# Universitatea Tehnică a Moldovei

# Facultatea Calculatoare Informatică și Microelectronică

# RAPORT

**Lucrarea de laborator nr. 3**

TMPS

**Tema: Structural Design Pattern**

A efectuat: Cucuietu Ion, gr. TI-202

A verificat: asis. univ. Gaidau Mihai

Chişinău 2023

## **Objectives:**

**1. Study and understand the Structural Design Patterns.**

**2. As a continuation of the previous laboratory work, think about the functionalities that your system will need to provide to the user.**

**3. Implement some additional functionalities, or create a new project using structural design patterns.**

## **Theoretical background:**

    Structural design patterns are a category of design patterns that focus on the composition of classes and objects to form larger structures and systems. They provide a way to organize objects and classes in a way that is both flexible and efficient, while allowing for the reuse and modification of existing code. Structural design patterns address common problems encountered in the composition of classes and objects, such as how to create new objects that inherit functionality from existing objects, how to create objects that share functionality without duplicating code, or how to define relationships between objects in a flexible and extensible way.

    Some examples of from this category of design patterns are:

* Adapter
* Bridge
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

**Sarcina practică:**

Pentru acest laborator am ales la fel limbajul C# pentru implementarea Design Patterns Structural, deoarece C# ofera:

* Reutilizarea codului: C# oferă posibilitatea de a defini clase și funcții reutilizabile, care pot fi folosite în mai multe locuri din programul tău.
* Interoperabilitate: C# poate fi folosit cu ușurință pentru crearea de aplicații Windows sau web, iar .NET Framework oferă suport pentru alte limbaje, cum ar fi Visual Basic sau C++.
* Tipizarea statică: C# este un limbaj tipizat static, ceea ce înseamnă că erorile de tip pot fi detectate mai devreme în procesul de dezvoltare și reduce șansele de erori la runtime.
* Compatibilitate: C# este un limbaj multiplatformă, ceea ce înseamnă că codul poate fi rulat pe diferite sisteme de operare, inclusiv Windows, Linux sau macOS.
* Performanță: C# este un limbaj compilat, ceea ce înseamnă că programul tău va fi mult mai rapid decât unul interpretat. În concluzie, C# poate fi o alegere bună pentru implementarea unui pattern de design structural în proiectul tău, datorită numeroaselor caracteristici și facilități pe care le oferă.

1. **Adapter**

Acest șablon permite utilizarea unei clase existente într-un context diferit, prin adaptarea acesteia pentru a se potrivi cu interfața cerută de sistem.

Acest șablon se bazează pe două elemente principale: adaptatul și adaptorul. Adaptatul este clasa existentă pe care vrem să o adaptăm pentru a se potrivi cu interfața de care avem nevoie. Adaptorul este clasa nouă care va oferi interfața compatibilă cu sistemul nostru și va utiliza metodele și proprietățile adaptatului pentru a realiza această adaptare.

Adaptorul poate fi de două tipuri: adaptor de obiect sau adaptor de clasă. Adaptorul de obiect folosește o instanță a adaptatului pentru a oferi interfața compatibilă, în timp ce adaptorul de clasă utilizează moștenirea pentru a obține aceeași funcționalitate. În C#, putem implementa șablonul Adapter folosind atât adaptorul de obiect, cât și adaptorul de clasă, în funcție de contextul specific al aplicației noastre.

În cazul proiectului meu, adapterul poate fi utilizat pentru a permite împrumutarea unor noi tipuri de obiecte care nu implementează interfața ILibraryItem, cum ar fi reviste sau e-book-uri.

Pentru a crea un adapter, trebuie să creăm o clasă care implementează interfața ILibraryItem și să o conectăm la clasa corespunzătoare a obiectului neadaptat. Clasa adapter trebuie să convertească apelurile metodelor ILibraryItem în apeluri la metodele obiectului neadaptat.

Pentru exemplificare, putem presupune că vrem să împrumutăm o revistă, care are proprietăți precum Title, Author și Pages. Putem crea o clasă Revista care implementează interfața ILibraryItem și un adapter RevistaAdapter care convertește apelurile metodelor ILibraryItem în apeluri la metodele Revistei.

public class Revista : ILibraryItem

{

public string Title { get; set; }

public string Author { get; set; }

public int Pages { get; set; }

}

public class RevistaAdapter : ILibraryItem

{

private readonly Revista \_revista;

public RevistaAdapter(Revista revista)

{

\_revista = revista;

}

public string Title => \_revista.Title;

public string Author => \_revista.Author;

public int Pages => \_revista.Pages;

}

Apelul metodelor pe obiectul de tip RevistaAdapter ar fi echivalent cu apelul metodelor corespunzătoare pe obiectul de tip Revista, deoarece obiectul adapter convertește apelurile la metodele interfeței ILibraryItem la apelurile corespunzătoare la metodele Revistei.

1. **Decorator**

Decorator este un pattern structural care permite adăugarea de comportamente suplimentare la un obiect într-un mod dinamic. În loc să modificăm clasa de bază a obiectului, putem crea o serie de clase Decorator care îl împachetează și îi adaugă comportamente suplimentare.

Prin urmare, decoratorul oferă o alternativă la moștenirea clasică, oferind astfel flexibilitate și extensibilitate prin încapsularea obiectelor în alte obiecte care adaugă comportamente suplimentare.

Decoratorul utilizează un obiect cu aceeași interfață ca obiectul original și îl învelește cu un decorator cu aceeași interfață, astfel încât decoratorul îndeplinește aceeași funcție ca și obiectul original și oferă o modalitate de adăugare a comportamentelor suplimentare.

Mai intii am creat o nouă interfață:

public interface ILibraryItemDecorator : ILibraryItem

{

ILibraryItem LibraryItem { get; }

}

Această interfață va fi implementată de toate clasele decorator pentru obiectele din bibliotecă.

Următorul pas este crearea clasei abstracte LibraryItemDecoratorBase care va conține funcționalitățile comune ale tuturor claselor decorator. Aceasta va implementa interfața ILibraryItemDecorator și va avea un constructor care primește ca parametru un obiect de tip ILibraryItem și îl va atribui proprietății LibraryItem.

public abstract class LibraryItemDecoratorBase : ILibraryItemDecorator

{

protected readonly ILibraryItem LibraryItem;

public LibraryItemDecoratorBase(ILibraryItem libraryItem)

{

LibraryItem = libraryItem;

}

public virtual string Title => LibraryItem.Title;

public virtual string Author => LibraryItem.Author;

public virtual int Pages => LibraryItem.Pages;

// Adaugă calificatorul "this." în fața câmpului "LibraryItem"

ILibraryItem ILibraryItemDecorator.LibraryItem => this.LibraryItem;

}

Clasele concrete de decorator vor extinde această clasă abstractă și vor suprascrie metodele pentru a adăuga funcționalitățile specifice fiecărei clase.

Următorul decorator pe care îl vom crea va adăuga un număr de inventar pentru fiecare element din bibliotecă. Clasa InventoryNumberDecorator va fi definită astfel:

public class InventoryNumberDecorator : LibraryItemDecoratorBase

{

private readonly int \_inventoryNumber;

public InventoryNumberDecorator(ILibraryItem libraryItem, int inventoryNumber) : base(libraryItem)

{

\_inventoryNumber = inventoryNumber;

}

public override string Title => $"({\_inventoryNumber}) {LibraryItem.Title}";

}

Acest decorator adaugă un număr de inventar la începutul titlului și suprascrie metoda Title pentru a afișa titlul cu numărul de inventar.

Mai departe, putem testa codul adăugând câteva instanțe de obiecte și decorându-le cu acest decorator astfel:

var inventoryNumber = 1001;

ILibraryItem item = new InventoryNumberDecorator(book, inventoryNumber);

// Afisam informatii despre carte cu numarul de inventar

Console.WriteLine($"'{item.Title}' de {item.Author} cu numarul de inventar {((InventoryNumberDecorator)item).Title}");

Putem vedea rezultatul.



1. **Proxy**

Proxy este un design pattern de tip structural care permite crearea unui obiect intermediar care acționează ca o "fațadă" pentru un obiect subiacent complex și protejează clienții de detaliile de implementare ale acestuia. Prin intermediul unui astfel de obiect intermediar, numit "proxy", clienții pot accesa obiectul subiacent și pot beneficia de funcționalitatea acestuia fără a cunoaște detaliile interne ale implementării sale.

Proxy-ul poate fi folosit pentru a controla accesul la obiectul subiacent sau pentru a oferi un comportament suplimentar înainte sau după apelurile către obiectul subiacent. Există mai multe tipuri de proxy, cum ar fi proxy-ul de acces, proxy-ul de filtrare, proxy-ul de obiecte virtuale, proxy-ul de securitate și proxy-ul de tranzacționare.

public class LoanProxy : ILoan

{

private readonly ILoan \_loan;

private readonly IUser \_user;

public LoanProxy(ILoan loan, IUser user)

{

\_loan = loan;

\_user = user;

}

public ILibraryItem Item => \_loan.Item;

public IUser Borrower => \_loan.Borrower;

public DateTime BorrowDate => \_loan.BorrowDate;

public void Borrow(IUser user)

{

if (user.Age >= 15) // verificăm dacă utilizatorul are vârsta minimă necesară pentru a împrumuta o carte

{

\_loan.Borrow(user);

}

else

{

Console.WriteLine("Nu puteți împrumuta această carte deoarece aveți mai puțin de 18 ani.");

}

}

public void Return()

{

\_loan.Return();

}

}

Proxy-ul primește ca parametru un obiect care implementează interfața ILoan și un obiect IUser care reprezintă utilizatorul care dorește să împrumute cartea. În metoda Borrow, proxy-ul verifică dacă utilizatorul are vârsta minimă necesară pentru a împrumuta cartea și, în caz afirmativ, apelează metoda Borrow a obiectului ILoan primit ca parametru. În caz contrar, afișează un mesaj de eroare.

Pentru a utiliza proxy-ul în codul existent, am înlocuit următoarele linii de cod din Program.cs:

ILoan loan = new BookLoan(book);

((ILoan)loan).Borrow(user);

Cu aceasta:

ILoan loan = new BookLoan(book);

ILoan loanProxy = new LoanProxy(loan, user);

loanProxy.Borrow(user);

În acest fel, obiectul loanProxy va acționa ca un proxy pentru obiectul loan și va efectua verificarea de vârstă înainte de a apela metoda Borrow pe acesta.

1. **Façade**

Facade este un pattern structural care ofera o interfata simplificata pentru a accesa un sistem complex de clase, obiecte si sub-sisteme. Acest pattern este utilizat pentru a ascunde complexitatea si detalii de implementare ale unui sistem prin oferirea unei interfete simple si unificate.

In esenta, Facade pattern ofera un punct de intrare unic intr-un sistem complex, prin care utilizatorii pot interactiona cu acesta, fara a fi necesar sa cunoasca toate detaliile interne ale sistemului. Acesta este util in cazul in care exista un sistem complex, cu multiple clase si sub-sisteme, care ar fi dificil de utilizat sau de inteles fara un punct de intrare unificat.

În programul meu, am creat o clasă Facade care oferă o interfață simplificată pentru utilizatorul care dorește să împrumute sau să returneze un obiect de bibliotecă. Această clasă ar trebui să ascundă detalii precum crearea și gestionarea împrumuturilor, precum și detaliile specifice ale obiectelor de bibliotecă.

public class LibraryFacade

{

private readonly Dictionary<int, ILoanable> \_items;

public LibraryFacade()

{

\_items = new Dictionary<int, ILoanable>();

}

public void AddItem(ILoanable item, int inventoryNumber)

{

\_items.Add(inventoryNumber, item);

}

public void BorrowItem(int inventoryNumber, IUser user)

{

if (\_items.TryGetValue(inventoryNumber, out var loanable))

{

loanable.Borrow(user);

if (loanable.Item is Book book)

{

Console.WriteLine($"Book '{book.Title}' a fost împrumutat de către {user.Name} la data de {loanable.BorrowDate}");

}

else

{

Console.WriteLine($"Obiectul '{loanable.Item.ToString()}' a fost împrumutat de către {user.Name} la data de {loanable.BorrowDate}");

}

}

}

public void ReturnItem(int inventoryNumber)

{

if (\_items.TryGetValue(inventoryNumber, out var loanable))

{

loanable.Return();

Console.WriteLine($"{loanable.GetType().Name} '{loanable.Item}' a fost returnat cu succes");

}

else

{

Console.WriteLine($"Nu există niciun obiect cu numărul de inventar {inventoryNumber}");

}

}

}

Această clasă definește trei metode publice: AddItem, BorrowItem și ReturnItem. Metoda AddItem permite adăugarea de obiecte de bibliotecă în dicționarul intern \_items. Acesta dicționar este folosit pentru a identifica obiectul corespunzător în cazul împrumutului sau returnării.

Metoda BorrowItem primește numărul de inventar al obiectului și utilizatorul care dorește să împrumute obiectul. Dacă obiectul există în dicționarul \_items, metoda cheamă metoda Borrow a obiectului respectiv și afișează un mesaj de confirmare. În caz contrar, afișează un mesaj de eroare. Metoda ReturnItem primește numărul de inventar al obiectului și cheamă metoda Return a obiectului corespunzător, dacă există în dicționarul \_items. Altfel, afișează un mesaj de eroare.

Metoda ReturnItem primește numărul de inventar al obiectului și cheamă metoda Return a obiectului corespunzător, dacă există în dicționarul \_items. Altfel, afișează un mesaj de eroare.

**Concluzie:**

În această lucrare de laborator, am explorat în detaliu patru dintre cele mai utilizate șabloane de design structurale: Adapter, Decorator, Proxy și Facade, și le-am implementat cu succes într-un proiect de Calculator. Am observat că fiecare dintre aceste șabloane are un scop specific și poate fi utilizat în diferite situații, în funcție de necesitățile și cerințele proiectului. Adapter-ul ne-a ajutat să adaptăm codul existent la o nouă interfață, fără a schimba codul sursă. Decorator-ul ne-a permis să adăugăm noi funcționalități la obiectele existente, fără a modifica comportamentul lor. Proxy-ul ne-a ajutat să controlăm accesul la resurse și să îmbunătățim performanța prin intermediul cache-ului. Facade-ul ne-a oferit o interfață simplificată către un subsistem mai complex, reducând complexitatea și creșterea modularității.