ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КПІ імені Ігоря Сікорського”

Кафедра програмного забезпечення комп’ютерних систем

**ЛАБОРАТОРНА РОБОТА №5**

з дисципліни

Програмування

C# .Net. Реалізація основних

принципів ООП мовою C#

Студента 2 курсу ФПМ

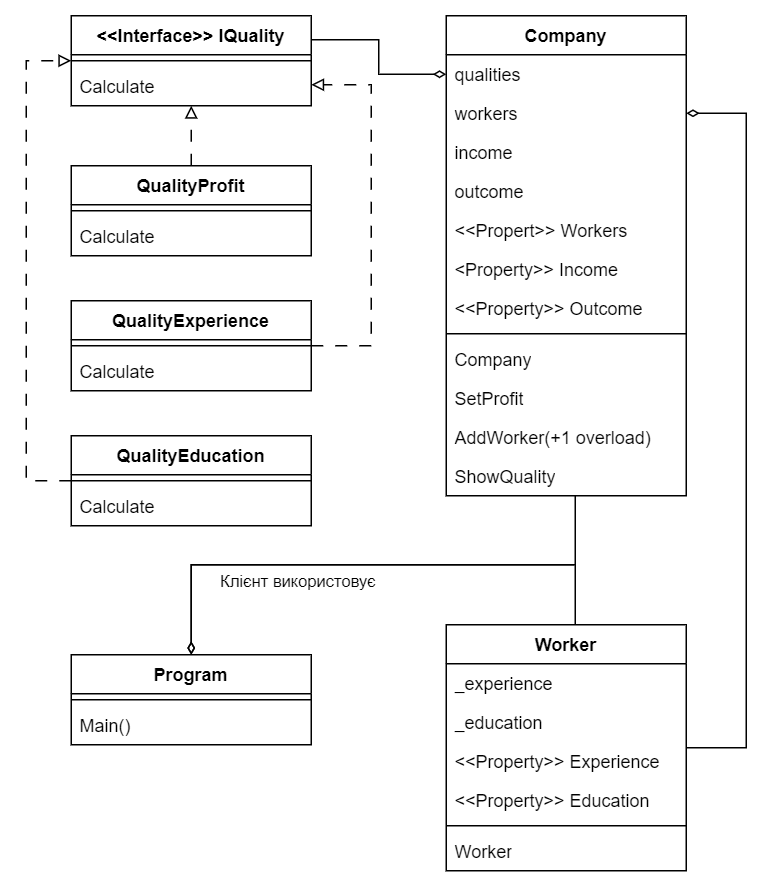
гр. КП-23 Верьовочкіна М. С.

Київ – 2024

**Варіант №6**

1. У компанії практикуються різні методи розрахунку показників якості роботи. Наприклад, розрахунок показників на основі врахування відношення прибутку до витрат; або врахування якості підготовки співробітників (досвід роботи, освіта тощо). За допомогою шаблону проєктування реалізувати можливість визначати якість роботи компанії різними способами.
2. Під час надходження замовлення на кухню ресторану його послідовно обробляють декілька кухарів. Наприклад, один кухар бере на себе відповідальність за приготування салату та передає замовлення наступному кухарю, який може відповідати за приготування м’ясної страви тощо. Таким чином, замовлення обходить всіх кухарів, які з нього вибирають ту частину блюд, яка лежить у зоні їх відповідальності. За допомогою шаблону проєктування реалізувати механізм обробки замовлення кухарями ресторану.
   1. Оскільки в сама задача – буквально імплементувати різні методи розрахунку якості роботи для спільної загальної системи, причому ці методи можуть відрізнятися значно, не в маленьких кроках (елементарних операціях), то доцільно використати для розв’язання шаблон «стратегія».

Представимо UML-діаграму:



Текст програми:

class Program

{

static void Main()

{

Console.WriteLine("Enter the count of workers");

int n = int.Parse(Console.ReadLine());

List<Worker> workers = new List<Worker>();

Random R = new Random();

for(int i = 0; i < n; i++)

Company.AddWorker(new Worker(R.NextDouble() \* 4000, (R.NextDouble() + 1) \* 6 ));

for (int i = 0; i < 3; i++)

Company.ShowQuality();

}

}

public interface IQuality

{

void Calculate();

}

public class QualityProfit : IQuality

{

public void Calculate()

{

Company.SetProfit();

Console.WriteLine("Quality by the paying off the company's outcome is {0:0.00}%", Company.Income / Company.Outcome \* 100);

}

}

public class QualityExperience : IQuality

{

public void Calculate()

{

Console.WriteLine("Quality by the middle work experiance level of the workers in the company is {0:0.00}",

Company.Workers.Sum(x => x.Experience) / Company.Workers.Count);

}

}

public class QualityEducation : IQuality

{

public void Calculate()

{

Console.WriteLine("Quality by the middle education level of the workers in the company is {0:0.00}",

Company.Workers.Sum(x => x.Education) / Company.Workers.Count);

}

}

public static class Company

{

private static Dictionary<string, IQuality> qualities = new Dictionary<string, IQuality>();

private static List<Worker> workers = new List<Worker>();

private static double income = 1;

private static double outcome = 1;

public static List<Worker> Workers => workers;

public static double Income => income;

public static double Outcome => outcome;

static Company()

{

qualities.Add("Quality by the paying off the company's outcome", new QualityProfit());

qualities.Add("Quality by the middle work experiance level of the workers in the company", new QualityExperience());

qualities.Add("Quality by the middle education level of the workers in the company", new QualityEducation());

}

public static void SetProfit()

{

double income = -1;

do

{

Console.Write("Enter the company income: ");

try { income = double.Parse(Console.ReadLine()); }

catch (FormatException) { Console.WriteLine("Wrong format..."); }

} while (income < 0);

double outcome = -1;

do

{

Console.Write("Enter the company outcome: ");

try { outcome = double.Parse(Console.ReadLine()); }

catch (FormatException) { Console.WriteLine("Wrong format..."); }

} while (outcome < 0);

Company.income = income;

Company.outcome = outcome;

}

public static void AddWorker(Worker worker) => workers.Add(worker);

public static void AddWorker(List<Worker> workers) => Company.workers = workers;

public static void ShowQuality()

{

int quality = 0;

Console.WriteLine("Calculate the company quality by:");

for (int i = 0; i < qualities.Count; i++)

Console.WriteLine("{0}) {1}", i + 1, qualities.Keys.ElementAt(i));

do

{

try { quality = int.Parse(Console.ReadLine()); }

catch (FormatException) { Console.WriteLine("Wrong format..."); }

} while (quality < 1 || quality > qualities.Count);

try

{ qualities[qualities.Keys.ElementAt(quality - 1)].Calculate(); }

catch (NullReferenceException) { Console.WriteLine("Workers list is null"); }

catch (Exception) { Console.WriteLine("Data error"); }

}

}

public class Worker

{

private double \_experience;

private double \_education;

public double Experience => \_experience;

public double Education => \_education;

public Worker(double experience, double education)

{

\_experience = experience;

\_education = education;

}

}

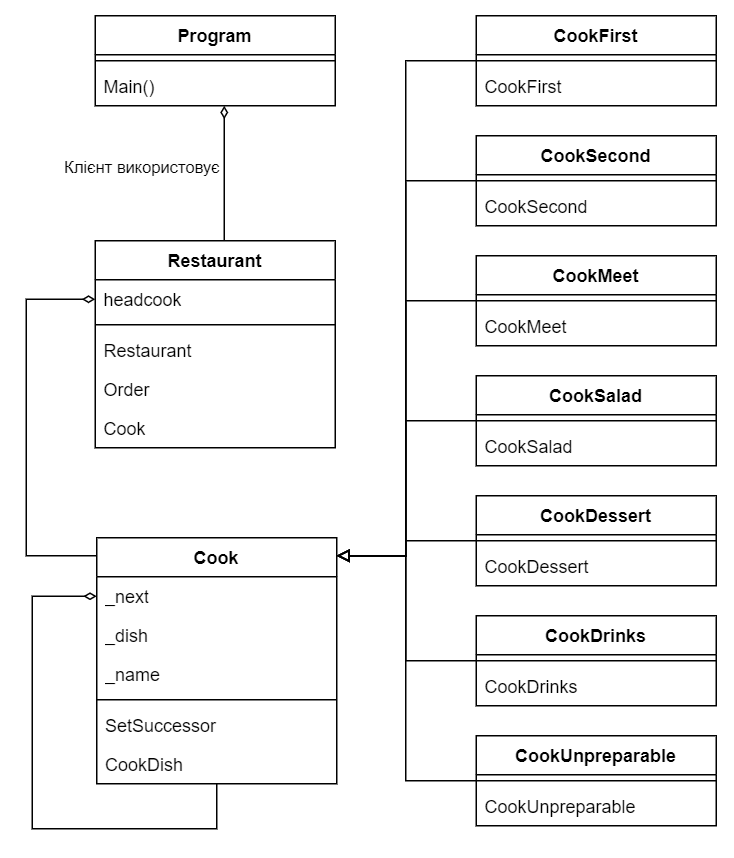
Приклад виконання розробленого коду:

A screenshot of a computer

Description automatically generated

* 1. Маємо послідовність однотипових виконавців, що роблять різні дії над отримним завданням відповідно до своїх обов’язків. Для цієї задачі оптимальним шаблоном є «ланцюжок обов’язків».

Преставимо UML-діаграму структури класів:



Текст програми:

class Program

{

static void Main()

{

Restaurant.Order();

}

}

public static class Restaurant

{

private static Cook headcook = null;

static Restaurant()

{

Cook cook1 = new CookFirst("Jack");

Cook cook2 = new CookSecond("Nick");

Cook cook3 = new CookMeet("Emily");

Cook cook4 = new CookSalad("Max");

Cook cook5 = new CookDessert("Liam");

Cook cook6 = new CookDrinks("Kate");

Cook no\_cook = new CookUnpreparable();

cook1.SetSuccessor(cook2);

cook2.SetSuccessor(cook3);

cook3.SetSuccessor(cook4);

cook4.SetSuccessor(cook5);

cook5.SetSuccessor(cook6);

cook6.SetSuccessor(no\_cook);

headcook = cook1;

}

public static void Order()

{

List<string> order = new List<string>();

Console.WriteLine("Menu:\n - first dish\n - second dish\n - meet\n - salad\n - dessert\n - drinks");

do

{

Console.Write("Choose the meals you want to order (separated by comma): ");

try { order = Console.ReadLine().Split(',').ToList(); }

catch (FormatException) { Console.WriteLine("Wrong format..."); }

} while (order == null);

for (int i = 0; i < order.Count; i++) order[i] = order[i].StartsWith(' ') ? order[i].Substring(1) : order[i];

Cook(order);

}

private static void Cook(List<string> dish)

{

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

List<string> order = headcook.CookDish(dish);

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

Console.WriteLine("Your order is prepared: ");

foreach (string meal in order)

Console.WriteLine(" - {0}", meal);

}

}

public abstract class Cook

{

protected Cook \_next = null;

protected string \_dish;

protected string \_name;

public void SetSuccessor(Cook successor) => \_next = successor;

public virtual List<string> CookDish(List<string> order)

{

List<string> dish = new List<string>();

if (order.Contains(\_dish))

{

Console.WriteLine("Cook {0} is preparing the {1}", \_name, \_dish);

for (int i = 0; i < order.FindAll(x => x == \_dish).Count; i++)

dish.Add(\_dish);

order.RemoveAll(x => x == \_dish);

}

if (order.Count > 0)

foreach (string meal in \_next.CookDish(order))

dish.Add(meal);

return dish;

}

}

public class CookFirst : Cook

{

public CookFirst(string name)

{

\_dish = "first dish";

\_name = name;

}

}

public class CookSecond : Cook

{

public CookSecond(string name)

{

\_dish = "second dish";

\_name = name;

}

}

public class CookMeet : Cook

{

public CookMeet(string name)

{

\_dish = "meet";

\_name = name;

}

}

public class CookSalad : Cook

{

public CookSalad(string name)

{

\_dish = "salad";

\_name = name;

}

}

public class CookDessert : Cook

{

public CookDessert(string name)

{

\_dish = "dessert";

\_name = name;

}

}

public class CookDrinks : Cook

{

public CookDrinks(string name)

{

\_dish = "drinks";

\_name = name;

}

}

public class CookUnpreparable : Cook

{

public override List<string> CookDish(List<string> order)

{

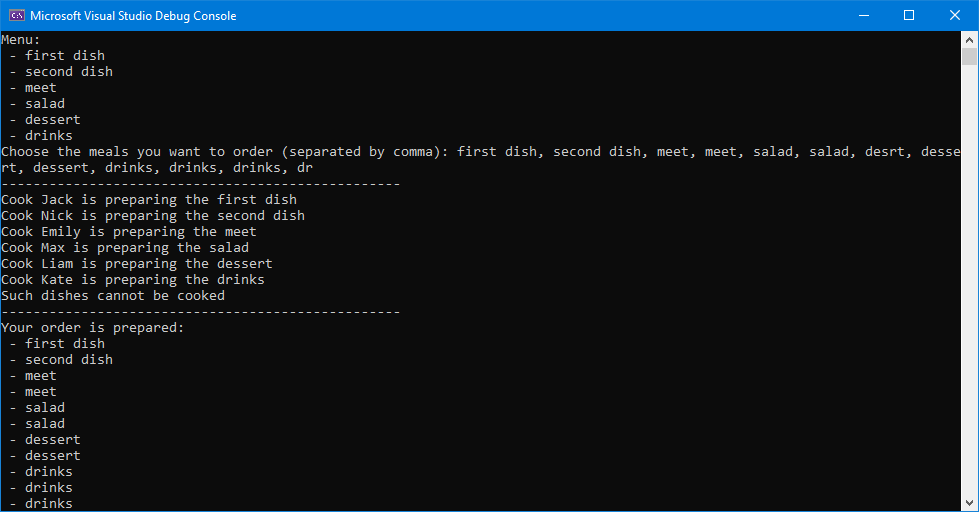
Console.WriteLine("Such dishes cannot be cooked");

return new List<string>();

}

}

Приклад виконання розробленого коду:



Як бачимо, програма коректно «приготувала» їжу відповідно до запиту користувача та вивела інформацію на екран.

**Висновки:** в ході виконання лабораторної роботи було опановано роботу з такими особливостями мови C#, як поведінкові шаблони, вивчено відмінності між ними та способи їх доцільного підбору. Були продемонстровані різні види UML-діаграм для обраних шаблонів. Безпосередньо в роботі були використані шаблони «стрітегія» та «ланцюжок обов’язків».