МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

МЕТОДИЧНІ ВКАЗІВКИ

до практичних робіт з курсу

«Об’єктно-орієнтоване програмування»

для студентів спеціальності 174 – Автоматизація, комп’ютерно-інтегровані технології та робототехніка усіх форм навчання

Харків
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

МЕТОДИЧНІ ВКАЗІВКИ

до практичних робіт з курсу

«Об’єктно-орієнтоване програмування»

для студентів спеціальності 174 **–** Автоматизація, комп’ютерно-інтегровані технології та робототехніка усіх форм навчання

Затверджено
редакційно-видавничою
радою університету,
протокол № 1 від 15.02.2024

Харків
2024

Методичні вказівки до практичних робіт з курсу «Об’єктно-орієнтоване програмування» 174 – Автоматизація, комп’ютерно-інтегровані технології та робототехніка усіх форм навчання /уклад. О. В. Пугановський, О. Г. Шутинський – Харків: НТУ “ХПІ”, 2024. – 52 с .

Укладачі: О. В. Пугановський
О. Г. Шутинський

Рецензент І.Л. Красніков

Кафедра автоматизації технологічних систем та екологічного моніторингу

ВСТУП

Методичні вказівки призначені для практичного закріплення знань студентами спеціальності 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка, що вивчають об'єктно-орієнтоване програмування.

Для виконання робіт потрібно мати передвстановлене середовище програмування C#. Альтернативною мовою, до якої можуть бути адаптовані матеріали вказівок є мова Java. Альтернативним варіантом є використання онлайн середовищ програмування. Тип операційної системи та інтернет-браузера (для онлайн редакторів) не має значення.

Вказівки містять мінімальний набір теоретичних відомостей тому перед виконанням практичних робіт рекомендується вивчити відповідні розділи з лекційного матеріалу. Проведення практичних занять з викладачем побудовано на поступовому розгортанні наведених у вказівках прикладів з поясненнями та модифікаціями у інтерактивному режимі.

При самостійному освоєнні практичних робіт, рекомендується перенести у середовище програмування наведений у даних рекомендаціях код і відлагодити його. Наступний крок – аналіз прикладу з конспектуванням власних досліджень. Звіт з практичної роботи при самостійному вивченні – готові програми та їх опис.

ПРАКТИЧНА РОБОТА 1
ЗАСТОСУВАННЯ МЕТОДІВ ТА ДЕЛЕГАТІВ

Мета роботи: закріпити знання і практичні навички у використанні методів та делегатів.

У минулій темі докладно було розглянуто делегати. Однак ці приклади, можливо, не показують істинної сили делегатів, тому що потрібні нам методи в даному випадку ми можемо викликати і безпосередньо без всяких делегатів. Однак найсильніший бік делегатів полягає в тому, що вони дають змогу делегувати виконання деякому коду ззовні. І на момент написання програми ми можемо не знати, що за код буде виконуватися. Ми просто викликаємо делегат. А який метод буде безпосередньо виконуватися при виклику делегата, буде вирішуватися потім.

Розглянемо докладний приклад. Нехай у нас є клас, що описує рахунок у банку:

```
public class Account
{   int sum; // Змінна для зберігання суми
    // через конструктор встановлюється початкова сума на рахунку
    public Account(int sum) => this.sum = sum;
    // додати кошти на рахунок
    public void Add(int sum) => this.sum += sum;
    // взяти гроші з рахунку
    public void Take(int sum)
    {
        // беремо гроші, якщо на рахунку достатньо коштів
        if (this.sum >=sum) this.sum -= sum;
    } }
```

У змінній sum зберігається сума на рахунку. За допомогою конструктора встановлюється початкова сума на рахунку. Метод `Add()` слугує для додавання на рахунок, а метод `Take` - для зняття грошей з рахунку.

Припустимо, у разі виведення грошей за допомогою методу `Take` нам треба якось повідомляти про це самого власника рахунка і, можливо, інші об'єкти. Якщо йдеться про консольну програму, і клас застосовуватиметься в тому самому проєкті, де його створено, то ми можемо написати просто:

```
public class Account
{   int sum;
    public Account(int sum) => this.sum = sum;
    public void Add(int sum) => this.sum += sum;
    public void Take(int sum)
    {
        if (this.sum >= sum)
        {
            this.sum -= sum;
            Console.WriteLine($"З рахунку списано
{sum} у.о.");
        }
    } }
```

Але що якщо наш клас планують використовувати в інших проєктах, наприклад, у графічному застосунку на Windows Forms або WPF, у мобільному застосунку, у веб-додатку. Там рядок повідомлення:

```
Console.WriteLine($"З рахунку списано {sum} у.о.");
```

не матиме великого сенсу.

Ба більше, наш клас Account буде використовуватися іншими розробниками у вигляді окремої бібліотеки класів. І ці розробники захочуть повідомляти про зняття коштів якимось іншим чином, про які ми навіть можемо не здогадуватися на момент написання класу. Тому примітивне повідомлення у вигляді рядка коду:

```
Console.WriteLine($"З рахунку списано {sum} у.о.");
```

не найкраще рішення в цьому випадку. І делегати дають змогу делегувати визначення дії з класу в зовнішній код, який використовуватиме цей клас.

Змінимо клас, застосувавши делегати:

```
// Оголошуємо делегат
public delegate void AccountHandler(string message);
public class Account
{   int sum;
    // Створюємо змінну делегата
    AccountHandler? taken;
    public Account(int sum) => this.sum = sum;
    // Реєструємо делегат
    public void RegisterHandler(AccountHandler del)
    {
        taken = del;
    }
    public void Add(int sum) => this.sum += sum;
    public void Take(int sum)
    {
        if (this.sum >= sum)
        {
            this.sum -= sum;
```

```
                // викликаємо делегат, передаючи йому
повідомлення
                taken?.Invoke($"З рахунку списано {sum}
у.о.");
            }
            else
            {
                taken?.Invoke($"Недостатньо коштів. Ба-
ланс: {this.sum} у.о.");
            } } }
```

Для делегування дії тут визначено делегат `AccountHandler`. Цей делегат відповідає будь-яким методам, які мають тип void і приймають параметр типу `string`.

```
public delegate void AccountHandler(string message);
```

У класі `Account` визначаємо змінну `taken`, яка представляє цей делегат:

```
AccountHandler? taken;
```

Тепер треба пов'язати цю змінну з конкретною дією, яка буде виконуватися. Ми можемо використовувати різні способи для передачі делегата в клас. У цьому випадку визначається спеціальний метод `RegisterHandler`, у якому в змінну `taken` передається реальна дія:

```
public void RegisterHandler(AccountHandler del)
{  taken = del; }
```

Таким чином, делегат встановлено, і тепер його можна викликати. Виклик делегата здійснюється в методі Take:

```
public void Take(int sum)
{    if (this.sum >= sum)
     {
         this.sum -= sum;
         // викликаємо делегат, передаючи йому
повідомлення
         taken?.Invoke($"З рахунку списано {sum}
у.о.");
     }
     else
     {
         taken?.Invoke($"Недостатньо коштів. Баланс:
{this.sum} у.о.");
     } }
```

Оскільки делегат AccountHandler як параметр приймає рядок, то під час виклику змінної taken() ми можемо передати в цей виклик конкретне повідомлення. Залежно від того, відбулося зняття грошей чи ні, у виклик делегата передаються різні повідомлення.

Тобто фактично замість делегата будуть виконуватися дії, які передані делегату в методі RegisterHandler. Причому знову ж таки підкреслю, при виклику делегата ми не знаємо, що це будуть за дії. Тут ми тільки передаємо в ці дії повідомлення про успішно або невдале зняття.

Тепер протестуємо клас в основній програмі:

```
// створюємо банківський рахунок
Account account = new Account(200);
```

```
    // Додаємо в делегат посилання на метод
PrintSimpleMessage
    account.RegisterHandler(PrintSimpleMessage);
    // Два рази поспіль намагаємося зняти гроші
    account.Take(100);
    account.Take(150);

    void PrintSimpleMessage(string message) => Con-
sole.WriteLine(message);
```

Тут через метод `RegisterHandler` змінній taken у класі `Account` передається посилання на метод `PrintSimpleMessage`. Цей метод відповідає делегату `AccountHandler`. Відповідно там, де в класі `Account` викликається делегат `taken`, у реальності буде виконуватися метод `PrintSimpleMessage`.

Через параметр `message` метод PrintSimpleMessage отримає передане з делегата повідомлення і виведе його на консоль:

```
З рахунку списано 100 у.о.
Недостатньо коштів. Баланс: 100 у.о.
```

Таким чином, ми створили механізм зворотного виклику для класу `Account`, який спрацьовує в разі зняття грошей. Тут ми виводимо повідомлення на консоль. Так, ми могли б просто виводити повідомлення на консоль і без делегатів. Однак із делегатом для класу `Account` не важливо, як це повідомлення виводиться. Класу `Account` навіть не відомо, що взагалі буде робитися в результаті списання грошей. Він просто надсилає повідомлення про це через делегат.

У результаті, якщо ми створюємо консольний додаток, ми можемо через делегат виводити повідомлення на консоль. Якщо ми створюємо графічний додаток Windows Forms або WPF, то можна виводити повідомлення у вигляді

графічного вікна. А можна не просто виводити повідомлення. А, наприклад, записати під час списання інформацію про цю дію у файл або надіслати повідомлення на електронну пошту. Загалом будь-якими способами обробити виклик делегата. І спосіб обробки не залежатиме від класу `Account`.

Додавання та видалення методів у делегаті

Хоча в прикладі наш делегат приймав адресу на один метод, насправді він може вказувати одразу на кілька методів. Крім того, за потреби ми можемо видалити посилання на адреси певних методів, щоб вони не викликалися під час виклику делегата. Отже, змінимо в класі `Account` метод `RegisterHandler` і додамо новий метод `UnregisterHandler`, який видалятиме методи зі списку методів делегата:

```
public delegate void AccountHandler(string message);
public class Account
{   int sum;
    AccountHandler? taken;
    public Account(int sum) => this.sum = sum;
    // Реєструємо делегат
    public void RegisterHandler(AccountHandler del)
    {
        taken += del;
    }
    // Скасування реєстрації делегата
    public void UnregisterHandler(AccountHandler del)
    {
        taken -= del; // видаляємо делегат
    }
    public void Add(int sum) => this.sum += sum;
    public void Take(int sum)
    {
```

```
            if (this.sum >= sum)
            {
                this.sum -= sum;
                taken?.Invoke($"З рахунку списано {sum}
у.о.");
            }
            іще
                taken?.Invoke($"Недостатньо коштів. Ба-
ланс: {this.sum} у.о.");
        } }
```

У першому методі об'єднує делегати `taken` і `del` в один, який потім присвоюється змінній `taken`. У другому методі зі змінної `taken` видаляється делегат `del`.

Застосуємо нові методи:

```
    Account account = new Account(200);
    // Додаємо в делегат посилання на методи
    account.RegisterHandler(PrintSimpleMessage);
    account.RegisterHandler(PrintColorMessage);
    // Два рази поспіль намагаємося зняти гроші
    account.Take(100);
    account.Take(150);
    // Видаляємо делегат
    account.UnregisterHandler(PrintColorMessage);
    // знову намагаємося зняти гроші
    account.Take(50);
    void PrintSimpleMessage(string message) => Con-
sole.WriteLine(message);
    void PrintColorMessage(string message)
```

```
{    // Встановлюємо червоний колір символів
     Console.ForegroundColor = ConsoleColor.Red;
     Console.WriteLine(message);
     // Скидаємо налаштування кольору
     Console.ResetColor();}
```

З метою тестування ми створили ще один метод – `PrintColorMessage`, який виводить те саме повідомлення тільки червоним кольором. Посилання на цей метод також передається в метод `RegisterHandler`, і таким чином його отримає змінна `taken`.

У рядку `account.UnregisterHandler(PrintColorMessage);` цей метод видаляється зі списку викликів делегата, тому цей метод більше не буде спрацьовувати. Консольний вивід матиме таку форму:

```
З рахунку списано 100 у.о.
З рахунку списано 100 у.о.
Недостатньо коштів. Баланс: 100 у.о.
Недостатньо коштів. Баланс: 100 у.о.
З рахунку списано 50 у.о.
```

ПРАКТИЧНА РОБОТА 2
ЗАСТОСУВАННЯ КЛАСІВ ТА СТРУКТУР

Мета роботи: Засвоїти практичні знання з використання класів та структур при розробці програм

Точкою входу до програми мовою C# є метод `Main`. Саме з цього починається виконання програми на C#. І програма на C# повинна обов'язково мати метод `Main`. Однак може виникнути питання, який ще метод `Main`, якщо, на-

приклад, Visual Studio 2022 за промовчанням створює проект консольної програми з наступним кодом:

```
Console.WriteLine("Hello, World!");
```

І ця програма жодних методів `Main` не містить. Проте насправді цей код неявно поміщається компілятором метод `Main`, який, своєю чергою, міститься у клас `Program`. Насправді назва класу може бути будь–якою (як правило, це клас `Program`, власне тому файл коду, що генерується за умовчанням, називається `Program.cs`). Але метод `Main` є обов'язковою частиною консольної програми. Тому вище поданий код фактично еквівалентний наступній програмі:

```
classProgram
{
staticMain(string[] args)
{
Console.WriteLine("Hello, World!");
}
}
```

Визначення методу `Main` обов'язково починається з статичного модифікатора, яке вказує, що метод `Main` – статичний. Пізніше ми докладніше розберемо, що це означає.

Типом методу `Main`, що повертається, обов'язково є тип `void`. Крім того, як параметр він приймає масив рядків – `string[] args` – в реальній програмі це параметри, через які при запуску програми з консолі ми можемо передати їй деякі значення. Усередині методу розташовуються дії, які виконує програма.

До Visual Studio 2022 усі попередні студії створювали за умовчанням приблизно такий код. Але, починаючи з Visual Studio 2022, нам необов'язково

вручну визначати клас `Program` і в ньому метод `Main` – компілятор генерує їх самостійно.

Якщо ми визначаємо якісь змінні, константи, методи та звертаємося до них, вони поміщаються у метод `Main`.

Однак слід враховувати, що визначення типів (зокрема класів) повинні йти в кінці файлу після інструкцій верхнього рівня.

Класи.Створення конструкторів

Конструктори можуть мати модифікатори, які зазначаються перед ім'ям конструктора. Так, у даному випадку, щоб конструктор був доступний поза класом Person, його визначено з модифікатором public.

Визначивши конструктор, ми можемо викликати його до створення об'єкта Person:

```
Person tom = new Person(); // Створення об'єкта Per-
son
```

В даному випадку вираз Person() якраз представляє виклик певного в класі конструктора (це більше не автоматичний конструктор за замовчуванням, якого клас тепер немає). Відповідно, при його виконанні на консолі буде виводитися рядок "Створення об'єкта Person"

```
Person tom = new Person(); // Створення об'єкта класу
Person
tom.Print(); // Ім'я: Tom Вік: 37

class Person
{
public string name;
public int age;
```

```
public Person()
{
Console.WriteLine("Створення об'єкта Person");
name = "Tom";
age = 37;
}
public void Print()
{
Console.WriteLine($"Ім'я: {name} Вік: {age}");
} }
```

Подібним чином ми можемо визначати інші конструктори в класі. Наприклад, змінимо клас `Person` наступним чином:

```
Person tom = newPerson(); // Виклик першого
конструктора без параметрів
Person bob = newPerson("Bob"); //виклик 2-го конст-
руктора з одним параметром
Person sam = newPerson("Sam", 25); // Виклик 3-го
конструктора з двома параметрами

tom.Print(); // Ім'я: Невідомо Вік: 18
bob.Print(); // Ім'я: Bob Вік: 18
sam.Print(); // Ім'я: Sam Вік: 25

classPerson
{
publicstring name;
publicint age;
```

```
    publicPerson() { name = "Невідомо"; age = 18; } // 1
конструктор
    publicPerson(string n) {name = n; age = 18; } // 2
конструктор
    publicPerson(string n, int a) {name = n; age = a; }
// 3 конструктор

    publicvoid Print()
    {
    Console.WriteLine($"Ім'я: {name} Вік: {age}");
    } }
```

Ключове слово `this`

Ключове слово `this` представляє посилання на поточний екземпляр/об'єкт класу.

```
    Person sam = new ("Sam", 25);
    sam.Print(); // Ім'я: Sam Вік: 25

    classPerson
    {
    publicstring name;
    publicint age;
    publicPerson() { name = "Невідомо"; age = 18; }
    publicPerson(string name) {this.name=name; age = 18;
}
    publicPerson(string name, int age)
    {
    this.name = name;
    this.age = age;
    }
```

```
    publicvoid Print() => Console.WriteLine($"Ім'я:
{name} Вік: {age}");
    }
```

У прикладі вище у другому та третьому конструкторі параметри називаються так само, як і поля класу. І щоб розмежувати параметри та поля класу, до полів класу звернення йде через ключове слово `this`. Так, у виразі

```
this.name = name;
```

перша частина – `this.name` означає, що `name` – це поле поточного класу, а не назва параметра `name`. Якби у нас параметри та поля називалися порізному, то використовувати слово це було б необов'язково. Також через ключове слово це можна звертатися до будь-якого поля або методу.

ПРАКТИЧНА РОБОТА 3
ВИКОРИСТАННЯ ПРИНЦИПІВ ООП. РЕФАКТОРІНГ

Мета роботи: Використати принципи ООП для створення програм, засвоїти термін «рефакторинг» на практичних прикладах

Основи ООП:

- Абстракція даних: подробиці внутрішньої логіки приховані кінцевого користувача. Користувачеві не потрібно знати, як працюють ті чи інші класи та методи, щоб їх використовувати. Відповідним прикладом із реального життя буде велосипед – коли ми їздимо на ньому або змінюємо деталь, нам не потрібно знати, як педаль приводить його в рух або як закріплений ланцюг.

- Спадкування: найпопулярніший принцип ОВП. Спадкування робить можливим повторне використання коду – якщо якийсь клас вже має якусь логіку та функції, нам не потрібно переписувати все це заново для створення нового класу, ми можемо просто включити старий клас у новий, цілком.
- Інкапсуляція: включення до класу об'єктів іншого класу, питання доступу до них, їх видимості.
- Поліморфізм: «полі» означає «багато», а «морфізм» – «зміна» або «варіативність», таким чином, «поліморфізм» – це властивість тих самих об'єктів і методів набувати різних форм.
- Обмін повідомленнями: здатність одних об'єктів викликати методи інших об'єктів, передаючи їм керування.

Давайте створимо консольний додаток і клас `Overload` з трьома методами `DisplayOverload` з параметрами, як нижче:

```
public class Overload
{
public void DisplayOverload(int a){ }
public void DisplayOverload(string a)
{ System.Console.WriteLine("DisplayOverload" + a); }
public void DisplayOverload(string a, int b)
{  System.Console.WriteLine("DisplayOverload"  +  a  +
b); } }
```

У головному методі Program.cs тепер напишемо наступне:

```
static void Main(string[] args)
{ Overload overload = new Overload();
overload.DisplayOverload(100);
```

```
overload.DisplayOverload("method overloading");
overload.DisplayOverload("method overloading", 100);
Console.ReadKey(); }
```

І тепер, коли ми це запустимо, результат буде наступним:

```
DisplayOverload 100
DisplayOverload method overloading
DisplayOverload method overloading 100
```

Клас `Overload` містить три методи, і всі вони називаються `DisplayOverload`, вони відрізняються тільки типами параметрів. У C# (як і більшості інших мов) ми можемо створювати методи з однаковими іменами, але різними параметрами, це і називається «перевантаження методів». Це означає, що нам немає потреби запам'ятовувати купу імен методів, які здійснюють однакові дії з різними типами даних.

Роль ключового слова `params` у поліморфізмі

Параметри можуть бути чотирьох різних видів:

- передане значення;
- передане посилання;
- параметр для виведення;
- масив властивостей.

З першими трьома ми начебто розібралися, тепер докладніше поглянемо на четвертий.

Якщо ми запустимо наступний код:

```
public void DisplayOverload(int a, string a) { }
public void Display(int a)
{string a;}
```

То отримаємо дві помилки:

```
Error1: Parametr name 'a' is a duplicate
```

```
Error2: Local local variable 'a' cannot be declared in this scope because it would give different meaning to 'a', which is already used in 'parent or current' scope to denote something else
```

Звідси випливає висновок: імена параметрів мають бути унікальними. Також не можуть бути однаковими ім'я параметра методу та ім'я змінної, створеної в цьому методі.

Тепер спробуємо запустити наступний код:

```
public class Overload
{
private string name = "Akhil";

public void Display()
{
Display2(ref name, ref name);
System.Console.WriteLine(name);
}

private void Display2(ref string x, ref string y)
{
System.Console.WriteLine(name);
x = "Akhil 1";
System.Console.WriteLine(name);
y = "Akhil 2";
```

```
System.Console.WriteLine(name);
name = "Akhil 3"; } }

class Program
{
static void Main(string[] args)
{
Overload overload = new Overload();
overload.Display();
Console.ReadKey();
} }
```

Ми отримаємо наступний вивід:

```
Akhil
Akhil 1
Akhil 2
Akhil 3
```

Ми можемо передавати однакові параметри посилання стільки разів, скільки захочемо. У методі `Display` рядок name має значення `Akhil`. Коли змінюємо значення x на «`Akhil1`», насправді ми змінюємо значення name, т.к. через параметр x передане посилання саме на нього. Те саме з `y` – всі ці три змінні посилаються на одне місце в пам'яті.

Тепер найцікавіше:

```
public class Overload
{
public void Display()
{
```

```
    DisplayOverload(100, "Akhil", "Mittal", "OOP");
    DisplayOverload(200, "Akhil");
    DisplayOverload(300);
    }

    private void DisplayOverload(int a, params string[]
parameterArray)
    {
    foreach (string str in parameterArray)
    Console.WriteLine(str + " " + a);
    } }

    class Program
    {
    static void Main(string[] args)
    {
    Overload overload = new Overload();
    overload.Display();
    Console.ReadKey();
    } }
```

Це дасть нам такий вивід:

```
    Akhil 100
    Mittal 100
    OOP 100
    Akhil 200
```

Нам часто потрібно передати методу n параметрів. C# таку можливість надає ключове слово params.

Важливо: це ключове слово може бути застосовано тільки до останнього аргументу методу, так що метод нижче не працюватиме:

```
    private void DisplayOverload(int a, params string[]
parameterArray, int b) { }
```

У випадку `DisplayOverload` перший аргумент повинен бути цілим числом, а решта — скільки завгодно багато рядків або навпаки, жодної.

```
    DisplayOverload(200, 100);
    DisplayOverload(200); }
    private void DisplayOverload(int a, params int[]
parameterArray)
    { foreach (var i in parameterArray)
Console.WriteLine(i + " " + a); }}
```

Результат: 200 100 300 100 100 200

Важливо запам'ятати: C # досить розумний, щоб розділити звичайні параметри і масив параметрів, навіть якщо вони одного типу.

Подивіться на наступні два методи:

```
    private void DisplayOverload(int a, params string[][]
parameterArray)
     { }
    private void DisplayOverload(int a, params string[,]
parameterArray)
    { }
```

Різниця між ними в тому, що перший запуститься, і така синтаксична конструкція матиме на увазі, що метод передаватиме n масивів рядків. Друга ж видасть помилку:

```
Error: The parameter array must be a single
dimensional array
```

Запам'ятайте: масив параметрів має бути одновимірним.

Слід згадати, що останній аргумент не обов'язково заповнювати окремими об'єктами, можна його використати, ніби це звичайний аргумент, який приймає масив, тобто:

```
public class Overload
{
public void Display()
{
string[] names = {"Akhil", "Ekta", "Arsh"};
DisplayOverload(3, names);
}

private void DisplayOverload(int a, params string[]
parameterArray)
{
foreach (var s in parameterArray)
Console.WriteLine(s + " " + a);
} }

class Program
{
static void Main(string[] args)
```

```
{
Overload overload = new Overload();
overload.Display();
Console.ReadKey();
} }
```

Виведення буде наступним:

```
Akhil 3
Ekta 3
Arsh 3
```

Однак такий код:

```
public class Overload
{
public void Display()
{
string [] names = {"Akhil", "Arsh"};
DisplayOverload(2, names, "Ekta");
}

private void DisplayOverload(int a, params string[]
parameterArray)
{
foreach (var str in parameterArray)
Console.WriteLine(str + " " + a);
} }
```

Вже викликає помилку:

```
    Error: The best overloaded method match for 'Inheri-
tanceAndPolymorphism.Overload.DisplayOverload(int, params
string[])' has invalid arguments

    Error:Argument 2: cannot convert from 'string[]' to
'string'
```

Змішувати передачу окремими параметрами та одним масивом не можна.

Тепер розглянемо поведінку наступної програми:

```
    public class Overload
    {
    public void Display()
    {
    int[] numbers = {10, 20, 30};
    DisplayOverload(40, numbers);
    Console.WriteLine(numbers[1]);
    }

    private void DisplayOverload(int a, params int[] pa-
rameterArray)
    {
    parameterArray[1] = 1000;
    } }

    class Program
    {
    static void Main(string[] args)
    {
```

```
Overload overload = new Overload();
overload.Display();
Console.ReadKey();
} }
```

Після її виконання ми отримаємо в консолі: 1000. Це відбувається через те, що при подібному синтаксисі масив передається за посиланням. Проте варто зазначити таку особливість:

```
public class Overload
{
public void Display()
{
int number = 102;
DisplayOverload(200, 1000, number, 200);
Console.WriteLine(number);
}

private void DisplayOverload(int a, params int[] pa-
rameterArray)
{
parameterArray[1] = 3000;
} }
```

Результатом виконання такого коду буде102. Адже з переданих параметрів C# автоматично формує новий, тимчасовий масив.

Тепер поговоримо про пріоритет мови у виборі методів. Припустимо, у нас є такий код:

```
public class Overload
```

```
{
public void Display()
{
DisplayOverload(200);
DisplayOverload(200, 300);
DisplayOverload(200, 300, 500, 600);
}

private void DisplayOverload(int x, int y)
{
Console.WriteLine("The 2 integers" + x + " " + y);
}

private void DisplayOverload(params int[] parameter-
Array)
{
Console.WriteLine("parameterArray");
}

}
///Program.cs все той же
```

C# розглядає методи з масивом параметрів останніми, так що в другому випадку буде викликаний метод, який приймає два цілих числа. У першому та третьому випадку буде викликаний метод з params, оскільки нічого крім нього запустити неможливо. Таким чином, на виході ми отримаємо:

```
parameterArray
The two integers 200 300
parameterArray
```

Тепер дещо цікаве. Як ви вважаєте, яким буде результат виконання наступної програми?

```
public class Overload
{
public static void Display(params object[] object-
ParamArray)
{
foreach (object obj in objectParamArray)
{
Console.Write(obj.GetType().FullName + " ");
}
Console.WriteLine();
} }

class Program
{
static void Main(string[] args)
{
object[] objArray = { 100, "Akhil", 200.300}; //Масив
object obj = objArray; //Масив як об'єкт
Overload.Display(objArray);
Overload.Display((object)objArray); //Масив,
приведений до об'єкту
Overload.Display(obj);
///Чому б не піти глибше? :D
Overload.Display((object[])obj); //Масив, як об'єкт,
приведений до масиву
Console.ReadKey(); } }
```

У консолі ми побачимо:

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

Тобто, в першому і четвертому випадках масив передається саме як масив, замінюючи собою об'єкт `ParamArray`, а в другому і третьому випадках масив передається як одиничний об'єкт, з якого створюється новий масив з одного елемента.

ПРАКТИЧНА РОБОТА 4
ВИКОРИСТАННЯ ПАТЕРНІВ, ЩО ПОРОДЖУЮТЬ

Мета роботи: Створення коду на основі діаграм патернів, що породжують.

Паттерн "Абстрактна фабрика" (Abstract Factory) надає інтерфейс створення сімейств взаємозалежних об'єктів з певними інтерфейсами без зазначення конкретних типів даних об'єктів, рис.4.1.

Концепт реалізації узято з сайту https://refactoring.guru/uk/design-patterns/abstract-factory/csharp/example

```
using System;
namespace
RefactoringGuru.DesignPatterns.AbstractFactory.Conceptual
```

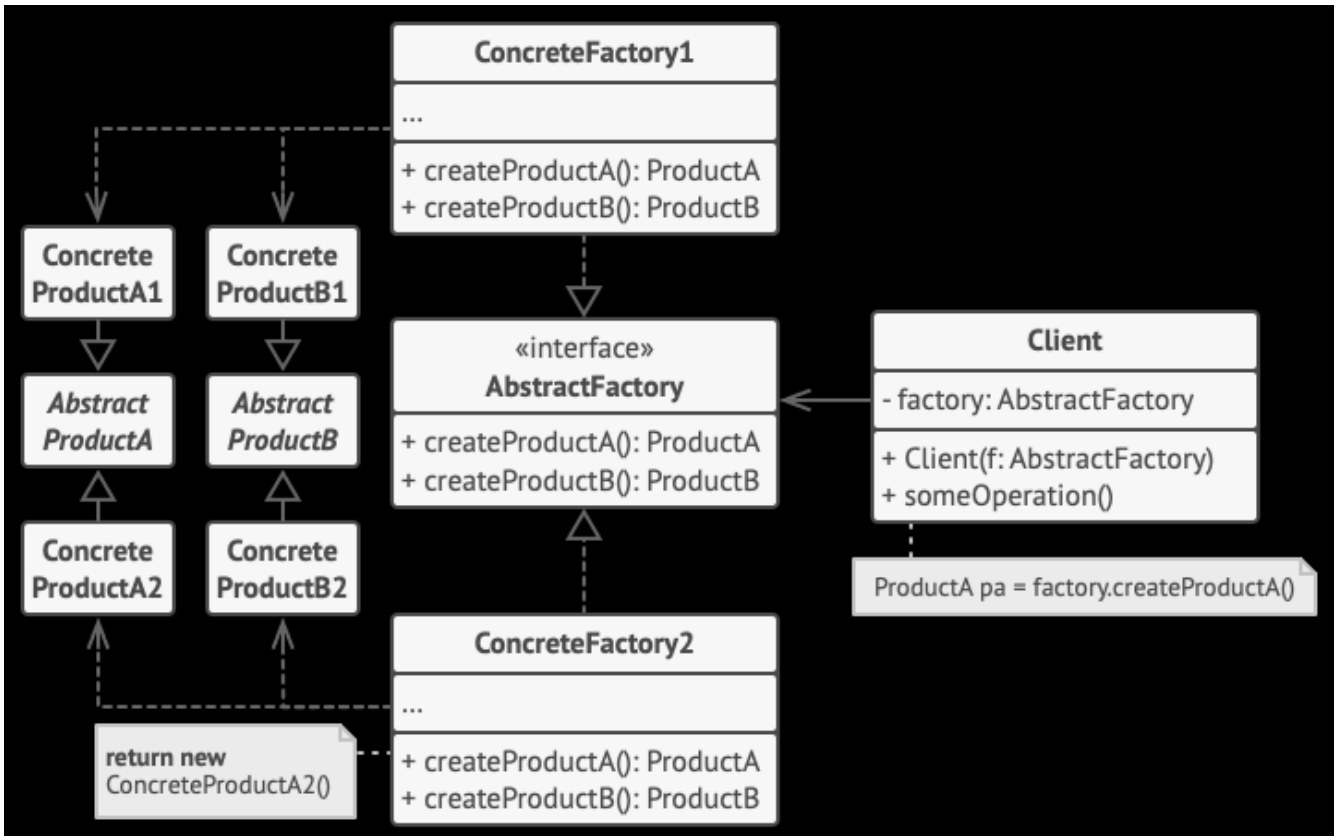

Рисунок 4.1 – Діаграма паттерну "Абстрактна фабрика"

```
    {
        // The Abstract Factory interface declares a set
of methods that return
        // different abstract products. These products
are called a family and are
        // related by a high-level theme or concept.
Products of one family are
        // usually able to collaborate among themselves.
A family of products may
        // have several variants, but the products of one
variant are incompatible
        // with products of another.
        public interface IAbstractFactory
```

```
        {
            IAbstractProductA CreateProductA();

            IAbstractProductB CreateProductB();
        }

        // Concrete Factories produce a family of
products that belong to a single
        // variant. The factory guarantees that resulting
products are compatible.
        // Note that signatures of the Concrete Factory's
methods return an abstract
        // product, while inside the method a concrete
product is instantiated.
        class ConcreteFactory1 : IAbstractFactory
        {
            public IAbstractProductA CreateProductA()
            {
                return new ConcreteProductA1();
            }

            public IAbstractProductB CreateProductB()
            {
                return new ConcreteProductB1();
            }
        }

        // Each Concrete Factory has a corresponding
product variant.
        class ConcreteFactory2 : IAbstractFactory
```

```
        {
            public IAbstractProductA CreateProductA()
            {
                return new ConcreteProductA2();
            }

            public IAbstractProductB CreateProductB()
            {
                return new ConcreteProductB2();
            }
        }

        // Each distinct product of a product family
should have a base interface.
        // All variants of the product must implement
this interface.
        public interface IAbstractProductA
        {
            string UsefulFunctionA();
        }

        // Concrete Products are created by corresponding
Concrete Factories.
        class ConcreteProductA1 : IAbstractProductA
        {
            public string UsefulFunctionA()
            {
                return "The result of the product A1.";
            }
        }
```

```
        class ConcreteProductA2 : IAbstractProductA
        {
            public string UsefulFunctionA()
            {
                return "The result of the product A2.";
            }
        }

        // Here's the the base interface of another
product. All products can
        // interact with each other, but proper
interaction is possible only between
        // products of the same concrete variant.
        public interface IAbstractProductB
        {
            // Product B is able to do its own thing...
            string UsefulFunctionB();

            // ...but it also can collaborate with the
ProductA.
            //
            // The Abstract Factory makes sure that all
products it creates are of
            // the same variant and thus, compatible.
            string
AnotherUsefulFunctionB(IAbstractProductA collaborator);
        }
```

```
        // Concrete Products are created by corresponding
Concrete Factories.
        class ConcreteProductB1 : IAbstractProductB
        {
            public string UsefulFunctionB()
            {
                return "The result of the product B1.";
            }

            // The variant, Product B1, is only able to
work correctly with the
            // variant, Product A1. Nevertheless, it
accepts any instance of
            // AbstractProductA as an argument.
            public string
AnotherUsefulFunctionB(IAbstractProductA collaborator)
            {
                var result =
collaborator.UsefulFunctionA();

                return $"The result of the B1
collaborating with the ({result})";
            }
        }

        class ConcreteProductB2 : IAbstractProductB
        {
            public string UsefulFunctionB()
            {
                return "The result of the product B2.";
```

```
            }

            // The variant, Product B2, is only able to
work correctly with the
            // variant, Product A2. Nevertheless, it
accepts any instance of
            // AbstractProductA as an argument.
            public string
AnotherUsefulFunctionB(IAbstractProductA collaborator)
            {
                var result =
collaborator.UsefulFunctionA();

                return $"The result of the B2
collaborating with the ({result})";
            }
        }

        // The client code works with factories and
products only through abstract
        // types: AbstractFactory and AbstractProduct.
This lets you pass any
        // factory or product subclass to the client code
without breaking it.
        class Client
        {
            public void Main()
            {
                // The client code can work with any
concrete factory class.
```

```
                Console.WriteLine("Client: Testing client
code with the first factory type...");
                ClientMethod(new ConcreteFactory1());
                Console.WriteLine();

                Console.WriteLine("Client: Testing the
same client code with the second factory type...");
                ClientMethod(new ConcreteFactory2());
            }

            public void ClientMethod(IAbstractFactory
factory)
            {
                var productA = factory.CreateProductA();
                var productB = factory.CreateProductB();


Console.WriteLine(productB.UsefulFunctionB());

Console.WriteLine(productB.AnotherUsefulFunctionB(product
A));
            }
        }

        class Program
        {
            static void Main(string[] args)
            {
                new Client().Main();
            }  }  }
```

При правильному створенні коду, результат виконання:

```
Client: Testing client code with the first factory
type...
The result of the product B1.
The result of the B1 collaborating with the (The result
of the product A1.)

Client: Testing the same client code with the second
factory type...
The result of the product B2.
The result of the B2 collaborating with the (The result
of the product A2.)
```

ПРАКТИЧНА РОБОТА 5
ВИКОРИСТАННЯ СТРУКТУРНИХ ПАТЕРНІВ

Мета роботи: Створити код програми на основі структурного патерну Фасад.

Фасад — це структурний патерн, який надає простий (але урізаний) інтерфейс до складної системи об’єктів, бібліотеки або фреймворку, рис. 5.1.

Крім того, що Фасад дозволяє знизити загальну складність програми, він також допомагає винести код, який залежить від зовнішньої системи, в одне місце.

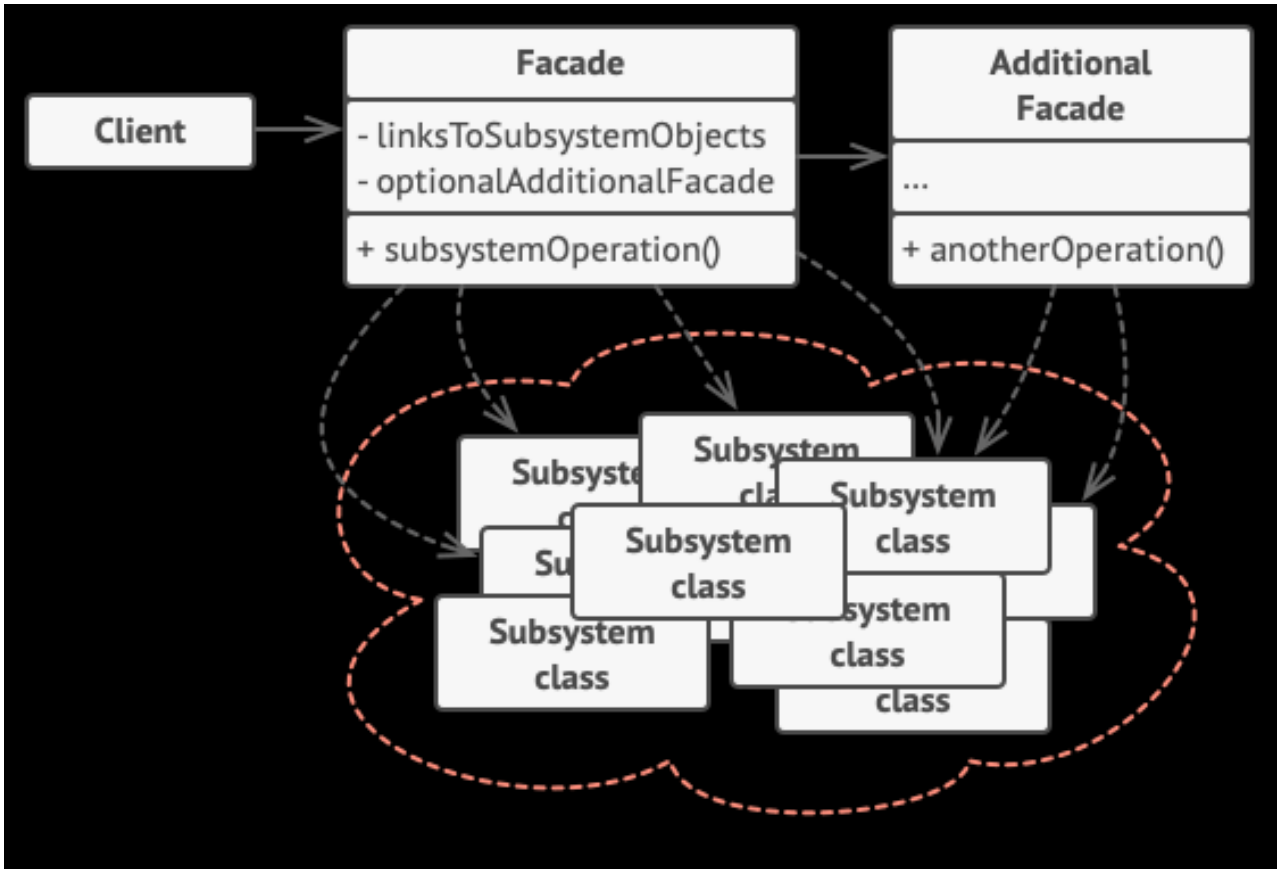

Рисунок 5.1 – Діаграма паттерну "Фасад"

Концепт реалізації взято з ресурсу https://refactoring.guru/ru/design-patterns/facade

```
using System;

namespace
RefactoringGuru.DesignPatterns.Facade.Conceptual
{
```

```
        // The Facade class provides a simple interface
to the complex logic of one
        // or several subsystems. The Facade delegates
the client requests to the
        // appropriate objects within the subsystem. The
Facade is also responsible
        // for managing their lifecycle. All of this
shields the client from the
        // undesired complexity of the subsystem.
        public class Facade
        {
            protected Subsystem1 _subsystem1;

            protected Subsystem2 _subsystem2;

            public Facade(Subsystem1 subsystem1,
Subsystem2 subsystem2)
            {
                this._subsystem1 = subsystem1;
                this._subsystem2 = subsystem2;
            }

            // The Facade's methods are convenient
shortcuts to the sophisticated
            // functionality of the subsystems. However,
clients get only to a
            // fraction of a subsystem's capabilities.
            public string Operation()
            {
```

```
                string result = "Facade initializes
subsystems:\n";
                result += this._subsystem1.operation1();
                result += this._subsystem2.operation1();
                result += "Facade orders subsystems to
perform the action:\n";
                result += this._subsystem1.operationN();
                result += this._subsystem2.operationZ();
                return result;
            }
        }

        // The Subsystem can accept requests either from
the facade or client
        // directly. In any case, to the Subsystem, the
Facade is yet another
        // client, and it's not a part of the Subsystem.
        public class Subsystem1
        {
            public string operation1()
            {
                return "Subsystem1: Ready!\n";
            }

            public string operationN()
            {
                return "Subsystem1: Go!\n";
            }
        }
```

```
        // Some facades can work with multiple subsystems
at the same time.
        public class Subsystem2
        {
            public string operation1()
            {
                return "Subsystem2: Get ready!\n";
            }

            public string operationZ()
            {
                return "Subsystem2: Fire!\n";
            }
        }


        class Client
        {
            // The client code works with complex
subsystems through a simple
            // interface provided by the Facade. When a
facade manages the lifecycle
            // of the subsystem, the client might not
even know about the existence
            // of the subsystem. This approach lets you
keep the complexity under
            // control.
            public static void ClientCode(Facade facade)
            {
                Console.Write(facade.Operation());
```

```
            }
        }

        class Program
        {
            static void Main(string[] args)
            {
                // The client code may have some of the
subsystem's objects already
                // created. In this case, it might be
worthwhile to initialize the
                // Facade with these objects instead of
letting the Facade create
                // new instances.
                Subsystem1 subsystem1 = new Subsystem1();
                Subsystem2 subsystem2 = new Subsystem2();
                Facade facade = new Facade(subsystem1,
subsystem2);
                Client.ClientCode(facade);
            }
        }
    }
```

Якщо програма створена без помилок, отримаємо:

```
Facade initializes subsystems:
Subsystem1: Ready!
Subsystem2: Get ready!
Facade orders subsystems to perform the action:
Subsystem1: Go!
```

```
Subsystem2: Fire!
```

ПРАКТИЧНА РОБОТА 6
ВИКОРИСТАННЯ ПАТЕРНІВ ПОВЕДІНКИ

Мета роботи: Створити програму на основі патерну поведінки Спостерігач.

Спостерігач — це поведінковий патерн, який дозволяє об’єктам повідомляти інші об’єкти про зміни свого стану, рис. 6.1.

При цьому спостерігачі можуть вільно підписуватися і відписуватись від цих повідомлень.

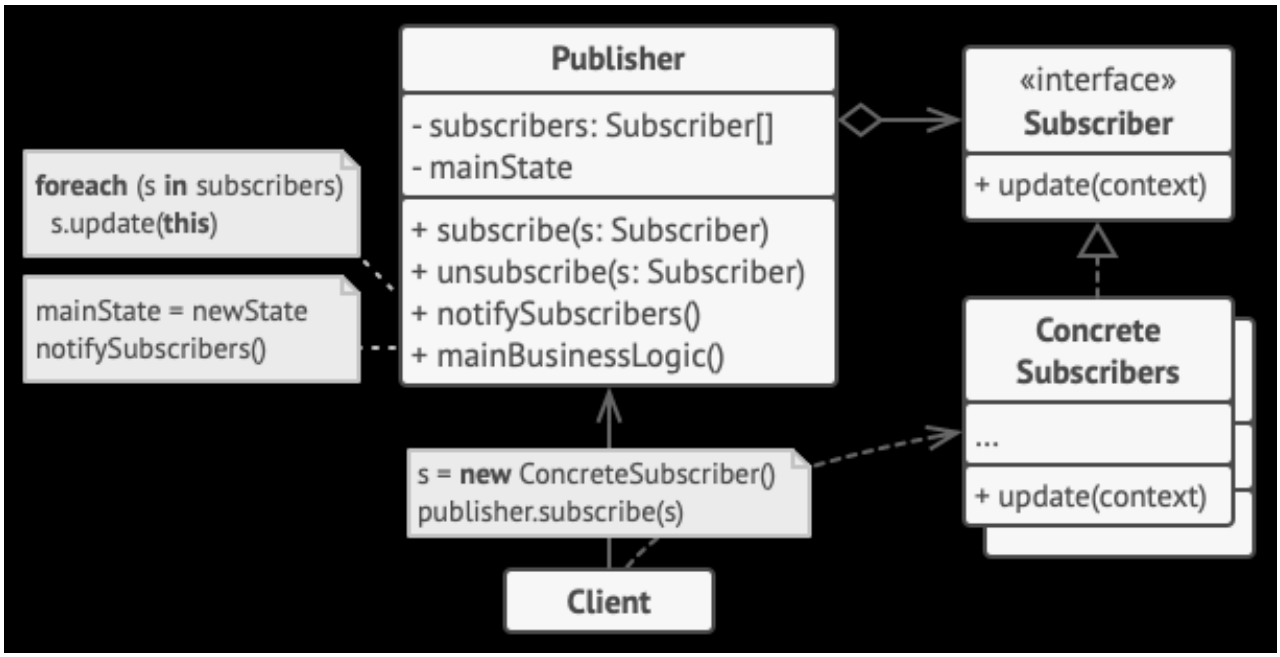

Рисунок 6.1 – Діаграма паттерну "Спостерігач "

Концепт реалізації взято з ресурсу https://refactoring.guru/ru/design-patterns/observer

```
using System;
```

```
    using System.Collections.Generic;
    using System.Threading;

    namespace
RefactoringGuru.DesignPatterns.Observer.Conceptual
    {
        public interface IObserver
        {
            // Receive update from subject
            void Update(ISubject subject);
        }

        public interface ISubject
        {
            // Attach an observer to the subject.
            void Attach(IObserver observer);

            // Detach an observer from the subject.
            void Detach(IObserver observer);

            // Notify all observers about an event.
            void Notify();
        }

        // The Subject owns some important state and
notifies observers when the
        // state changes.
        public class Subject : ISubject
        {
```

```
            // For the sake of simplicity, the Subject's
state, essential to all
            // subscribers, is stored in this variable.
            public int State { get; set; } = -0;

            // List of subscribers. In real life, the
list of subscribers can be
            // stored more comprehensively (categorized
by event type, etc.).
            private List<IObserver> _observers = new
List<IObserver>();

            // The subscription management methods.
            public void Attach(IObserver observer)
            {
                Console.WriteLine("Subject: Attached an
observer.");
                this._observers.Add(observer);
            }

            public void Detach(IObserver observer)
            {
                this._observers.Remove(observer);
                Console.WriteLine("Subject: Detached an
observer.");
            }

            // Trigger an update in each subscriber.
            public void Notify()
            {
```

```
            Console.WriteLine("Subject: Notifying
observers...");

            foreach (var observer in _observers)
            {
                observer.Update(this);
            }
        }

        // Usually, the subscription logic is only a
fraction of what a Subject
        // can really do. Subjects commonly hold some
important business logic,
        // that triggers a notification method
whenever something important is
        // about to happen (or after it).
        public void SomeBusinessLogic()
        {
            Console.WriteLine("\nSubject: I'm doing
something important.");
            this.State = new Random().Next(0, 10);

            Thread.Sleep(15);

            Console.WriteLine("Subject: My state has
just changed to: " + this.State);
            this.Notify();
        }
    }
```

```
        // Concrete Observers react to the updates issued
by the Subject they had
        // been attached to.
        class ConcreteObserverA : IObserver
        {
            public void Update(ISubject subject)
            {
                if ((subject as Subject).State < 3)
                {
                    Console.WriteLine("ConcreteObserverA:
Reacted to the event.");
                }
            }
        }

        class ConcreteObserverB : IObserver
        {
            public void Update(ISubject subject)
            {
                if ((subject as Subject).State == 0 ||
(subject as Subject).State >= 2)
                {
                    Console.WriteLine("ConcreteObserverB:
Reacted to the event.");
                }
            }
        }

        class Program
        {
```

```
        static void Main(string[] args)
        {
            // The client code.
            var subject = new Subject();
            var observerA = new ConcreteObserverA();
            subject.Attach(observerA);

            var observerB = new ConcreteObserverB();
            subject.Attach(observerB);

            subject.SomeBusinessLogic();
            subject.SomeBusinessLogic();

            subject.Detach(observerB);

            subject.SomeBusinessLogic();
        }
    }
}
```

Якщо програма створена без помилок, отримаємо:

```
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 2
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
```

```
ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.
Subject: My state has just changed to: 1
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
Subject: Detached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 5
Subject: Notifying observers...
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Повний посібник з C# 10 та .NET 6 [Електронний ресурс] // Krypton. – 2022. – Режим доступу до ресурсу: https://krypton.com.ua/tutorial/ci-10-net-6/.

2. Паттерни проектування у C# та .NET [Електронний ресурс] // Krypton. – 2022. – Режим доступу до ресурсу: https://krypton.com.ua/tutorial/ci-10-net-6/.

3. C# практичні / http://www.e-helper.com.ua/c_sharp_programmind_labs

4. C# лекції / http://www.e-helper.com.ua/c_sharp_programming_lections

5. Підручник з Umbrello UML Modeller https://docs.kde.org/trunk5/uk/umbrello/umbrello/index.html

6. Booch G. Rumbaugh J, Jacobson I. The Unified Modeling Language User Guide (Object Technology Series): 2nd Edition, Addison-Wesley Professional, 2005, 494 p.

7. Troelsen A. Japikse P. C# 9 .NET 5: F : 10 , A , 2021, 1411 p.

ЗМІСТ

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ

до практичних робіт з курсу «Об'єктно-орієнтоване програмування»

для студентів спеціальності 174 **–** Автоматизація, комп'ютерно-інтегровані технології та робототехніка

Укладачі: ПУГАНОВСЬКИЙ Олег Валентинович
ШУТИНСЬКИЙ Олексій Григорович

Відповідальний за випуск : О.М. Дзевочко
Роботу до видання рекомендував: І.Л. Красніков

Редактор

План 2024 р., поз .Підп. до друку . Формат 60х84 1/16.
Папір офсет-ний. ГарнітураTimes.
Ум. друк. арк. 3. Наклад 25 прим. Зам. №__. Ціна договірна.

Видавець НТУ"ХПІ", 61002, Харків вул. Кипичова, 2
Свідоцтво про державну реєстрацію ДК № 5478 от 21.08.2017 г.

Надруковано з готового оригінал-макету у друкарні ФОП В.В. Петров Єдиний державний реєстр юридичних осіб та фізичних осіб – підпри-ємців.
Запис №24800000000106167 від 08.01.2009 р.
61144, м. Харків, вул. Гв. Широнінців, 79в, к. 137, тел. (057)78-17-137.
e-mail: bookfabrik@mail.ua