Ministerul Educației al Republicii Moldova

Universitatea Tehnica a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software si Automatica

**Raport**

Laborator nr. 2

Disciplina: Tehnici si mecanisme de proiectare software.

Tema: „Design Patterns.”

A efectuat: st.gr.TI–202

Rusu Felicia

A verificat: asist.univ.

Gaidau Mihai

Chişinău 2023

### \*\*Tiparele de proiectare\*\*

(sau "design patterns" în engleză) sunt soluții comune și testate în practică pentru problemele care apar în procesul de dezvoltare software. Acestea oferă o abordare clară și structurată pentru proiectarea și implementarea software-ului. Tiparele de proiectare sunt împărțite în trei categorii principale: creationale, structurale și comportamentale Tiparele creationale sunt utilizate pentru crearea obiectelor, tiparele structurale pentru compunerea obiectelor, iar tiparele comportamentale pentru interacțiunea între obiecte.

Exemple de \*\*tipare de proiectare creationale\*\* includ \*Singleton, Factory Method, Abstract Factory, Builder, Prototype și Object Pool\*. Aceste tipare oferă o modalitate clară de a crea și gestiona obiecte într-un mod structurat și eficient, îmbunătățind astfel performanța și extensibilitatea aplicației.

#### \*\*Singleton\*\*:

Este unul dintre cele mai simple modele de design. Înseamnă că o singură instanță poate fi instanțiată dintr-o clasă dată.

Se asigură că o clasă are o singură instanță și oferă un punct de acces global la această instanță particulară. Modelul Singleton ajută la salvarea memoriei ,

deoarece obiectul nu este creat la fiecare cerere. Fie creează un obiect nou, fie returnează un obiect existent care a fost deja creat.

#### \*\*Builder\*\*:

Separă construcția obiectului complex de reprezentarea sa. Acesta ascunde structura internă de utilizatorul final și oferă o interfață ușoară cu lumea exterioară.

Modelul Builder se concentrează pe construirea progresivă a unui produs complex . Este conceput ca o compoziție a unuia sau mai multor obiecte similare cu

funcționalitate similară. Ne permite să producem diferite tipuri de obiect folosind același cod.

#### \*\*Prototype\*\*:

Creează o instanță complet inițializată care poate fi clonată. Ne permite să copiem obiecte deja existente fără a face codul dependent de alte clase.

Un model de prototip construiește un produs prin copierea stării inițiale a unui obiect prototip.

#### \*\*Object Pooling\*\*:

Ajută la reutilizarea obiectelor care sunt costisitoare de creat prin reciclarea obiectelor care nu mai sunt utilizate.

Este cel mai eficient în situațiile în care rata și costul instanțierii unei clase sunt ridicate. Un caz comun de utilizare a modelului Object Pool este atunci

când există mai mulți clienți care au nevoie de aceeași resursă în momente diferite. De exemplu, dacă trebuie să deschidem multe conexiuni la o bază de date,

este nevoie de mai mult timp pentru a crea o instanță nouă și serverul bazei de date va fi supraîncărcat.

#### \*\*Factory Method\*\*:

Creează o instanță a mai multor clase derivate aparținând unei singure familii de obiecte . Ajută la crearea de obiecte într-o superclasă, dar permite și subclaselor

să modifice tipul de obiecte care sunt create. Putem ascunde existența claselor de utilizatorul final folosind modelul Factory Method.

#### \*\*Abstract Factory\*\*:

Creează un obiect din mai multe familii de clase . Ne permite să producem familii de obiecte înrudite fără a specifica clasele lor concrete.

\*\*\*In aceasta lucrare de laborator am creat un proiect in care am utilizat Singleton, Builder si Abstract Factory.\*\*\*

Fisierul principal Program.cs conține metoda Main, care este punctul de intrare în aplicație. Acesta demonstrează utilizarea modelului de design Singleton,

Builder și Abstract Factory prin crearea și utilizarea obiectelor din clasele School, Student și Teacher.

Fisierul Program.cs:

using System;

namespace DPatterns

{

class Program

{

static void Main(string[] args)

{

// Create a singleton instance of the school

School school = School.Instance;

// Construct a student using the builder pattern

StudentBuilder builder = new StudentBuilder();

Student student = builder.SetId(1)

.SetName("John Doe")

.SetAge(18)

.SetAddress("123 Main St.")

.Build();

// Create student and teacher objects using the abstract factory pattern

PersonFactory factory = new SchoolPersonFactory();

Student student2 = factory.CreateStudent();

Teacher teacher = factory.CreateTeacher();

// Display the objects

Console.WriteLine("School name: " + school.Name);

Console.WriteLine("Student name: " + student.Name);

Console.WriteLine("Student 2 name: " + student2.Name);

Console.WriteLine("Teacher name: " + teacher.Name);

}

}

}

Clasa School.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DPatterns

{

public class School

{

private static School instance;

public string Name { get; set; }

private School()

{

Name = "My School";

}

public static School Instance

{

get

{

if (instance == null)

{

instance = new School();

}

return instance;

}

}

// Singleton pattern

}

}

Clasa Student.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DPatterns

{

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public string Address { get; set; }

public Student(int id, string name, int age, string address)

{

Id = id;

Name = name;

Age = age;

Address = address;

}

// Builder pattern

}

}

Clasa Teacher.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DPatterns

{

public class Teacher

{

public string Name { get; set; }

public Teacher()

{

Name = "Jane Smith";

}

// abstract factory pattern

}

}

Singleton: Clasa School este accesată folosind proprietatea School.Instance, care returnează instanța singleton a clasei. Proprietatea Instance este implementată

folosind modelul de design Singleton pentru a asigura că se creează doar o singură instanță a clasei pe parcursul duratei de viață a aplicației.

Builder: Clasa StudentBuilder este utilizată pentru a construi un obiect Student. Aceasta oferă o abordare pas cu pas pentru construirea unui obiect complex prin

setarea proprietăților sale. StudentBuilder este responsabil pentru inițializarea proprietăților obiectului Student și returnarea obiectului construit.

Clasa StudentBuilder.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DPatterns

{

public class StudentBuilder

{

private int id;

private string name;

private int age;

private string address;

public StudentBuilder SetId(int id)

{

this.id = id;

return this;

}

public StudentBuilder SetName(string name)

{

this.name = name;

return this;

}

public StudentBuilder SetAge(int age)

{

this.age = age;

return this;

}

public StudentBuilder SetAddress(string address)

{

this.address = address;

return this;

}

public Student Build()

{

return new Student(id, name, age, address);

}

}

// builder pattern

}

Abstract Factory: este utilizat în clasele Student și Teacher. Acest pattern este utilizat atunci când dorim să creăm o familie de obiecte care trebuie să fie

compatibile între ele, astfel încât să putem utiliza aceleași metode sau proprietăți în toate obiectele create. În acest exemplu, clasa abstractă PersonFactory

definește metodele pentru crearea obiectelor Student și Teacher, în timp ce clasa SchoolPersonFactory oferă implementarea concretă pentru aceste metode, creând

obiecte Student și Teacher specifice unei școli. Acest pattern ne permite să schimbăm comportamentul de creare a obiectelor fără a afecta codul client.

Clasa SchoolPersonFactory.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DPatterns

{

public class SchoolPersonFactory : PersonFactory

{

public override Student CreateStudent()

{

return new Student(2, "Mary Johnson", 17, "456 Oak St.");

}

public override Teacher CreateTeacher()

{

return new Teacher();

}

}

// abstract factory pattern

}

Clasa PersonFactory.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DPatterns

{

public abstract class PersonFactory

{

public abstract Student CreateStudent();

public abstract Teacher CreateTeacher();

}

// abstract factory pattern

}

**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.