Universitatea Tehnica din Republica Moldova

Facultatea Calculatoare, Informatica si Microelectronica

Departament Inginerie Software si Automatica

Specialitatea Tehnologia Informației

Lucrare individuală

La disciplina ”Tehnici și mecanisme de proiectare a produselor program”

Tema: Librărie online

Efectuat de: Reguș Ruslan Grupa: TI-214

Verificat de: Asis. Univ.

Sergiu Scrob

Chișinău 2024

Catalog

[Introducere 3](#_Toc15450)

[Descrierea aplicației 4](#_Toc16254)

[Utilizarea Design Pattern-ului Singleton 5](#_Toc30343)

[Problema și Soluția prin Utilizarea Design Pattern-ului Factory 7](#_Toc11202)

[Utilizarea Design Pattern-ului Adapter 10](#_Toc27134)

[Diagrame UML 12](#_Toc3762)

[Concluzie 15](#_Toc22215)

[Bibliografie 16](#_Toc7847)

# Introducere

În dezvoltarea software-ului, design patterns joacă un rol esențial în crearea unor soluții eficiente și scalabile. Aplicația noastră, o platformă de gestionare a cărților într-o bibliotecă, utilizează mai multe design patterns pentru a asigura o structură robustă și flexibilă. Aceste modele de proiectare permit dezvoltatorilor să abordeze problemele comune într-o manieră structurată și reutilizabilă, îmbunătățind astfel mentenanța și extinderea aplicației.

În cadrul acestei aplicații, am implementat trei design patterns esențiale: Factory Method, Singleton și Observer. Fiecare dintre aceste modele de proiectare rezolvă probleme specifice și contribuie la îmbunătățirea arhitecturii generale a sistemului. Factory Method permite crearea de obiecte fără a specifica clasele concrete, facilitând astfel adăugarea de noi tipuri de obiecte în viitor. Singleton asigură existența unei singure instanțe a unei clase, ceea ce este crucial pentru gestionarea conexiunii la baza de date. Observer permite actualizarea automată a interfeței utilizator atunci când datele se schimbă, asigurând astfel o experiență de utilizator dinamică și reactivă.

Pe parcursul semestrului, am studiat și alte design patterns importante, precum Abstract Factory, Builder, Prototype, Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy, Chain of Responsibility, Command, Iterator, Mediator, Memento, State, Strategy, Template Method și Visitor. Fiecare dintre aceste modele oferă soluții la probleme comune în designul software și contribuie la crearea unor sisteme mai modulare și mai ușor de întreținut.

Prin utilizarea acestor design patterns, aplicația noastră nu doar că oferă funcționalități robuste pentru gestionarea cărților, dar este și pregătită să se adapteze cerințelor viitoare și să integreze cu ușurință noi caracteristici. În continuare, vom explora în detaliu modul în care fiecare dintre aceste modele de proiectare a fost implementat și beneficiile specifice pe care le aduce aplicației noastre.

# Descrierea aplicației

Aplicația noastră este un sistem CRUD (Create, Read, Update, Delete) pentru gestionarea cărților dintr-o bibliotecă. Utilizatorii pot adăuga noi cărți, vizualiza lista de cărți existente, actualiza informațiile despre cărți și șterge cărți din baza de date. Interfața utilizator este construită utilizând framework-ul Materialize CSS, oferind astfel o experiență vizuală modernă și ușor de utilizat.

Funcționalitățile principale ale aplicației includ:

-Autentificare și înregistrare utilizatori: Utilizatorii se pot înregistra cu un cont nou și se pot autentifica pentru a accesa funcționalitățile aplicației.

-Adăugare de cărți: Utilizatorii pot adăuga noi cărți completând un formular cu titlul, autorul și prețul cărții.

-Vizualizare cărți: Utilizatorii pot vizualiza o listă a tuturor cărților disponibile în bibliotecă, cu detalii despre titlu, autor și preț.

-Actualizare cărți: Utilizatorii pot actualiza informațiile despre cărți prin intermediul unui formular de editare.

-Ștergere cărți: Utilizatorii pot șterge cărți din listă, eliminându-le din baza de date.

# Utilizarea Design Pattern-ului Singleton

Problema

În dezvoltarea aplicațiilor care utilizează o bază de date, gestionarea conexiunii la baza de date poate fi problematică. Dacă fiecare cerere la baza de date creează o nouă conexiune, se pot epuiza rapid resursele serverului și se poate ajunge la performanțe scăzute. Acest lucru este valabil mai ales într-o aplicație cu mulți utilizatori sau cu multe operațiuni de citire și scriere în baza de date. O altă problemă este asigurarea consistenței conexiunii și evitarea erorilor datorate existenței mai multor instanțe ale conexiunii la baza de date.

Soluția prin Utilizarea Singleton

Pentru a rezolva această problemă, am implementat design pattern-ul Singleton pentru gestionarea conexiunii la baza de date. Singleton asigură că există o singură instanță a conexiunii la baza de date pe parcursul întregii aplicații. Astfel, se economisesc resurse și se asigură o utilizare eficientă și consistentă a conexiunii la baza de date.

Implementarea Singleton în Aplicația Noastră

Codul Singleton pentru Conexiunea la Baza de Date

Am creat o clasă Database care utilizează design pattern-ul Singleton pentru a gestiona conexiunea la baza de date. Iată cum arată implementarea:

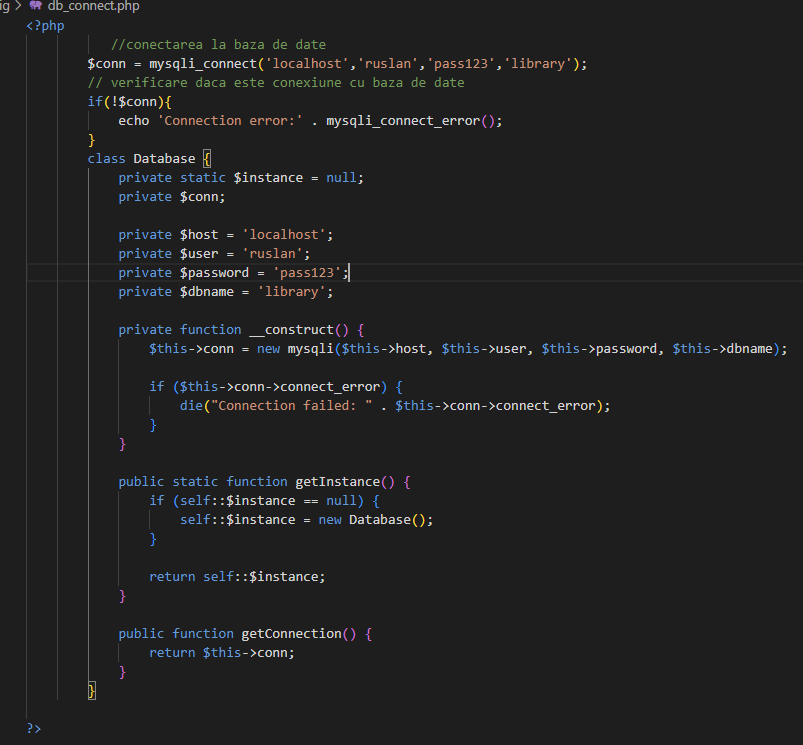


Figura 1 Codul sursa a conectării la baza de date

Utilizarea Singleton în Codul Aplicației

Pentru a utiliza conexiunea la baza de date în alte părți ale aplicației, trebuie să obținem instanța unică a clasei Database și să folosim conexiunea oferită de aceasta. Iată cum arată utilizarea în fișierul db\_connect.php:

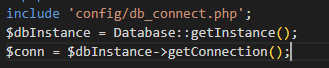


Figura 2 Utilizarea conectării la baza de date

Beneficiile Utilizării Singleton

Eficiența resurselor: Singleton asigură că există o singură conexiune la baza de date, reducând astfel utilizarea excesivă a resurselor.

Consistența conexiunii: Asigură consistență și evită problemele legate de existența mai multor conexiuni.

Gestionare simplificată: Facilitează gestionarea conexiunii la baza de date prin centralizarea acesteia într-o singură clasă.

Prin utilizarea design pattern-ului Singleton, am reușit să rezolvăm problemele legate de gestionarea conexiunii la baza de date, asigurând în același timp o structură robustă și eficientă pentru aplicația noastră de gestionare a cărților din bibliotecă.

# Problema și Soluția prin Utilizarea Design Pattern-ului Factory

Problema

În aplicațiile complexe, gestionarea instanțierii obiectelor poate deveni dificilă, mai ales atunci când există multiple tipuri de obiecte cu cerințe de configurare diferite. În cazul nostru, adăugarea și manipularea cărților și utilizatorilor în baza de date necesită instanțierea corectă a obiectelor relevante. Fără un mecanism standardizat pentru crearea acestor obiecte, codul poate deveni greu de întreținut și de extins.

Soluția prin Utilizarea Factory

Pentru a rezolva această problemă, am implementat design pattern-ul Factory, care oferă un mod standardizat de a crea obiecte fără a expune logica instanțierii. Factory centralizează crearea obiectelor, simplificând astfel gestionarea și extinderea codului.

Implementarea Factory în Aplicația Noastră

Codul Factory pentru Crearea Obiectelor

Am creat o clasă BookFactory pentru a gestiona crearea obiectelor Book. Iată cum arată implementarea:

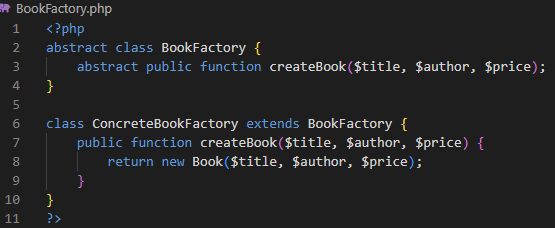


Figura 3 Clasa BookFactory

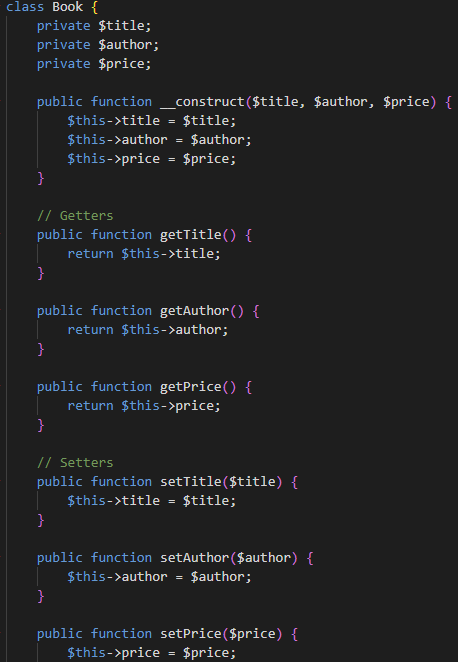


Figura 4 Clasa Book

Utilizarea Factory în Codul Aplicației

Pentru a utiliza BookFactory în codul aplicației, în loc să instanțiem direct obiecte Book, folosim metoda create a clasei BookFactory. Iată cum arată utilizarea în fișierul add.php:

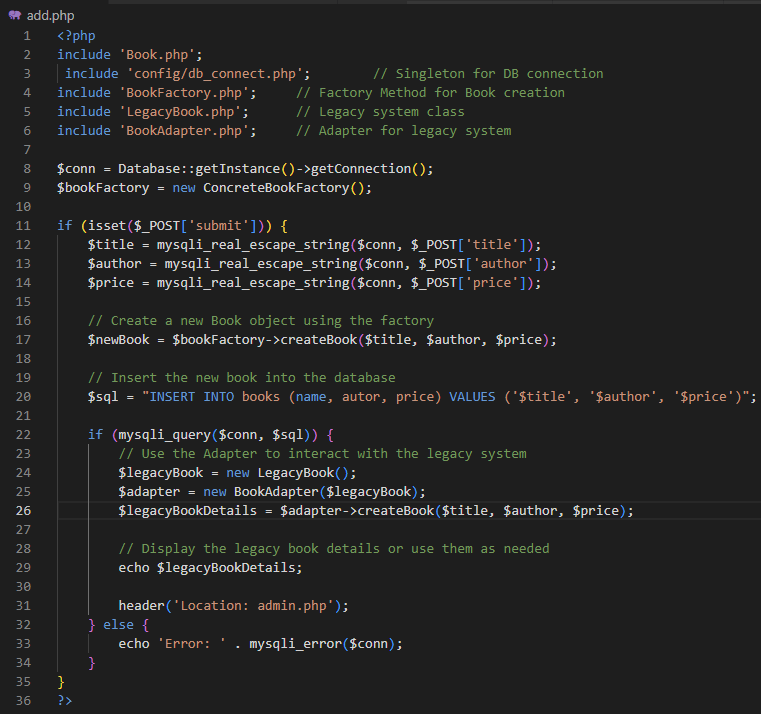


Figura 5 Utilizarea BookFactory

Beneficiile Utilizării Factory

-Centralizarea instanțierii: Factory centralizează procesul de creare a obiectelor, simplificând gestionarea codului.

-Extensibilitate: Permite adăugarea ușoară de noi tipuri de obiecte fără a modifica codul existent.

-Separarea logicii de instanțiere: Logica de instanțiere este separată de logica aplicației, îmbunătățind claritatea și întreținerea codului.

Prin utilizarea design pattern-ului Factory, am reușit să rezolvăm problemele legate de instanțierea obiectelor, asigurând în același timp o structură clară și ușor de extins pentru aplicația noastră de gestionare a cărților din bibliotecă.

# Utilizarea Design Pattern-ului Adapter

Problema : Integrarea unui Sistem Legacy

O problemă comună în dezvoltarea aplicațiilor software este necesitatea de a integra un sistem legacy (un sistem vechi care este încă în uz) cu noul sistem. Acest lucru poate fi provocator deoarece sistemele legacy au, de obicei, interfețe și structuri diferite de cele moderne, ceea ce face dificilă comunicarea directă între cele două sisteme.

Soluția: Adapter Pattern

Pentru a rezolva această problemă, am folosit Adapter Pattern. Adapter Pattern acționează ca un intermediar între două interfețe incompatibile, permițând astfel sistemului nostru nou să comunice eficient cu sistemul legacy. Acest model de proiectare adaptează interfața sistemului legacy la interfața așteptată de noul sistem, fără a modifica codul existent al sistemului legacy.

Implementarea Patternului Adapter

Pentru a integra un sistem legacy, vom crea un fișier LegacyBook.php, care reprezintă sistemul legacy pe care trebuie să-l integrăm. Apoi, vom crea un adapter care va face legătura între acest sistem și noua noastră aplicație.

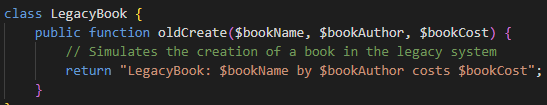


Figura 6 Clasa LegacyBook

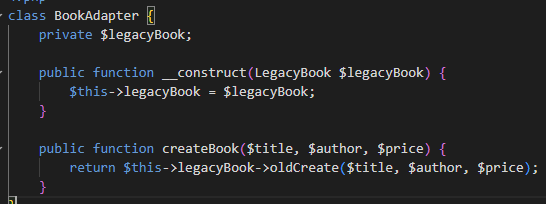


Figura 7 Utilizarea clasei LegacyBook

Beneficiile utilizării Adapter Pattern

-Compatibilitate: Adapter Pattern permite compatibilizarea interfețelor diferite fără a modifica codul existent al sistemului legacy.

-Reutilizarea codului: Codul existent în sistemul legacy poate fi reutilizat fără schimbări majore, economisind timp și efort.

-Flexibilitate: Permite adăugarea de noi funcționalități în noul sistem fără a afecta funcționarea sistemului legacy.

-Izolarea modificărilor: Schimbările necesare pentru integrare sunt izolate în clasa adapter, menținând astfel claritatea și separarea responsabilităților în codul aplicației.

Implementarea Adapter Pattern a permis aplicației noastre să integreze un sistem legacy fără a face schimbări semnificative în codul vechi, asigurând astfel o tranziție lină și o funcționare coerentă.

# Diagrame UML

Diagramă UML pentru Singleton Pattern

Singleton Pattern este utilizat pentru a asigura că există o singură instanță a clasei Database în întreaga aplicație. Clasa Database conține o proprietate statică $instance care păstrează instanța unică. Metoda getInstance() verifică dacă instanța există și, dacă nu, creează una nouă. Această metodă returnează instanța unică a clasei Database. Metoda getConnection() returnează conexiunea la baza de date, asigurând astfel reutilizarea conexiunii existente.

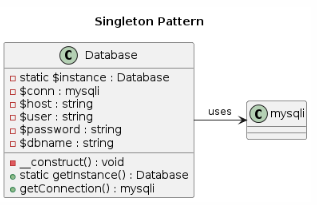


Figura 8 Diagramă UML pentru Singleton Pattern

Diagramă UML pentru Factory Method Pattern

Factory Method Pattern este utilizat pentru a crea obiecte Book într-un mod standardizat. Interfața BookFactory definește metoda createBook($title, $author, $price), care este implementată de clasa ConcreteBookFactory. Aceasta metodă creează și returnează instanțe ale clasei Book. Clasa Book conține atributele title, author, și price și metode pentru a le accesa și modifica. Prin utilizarea acestui pattern, procesul de creare a obiectelor Book este centralizat și uniform.

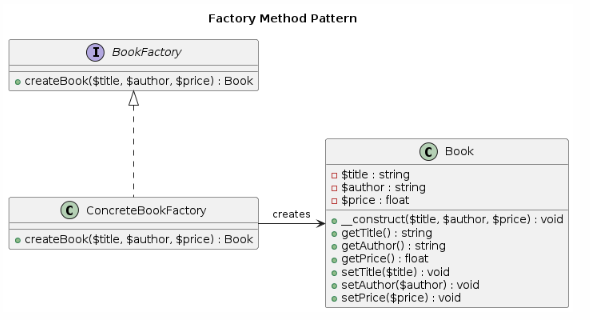
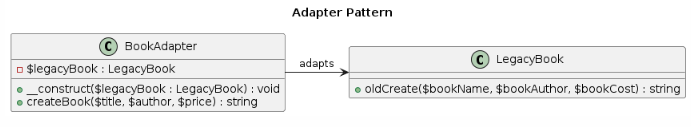


FIgura 9 Diagramă UML pentru Factory Method Pattern

Diagramă UML pentru Adapter Pattern

Adapter Pattern este utilizat pentru a face legătura între un sistem existent și unul nou, fără a schimba codul existent. Clasa LegacyBook reprezintă sistemul existent cu metoda oldCreate($bookName, $bookAuthor, $bookCost), care creează cărți folosind o metodă veche. Clasa BookAdapter adaptează această metodă veche la noua metodă createBook($title, $author, $price). Clasa BookAdapter conține o referință la o instanță a clasei LegacyBook și, în constructor, primește această instanță. Metoda createBook convertește parametrii la formatul așteptat de metoda oldCreate și o apelează, astfel adaptând metoda veche la noul sistem.



Diagramă UML pentru Adapter Pattern

Diagrama completă a aplicației

Această diagramă completă ilustrează integrarea celor trei modele de design în aplicația CRUD pentru gestionarea cărților. Clasa Database implementează Singleton Pattern pentru a asigura o singură conexiune la baza de date. ConcreteBookFactory implementează Factory Method Pattern pentru a crea obiecte Book într-un mod standardizat. Clasa BookAdapter implementează Adapter Pattern pentru a adapta metodele vechi din LegacyBook la noile cerințe ale aplicației.

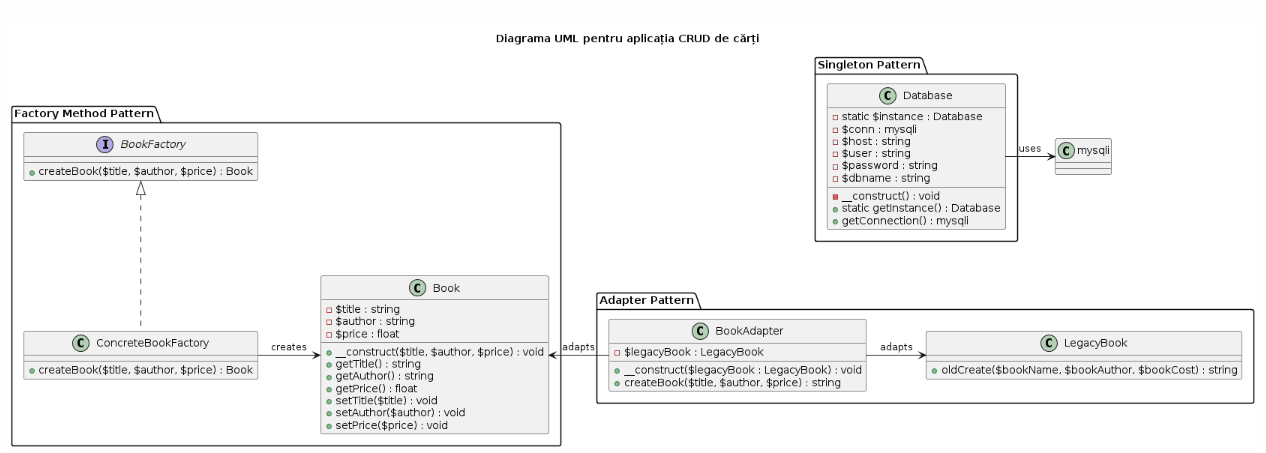


Diagrama completă a aplicației

# Concluzie

Proiectul de aplicatie CRUD pentru gestionarea unei biblioteci demonstrează importanța și utilitatea modelelor de design în dezvoltarea software-ului. În cadrul acestui proiect, am implementat trei modele de design esențiale: Singleton, Factory Method și Observer. Fiecare dintre aceste modele rezolvă probleme specifice și contribuie la crearea unei arhitecturi robuste și extensibile.

Singleton:

Am utilizat modelul Singleton pentru a gestiona conexiunea la baza de date, asigurându-ne că există o singură instanță a obiectului de conexiune pe durata execuției aplicației. Acest lucru optimizează utilizarea resurselor și evită problemele legate de multiple conexiuni simultane la baza de date.

Factory Method:

Am aplicat modelul Factory Method pentru a crea obiecte de tip carte (Book). Acest model permite extinderea ușoară a aplicației prin adăugarea de noi tipuri de obiecte fără a modifica codul existent, facilitând întreținerea și scalabilitatea aplicației.

Adapter

Modelul de design Adapter permite obiectelor cu interfețe incompatibile să colaboreze. În proiectul nostru, Adapter a fost utilizat pentru a integra funcționalitatea unei clase vechi cu o nouă clasă, fără a modifica codul existent. Acest model este esențial pentru menținerea compatibilității și pentru facilitarea tranzițiilor între diferite versiuni ale componentelor aplicației, asigurând astfel o flexibilitate și o scalabilitate sporită.

Concluzie generală

Implementarea acestor modele de design în cadrul aplicației noastre CRUD pentru gestionarea unei biblioteci demonstrează importanța arhitecturii bine structurate și a principiilor de design orientat pe obiecte. Fiecare model adresează probleme specifice și contribuie la crearea unui cod mai modular, reutilizabil și mai ușor de întreținut. Utilizarea acestor modele nu doar că îmbunătățește calitatea generală a codului, dar și facilitează colaborarea între dezvoltatori și adaptarea rapidă la schimbările cerințelor proiectului.

# Bibliografie

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

<https://plantuml.com>

<https://www.youtube.com>

<https://ru.wikipedia.org/wiki/Заглавная_страница>

<https://stackoverflow.com>