Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Ingineria Software și Automatică

RAPORT

Lucrare de laborator nr.3

Disciplina: Tehnici și Mecanisme de Proiectare Software.

Tema: Structural Design Patterns.

A efectuat: Pîntea Adina, gr. TI-202

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.

**Noțiuni teoretice:**

Modelele de proiectare structurală sunt o categorie de modele de proiectare care se concentrează pe compoziția claselor și a obiectelor pentru a forma structuri și sisteme mai mari. Ele oferă o modalitate de a organiza obiectele și clasele într-un mod flexibil și eficient, permițând în același timp reutilizarea și modificarea codului existent. Modelele de proiectare structurală abordează probleme comune întâlnite în compoziția claselor și a obiectelor, cum ar fi modul de a crea noi obiecte care moștenesc funcționalitatea de la obiectele existente, cum să creeze obiecte care partajează funcționalitatea fără a duplica codul sau cum să definești relațiile dintre obiecte într-un mod flexibil. și modul extensibil.

Câteva exemple din această categorie de modele de design sunt:

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

Noțiunile Șabloanelor Structurale:

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. De exemplu, în cazul în care o aplicație are nevoie să utilizeze o bibliotecă externă care oferă o interfață diferită de cea a aplicației, se poate utiliza un adaptor pentru a face ca cele două interfețe să fie compatibile.
2. Bridge: Acest șablon separă abstractizarea de implementare, permițându-le să evolueze separat și reducând dependența dintre acestea. Un exemplu ar fi separarea interfeței de utilizator de implementarea acesteia. Astfel, dacă se dorește schimbarea implementării interfeței, nu va fi necesară modificarea interfeței de utilizator.
3. Composite: Acest șablon permite crearea de structuri complexe prin compunerea obiectelor simple în structuri ierarhice. De exemplu, o structură ierarhică de obiecte grafice poate fi creată prin compunerea diferitelor forme (dreptunghi, cerc etc.) într-un obiect mai mare, cum ar fi o imagine.
4. Decorator: Acest șablon permite adăugarea de comportamente suplimentare unui obiect existent prin încapsularea acestuia într-un decorator care oferă o interfață suplimentară. De exemplu, în cazul în care se dorește adăugarea de o funcționalitate suplimentară pentru o clasă existentă, se poate utiliza un decorator pentru a adăuga această funcționalitate fără a modifica clasa originală.
5. Facade: Acest șablon oferă o interfață simplificată pentru o parte complexă a sistemului, astfel încât utilizatorii să nu fie nevoiți să interacționeze direct cu toate elementele componente ale acesteia. De exemplu, în cazul unei aplicații cu o structură complexă, se poate utiliza un fațadă pentru a oferi o interfață simplificată utilizatorilor.
6. Flyweight: Acest șablon permite utilizarea eficientă a obiectelor prin partajarea lor între mai multe instanțe. De exemplu, în cazul unui editor grafic, se pot partaja obiecte cum ar fi culorile sau fonturile între mai multe instanțe pentru a reduce utilizarea resurselor.
7. Proxy: Acest șablon permite controlul accesului la un obiect prin intermediul unui obiect proxy care acționează ca un înlocuitor al obiectului original. De exemplu, în cazul în care un obiect costisitor în resurse trebuie accesat din mai multe locații, se poate utiliza un proxy pentru a controla accesul la obiect și pentru a reduce utilizarea resurselor.

**Sarcina practică:**

La implementarea Design Patterns Structural, am ales creare unei aplicații cu un funcțional simplu în limbajul C#, deoarece acesta are următoarele avantaje:

1. OOP: C# este un limbaj orientat pe obiecte, ceea ce face ca implementarea șabloanelor pattern structurale să fie ușoară și intuitivă.
2. Biblioteci standard: C# are o serie de biblioteci standard, cum ar fi System.Collections, care oferă implementări de referință pentru multe șabloane pattern structurale comune. Acestea pot fi utilizate în implementarea modelelor de design, reducând timpul și efortul necesar dezvoltării acestora.
3. Interoperabilitatea cu alte limbaje: C# poate fi utilizat într-o varietate de scenarii, de la aplicații desktop la aplicații web și chiar la dezvoltarea de aplicații pentru dispozitive mobile. În plus, C# poate fi integrat cu alte limbaje, cum ar fi C++, prin intermediul platformei .NET, ceea ce face posibilă dezvoltarea de aplicații complexe care implică mai multe limbaje și tehnologii.

Primul pas în implementarea șabloanelor Structurale, a fost de a crea un nou proiect.

Prin urmare:

1. Am creat un nou proiect de tipul Console App, în C#;
2. L-am denumit ca ”Calculator”, ceea ce presupune și funcționalitatea proiectului;
3. Am selectat 4 șabloane la dorință: Adapter, Decorator, Proxy și Facade;
4. Am structurat proiectul pe fișiere .cs pentru a fi mai înțeles și mai ușor de gestionat.
5. Pentru început am creat fișierul Calculator.cs, și am definit o clasă cu următoarea interfață publică:

using System;

namespace Calculator

{

public class Calculator

{

public double Add(double num1, double num2)

{

return num1 + num2;

}

public double Subtract(double num1, double num2)

{

return num1 - num2;

}

public double Multiply(double num1, double num2)

{

return num1 \* num2;

}

public double Divide(double num1, double num2)

{

if (num2 == 0)

{

throw new DivideByZeroException();

}

return num1 / num2;

}

}

}

Acest fișier Calculator.cs conține o clasă Calculator, care definește operațiile aritmetice de bază (adunare, scădere, înmulțire și împărțire) utilizate într-un calculator.

Clasa Calculator este considerată clasa de bază deoarece definește funcționalitatea fundamentală pe care o așteptăm de la un calculator. Scopul clasei Calculator este de a oferi o implementare simplă și reutilizabilă a funcțiilor de bază ale unui calculator.

1. Următorul fișier .cs creat, este CalculatorAdapter.cs.

Design pattern-ul "Adapter" ne va ajuta să adaptăm funcționalitatea existentă a clasei "Calculator" într-o altă interfață, astfel încât aceasta să poată fi utilizată într-un context diferit.

În interiorul fișierului, am definit o interfață ICalculator:

public interface ICalculator

{

double Calculate(string expression);

}

Am deifit o clasă ”CalculatorAdapter” care implementează interfața ”ICalculator” și utilizează clasa ”Calculator” pentru a realiza operațiile de calcul:

public class CalculatorAdapter : ICalculator

{

private readonly Calculator \_calculator = new Calculator();

public double Calculate(string expression)

{

// Verificăm dacă expresia este într-un format corect

var regex = new Regex(@"^\d+(\.\d+)?\s\*[-+\*/]\s\*\d+(\.\d+)?$");

if (!regex.IsMatch(expression))

{

throw new ArgumentException("Invalid input format");

}

// Parsăm expresia și calculăm rezultatul

var operands = expression.Split(' ');

var operand1 = double.Parse(operands[0]);

var operand2 = double.Parse(operands[2]);

var op = operands[1];

double result;

switch (op)

{

case "+":

result = operand1 + operand2;

break;

case "-":

result = operand1 - operand2;

break;

case "\*":

result = operand1 \* operand2;

break;

case "/":

result = operand1 / operand2;

break;

default:

throw new ArgumentException("Invalid operator");

}

return result;

}

}

1. Design pattern-ul ”Decorator”.

Am creat un fișier ”CalculatorDecorator.cs”.

Fișierul CalculatorDecorator.cs are o importanță mare în implementarea pattern-ului decorator. Funcționalitatea sa este aceea de a permite adăugarea de funcționalități noi pentru calcularea unor expresii matematice, fără a afecta codul deja existent în ceea ce privește calcularea expresiilor matematice simple.

În interiorul fișierului, am definit o clasă abstractă ”CalculatorDecorator”, care implementează interfața ”ICalculator”:

// Clasa abstracta CalculatorDecorator care implementeaza interfata ICalculator

public abstract class CalculatorDecorator : ICalculator

{

private readonly ICalculator \_calculator;

//Constructorul pentru clasa abstracta CalculatorDecorator care primeste un obiect de tip ICalculator

protected CalculatorDecorator(ICalculator calculator)

{

\_calculator = calculator;

}

// Metoda virtuala Calculate care va fi suprascrisa in clasele derivate

public virtual double Calculate(string expression)

{

return \_calculator.Calculate(expression);

}

}

În continuare, am definit o clasă ”LoggingCalculatorDecorator” care extinde clasa ”CalculatorDecorator” și adaugă funcționalitatea de înregistrare a tututot expresiilor matematice și rezultatelor corespunzătoare într-un fișier de jurnal:

//clasa LoggingCalculatorDecorator care mosteneste clasa abstracta CalculatorDecorator

public class LoggingCalculatorDecorator : CalculatorDecorator

{

private readonly StreamWriter \_logFile;

public LoggingCalculatorDecorator(ICalculator calculator, string logFileName) : base(calculator)

{

\_logFile = new StreamWriter(logFileName, append: true);

}

//Metoda Calculate pentru a inregistra operatiile efectuate in fisierul de log

public override double Calculate(string expression)

{

var result = base.Calculate(expression);

\_logFile.WriteLine($"{expression} = {result}");

return result;

}

//Închiderea fișierului log

public void CloseLogFile()

{

\_logFile.Close();

}

}

1. Design pattern-ul ”Proxy”

Design pattern-ul "Proxy" ne va ajuta să controlăm accesul la resursele costisitoare sau restricționate (în cazul nostru, fișierul de jurnal creat de decoratorul "LoggingCalculatorDecorator").

Am creat fișierul ”CalculatorProxy.cs”. În interiorul fișierului, am definit o clasă ”CalculatorProxy” care implementează interfața ”ICalculator”:

public class CalculatorProxy : ICalculator

{

private readonly ICalculator \_calculator; // obiectul calculatorului original care va fi proxy

private readonly LoggingCalculatorDecorator \_loggingDecorator; // decoratorul pentru înregistrarea în fișierul de log

// Constructorul inițializează obiectul calculatorului original și obiectul decoratorului de înregistrare

public CalculatorProxy(ICalculator calculator)

{

\_calculator = calculator;

\_loggingDecorator = new LoggingCalculatorDecorator(calculator,"calculation.log");

}

// Această metodă calculează expresia, înregistrează operația în fișierul de log și returnează rezultatul

public double Calculate(string expression)

{

var result = \_calculator.Calculate(expression);

\_loggingDecorator.CloseLogFile();

return result;

}

}

1. Design Pattern ”Facade”

Acest pattern, ne ajută să ascundem complexitatea sistemului, și oferă o interfață simplă către clienți.

Am creat un fișier ”CalculatorFacade.cs”. În acest fișier am definit o clasă CalculatorFacade, care expune o interfață simplă de Calculator către utilizatori, iar aceasta utilizează serviciile de calculator existente pentru efectuarea calculelor.

public class CalculatorFacade

{

//Instanțele claselor Calculator și CalculatorAdapter

private readonly Calculator \_calculator = new Calculator();

private readonly CalculatorAdapter \_calculatorAdapter = new CalculatorAdapter();

public double Add(double num1, double num2)

{

return \_calculator.Add(num1, num2);

}

public double Subtract(double num1, double num2)

{

return \_calculator.Subtract(num1, num2);

}

public double Multiply(double num1, double num2)

{

return \_calculator.Multiply(num1, num2);

}

public double Divide(double num1, double num2)

{

return \_calculator.Divide(num1, num2);

}

public double Calculate(string expression)

{

return \_calculatorAdapter.Calculate(expression);

}

}

1. Cel mai principal funcțional în orice proiect o are fișierul ”Program.cs”.

Acesta este primul fișier care este executat atunci când aplicația este pornită și este responsabil pentru inițializarea aplicației și apelarea altor funcții și metode din celelalte fișiere ale proiectului.

În acest fișier, este creat obiectul Calculator și sunt apelate metodele din acesta. De asemenea, este creat un obiect CalculatorAdapter, care permite integrarea clasei Calculator cu clasa CalculatorLogger prin intermediul pattern-ului Adapter.

În final, fișierul Program.cs conține afișarea rezultatului final al operațiilor aritmetice pe ecran, prin intermediul consolei.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Calculator Adapter");

Console.WriteLine("-------------------");

Console.WriteLine("Enter an expression (e.g. 2 + 3):");

var input = Console.ReadLine();

Console.WriteLine($"Input: {input}");

var adapter = new CalculatorAdapter();

try

{

var result = adapter.Calculate(input);

Console.WriteLine($"Result: {result}");

}

catch (Exception ex)

{

Console.WriteLine($"Error: {ex.Message}. Expression: {input}");

}

Console.WriteLine("-------------------");

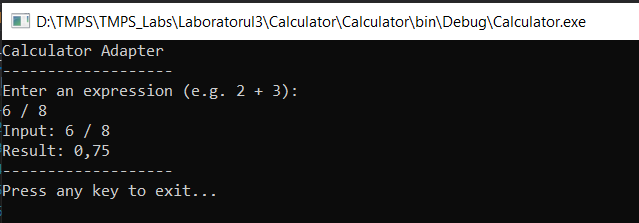
Console.WriteLine("Press any key to exit...");

Console.ReadKey();

}

}

**Rezultatul rulării proiectului:**



**Concluzie:**

La această lucrare de laborator, am avut ocazia să înțeleg în detaliu patru dintre cele mai cunoscute șabloane de design structurale: Adapter, Decorator, Proxy și Facade, pe care le-am implementat cu succes într-un proiect, care are o funcționalitate de Calculator.

De asemenea, am observat că fiecare dintre aceste șabloane are un rol specific și poate fi utilizat în diverse situații, în funcție de nevoile și cerințele proiectului.

* Adapter-ul ne-a ajutat să adaptăm codul existent la o nouă interfață;
* Decorator-ul ne-a permis să extindem comportamentul obiectelor existente;
* Proxy-ul ne-a ajutat să controlăm accesul la resurse;
* Facade-ul ne-a oferit o interfață simplificată către un subsistem mai complex.

În concluzie, implementarea acestor șabloane structurale în proiectul de Calculator mi-a permis să cresc flexibilitatea și extensibilitatea codului, precum și să îmbunătățesc înțelegerea și organizarea arhitecturii software-ului.