comportamentul și interacțiunea între obiecte și entități într-un sistem software. Acestea oferă soluții pentru gestionarea fluxului de execuție, comunicarea și gestionarea responsabilităților între obiecte într-un mod flexibil și extensibil.

Pattern-urile de design comportamentale abordează probleme legate de modul în care obiectele cooperează, comunică și se sincronizează într-un sistem. Ele oferă un ghid și o structură pentru implementarea și organizarea comportamentului într-un mod modular și reutilizabil.

Prin utilizarea pattern-urilor de design comportamentale, dezvoltatorii pot obține un cod mai flexibil, modular și ușor de întreținut. Acestea promovează separarea preocupărilor și permit schimbarea și extinderea comportamentului fără a afecta structura generală a sistemului.

În general, pattern-urile de design comportamentale sunt utile în dezvoltarea software pentru a rezolva probleme legate de comportamentul și interacțiunea obiectelor într-un mod elegant, eficient și reutilizabil.[5]

### Chain of Responsibility

Chain of Responsibility este un pattern de design care permite transmiterea secvențială a unei cereri printr-o serie de obiecte sau noduri care pot trata cererea sau o pot pasa mai departe în lanț. Acesta aparține categoriei pattern-urilor comportamentale și este utilizat atunci când se dorește decuplarea expeditorului și a destinatarului unei cereri și posibilitatea de a procesa cererea într-un mod flexibil și extensibil.

Definiția sa implică utilizarea unei structuri de tip lanț, formată din obiecte numite "handleri". Fiecare handler are abilitatea de a trata cererea sau de a o pasa mai departe către următorul handler din lanț. Clientul trimite cererea către primul handler din lanț, care decide dacă poate trata cererea sau o va pasa către următorul handler din lanț. Aceasta continuă până când cererea este tratată sau până când lanțul este epuizat.

Beneficiile pattern-ului Chain of Responsibility includ flexibilitatea, extensibilitatea și ușurința în adăugarea și modificarea comportamentului de tratare a cererilor. Acesta permite definirea unui lanț complex de handleri și gestionarea dinamică a tratării cererilor în funcție de necesități. De asemenea, facilitează reducerea cuplării și separarea responsabilităților între handleri.

Este important de menționat că trebuie avută grijă la configurarea corectă a lanțului și la evitarea ciclurilor infinite în tratarea cererilor. De asemenea, handlerii trebuie să fie bine definiciți și să aibă claritate asupra responsabilităților lor pentru a asigura un flux corespunzător al cererilor prin lanț.

Astfel, Chain of Responsibility este folosit în situații în care se dorește decuplarea expeditorului și a destinatarului unei cereri și tratarea acesteia într-un mod flexibil și extensibil prin intermediul unui lanț de handleri.

**Problemă**

Există un număr potențial variabil de obiecte „handler” sau „element de procesare” sau „nod” și un flux de cereri care trebuie tratate. Necesitatea de a procesa în mod eficient cererile, fără relații și prioritate de gestionare a cablurilor, sau mapări cerere-la-handler.

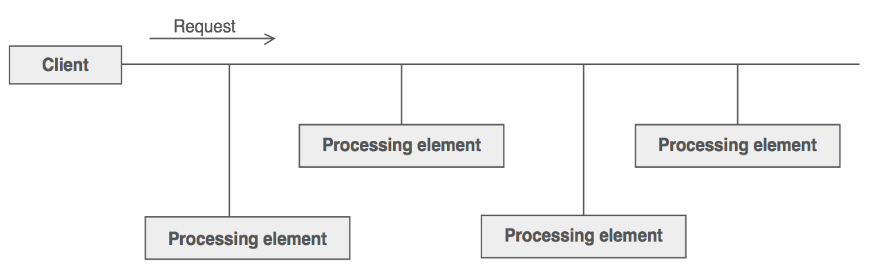


Figura 4.1 – handler.

**Structura**

Clasele derivate știu cum să satisfacă cererile Clientului. Dacă obiectul „actual” nu este disponibil sau suficient, atunci el delegă la clasa de bază, care delegă la obiectul „următorul”, iar cercul vieții continuă.

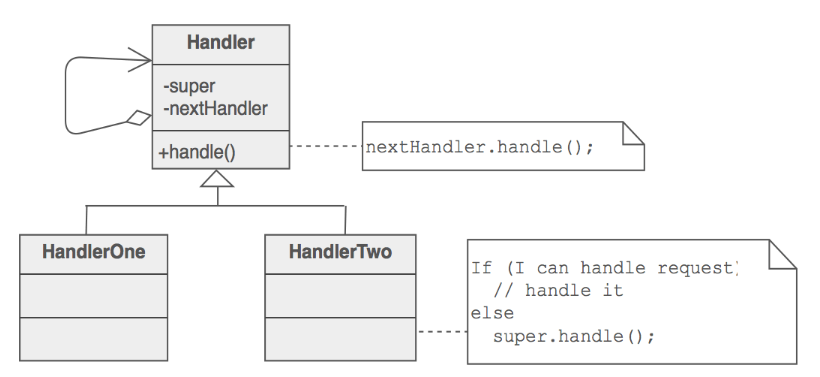


Figura 4.2 – Structura Chain of Responsibility.

Mai mulți operatori ar putea contribui la tratarea fiecărei cereri. Solicitarea poate fi transmisă pe toată lungimea lanțului, ultima verigă având grijă să nu delege unui „null next”.[11]

### Command Design Pattern

Command Design Pattern este un pattern de design comportamental care încapsulează o solicitare sub forma unui obiect, permițând astfel parametrizarea clienților cu diferite solicitări, coada sau înregistrarea solicitărilor și suportul pentru operații anulate. Acesta aparține categoriei pattern-urilor comportamentale și este utilizat pentru a separa solicitarea de executantul acesteia, permițând astfel flexibilitate și extensibilitate în gestionarea și tratarea solicitărilor.

Definiția sa implică utilizarea a patru componente principale: comanda, invocatorul, executantul și receptorul. Comanda este reprezentată sub forma unui obiect și conține informații despre acțiunea solicitată și parametrii necesari. Invocatorul este responsabil pentru invocarea comenzii și poate menține o coadă sau o listă de comenzi pentru a le executa ulterior sau pentru a le anula. Executantul este cel care efectuează acțiunea specifică solicitată de comandă, iar receptorul este obiectul asupra căruia se efectuează acțiunea.

Beneficiile pattern-ului Command includ decuplarea între expeditorul comenzii și receptorul acțiunii, flexibilitatea și extensibilitatea în gestionarea solicitărilor, posibilitatea de înregistrare și anulare a comenzilor, și reutilizarea și parametrizarea ușoară a acțiunilor.

Este important de menționat că Command Design Pattern poate fi implementat în mai multe moduri, cum ar fi utilizarea interfețelor, funcțiilor pointer sau expresiilor lambda, în funcție de limbajul de programare și nevoile specifice ale aplicației.

Astfel, Command Design Pattern este utilizat în situații în care se dorește separarea clară între expeditorul comenzii și receptorul acțiunii, gestionarea și tratarea flexibilă a solicitărilor și suportul pentru înregistrarea și anularea comenzilor.

**Problemă**

Necesitatea de a emite cereri către obiecte fără a ști nimic despre operațiunea solicitată sau despre destinatarul cererii.

**Structura**

Clientul care creează o comandă nu este același client care o execută. Această separare oferă flexibilitate în sincronizarea și succesiunea comenzilor. Materializarea comenzilor ca obiecte înseamnă că acestea pot fi transmise, puse în scenă, partajate, încărcate într-un tabel și instrumentate sau manipulate în alt mod ca orice alt obiect.

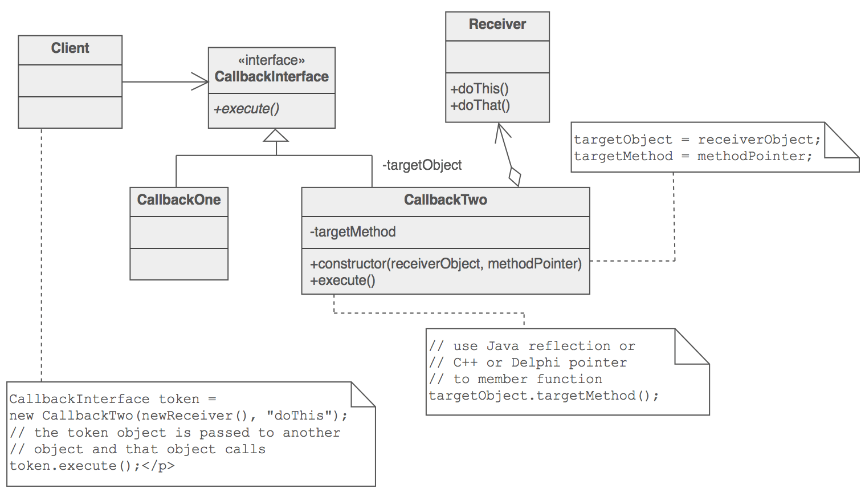


Figura 4.3 – Structura Command Design Pattern.

Obiectele de comandă pot fi considerate „token-uri” care sunt create de un client care știe ce trebuie făcut și transmise unui alt client care are resursele pentru a face acest lucru.[12]

### Observer Design Pattern

Observer Design Pattern este un pattern de design comportamental care permite notificarea automată a mai multor obiecte (numite observatori) atunci când starea unui obiect (numit subiect) se schimbă. Acesta aparține categoriei pattern-urilor comportamentale și este utilizat pentru a realiza o comunicare slab cuplată între obiecte într-un sistem software.

Definiția sa implică două componente principale: subiectul și observatorii. Subiectul este obiectul ale cărui modificări de stare trebuie să fie observate, în timp ce observatorii sunt obiectele care sunt notificate atunci când starea subiectului se schimbă. Atunci când subiectul se modifică, observatorii săi sunt notificați automat și pot reacționa în consecință.

Beneficiile pattern-ului Observer includ decuplarea între subiect și observatori, posibilitatea de a notifica automat mai mulți observatori la modificarea stării subiectului, flexibilitatea în adăugarea și eliminarea observatorilor și posibilitatea de reutilizare și extensibilitate a subiectului și a observatorilor.

Este important de menționat că Observer Design Pattern poate fi implementat în diferite moduri, cum ar fi utilizarea interfețelor sau clase abstracte, în funcție de limbajul de programare și nevoile specifice ale aplicației.

Astfel, Observer Design Pattern este utilizat în situații în care se dorește comunicarea slab cuplată între obiecte și notificarea automată a observatorilor atunci când starea unui obiect se schimbă.

**Problemă**

Un design monolitic mare nu se scalează bine, deoarece sunt impuse cerințe noi de graficare sau monitorizare.

**Structura**

Subiectul reprezintă abstracția de bază (sau independentă sau comună sau motor). Observer reprezintă abstractizarea variabilei (sau dependentă sau opțională sau interfață utilizator). Subiectul solicită obiectelor Observator să facă treaba lor. Fiecare observator poate apela înapoi la Subiect după cum este necesar.[13]

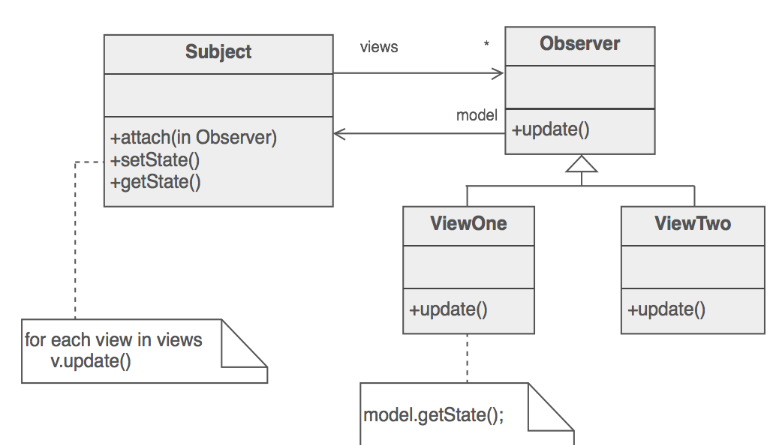


Figura 4.4 – Structura Observer Design Pattern.

# Prezentarea Aplicației

În dezvoltarea unei aplicații de consolă pentru gestionarea produselor, cum ar fi cărți, filme și albume muzicale, am folosit o abordare orientată pe obiecte (OOP) și am aplicat diferite pattern-uri de proiectare pentru a îmbunătăți structura și flexibilitatea aplicației noastre. Utilizând limbajul Python, am implementat următoarele pattern-uri:

* Modelele de design creațional:
* Metoda Factory;
* Builder ;
* Singleton.
* Modelele de proiectare structurală:
* Decorator;
* Proxy.
* Behavioral design patterns:
* Command;
* Chain of Responsibility;
* Observator.

Prin utilizarea acestor pattern-uri de proiectare în dezvoltarea aplicației noastre de consolă pentru produse, am reușit să obținem o structură mai modulară, o separare clară a responsabilităților și o flexibilitate crescută în gestionarea diferitelor tipuri de produse.

## Modelele de design creațional

Implementarea modele de design creațional. Aceste modele sunt o categorie de modele de design software care se concentrează pe procesul de creare și inițializare a obiectelor. Ele oferă mecanisme pentru crearea obiectelor într-un mod flexibil și modular, ascunzând detaliile specifice ale creării obiectelor și permițând o flexibilitate sporită în procesul de creare.

Aceste modele de design creațional oferă avantaje precum encapsularea procesului de creare a obiectelor, separarea construcției de reprezentarea obiectelor și creșterea flexibilității și extensibilității în procesul de creare a obiectelor.

### Metoda Factory

Acest cod reprezintă o implementare simplă a unui sistem de gestionare a resurselor, cum ar fi cărți, filme și muzică. Utilizează concepte de programare orientată pe obiecte, design patterns și observabilitate.

Principalele clase din cod sunt următoarele:

Clasa abstractă Resursa: Este o clasă abstractă care definește interfața comună pentru toate resursele (cărți, filme, muzică). Are metode abstracte și metode comune implementate, cum ar fi afisare.

Clasa Carte, Film, Muzica: Aceste clase reprezintă resursele specifice (carte, film, muzică) și extind clasa abstractă Resursa. Acestea au atribute specifice, cum ar fi titlu, autor, an de apariție, precum și metode pentru a seta prețul, procentajul de reducere și codul unic al resursei. De asemenea, implementează metodele abstracte definite în clasa de bază pentru a afișa informații specific

Clasa ResursaFactory: Este o clasă de fabrică utilizată pentru a crea noi instanțe de resurse, pe baza tipului specificat.

Această implementare permite gestionarea resurselor.

Anexa 1: Exemplu de sintaxă Metoda Factory

1. **class** Resursa(ABC):
2. @abstractmethod
3. **def** afisare(self):
4. **pass**
6. **class** Carte:
7. **def** \_\_init\_\_(self, titlu, autor, editura):
8. self.tip = "carte"
9. self.titlu = titlu
10. **self.autor = autor**
11. self.editura = editura
12. **def** afisare(self):
13. **print**(f"Tip resursa: {self.tip}")
14. **print**(f"Aceasta este titlul cartii: '{self.titlu}'.")
15. **print(f"Scrisă de: '{self.autor}'.")**
16. **print**(f"Editată de: '{self.editura}'. ")

cum putem să observăm aveam reprezentat diagrama UML:

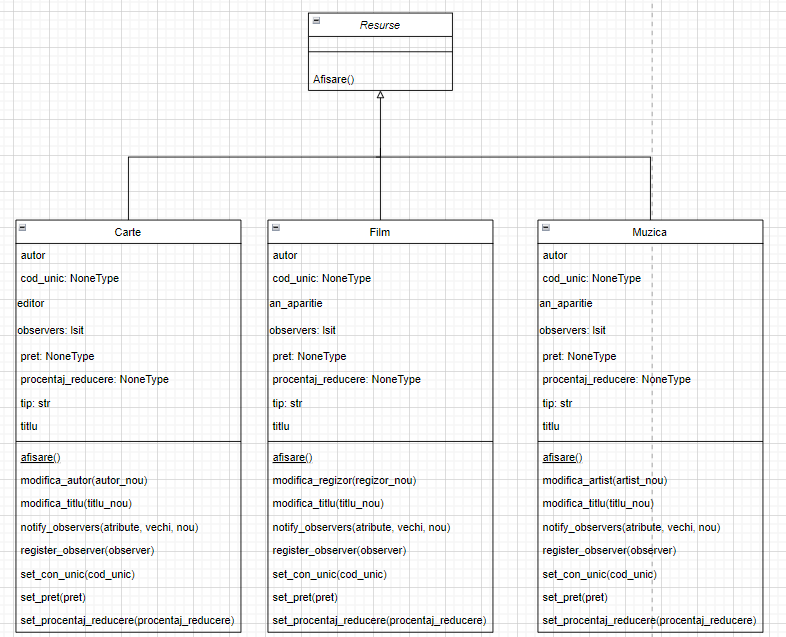


Figura 1.1 – Schema UML pentru Metoda Factory.

Clasa ResursaFactory utilizează clasele Carte, Film și Muzica pentru a crea resurse. Aceasta este o relație de asociere (association), reprezentată printr-o linie solidă între clasele ResursaFactory și Carte, Film, Muzica.

### Builder

Clasa "Autor" reprezintă un autor și are câteva atribute, precum ID-ul autorului, numele, prenumele, naționalitatea și anul nașterii. Clasa are și câteva metode, cum ar fi "str" pentru a returna o reprezentare sub formă de șir de caractere a autorului, "get\_full\_name" pentru a obține numele complet al autorului și "modifica\_autor" pentru a modifica autorul cu unul nou.

Clasa "AutorBuilder" este folosită pentru a construi obiecte de tip Autor. Aceasta implementează metode pentru setarea diferitelor atribute ale autorului, cum ar fi numele, prenumele, naționalitatea și anul nașterii. Metoda "build" este responsabilă pentru construirea obiectului Autor pe baza atributelor setate și pentru a insera autorul într-o bază de date folosind un obiect de conexiune la baza de date.

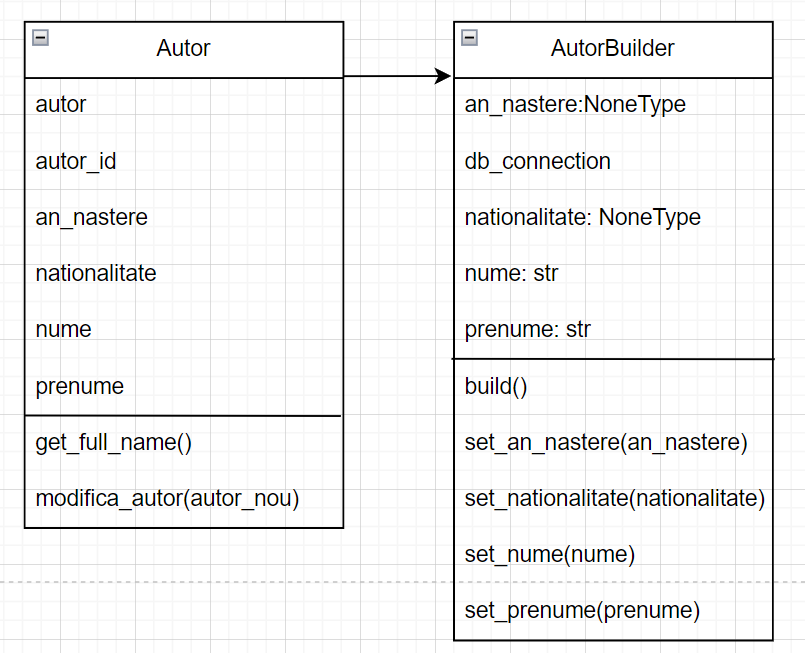


Figura 1.2 – Schema UML pentru Builder.

Clasa "AutorBuilder" este responsabilă de construirea obiectelor de tip "Autor". Ea are o relație de asociere cu clasa "Autor", deoarece utilizează obiecte de tip "Autor" pentru a le construi.

În implementarea dată, clasa "AutorBuilder" conține o metodă "build()" care construiește și returnează un obiect de tip "Autor". Metoda "build()" utilizează valorile stocate în obiectul "AutorBuilder" pentru a crea un obiect "Autor" prin intermediul constructorului său.

Astfel, clasa "AutorBuilder" utilizează clasa "Autor" pentru a crea și returna obiecte de tip "Autor", iar clasa "Autor" este utilizată pentru a stoca și manipula informațiile despre un autor specific.

### Singleton

Pentru gestionarea conexiunii și crearea unei baze de date SQLite. Clasa "DatabaseConnection" se ocupă de conexiunea la baza de date și crearea tabelei "autor", iar clasa "DatabaseConnectionFactory" implementează un Singleton pentru obținerea unei singure instanțe a conexiunii la baza de date.

Clasa "DatabaseConnection" are metode pentru a crea și gestiona conexiunea la baza de date SQLite. Ea conține metode pentru conectarea la baza de date, crearea tabelei "autor" dacă nu există și închiderea conexiunii.

Clasa "DatabaseConnectionFactory" implementează un Singleton pentru a asigura că există o singură instanță a conexiunii la baza de date. Ea oferă o metodă statică "get\_instance" care primește numele bazei de date și returnează instanța conexiunii la baza de date. Dacă nu există o instanță, se creează una nouă și se efectuează conexiunea la baza de date.

În ultima parte a codului, se obține o instanță a conexiunii la baza de date folosind Singleton-ul implementat în clasa "DatabaseConnectionFactory". Se specifică numele bazei de date "mydatabases.db".

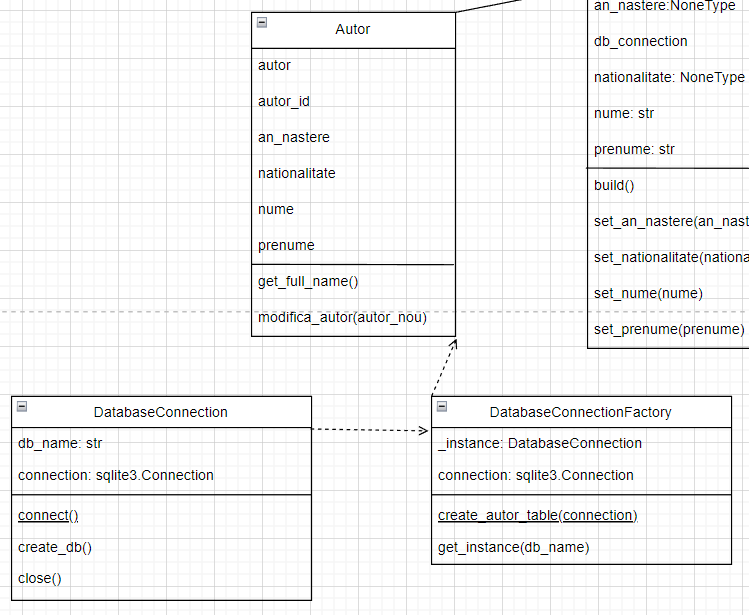


Figura 1.3 – Schema UML pentru Singleton.

Clasa DatabaseConnection reprezintă o conexiune la baza de date și este asociată cu clasa DatabaseConnectionFactory printr-o relație de asociere simplă, deoarece DatabaseConnectionFactory folosește o instanță de DatabaseConnection pentru a crea conexiunea la baza de date.

Clasa DatabaseConnectionFactory este asociată cu clasa Autor printr-o relație de asociere simplă, deoarece DatabaseConnectionFactory utilizează instanțe ale clasei Autor pentru a crea tabela "autor" în baza de date.

Clasa Autor reprezintă un autor din baza de date și este asociată cu clasa DatabaseConnectionFactory printr-o relație de asociere simplă, deoarece obiectele Autor sunt create și utilizate în cadrul DatabaseConnectionFactory pentru a interacționa cu tabela "autor".

## Modelele de proiectare structurală

Modelele de proiectare structurală sunt un set de pattern-uri care se concentrează pe organizarea și relațiile dintre obiecte pentru a obține o structură flexibilă și ușor de înțeles. Aceste modele sunt utilizate în principal pentru a obține o separare clară între diferitele componente ale unui sistem și pentru a permite extinderea și modificarea ulterioară a acestuia fără a afecta întreaga structură.

Unul dintre modelele de proiectare structurală menționat în descrierea dată este Decoratorul (Decorator pattern). Acesta permite adăugarea de comportamente suplimentare la obiectele existente prin încapsularea acestora în obiecte decorator. Aceste obiecte decorator extind funcționalitatea obiectului de bază prin adăugarea de funcționalități suplimentare într-un mod flexibil și modular. În cazul dat, clasele ResursaDecorator, ReducereDecorator și CodUnicDecorator acționează ca decoratori pentru clasele de resurse existente (Carte, Film, Muzica), adăugând funcționalități precum afișarea prețului, aplicarea reducerilor sau generarea unui cod unic.

Un alt model de proiectare menționat este Proxy (Proxy pattern). Acesta acționează ca un intermediar între client și obiectul real, controlând accesul la acesta și furnizând o interfață simplificată. Clasa ProxyResursa acționează ca un proxy pentru resurse, verificând utilizatorii autorizați și oferind opțiuni pentru crearea, afișarea și modificarea resurselor. Acest model permite gestionarea și controlul accesului la resurse și poate fi utilizat, de exemplu, pentru autentificare sau validare.

### Decorator

Codul dat definește trei clase decorator: ResursaDecorator, ReducereDecorator și CodUnicDecorator. Aceste clase sunt utilizate pentru a extinde comportamentul unei clase de bază Resursa prin adăugarea de funcționalități suplimentare.

ResursaDecorator primește o instanță a clasei de bază resursa și un pret și adaugă funcționalitatea de afișare a resursei, urmată de afișarea prețului.

ReducereDecorator primește o instanță a clasei de bază resursa și un procentaj\_reducere și adaugă funcționalitatea de afișare a resursei, urmată de afișarea prețului cu reducerea aplicată.

CodUnicDecorator primește o instanță a clasei de bază resursa și adaugă funcționalitatea de afișare a resursei, urmată de afișarea unui cod unic generat pentru produsul respectiv.

Fiecare clasă decorator implementează metoda afisare(), care apelează metoda afisare() a resursei de bază și adaugă comportamentul suplimentar specific decoratorului respectiv.

Anexa 2: Exemplu de sintaxă pattern-ul Decorator

1. **class** ResursaDecorator:
2. **def** \_\_init\_\_(self, resursa, pret):
3. self.resursa = resursa
4. self.pret = pret
6. **def** afisare(self):
7. self.resursa.afisare()
8. *# print(f"Pret: {self.pret} USD")*

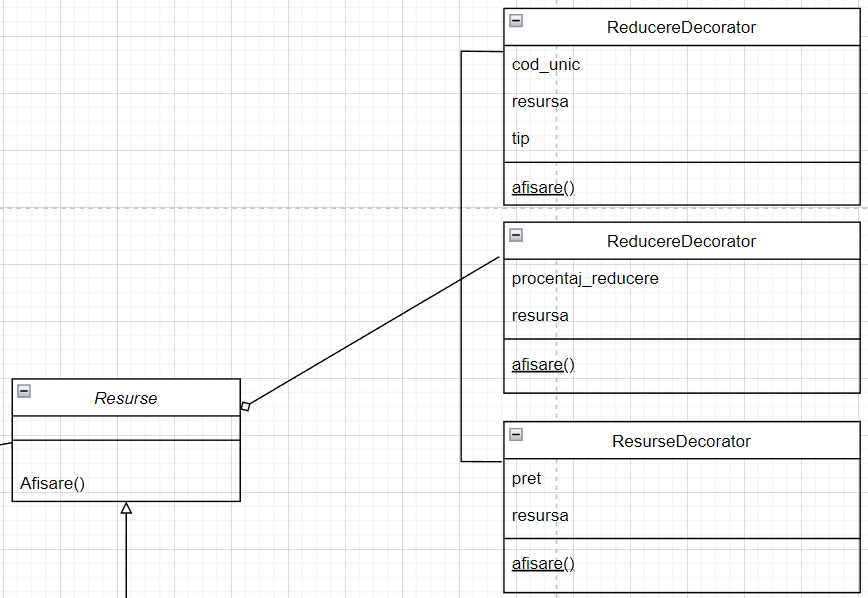


Figura 2.1 – Schema UML pentru Decorator.

Clasele ResursaDecorator, ReducereDecorator și CodUnicDecorator sunt decoratoare pentru clasele de resurse existente (Carte, Film, Muzica). Acestea sunt relații de agregare (aggregation), reprezentate printr-o săgeată cu un diamant goală pe capătul claselor de decoratoare și o săgeată solidă pe capătul claselor de resurse existente (Carte, Film, Muzica).

### Proxy

Clasa ProxyResursa este o clasă proxy care acționează ca un intermediar între utilizatori și resurse. Aceasta verifică utilizatorii autorizați și oferă opțiuni pentru crearea, afișarea și modificarea resurselor. Aceasta utilizează obiecte de tip observator pentru a înregistra și notifica evenimente legate de resurse.

Codul conține și o ierarhie de clase de comenzi (ModificaTitluCarteCommand, ModificaRegizorFilmCommand, etc.) care sunt folosite pentru a modifica diverse aspecte ale resurselor.

În funcția ProxyResursa.afisare(), utilizatorul este întâmpinat cu un meniu interactiv care permite crearea, afișarea și modificarea resurselor. Utilizatorii autorizați pot efectua aceste acțiuni în funcție de opțiunile disponibile.

În esență, acest cod oferă o interfață simplă pentru gestionarea resurselor prin intermediul unui proxy și utilizează design patterns pentru a extinde funcționalitatea resurselor și a implementa logica de validare și notificare.

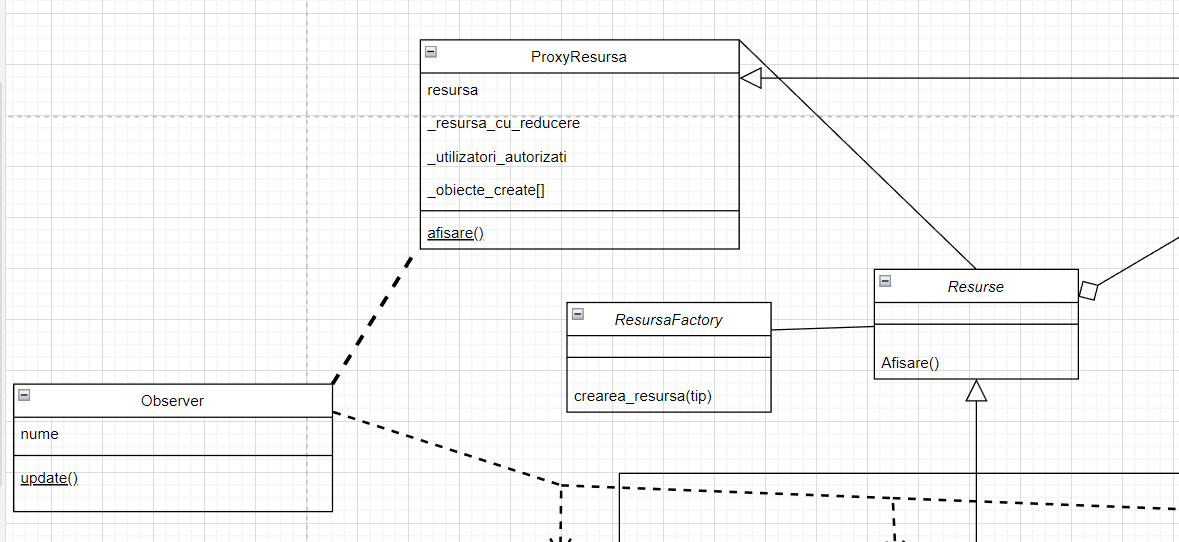


Figura 2.1 – Schema UML pentru Proxy.

* Asociație:

Relația de asociație este reprezentată prin o linie solidă între clasele implicate.

În acest caz, puteți utiliza o relație de asociație între clasa ProxyResursa și clasa Resursa.

* Moștenire:

Relația de moștenire este reprezentată prin o linie săgeată de la clasa derivată (clasa ProxyResursa) la clasa de bază (clasa CodUnicDecorator).

Această relație indică faptul că clasa ProxyResursa moștenește comportamentul și atributele clasei (CodUnicDecorator).

* Implementare:

Relația de implementare este reprezentată prin o linie întreruptă cu o săgeată de la clasa care implementează interfața (clasa ProxyResursa) la interfață (Observer).

Această relație indică faptul că clasa ProxyResursa implementează interfața Observer.

## Behavioral design patterns

Comportamentale design patterns sunt esențiale în dezvoltarea software pentru a organiza și gestiona comportamentul obiectelor într-un mod flexibil și modular. Aceste tipare ne permit să separăm responsabilitățile, să îndeplinim cerințele de interacțiune și să extindem funcționalitatea într-un mod coerent și ușor de întreținut.

Pattern-ul Command oferă o modalitate elegantă de a încapsula solicitările în obiecte și de a le executa într-un mod decuplat. Prin intermediul claselor ModificaTitluCarteCommand și ModificaAutorCarteCommand, putem modifica titlul și autorul unei cărți într-un mod controlat și uniform. Aceste clase permit gestionarea simplă a operațiilor de modificare și asigură un nivel înalt de flexibilitate în sistem.

Pattern-ul Chain of Responsibility facilitează validarea structurată și modulară a diferitelor aspecte ale cărților, filmelor și albumelor muzicale. Prin intermediul claselor de validare precum ValidareTitlu, ValidareAutor sau ValidareRegizor, putem verifica fiecare aspect într-o ordine specifică și putem extinde ușor funcționalitatea prin adăugarea de noi clase de validare. Astfel, se obține un flux coerent și modular pentru validarea obiectelor.

Pattern-ul Observer permite implementarea unui sistem de notificare și actualizare eficient pentru resursele observate. Prin intermediul clasei Observer și a metodei sale update, putem adăuga observatori și să le notificăm în momentul în care o modificare are loc în resursă. Acest pattern ne oferă flexibilitatea de a extinde și personaliza observatorii prin crearea de subclase și suprascrierea metodei update.

### Command

Avem reprezentat doua clase ModificaTitluCarteCommand și ModificaAutorCarteCommand, care sunt folosite pentru a modifica titlul și respectiv autorul unui obiect Carte (carte).

Aceste clase de comandă încapsulează operațiile de modificare și oferă o modalitate uniformă de a le executa. Prin crearea de instanțe ale acestor clase și apelarea metodelor lor execute, puteți modifica titlul sau autorul unui obiect Carte într-un mod controlat și decuplat.

Anexa 3: Exemplu de sintaxă pattern-ul Command

1. **class** ModificaTitluCarteCommand:
2. **def** \_\_init\_\_(self, carte, titlu\_nou):
3. self.carte = carte
4. self.titlu\_nou = titlu\_nou
6. **def** execute(self):
7. self.carte.modifica\_titlu(self.titlu\_nou)

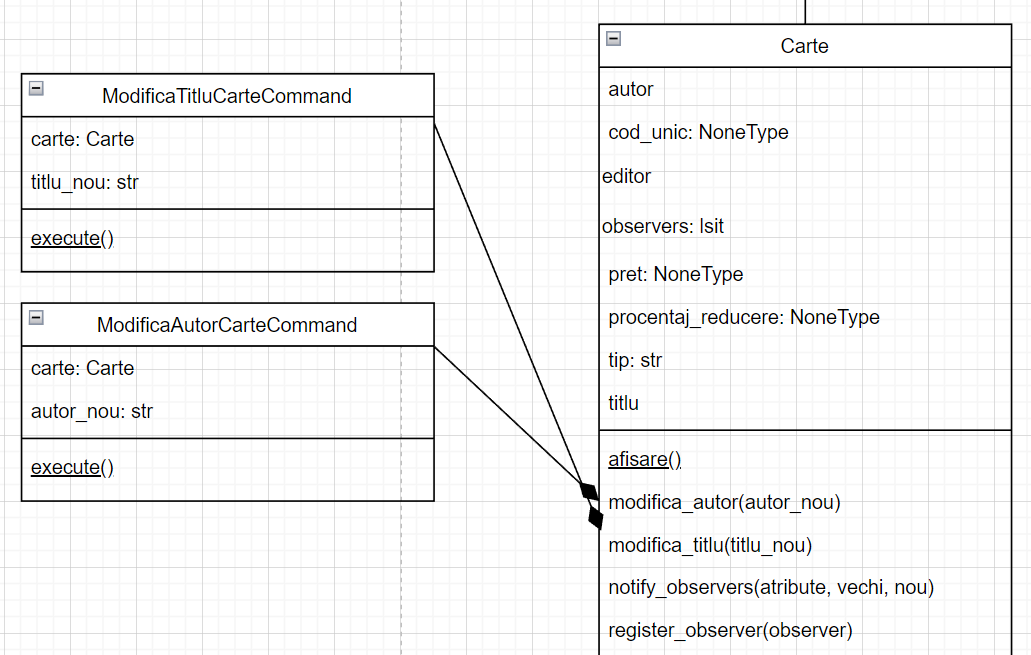


Figura 2.1 – Schema UML pentru Decorator.

În diagramă, ModificaTitluCarteCommand și ModificaAutorCarteCommand sunt asociate cu clasa Carte. Au o relație de compoziție cu clasa Carte, deoarece dețin referințe la obiecte Carte (atribut carte). Asocierea este reprezentată de o linie continuă cu un diamant umplut la capătul Carte.

Atât ModificaTitluCarteCommand, cât și ModificaAutorCarteCommand au o metodă execute() care permite executarea operației de modificare corespunzătoare asupra obiectului Carte asociat. La fel este și cu Film iș cu Muzica.

### Chain of Responsibility

Clasele prezentate definesc o structură de tip lanț de responsabilitate pentru validarea diferitelor aspecte ale cărților, filmelor și albumelor muzicale.

Clasele ValidareTitlu, ValidareAutor, ValidareRegizor, ValidareArtist, ValidareEditura și ValidareAn sunt responsabile de validarea specifică a titlului, autorului, regizorului, artistului, editurii și anului de apariție. Aceste clase implementează o metodă valideaza care verifică dacă elementul respectiv îndeplinește criteriile de validare specifice și returnează True sau False în funcție de rezultatul validării. Fiecare clasă are, de asemenea, o referință către succesorul său în lanțul de responsabilitate pentru a permite continuarea validării în cazul în care elementul respectiv nu a trecut validarea curentă.

Clasa ValidareChain este clasa principală care conține instanțe ale claselor de validare și controlează fluxul de validare. Aceasta are metode separate pentru validarea cărților, filmelor și albumelor muzicale. În aceste metode, se apelează în lanț metodele de validare pentru fiecare aspect (titlu, autor, editură, an, regizor/artist) și se verifică dacă toate aspectele sunt valide.

În funcție de rezultatul validării, se afișează mesaje corespunzătoare pentru a indica dacă obiectul respectiv (carte, film, album) este valid sau nu.

Prin utilizarea unei astfel de structuri de lanț de responsabilitate, puteți adăuga ușor noi clase de validare și puteți configura ordinea validărilor specifice în lanț, asigurând un flux coerent și modular pentru validarea diferitelor obiecte.

Anexa 4: Exemplu de sintaxă pattern-ul Chain of Responsibility

1. **class** ValidareAutor:
2. **def** \_\_init\_\_(self, succesor=None):
3. self.succesor = succesor
5. **def valideaza(self, autor):**
6. *# Implementați logica de validare pentru autor (nume și prenume)*
7. *# Returnați True dacă autorul este valid sau apelați succesorul pentru a continua validarea*
8. **if** len(autor.nume) >= 5 **and** len(autor.prenume) >= 5:
9. **if** self.succesor **is** **not** None:
10. **return self.succesor.valideaza(autor)**
11. **else**:
12. **return** True
13. **else**:
14. **return** False
15. **class ValidareChain:**
16. **def** \_\_init\_\_(self):
17. self.validare\_titlu = ValidareTitlu()
18. self.validare\_autor = ValidareAutor()
19. self.validare\_editura = ValidareEditura()
20. **self.validare\_an = ValidareAn()**
21. self.validare\_regizor = ValidareRegizor()
23. **def** valideaza\_carte(self, carte):
24. *# Începeți validarea în lanț pentru carte*
25. **if self.validare\_titlu.valideaza(carte.titlu) and \**
26. self.validare\_autor.valideaza(carte.autor) **and** \
27. self.validare\_editura.valideaza(carte.editura):
28. **print**("Cartea este validă.")
29. **else**:
30. **print("Cartea nu este validă.")**

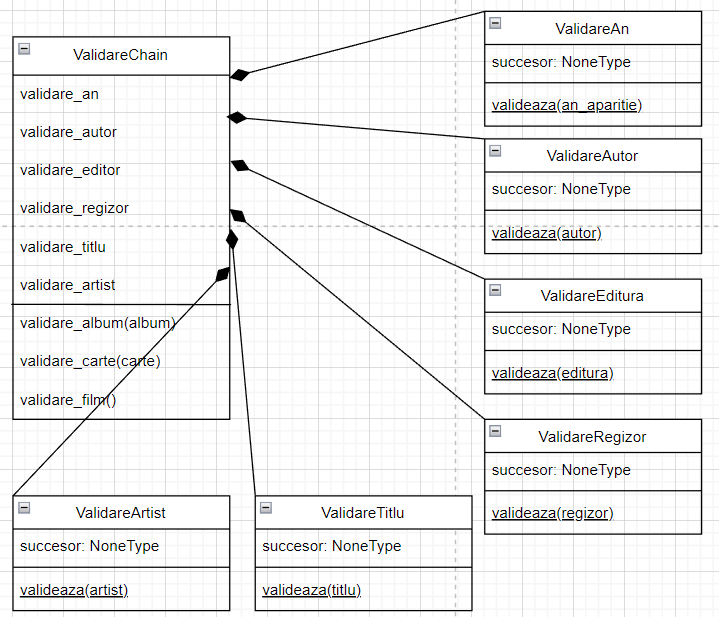


Figura 2.2 – Schema UML pentru Chain of Responsibility.

Clasa ValidareChain reprezintă clasa principală care controlează fluxul de validare. Aceasta conține instanțe ale claselor de validare și are metode separate pentru validarea cărților, filmelor și albumelor muzicale. În aceste metode, se apelează în lanț metodele de validare pentru fiecare aspect (titlu, autor, editură, an, regizor/artist) și se verifică dacă toate aspectele sunt valide.

Clasa creeaza\_carte este o metodă care folosește design pattern-ul de fabrică pentru a crea o carte. În interiorul acestei metode, se utilizează ResursaFactory pentru a crea instanța unei cărți, se obțin informațiile despre preț și procentajul de reducere de la utilizator și se aplică diverse decorații și setări pentru carte. La final, se afișează cartea și se validează folosind ValidareChain.

### Observator

Clasa Observer definește o metodă update care este apelată atunci când o modificare are loc într-o resursă observată. Această metodă primește patru argumente: attribute (atributul care a fost modificat), resource (resursa care a fost modificată), vechi (valoarea veche a atributului) și nou (valoarea nouă a atributului).

Clasa Observer are un atribut nume care poate fi folosit pentru a specifica numele observatorului.

Metoda update afișează un mesaj care indică că observatorul a primit o notificare cu privire la modificarea unui atribut al unei resurse. Mesajul conține numele observatorului, numele atributului, tipul resursei, valoarea veche și valoarea nouă ale atributului.

Observatorul (Observer) poate fi extins și personalizat prin crearea de subclase care moștenesc clasa Observer. Aceste subclase pot suprascrie metoda update pentru a implementa comportamente specifice atunci când sunt notificate despre modificări în resursele observate.

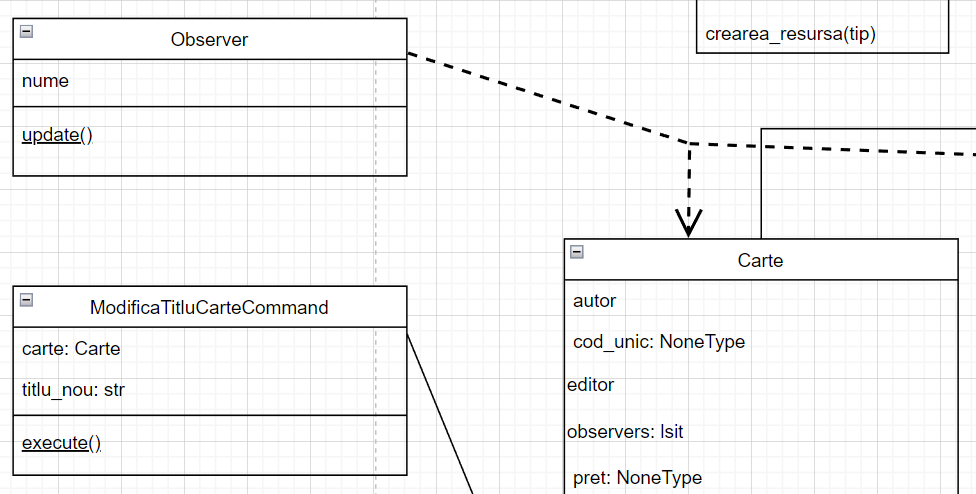


Figura 2.3 – Schema UML pentru Observator.

Clasa Observer are un singur atribut privat nume și o metodă publică update() care trebuie implementată în clasele concrete care extind Observer. Metoda update() este apelată de către obiectul Carte atunci când are loc o modificare și primește informații despre atributul modificat și noile valori.

Clasa Carte reprezintă obiectul observabil. Are mai multe atribute, inclusiv titlu, autor, editura, pret, procentaj\_reducere, cod\_unic și o listă de observatori observers. Metodele sale includ setarea prețului, procentului de reducere și codului unic, afișarea detaliilor cărții, modificarea titlului și autorului, notificarea observatorilor și înregistrarea unui observator nou.

Relația dintre Observer și Carte este o relație de dependență. Carte depinde de Observer pentru a notifica și actualiza observatorii săi, iar Observer depinde de Carte pentru a primi actualizările.

La fel este și cu Film și cu Album.

# Diagrama UML Generală:

O diagramă UML (Unified Modeling Language) este o formă standardizată de reprezentare grafică a elementelor și relațiilor dintr-un sistem software sau un proces de dezvoltare software. UML oferă un set de simboluri și convenții pentru a descrie structura, comportamentul și interacțiunea componentelor unui sistem.

Diagrama UML ajută la comunicarea și înțelegerea sistemelor complexe prin utilizarea unui limbaj grafic. Aceasta oferă o vedere abstractă asupra sistemului, evidențiind entitățile implicate, relațiile dintre acestea și comportamentul lor în cadrul sistemului.

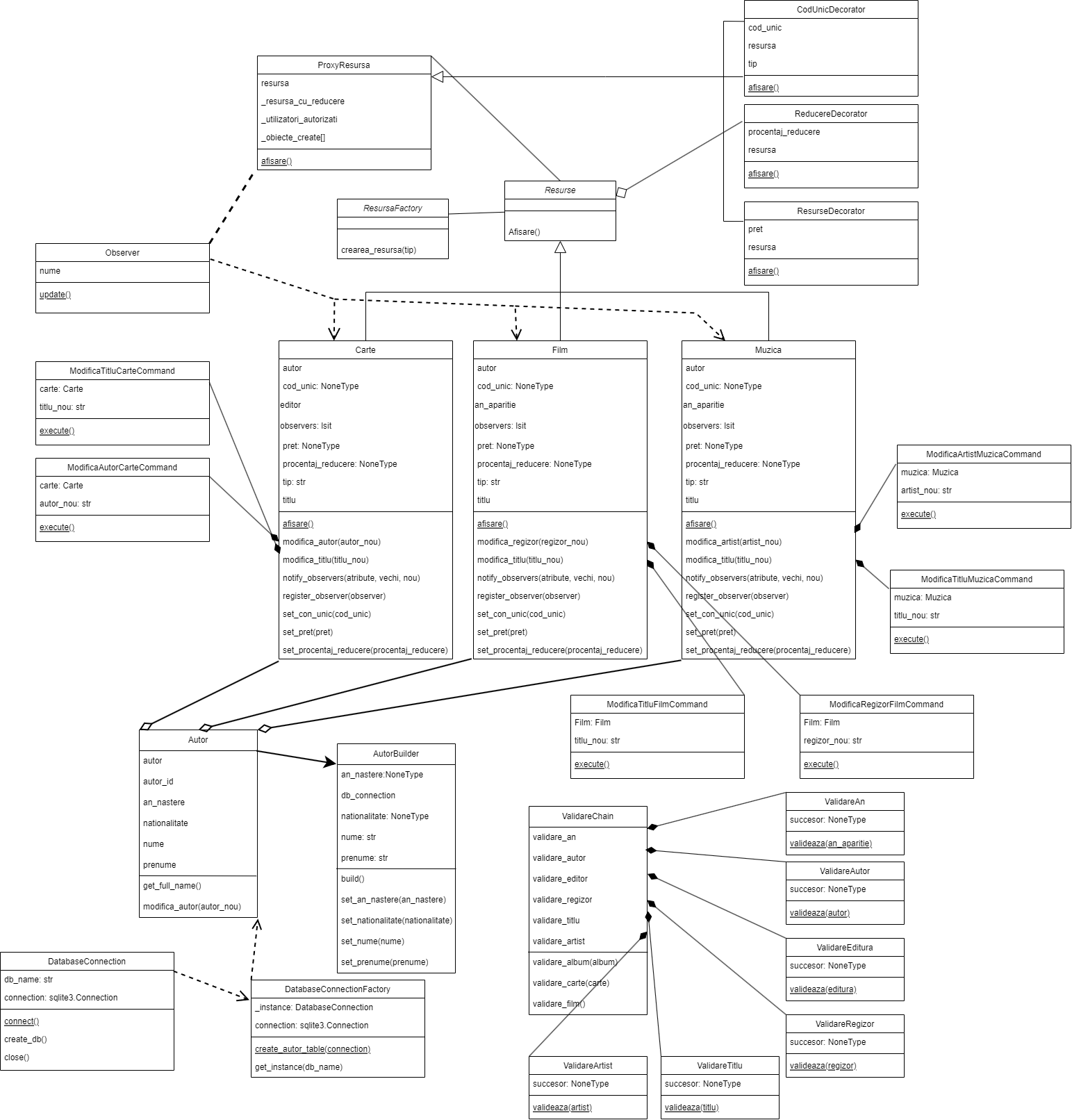


Figura 3.1 – Diagrama UML generală.

# Concluzie

În concluzie, utilizarea abordării orientate pe obiecte (OOP) și aplicarea diferitelor pattern-uri de proiectare, precum modelele de design creațional (Metoda Factory, Builder, Singleton), modelele de proiectare structurală (Decorator, Proxy) și modelele de proiectare comportamentală (Command, Chain of Responsibility, Observator), în dezvoltarea unei aplicații de consolă pentru gestionarea produselor (cărți, filme, albume muzicale) a dus la obținerea unei structuri mai modulară, o separare clară a responsabilităților și o flexibilitate crescută în gestionarea diferitelor tipuri de produse.

Prin utilizarea modelelor de design creațional, am reușit să encapsulăm procesul de creare a obiectelor, oferindu-ne un mecanism flexibil și modular pentru crearea și inițializarea acestora. Astfel, am putut gestiona diferite tipuri de produse (cărți, filme, muzică) utilizând clasa abstractă Resursa și clasele concrete care extind această clasă de bază.

Modelele de proiectare structurală, precum Decoratorul și Proxy-ul, ne-au permis să extindem funcționalitatea obiectelor existente într-un mod flexibil și modular. Prin intermediul decoratoarelor, am adăugat funcționalități suplimentare (afișarea prețului, aplicarea reducerilor, generarea unui cod unic) la clasele de resurse existente (Carte, Film, Muzica), fără a le modifica direct. Proxy-ul ne-a oferit un intermediar între utilizatori și resurse, permițând controlul și gestionarea accesului la acestea.

Modelele de proiectare comportamentală, cum ar fi Command, Chain of Responsibility și Observatorul, ne-au ajutat să gestionăm și să răspundem la diverse acțiuni și evenimente legate de resursele noastre. Prin intermediul comenzilor, am putut implementa acțiuni specifice pentru modificarea resurselor, iar prin Chain of Responsibility am putut configura un lanț de obiecte care să preia responsabilități în funcție de nevoile aplicației. Observatorul ne-a oferit posibilitatea de a înregistra și notifica evenimentele legate de resurse, facilitând integrarea de funcționalități suplimentare.

În ansamblu, utilizarea acestor pattern-uri de proiectare în dezvoltarea aplicației noastre de consolă pentru gestionarea produselor ne-a permis să obținem o structură modulară și flexibilă, cu o separare clară a responsabilităților și cu posibilitatea extinderii și modificării ulterioare fără a afecta întreaga aplicație. Astfel, am putut crea o aplicație scalabilă și ușor de întreținut, adaptată pentru gestionarea eficientă a diferitelor tipuri de produse.

# Bibliografie:

1. Codul sursă [01.03.2023] Disponibil:

<https://github.com/BunescuGabriel/TMPS>

1. Design Patterns [27.05.2023] Disponibil:

<https://sourcemaking.com/design_patterns> [1], [2] , [3] ], [4] ], [5]

1. Abstract Factory Design Pattern [27.05.2023] Disponibil:

[https://sourcemaking.com/design\_patterns/abstract\_factory [6](https://sourcemaking.com/design_patterns/abstract_factory%20%5b6)]

1. Builder Design Pattern [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/builder [7]

1. Singleton Design Pattern [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/singleton [8]

1. Decorator Design Pattern [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/decorator [9]

1. Proxy Design Pattern [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/proxy [10]

1. Chain of Responsibility [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/chain\_of\_responsibility [11]

1. Command Design Pattern [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/command [12]

1. Observer Design Pattern [27.05.2023] Disponibil:

https://sourcemaking.com/design\_patterns/observer [13]