Ministerul Educației al Republicii Moldova

Universitatea Tehnica a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software si Automatica

**Raport**

Laborator nr. 3

Disciplina: Tehnici si mecanisme de proiectare software.

Tema: „Structural Design Patterns.”

A efectuat: st.gr.TI–202

Cozac Anastasia

A verificat: asist.univ.

Gaidau Mihai

Chişinău 2023

**Obiective:**

1. Studiați și înțelegeți modelele de proiectare structurală.

2. Ca o continuare a lucrărilor anterioare de laborator, gândiți-vă la funcționalitățile pe care sistemul dumneavoastră va trebui să le ofere utilizatorului.

3. Implementați unele funcționalități suplimentare sau creați un nou proiect folosind modele de proiectare structurală.

**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

***Modelul Adapter*** ajută la transformarea interfeței unei clase într-o altă interfață în funcție de cerințele clientului. Deci, practic, oferind ceea ce este cerut de client prin utilizarea serviciului unei clase cu o interfață diferită. Modelul adaptorului este cunoscut și sub numele de Wrapper.

*Cand sa folositi Adapter Pattern:*

1. Când doriți să creați o clasă reutilizabilă care poate coopera cu ușurință cu alte clase, care nu are nicio interfață compatibilă.
2. Când doriți să utilizați un obiect într-un anumit mediu care se așteaptă la o interfață diferită de interfața obiectului său.
3. Când doriți să utilizați o clasă existentă, dar interfața acesteia nu se potrivește cu interfața de care are nevoie clasa.

deci, practic, modelul adaptorului acționează ca o stare intermediară între două clase.

Funcția principală a modelului ***Bridge*** este de a separa partea de implementare de partea de abstractizare, astfel încât abstracția și implementarea unei clase să poată varia independent.

*Există două părți în Bridge Pattern:*

-Implementare sau Corp

-Abstracție sau manipulare

*Când să folosiți Bridge Pattern:*

1. Când atât implementarea, cât și nevoile sale de abstractizare funcțională sunt extinse folosind o altă subclasă.
2. Când nu doriți să afectați clienții din cauza modificărilor de implementare.
3. Când doriți să evitați legarea permanentă între implementare și abstractizarea sa funcțională.

Într-un model ***Composite*,** clientul este capabil să opereze obiecte care pot reprezenta sau nu ierarhia acestuia. În termeni simpli, modelul compozit vă permite să creați un arbore ierarhic cu complexități unice, care permite tratarea fiecărui obiect individual.

Câ*nd să utilizați Composite Pattern:*

1. Atunci când doriți să adăugați noi tipuri de componente unor obiecte specifice
2. Oferă interfețe gestionabile și flexibilitate în structura sa.
3. Ajută la definirea ierarhiilor care conțin obiecte complexe și primitive.

În ***Decorator Pattern***, putem adăuga sau elimina funcționalitatea obiectului fără a modifica funcția sau aspectul exterior al obiectului. Prin urmare, cu ajutorul unui model de decorator, putem adăuga responsabilități suplimentare unui obiect fără a modifica funcționalitățile obiectului.

*Când să folosiți Decorator Pattern:*

1. Îmbunătățește extensibilitatea unui obiect, deoarece modificările sunt făcute prin codificarea unor noi clase.
2. Oferă o flexibilitate mai mare în comparație cu moștenirea statică.
3. Mai degrabă decât codificarea întregului comportament într-un singur obiect Fiecare parte specifică a funcționalității este simplificată prin codificarea unei serii de clase.

În ***Facade Pattern***, oferim o interfață simplificată unui set de interfețe ale unui subsistem pentru a ascunde complexitatea subsistemului de client. Cu cuvinte simple, cu ajutorul modelului de fațadă, facem subsistemele mai ușor de utilizat prin descrierea unei interfețe de nivel superior.

*Când să utilizați Facade Pattern:*

1. Când există multe dependențe între clasele de implementare de abstractizare și cerințele clientului.
2. Când doriți să aveți straturi în subsistemele dvs.
3. Când doriți să furnizați o interfață simplă în loc de un subsistem complex.

***Flyweight Patterns*** ne ajută să reutilizam tipuri similare de obiecte existente prin stocarea și crearea de noi obiecte atunci când obiecte similare nu sunt găsite. Prin urmare, în termeni simpli, ne ajută să reducem costul mai multor instanțe care conțin aceleași date.

*Când să utilizați Flyweight Pattern:*

1. Pentru a reduce costurile de depozitare, prin reutilizarea obiectelor similare.
2. Când este necesar un număr mare de obiecte similare.
3. Când aplicația depinde de identitatea unică a obiectului.

În ***Proxy Pattern*** oferim un substituent sau un obiect surogat pentru a controla accesul la obiectul original. În termeni simpli, furnizarea unui proxy sau un strat inactiv de informații înaintea clientului înainte de a accesa datele originale.

*Când să utilizați ce model de proxy:*

1. Proxy-ul de la distanță poate fi utilizat pentru a face resursele interfeței disponibile de la distanță, cum ar fi un serviciu web.
2. Virtual Proxy poate fi utilizat atunci când sunt efectuate mai multe apeluri la baza de date pentru extragerea unui fișier uriaș. Deoarece este un proces costisitor, putem folosi un proxy virtual pentru a reduce costurile.
3. Protective Proxy poate fi utilizat atunci când este necesară securitatea sau confidențialitatea datelor. Cu ajutorul unui model proxy, putem verifica dacă utilizatorul are sau nu acces la date.
4. Smart Proxy poate fi folosit atunci când avem nevoie de un strat suplimentar de securitate pentru un anumit obiect.

In acest proiect am utilizat trei design pattern-uri structurale: Adapter, Proxy și Composite.

**Decorator Pattern:**

// Base vehicle interface

public interface IVehicle

{

void Rent(string customer);

}

// Concrete vehicle class

public class Car : IVehicle

{

public void Rent(string customer)

{

Console.WriteLine($"Car rented to {customer}");

}

}

// Decorator interface

public interface IVehicleDecorator : IVehicle

{

}

// Concrete decorator classes

public class GpsDecorator : IVehicleDecorator

{

private readonly IVehicle \_vehicle;

public GpsDecorator(IVehicle vehicle)

{

\_vehicle = vehicle;

}

public void Rent(string customer)

{

\_vehicle.Rent(customer);

Console.WriteLine("GPS added to the car");

}

}

public class MusicDecorator : IVehicleDecorator

{

private readonly IVehicle \_vehicle;

public MusicDecorator(IVehicle vehicle)

{

\_vehicle = vehicle;

}

public void Rent(string customer)

{

\_vehicle.Rent(customer);

Console.WriteLine("Music player added to the car");

}

}

// Client code

static void Main(string[] args)

{

// Rent a basic car

IVehicle car = new Car();

car.Rent("John");

// Rent a car with GPS

IVehicle carWithGps = new GpsDecorator(new Car());

carWithGps.Rent("Alice");

// Rent a car with GPS and music player

IVehicle carWithAccessories = new MusicDecorator(new GpsDecorator(new Car()));

carWithAccessories.Rent("Bob");

**Adapter Pattern:**

// Third-party car rental service interface

public interface IThirdPartyRentalService

{

void RentCar(string customerName, string carModel);

}

// Adapter class

public class ThirdPartyRentalAdapter : IVehicle

{

private readonly IThirdPartyRentalService \_service;

public ThirdPartyRentalAdapter(IThirdPartyRentalService service)

{

\_service = service;

}

public void Rent(string customer)

{

// Map our interface to the third-party service interface

\_service.RentCar(customer, "basic car");

}

}

// Client code

static void Main(string[] args)

{

// Rent

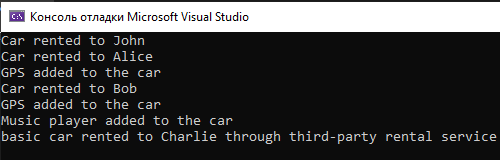


Figura1 – Rezultatul executiei a acestui proiect

**Concluzie:**

În concluzie, utilizarea acestor design pattern-uri îmbunătățește modularitatea și flexibilitatea proiectului, permițând o mai bună separare a responsabilităților între clase și reducând acoplarea între ele. De asemenea, permite adaptarea mai ușoară a proiectului la schimbările viitoare și îmbunătățește performanța și eficiența utilizării resurselor de sistem.