Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Proiect de curs**

*Disciplina: Tehnici și Mecanisme de Proiectare Software*

Au efectuat: Oselschi Andrei, TI-204

A verificat: asist. univ. Gaidau Mihai

Chișinău – 2023

Cuprins

[**Introducere** 2](#_Toc136940193)

[**Design Pattern** 3](#_Toc136940194)

[**1.1 Modele de design creațional** 4](#_Toc136940195)

[**1.2 Modele de proiectare structurală** 5](#_Toc136940196)

[**1.3 Modele de design comportamental** 6](#_Toc136940197)

[**Diagrame UML** 8](#_Toc136940198)

[**2.1 Modele de design creațional** 9](#_Toc136940199)

[**2.2 Modele de proiectare structurală** 11](#_Toc136940200)

[**2.3 Modele de design comportamental** 13](#_Toc136940201)

[**Concluzie :** 15](#_Toc136940202)

[**Bibliografie :** 16](#_Toc136940203)

**Introducere**

Aplicația este un sistem simplu de gestionare a unui magazin, în special în ceea ce privește categoriile și produsele. Oferă o serie de funcționalități prin intermediul unui API RESTful, permițând crearea, recuperarea, actualizarea și ștergerea categoriilor și produselor. Scopul său este de a facilita gestionarea inventarului pentru magazinele online, permițând utilizatorilor să organizeze în mod eficient produsele pe categorii și să efectueze diverse operațiuni asupra acestora.

Obiectiv și caracteristici:

Obiectivul aplicației este de a simplifica gestionarea categoriilor și a produselor în cadrul unui magazin. Aceasta oferă o soluție convenabilă pentru organizarea și menținerea unui catalog de articole, simplificând pentru administratorii magazinelor adăugarea, modificarea și eliminarea produselor, precum și gestionarea categoriilor asociate.

Caracteristici cheie: - Funcții de bază: - Funcții de gestionare a magazinelor:

1. Gestionarea categoriilor: Utilizatorii pot crea noi categorii, pot prelua categoriile existente, pot actualiza detaliile categoriei (cum ar fi titlul și descrierea) și pot șterge categoriile atunci când este necesar.

2. Gestionarea produselor: Produsele pot fi create în cadrul unor categorii specifice, pot fi recuperate pe baza categoriei, pot fi actualizate cu informații noi și pot fi șterse din sistem.

3. Recuperarea datelor: Aplicația oferă puncte finale pentru recuperarea listelor de categorii, a categoriilor individuale identificate prin identificatori unici și a produselor aparținând unor categorii specifice.

4. API RESTful: Aplicația aderă la principiile Transferului de stat reprezentațional (REST) și oferă o interfață API bine definită și standardizată pentru interacțiunea cu sistemul de gestionare a magazinului.

5. Metode HTTP acceptate: Metode HTTP:: Aplicația acceptă diverse metode HTTP, inclusiv GET (pentru recuperarea datelor), POST (pentru crearea de date noi), PATCH (pentru actualizarea datelor existente) și DELETE (pentru ștergerea datelor).

Această aplicație servește ca bază pentru construirea infrastructurii backend a unui magazin online, permițând o integrare perfectă cu interfețe front-end, aplicații mobile sau alte sisteme care necesită acces la inventarul magazinului. În general, aplicația Store Management App simplifică procesul de organizare și gestionare a produselor, oferind o soluție eficientă pentru menținerea unui inventar actualizat și bine structurat pentru orice magazin online.

**Design Pattern**

Modelele de proiectare, cunoscute și sub denumirea de Design Patterns, sunt soluții reutilizabile pentru probleme comune care apar în dezvoltarea software-ului. Acestea reprezintă concepte generale și abstracte care pot fi aplicate în diverse situații pentru a rezolva provocări de proiectare specifice.

Originea Design Patterns se regăsește în cartea intitulată "Design Patterns: Elements of Reusable Object-Oriented Software" scrisă de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides, cunoscută și sub numele de Cartea Gang of Four. Această carte descrie 23 de modele clasice de proiectare care au fost dezvoltate și îmbunătățite de-a lungul timpului.

Un Design Pattern furnizează o soluție elegantă pentru o problemă comună, stabilind o abordare recomandată pentru structurarea și organizarea codului. Acestea pot acoperi diferite aspecte ale proiectării software-ului, inclusiv structura claselor, relațiile dintre obiecte, gestionarea fluxului de date și comunicarea între componente.

Utilizarea Design Patterns aduce numeroase beneficii în dezvoltarea software-ului, cum ar fi reutilizabilitatea crescută, ușurința înțelegerii și menținerea codului, reducerea complexității și sporirea flexibilității. Totuși, este important să înțelegi în mod adecvat fiecare Design Pattern și să-l aplici corespunzător în contextul problemei specifice pentru a putea beneficia de avantajele pe care le oferă.

## **Modele de design creațional**

Acestea sunt cele șase Design Patterns creational clasice. Fiecare dintre ele se concentrează pe diferite aspecte ale creării obiectelor și oferă soluții optimizate pentru situații specifice. Este crucial să înțelegi contextul și nevoile tale pentru a alege Design Pattern-ul creational potrivit în dezvoltarea software-ului tău.

1. Singleton (Singleton): Singleton este un Design Pattern creational care garantează că o clasă are o singură instanță și oferă un punct global de acces la această instanță. Este util în cazurile în care trebuie să existe o singură instanță a unei clase care să ofere acces global la acea instanță.

2. Factory Method (Metoda Fabrică): Factory Method este un Design Pattern creational care furnizează o interfață pentru crearea de obiecte, dar permite subclaselor să decidă ce clasă să instanțieze. În esență, acesta abstractizează procesul de creare a obiectelor și oferă o flexibilitate mai mare în alegerea clasei concrete pe care să o instanțiezi.

3. Abstract Factory (Fabrică Abstractă): Abstract Factory este un Design Pattern creational care furnizează o interfață pentru crearea unei familii de obiecte legate sau dependente, fără a specifica clasele concrete. Prin intermediul acestuia, poți crea obiecte care sunt concepute să funcționeze împreună și să fie coerente în cadrul unui context.

4. Builder (Constructor): Builder este un Design Pattern creational care separă procesul de construcție al unui obiect complex de reprezentarea sa. Acesta permite crearea obiectului pas cu pas, oferind o metodă flexibilă de construire a unui obiect cu opțiuni diferite.

5. Prototype (Prototip): Prototype este un Design Pattern creational care permite crearea de obiecte noi prin clonarea unui obiect existent, numit prototip. Aceasta elimină dependența de crearea de obiecte folosind constructori și oferă o modalitate mai flexibilă de generare a obiectelor.

6. Object Pool (Pool de Obiecte): Object Pool este un Design Pattern creational care oferă un set predefinit de obiecte inițializate și gata de utilizare, numite "pool", pentru a fi reutilizate. În loc să creezi și să distrugi obiecte repetat, poți recicla obiectele din pool, economisind astfel timp și resurse.

## **Modele de proiectare structurală**

Acestea sunt cele șapte Design Patterns structurale clasice. Fiecare dintre ele se ocupă de diferite aspecte ale organizării și interacțiunii între obiecte și oferă soluții pentru probleme specifice. Este esențial să înțelegi cerințele de proiectare și să alegi Design Pattern-ul structural potrivit pentru a obține o arhitectură software flexibilă și ușor de întreținut.

1. Adapter (Adaptor): Adapter este un Design Pattern structural care permite comunicarea între două clase care au interfețe incompatibile. Acționează ca un intermediar între aceste clase, convertind solicitările de la una într-un format acceptat de cealaltă.

2. Bridge (Pod): Bridge este un Design Pattern structural care separă abstractizarea de implementare, permițându-le să evolueze independent. Este util atunci când există mai multe abstracții și mai multe implementări și trebuie să le combinăm într-un mod flexibil.

3. Composite (Compus): Composite este un Design Pattern structural care permite tratarea uniformă a obiectelor individuale și a compozițiilor acestora. În esență, tratează un obiect individual și o colecție de obiecte în același mod, permițând manipularea lor într-un mod transparent.

4. Decorator (Decorator): Decorator este un Design Pattern structural care permite adăugarea de funcționalități suplimentare la un obiect în mod dinamic, fără a modifica clasa de bază. Utilizează încapsularea și compunerea pentru a extinde funcționalitatea unui obiect la runtime.

5. Facade (Fațadă): Facade este un Design Pattern structural care oferă o interfață simplificată pentru un subsistem complex de clase. Ascunde complexitatea din spatele unei interfețe unificate, permițând astfel utilizarea mai ușoară și mai intuitivă a subsistemului.

6. Flyweight (Obiect Ușor): Flyweight este un Design Pattern structural care optimizează utilizarea memoriei prin partajarea eficientă a obiectelor. În loc să creați un nou obiect pentru fiecare instanță, acesta reutilizează obiecte existente similare, economisind astfel resurse.

7. Proxy (Procuror): Proxy este un Design Pattern structural care furnizează o înlocuire sau o reprezentare a unui obiect real. Poate fi util în situații în care se dorește controlul suplimentar al accesului la obiectul real, cum ar fi accesul la distanță sau implementarea unui nivel de securitate.

## **1.3 Modele de design comportamental**

Acestea sunt cele nouă Design Patterns behaviorale clasice. Fiecare dintre ele abordează diferite aspecte ale comportamentului și interacțiunii între obiecte în cadrul unei aplicații software. Alegerea potrivită a Design Pattern-urilor behaviorale poate duce la un design mai flexibil, modular și ușor de extins.

1. Observer (Observator): Observer este un Design Pattern behavioral care stabilește o relație de tip unu-la-mulți între obiecte. Atunci când un obiect se schimbă, toate obiectele dependente de acesta sunt notificate automat și actualizate.

2. Strategy (Strategie): Strategy este un Design Pattern behavioral care permite schimbarea comportamentului unei clase prin înlocuirea unei strategii. Acesta separă algoritmul de implementarea specifică, permițând clasei să fie mai flexibilă și ușor de adaptat la diferite scenarii.

3. Chain of Responsibility (Lanț de Responsabilitate): Chain of Responsibility este un Design Pattern behavioral care permite transmiterea unei solicitări de-a lungul unei lanț de obiecte până când aceasta este gestionată de unul dintre aceste obiecte. Acesta reduce cuplarea între obiectul care solicită și obiectul care o prelucrează, oferind o flexibilitate mai mare.

4. Command (Comandă): Command este un Design Pattern behavioral care încapsulează o solicitare ca un obiect, permițând astfel parametrizarea clientului cu diferite solicitări, coadă sau înregistrare de solicitări și posibilitatea de a anula o solicitare. Acesta separă solicitarea de execuția acesteia.

5. Iterator (Iterare): Iterator este un Design Pattern behavioral care oferă o modalitate de a accesa elementele unui obiect de colecție într-un mod secvențial, fără a expune detaliile de implementare ale acestuia. Acesta permite parcurgerea eficientă a unei colecții fără a cunoaște structura internă.

6. Memento (Memento): Memento este un Design Pattern behavioral care capturează și salvează starea internă a unui obiect fără a încălca încapsularea. Acesta permite restaurarea obiectului la o stare anterioară și oferă un mecanism de revenire la stări anterioare.

7. State (Stare): State este un Design Pattern behavioral care permite unui obiect să-și schimbe comportamentul în funcție de starea internă. Acesta abstrage stările posibile ale unui obiect în clase separate și îi permite să treacă de la o stare la alta într-un mod controlat.

8. Template Method (Metodă Șablon): Template Method este un Design Pattern behavioral care definește scheletul unei algoritme într-o clasă de bază, lăsând implementarea detaliilor specifice subclaselor. Acesta oferă un mod convenabil de a defini o structură comună pentru mai multe clase.

9. Visitor (Vizitator): Visitor este un Design Pattern behavioral care separă algoritmul de obiectele pe care le operează. Permite adăugarea de noi operații la o ierarhie de obiecte fără a le modifica.

# **Diagrame UML**

O diagramă UML (Unified Modeling Language) este o reprezentare vizuală standardizată a unui sistem sau a unei aplicații software folosind simboluri și notații specifice. Scopul diagramei UML este de a comunica și documenta diverse aspecte ale structurii, comportamentului și interacțiunilor sistemului.

Diagramele UML oferă o modalitate de a vizualiza diferite perspective ale sistemului și de a evidenția elementele și relațiile cheie. Ele servesc ca un limbaj comun pentru comunicarea și înțelegerea arhitecturii și designului sistemului între dezvoltatori, designeri și alte părți interesate.

Există mai multe tipuri de diagrame UML, fiecare cu propriul său scop și obiectiv. Iată câteva exemple comune de diagrame UML:

1. Diagrama de clasă: reprezintă structura statică a sistemului, inclusiv clasele, atributele, metodele și relațiile dintre ele.

2. Diagrama de caz de utilizare: ilustrează interacțiunile dintre actori (utilizatori sau sisteme externe) și sistem, concentrându-se pe funcționalitatea sistemului și pe obiectivele utilizatorului.

3. Diagrama de secvență: prezintă interacțiunile între obiecte pe parcursul timpului, evidențiind ordinea mesajelor schimbate între ele.

4. Diagrama de activități: ilustrează fluxul activităților sau proceselor din sistem, prezentând secvența acțiunilor și deciziilor.

5. Diagrama mașinii de stări: reprezintă diversele stări și tranziții ale unui obiect sau sistem, captând comportamentul acestuia în răspuns la evenimente.

6. Diagrama componentelor: vizualizează componentele fizice sau logice ale sistemului și dependențele dintre ele.

7. Diagrama de implementare: arată componentele hardware și software ale sistemului și modul în care acestea sunt aranjate fizic într-o rețea sau mediu de calcul.

Aceste diagrame, împreună cu altele precum diagramele pachetelor, diagramele de comunicare și diagramele de colaborare, oferă o vedere completă a arhitecturii, designului și interacțiunilor sistemului.

Prin utilizarea diagramelelor UML, profesioniștii în software pot comunica și împărtăși idei în mod eficient, pot analiza și valida proiectele de sistem și pot facilita colaborarea între membrii echipei implicați în dezvoltarea și întreținerea sistemului.

## **2.1 Modele de design creațional**

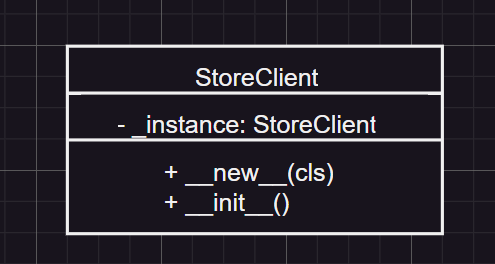


Figura 1.1 – Diagrama Singleton

În diagramă:

* Clasa StoreClient este reprezentată cu numele său.
* Atributul \_instance este notat ca o variabilă privată.
* Metodele \_\_new\_\_ și \_\_init\_\_ sunt indicate ca metode publice.

Modelul Singleton asigură faptul că există doar o singură instanță a clasei StoreClient în întregul program. Variabila privată `\_instance` păstrează această instanță unică a clasei. Prin suprascrierea metodei `\_\_new\_\_`, se verifică dacă o instanță există deja și, în caz contrar, se creează o instanță nouă. Metoda `\_\_init\_\_` este utilizată pentru a inițializa obiectul.

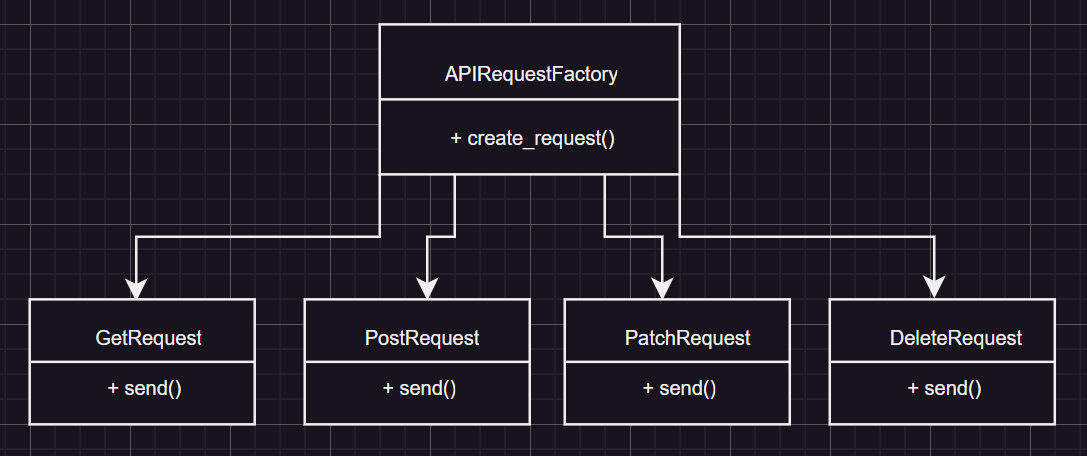


Figura 1.2 – Diagrama Factory

În diagrama prezentată:

- Clasa APIRequestFactory este reprezentată ca o clasă fabrică responsabilă de crearea diferitelor tipuri de solicitări.

- Metoda create\_request() este metoda fabricii care se ocupă de crearea și returnarea obiectelor de solicitare specifice.

- Clasele de solicitări concrete (GetRequest, PostRequest, PatchRequest, DeleteRequest) sunt clasele de produs care implementează interfața comună (metoda send()) definită de fabrică.

Modelul Factory Method permite clasei APIRequestFactory să deleghe crearea diferitelor tipuri de obiecte de solicitare către clasele de solicitare concrete. Metoda create\_request() din fabrică este responsabilă de crearea obiectului de solicitare adecvat în funcție de parametrii furnizați, cum ar fi metoda de solicitare. Această abordare oferă flexibilitate și extensibilitate ușoară atunci când se introduc noi tipuri de solicitări în viitor.

## **2.2 Modele de proiectare structurală**

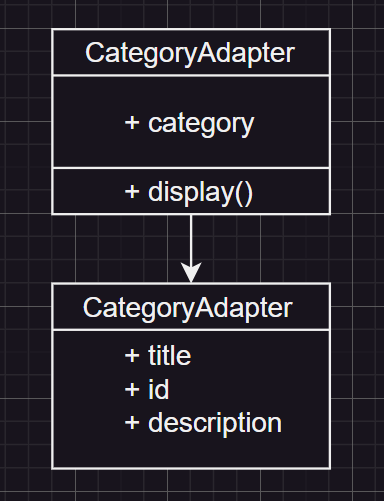


Figura 1.3 – Diagrama Adapter

În diagramă:

- Clasa CategoryAdapter acționează ca un adaptor. Ea adaptează obiectul Category la o interfață diferită.

- Atributul "categorie" din CategoryAdapter reprezintă obiectul Category care este împachetat.

- Metoda "display()" din CategoryAdapter adaptează comportamentul obiectului Category la interfața dorită.

Modelul Adapter permite clasei CategoryAdapter să adapteze un obiect Category existent la o interfață diferită. Clasa CategoryAdapter include obiectul Category și furnizează un set diferit de metode sau comportament prin intermediul metodei "display()". Aceasta permite utilizarea obiectului Category în contexte care așteaptă interfața adaptată.

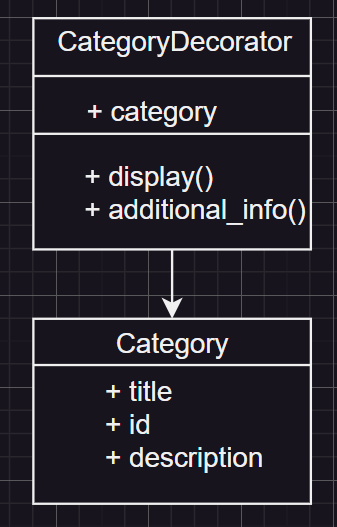


Figura 1.4 – Diagrama Adapter

În diagramă:

- Clasa CategoryDecorator reprezintă Decoratorul. Aceasta încorporează obiectul Category și adaugă un comportament suplimentar.

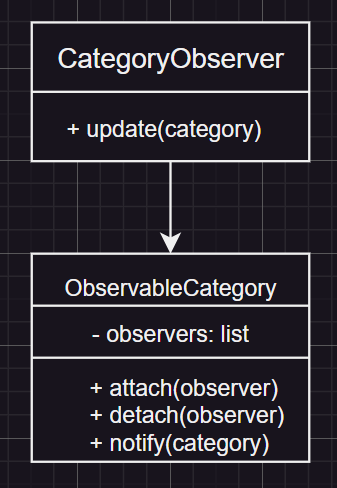
- Atributul "categorie" din CategoryDecorator reprezintă obiectul Category care este împachetat.

- Metoda "display()" din CategoryDecorator suprascrie comportamentul metodei "display()" a obiectului Category.

- Metoda "additional\_info()" din CategoryDecorator adaugă un comportament suplimentar metodei "display()".

Modelul Decorator permite clasei CategoryDecorator să adauge funcționalități noi sau să modifice comportamentul obiectului Category fără a-l modifica direct. Prin împachetarea obiectului Category, CategoryDecorator poate îmbunătăți metoda "display()" prin adăugarea de informații suplimentare utilizând metoda "additional\_info()".

## **2.3 Modele de design comportamental**



Figuta 1.5 – Diagrama Observer

În diagramă:

- Clasa CategoryObserver reprezintă Observatorul. Aceasta definește metoda "update()" pentru a primi actualizări de la obiectul observabil.

- Clasa ObservableCategory reprezintă Observable. Menține o listă de observatori și oferă metode pentru gestionarea lor (attach(), detach(), notify()).

- Alte clase (OtherClass, AnotherClass etc.) pot acționa ca observatori prin implementarea metodei "update()" și înregistrarea cu obiectul observabil.

Modelul Observer permite obiectului ObservableCategory să notifice observatorii săi înregistrați (cum ar fi OtherClass, AnotherClass etc.) atunci când există actualizări. Observatorii implementează metoda "update()", care este apelată de obiectul observabil pentru a furniza informațiile actualizate despre categorie.

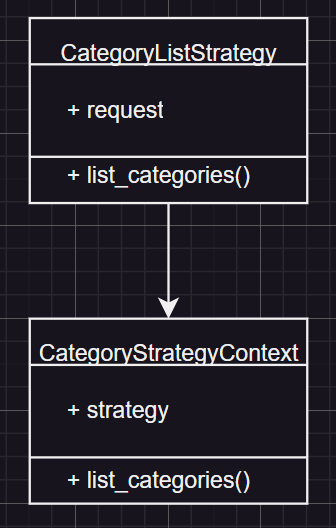


Figura 1.6 – Diagrama Strategy

În diagramă:

- Clasa CategoryListStrategy reprezintă Strategia. Aceasta încapsulează algoritmul pentru listarea categoriilor pe baza unei cereri specifice.

- Atributul de solicitare din CategoryListStrategy reprezintă obiectul de solicitare utilizat pentru preluarea categoriilor.

- Metoda list\_categories() din CategoryListStrategy execută algoritmul pentru listarea categoriilor pe baza cererii furnizate.

- Clasa CategoryStrategyContext reprezintă Contextul. Menține o referință la strategia curentă și oferă o metodă pentru a executa strategia (list\_categories()).

Modelul Strategy permite CategoryStrategyContext să delege algoritmul pentru listarea categoriilor către diferite strategii, precum CategoryListStrategy. CategoryListStrategy încapsulează algoritmul specific pentru listarea categoriilor pe baza solicitării furnizate. CategoryStrategyContext oferă o modalitate de a seta strategia curentă și de a o executa, ascunzând detaliile specifice ale implementării strategiei de la client.

# **Concluzie :**

Aplicația de gestionare a magazinului este un instrument valoros pentru a organiza și gestiona eficient produsele într-un magazin online. Cu un API prietenos și funcționalitate extinsă, aplicația simplifică procesul de creare, actualizare și ștergere a categoriilor și produselor. Administratorii beneficiază de o interfață convenabilă pentru a menține un inventar precis și bine structurat.

Cu ajutorul funcțiilor aplicației, proprietarii de magazine pot adăuga cu ușurință categorii noi, pot atribui produse la categorii specifice, pot actualiza informațiile despre produse și pot elimina articolele vechi sau întrerupte. Designul API RESTful al aplicației asigură compatibilitatea cu diverse interfețe front-end, aplicații mobile și sisteme externe, facilitând integrarea în ecosistemul magazinului.

Aplicația de gestionare a magazinului îmbunătățește productivitatea prin optimizarea gestionării stocurilor, reducerea efortului manual și oferirea unui sistem centralizat pentru operațiuni legate de produse. Aceasta contribuie la creșterea eficienței generale, facilitând administratorilor menținerea unui catalog actualizat și bine organizat.

În concluzie, aplicația de gestionare a magazinului oferă o soluție fiabilă și eficientă pentru gestionarea categoriilor și produselor într-un magazin online. Cu funcționalitatea sa robustă, API-ul prietenos și suportul pentru operațiuni esențiale, această aplicație oferă proprietarilor de magazine puterea de a oferi o experiență de cumpărături organizată și impecabilă pentru clienții lor.

# **Bibliografie :**

1. <http://elf.cs.pub.ro/poo/laboratoare/design-patterns>
2. <https://app.diagrams.net/>
3. <https://www.youtube.com/watch?v=tAuRQs_d9F8>
4. <https://refactoring.guru/design-patterns/python>
5. <https://www.geeksforgeeks.org/python-design-patterns/>
6. <https://github.com/PacktPublishing/Mastering-Python-Design-Patterns-Second-Edition>
7. <https://www.developer.com/design/using-design-patterns-in-uml/>