ІНДЗ-6

Система оцінювання

№	Тема	К-ть балів
1.	ІНДЗ-1	1
2.	ІНДЗ-2	1
3.	ІНДЗ-3	1,5
4.	ІНДЗ-4	1,5
	Всього	**5**

Основою об’єктно-орієнтованого програмування, на якій побудовано базові принципи ООП, є *вказівник this* та *динамічна диспетчеризація (dynamic dispatch)*. Вказівник this неявно передається при кожному виклику функції рівня екземпляру (instance-level function), тобто кожна функція вже має об’єкт, над яким зможе виконувати операції. Такий підхід здійснює перехід до коду без глобальних функцій, що, як наслідок, формує операції як частину структур даних.

Динамічна диспетчеризація покладається на наявність у кожного об’єкта метаданих щодо свого типу. Цей тип слідкує за таблицею своїх віртуальних функцій, що дає змогу заміщати реалізацію в успадкованих типах. При виклику функції, насправді, невідомо, який код буде виконуватись. Середовище виконання звернеться до об’єкта, який викликає функцію, звернеться до таблиці віртуальних функцій та знайде адресу реалізації функції для виклику. Таким чином, адреси функцій для виклику визначаються *динамічно*, а не статично, що було характерним для глобальних функцій. Це означає, що існує можливість писати об’єктно-орієнтований код не лише в мовах структурованого програмування, на зразок C, а й навіть у вигляді макросів ассемблера. Тобто об’єктна-орієнтованість коду не завжди потребує об’єктно-орієнтованої мови програмування.

У результаті можемо зазначити, що об’єктно-орієнтоване мислення будується на двох основних питаннях:

- Визначення об’єктів: які дані поєднуються з якими операціями?
- Використання поліморфізму: які операції вимагають динамічної диспетчеризації?

Все решта є наслідком відповідей на ці запитання: узагальнені типи, структурована обробка винятків і т. д.

Розглянемо кілька практичних порад з покращення дизайну об’єктно-орієнтованого коду:

- *Уникнення інструкцій галуження* шляхом їх заміни на об’єкти. Це покращує стійкість (resilience) та гнучкість реалізації.
- *Уникнення циклів* шляхом їх вбудовування в структури даних. Це покращує реюзабельність коду та дозволяє реалізовувати ширшу функціональність на основі простішої.
- *Перетворення структур даних в об’єкти*
- *Перетворення алгоритмів у повторно використовувані модулі коду.*
- *Уникнення null-посилань,* оскільки null не є об’єктом.

Візьмемо приклад поганого об’єктно-орієнтованого коду:

```
class Program
{

    static int Sum(int[] values)
    {
        int sum = 0;

        for (int i = 0; i < values.Length; i++)
            sum += values[i];

        return sum;
    }

    static void Main(string[] args)
    {

        Console.ReadLine();

    }
}
```

Йому не вистачає гнучкості. Наприклад, клієнт приходить з вимогою додавати тільки непарні числа. Реалізувати цю вимогу нескладно: додати в цикл for перевірку умови

if(values[i] % 2 != 0)

Проте клієнту може знадобитись така потреба не завжди, а тільки в деяких випадках. Доведеться передавати в метод булевий прапорець (тут – oddNumbersOnly) та ускладнювати перевірку:

if(!OddNumbersOnly || values[i] % 2 != 0)

Це працюючий спосіб, проте він не є хорошим. Якщо клієнту знадобиться також інколи додавати тільки парні числа, попередній підхід не спрацює. Вирішення завдання все більше ускладнюватиметься та розростатиметься, що в подальшому призведе до неможливості сказати, чи працює код коректно, і до неможливості супроводу такого коду в цілому.

Відповідь на проблеми коду – відсутність у його поточній версії динамічної диспетчеризації. Статичне рішення не підтримує можливі зміни вимог, тому вимагає зміни існуючого коду, що є максимально небажаною дією.

Введемо об’єкт selector, який буде підтримувати різні клієнтські вимоги:

```
class Program
{

    static int Sum(int[] values, selector)
    {
        int sum = 0;

        foreach (int value in selector.Pick(values))
            sum += value;

        return sum;
    }

    static void Main(string[] args)
    {

        Console.ReadLine();

    }
}
```

Також слід зняти потребу в додаванні тільки цілих чисел, замість цього краще додавати елементи вмісту об’єкта. Нехай масив тепер буде об’єктом:

```
static int Sum(int[] values, selector)
{
    int sum = values.Sum(selector);
    return sum;
}
```

У такому випадку навіщо взагалі потрібний такий метод? Він не належить жодному об’єкту, тобто в такому вигляді є глобальною функцією. Звідси, функцію Sum() слід взагалі видалити та перенести в клас Array!

Інший приклад: конструкція if-else для перевірки рівності значення null. Метод ShowIt() приймає рядок та перетворює всі його літери у великі:

```
class Program
{

    static void ShowIt(string data)
    {
        string upper;

        if (data == null)
        {
            upper = null;
        }
        else
        {
            upper = data.ToUpper();
        }

        Console.WriteLine(upper);
    }

    static void Main(string[] args)
    {

        Console.ReadLine();
```

Оскільки null не є об’єктом, ми матимемо клас, проте не зможемо коректно утворити екземпляр цього класу. Отже, слід передавати такий об’єкт у метод ShowIt, який матиме вбудовану перевірку. Нехай, назвемо відповідний тип MaybeString, і він буде «думати», як правильно перетворити рядок:

```
static void ShowIt(MaybeString data)
{
    MaybeString upper = data.ToMaybeUpper();
    Console.WriteLine(upper);
}

static void Main(string[] args)
{

    Console.ReadLine();

}
```

Завдання 1. Уявіть, що клієнт замовив написати клас, що описуватиме грошові рахунки, зокрема, в банках. Повинні бути доступні операції внеску та зняття грошей. Вимоги до цього класу наступні:

- Гроші можна вносити в будь-який момент
- Знімати гроші можна тільки після підтвердження особистості власника рахунку
- Власнику дозволяється закривати рахунок у будь-який час.
- Закритий рахунок не дозволяє знімати чи вносити гроші.

Було запропоновано наступну тривіальну реалізацію:

```
class Account
{
    public decimal Balance { get; private set; }
    private bool IsVerified { get; set; }
    private bool IsClosed { get; set; }

    // Тест 1: Deposit 10, Close, Deposit 1 - Balance == 10
    // Тест 2: Deposit 10, Deposit 1 - Balance == 11
    public void Deposit(decimal amount)
    {
        if (this.IsClosed)
            return; // чи щось інше
        this.Balance += amount;
    }

    // Тест 3: Deposit 10, Withdraw 1 - Balance == 10
    // Тест 4: Deposit 10, Close, Withdraw 1 - Balance == 10
    // Тест 5: Deposit 10, Verify, Withdraw 1 - Balance == 9
    public void Withdraw(decimal amount)
    {
        if (!this.IsVerified)
            return; // чи щось інше
        if (this.IsClosed)
            return; // чи щось інше
        this.Balance -= amount;
    }

    public void HolderVerified()
    {
        this.IsVerified = true;
    }

    public void Close()
    {
        this.IsClosed = true;
    }
}
```

Нова вимога: рахунок повинен бути замороженим, якщо не використовувався деякий час. Вводимо новий прапорець IsFrozen та метод Freeze(), який заморожуватиме рахунок. Проте метод Freeze() повинен нічого не змінювати, якщо рахунок уже закрито або власника рахунку ще не було верифіковано. Подібна вимога також приховано додає потребу в розмороженні рахунку, якщо на нього було покладено гроші, причому разом з цим має викликатись функція зворотного відгуку (тут – OnUnfreeze()):

```
class Account
{
    public decimal Balance { get; private set; }
    private bool IsVerified { get; set; }
    private bool IsClosed { get; set; }
    private bool IsFrozen { get; set; }

    private Action OnUnfreeze { get; }

    public Account(Action onUnfreeze)
    {
        this.OnUnfreeze = onUnfreeze;
    }

    // Тест 1: Deposit 10, Close, Deposit 1 - Balance == 10
    // Тест 2: Deposit 10, Deposit 1 - Balance == 11
    // Тест 6: Deposit 10, Freeze, Deposit 1 - Викликано OnUnfreeze
    // Тест 7: Deposit 10, Freeze, Deposit 1 - IsFrozen == false
    // Тест 8: Deposit 10, Deposit 1 - OnUnfreeze не викликався
```

```
        public void Deposit(decimal amount)
        {
            if (this.IsClosed)
                return; // чи щось інше
            if (this.IsFrozen)
            {
                this.IsFrozen = false;
                this.OnUnfreeze();
            }
            this.Balance += amount;
        }

        // Тест 3:  Deposit 10, Withdraw 1 - Balance == 10
        // Тест 4:  Deposit 10, Close, Withdraw 1 - Balance == 10
        // Тест 5:  Deposit 10, Verify, Withdraw 1 - Balance == 9
        // Тест 9:  Deposit 10, Verify, Freeze, Withdraw 1 - Викликано OnUnfreeze
        // Тест 10: Deposit 10, Verify, Freeze, Withdraw 1 - IsFrozen == false
        // Тест 11: Deposit 10, Verify, Withdraw 1 - OnUnfreeze не викликався
        public void Withdraw(decimal amount)
        {
            if (!this.IsVerified)
                return; // чи щось інше
            if (this.IsClosed)
                return; // чи щось інше
            if (this.IsFrozen)
            {
                this.IsFrozen = false;
                this.OnUnfreeze();
            }
            this.Balance -= amount;
        }

        public void HolderVerified()
        {
            this.IsVerified = true;
        }

        public void Close()
        {
            this.IsClosed = true;
        }

        public void Freeze()
        {
            if (this.IsClosed)
                return; // рахунок повинен бути незакритим
            if (!this.IsVerified)
                return; // рахунок повинен бути перевіреним
            this.IsFrozen = true;
        }
    }
```

Кількість тестів для роботи класу значно зросла, причому ще не розглядалась інша логіка роботи рахунку та можливі нові вимоги. *Виправте проблеми з кодом, переглянувши відеоуроки 3.3-3.5 з курсу "Making Your C# Code More Object-oriented". Обов'язково доповніть звіт поясненнями виконаного рефакторингу з відео 3.6. Продемонструйте роботу отриманого коду.*

Завдання 2 (фокусування на логіці предметної області за допомогою послідовностей). Нехай визначено інтерфейс IPainter, який описуватиме можливий «контракт» з маляром:

```
interface IPainter
```

```
    {
        // визначення доступності маляра
        bool IsAvailable { get; }
        // оцінка часу на фарбування деякої поверхні
        TimeSpan EstimateTimeToPaint(double sqMeters);
        // оцінка оплати за пофарбовану поверхню
        double EstimateCompensation(double sqMeters);
    }
```

У клієнтському коді бажано написати функцію, яка обиратиме з набору малярів того, хто візьме найменше грошей за пофарбування поверхні заданої площі:

```
        private static IPainter FindCheapestPainter(double sqMeters,
                                                    IEnumerable<IPainter> painters)
        {
            double bestPrice = 0.0;
            IPainter cheapest = null;

            foreach (IPainter painter in painters)
            {
                if(painter.IsAvailable)
                {
                    double price = painter.EstimateCompensation(sqMeters);
                    if (cheapest == null || price < bestPrice)
                    {
                        cheapest = painter;
                    }
                }
            }

            return cheapest;
        }
```

Питання до такого коду:

- Чи зможете Ви сказати, що цей код робить через деякий час?
- Скільки часу піде на перевірку коректності методу?
- Чи можливий знайдений баг буде відноситись до цього коду, чи надходити десь ззовні?

Бажаний вигляд коду буде наближено таким:

```
        private static IPainter FindCheapestPainter(double sqMeters,
                                                    IEnumerable<IPainter> painters)
        {
            return painters
                        .ThoseAvailable()
                        .WithMinimum(painter.EstimateCompensation(sqMeters));
        }
```

Аналіз багів у такій реалізації методу спрощується. У подібний вигляд звести код вручну не завжди вдається, тому часто застосовуються бібліотеки, на зразок LINQ. Конкретно для даної реалізації перевірку доступності малярів будемо здійснювати за допомогою LINQ-методу Where():

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

namespace task2
{
    class Program
    {
        private static IPainter FindCheapestPainter(double sqMeters,
IEnumerable<IPainter> painters)
        {
```

```
            return painters
                        .Where(painter => painter.IsAvailable)
                        .WithMinimum(painter.EstimateCompensation(sqMeters));
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Варіанти знаходження маляра з мінімальною оплатою можуть бути різними. Наївний підхід полягає в сортуванні елементів набору від меншого до більшого та вибору першого з елементів:

```
return painters
        .Where(painter => painter.IsAvailable)
        .OrderBy(painter => painter.EstimateCompensation(sqMeters))
        .FirstOrDefault();
```

Метод FirstOrDefault() дозволяє уникнути виняткового стану, коли в послідовності не виявиться елементів (буде null).

Проте очевидно, що сортування – не найкращий спосіб знаходження мінімуму, оскільки найкраща асимптотична оцінка алгоритму буде $O(N \log N)$, де N – довжина послідовності. Можливо, малярів не буде особливо багато, проте ігнорувати покращення продуктивності коду небажано. Таким чином, слід при пошуку мінімуму перевіряти кожний елемент послідовності лише один раз, щоб асимптотична складність алгоритму була $O(n)$.

Розв'язок буде базуватись на методі Aggregate(), яка виконуватиме загальне порівняння поточного елементу послідовності з мінімумом та повертатиме мінімум після завершення перебору елементів.

```
return painters
        .Where(painter => painter.IsAvailable)
        .Aggregate((best, cur) =>
           best.EstimateCompensation(sqMeters) < cur.EstimateCompensation(sqMeters)
           ? best : cur);
```

У такій реалізації метод EstimateCompensation() викликатиметься двічі, а якщо послідовність буде порожня, повертатиметься null, і в результаті викидатиметься виняток.

Перегляньте відеоуроки 4.4-4.6 та зробіть код більш зручним для читання та продуктивним. Обов'язково додайте опис виконаного рефакторингу в звіт! Продемонструйте роботу отриманого коду.

Завдання 3 (відокремлення структури бізнес-даних від операцій над ними). Використовуючи тип IEnumerable<T>, ми не вказуємо, яка конкретно колекція лежить в основі послідовності об'єктів: словник, список, хеш-таблиця тощо. Тобто, IEnumberable<T> є найменш зобов'язуючим типом для представлення послідовності елементів.

Визначимо 2 додаткових методи відносно завдання 2, які будуть знаходити найшвидшого маляра (метод FindFasterPainter()) та формувати оплату роботи групу малярів (метод WorkTogether):

```
class Program
{
    private static IPainter FindCheapestPainter(double sqMeters,
                                                IEnumerable<IPainter> painters)
    {
        return painters
```

```
                    .Where(painter => painter.IsAvailable)
                    .WithMinimum(painter => painter.EstimateCompensation(sqMeters));
    }

    private static IPainter FindFasterPainter(double sqMeters,
                                              IEnumerable<IPainter> painters)
    {
        return painters
                    .Where(painter => painter.IsAvailable)
                    .WithMinimum(painter => painter.EstimateTimeToPaint(sqMeters));
    }

    private static void WorkTogether(double sqMeters, IEnumerable<IPainter> painters)
    {
        // припускаємо, що швидкість роботи малярів стала
        // кожний маляр фарбує частину стіни
        // оцінюємо швидкість, з якою кожний маляр пофарбує всю стіну
        TimeSpan time = TimeSpan.FromHours( 1 /
          painters
            .Where(painter => painter.IsAvailable)
            .Select(painter => 1 / painter.EstimateTimeToPaint(sqMeters).TotalHours)
            .Sum());

        // нехай оплата маляра залежить від часу
        // обчислюємо погодинну оплату кожного маляра
        // та множимо на визначений час фарбування групою малярів
        double cost = painters
            .Where(painter => painter.IsAvailable)
            .Select(painter =>
                painter.EstimateCompensation(sqMeters) /
                painter.EstimateTimeToPaint(sqMeters).TotalHours * time.TotalHours)
            .Sum();
    }

    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Зауваження щодо поточної реалізації:

- У методах весь час повторюється одна операція – отримання доступних малярів.
- Метод WorkTogether() виконує оцінку часу та оплати роботи малярів. Це узгоджується з частиною інтерфейсу IPainter:

```
public interface IPainter
{
    bool IsAvailable { get; }
    TimeSpan EstimateTimeToPaint(double sqMeters);
    double EstimateCompensation(double sqMeters);
}
```

Можливо, краще виокремити ці операції. У результаті, можемо зробити «віртуального» об’єкта-маляра, який даватиме лише 2 дані оцінки.

Виділимо клас ProportionalPainter, який реалізуватиме інтерфейс IPainter.

```
class ProportionalPainter : IPainter
{
    public TimeSpan TimePerSqMeter { get; set; }
    public double DollarsPerHour { get; set; }

    public bool IsAvailable => true;

    public double EstimateCompensation(double sqMeters) =>
        this.EstimateTimeToPaint(sqMeters).TotalHours * this.DollarsPerHour;
```

```
    public TimeSpan EstimateTimeToPaint(double sqMeters) =>
        TimeSpan.FromHours(this.TimePerSqMeter.TotalHours * sqMeters);
}
```

Об’єкти цього класу зможуть представляти як окремих малярів, так і бути фасадом для групи малярів. Таким чином, застосовується шаблон Компонувальник. У методі WorkTogether внесемо зміни у відповідь на впровадження нового класу:

```
private static IPainter WorkTogether(double sqMeters,
                                     IEnumerable<IPainter> painters)
{
    // припускаємо, що швидкість роботи малярів стала
    // кожний маляр фарбує частину стіни
    // оцінюємо швидкість, з якою кожний маляр пофарбує всю стіну
    TimeSpan time = TimeSpan.FromHours( 1 /
       painters
         .Where(painter => painter.IsAvailable)
         .Select(painter => 1 / painter.EstimateTimeToPaint(sqMeters).TotalHours)
         .Sum());

    // нехай оплата маляра залежить від часу
    // обчислюємо погодинну оплату кожного маляра
    // та множимо на визначений час фарбування групою малярів
    double cost = painters
        .Where(painter => painter.IsAvailable)
        .Select(painter =>
            painter.EstimateCompensation(sqMeters) /
            painter.EstimateTimeToPaint(sqMeters).TotalHours * time.TotalHours)
        .Sum();

    return new ProportionalPainter()
    {
        TimePerSqMeter = TimeSpan.FromHours(time.TotalHours / sqMeters),
        DollarsPerHour = cost / time.TotalHours
    };
}
```

Розгляньте подальші вдосконалення коду в відеоуроках 5.3-5.8, реалізуйте відповідний код та детально опишіть у звіті всі кроки зробленого рефакторингу. Продемонструйте роботу отриманого коду.

Завдання 4 (перетворення алгоритмів у об’єкти-стратегії). *Розгляньте подальші вдосконалення коду в відеоуроках 6.1-6.7, реалізуйте відповідний код та детально опишіть у звіті всі кроки зробленого рефакторингу.*