МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних 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 лабораторних 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 лабораторних робіт з курсу «Об’єктно-орієнтоване програмування» для студентів спеціальності 174 – Автоматизація, комп’ютерно-інтегровані технології та робототехніка усіх форм навчання /уклад. О. В. Пугановський, С. Д. Деменкова – Харків: НТУ “ХПІ”, 2024. – 56 с.

Укладачі: О. В. Пугановський
С. Д. Деменкова

Рецензент І. Л.Красников

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. Вказівки містять мінімальний набір теоретичних відомостей тому перед виконанням лабораторних робіт рекомендується вивчити відповідні розділи з лекційного матеріалу. На виконання робіт відводиться від 2 до 4 годин часу при виконанні у аудиторії. Лабораторні роботи є обов'язковими для успішного завершення курсу «Об'єктно-орієнтоване програмування».

Для виконання робіт потрібно мати пердвстановлене 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.

Результат виконання лабораторної роботи – відлагоджена програма. Рекомендується але не є обов'язковим використовувати коментарі або текстовий опис. Зарахування лабораторної роботи – за результатами опитування на теоретичні знання та практичні навички за темою роботи.

ЛАБОРОТОРНА РОБОТА 1
ВИКОРИСТАННЯ СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ НАПИСАННЯ КОДУ ПРОГРАМ

Мета роботи: перевірити придатність систем штучного інтелекту (ШІ) для створення програмних кодів.

1.1 Загальні відомості

Ця технологія відкриває перед програмістами широкий спектр можливостей, зробивши процес розробки та налагодження коду більш ефективним та зручним. Ось кілька способів, як чат-бот може бути використаний у програмуванні:

Розв'язання алгоритмічних задач. Якщо ви зіткнулися зі складним завданням або важко знайти оптимальне рішення, ви можете звернутися до *ChatGPT* з питанням про підходи або алгоритми, які можуть бути застосовані до вашої проблеми.

Написання коду. Ви можете просто описати свою ідею природною мовою, і чат-бот надасть відповідний програмний код. Це особливо зручно для написання простих або стандартних фрагментів коду, що дозволяє заощадити час та зусилля при створенні програм або веб-сайтів.

Швидкий пошук інформації. Коли у вас виникають питання про мови програмування, специфікації API, можливі рішення для певного завдання або інші технічні питання, чат-бот також може бути вашим швидким і надійним джерелом інформації. Просто поставте своє питання, і воно надасть вам актуальні та точні відомості, які допоможуть вам продовжити роботу без необхідності шукати інформацію в інших джерелах.

Креативне програмування. Іноді програмістам потрібно натхнення та креативні ідеї для своїх проектів. *ChatGPT* може допомогти вам згенерувати нестандартні підходи до вирішення завдання або запропонувати цікаві концепції

для вашої програми. Він здатний дати нові перспективи та допомогти збагатити вашу розробку ідеями, які ви могли б не розглянути у звичайних умовах.

Налагодження та виправлення помилок. Якщо ви зіткнулися з помилкою або багом у вашому коді, ви можете звернутися до чат-боту для отримання допомоги у його виправленні. Надайте деталі про проблему, і він намагатиметься запропонувати можливі рішення або підказки, як знайти та усунути помилку.

Однак, незважаючи на численні переваги використання *ChatGPT* у програмуванні, важливо пам'ятати, що він не замінює професійних знань та досвіду програміста. У деяких випадках він може надати неправильні чи неоптимальні рішення. Тому завжди рекомендується перевіряти згенерований код та інформацію, отриману таким чином.

1.2 Порядок виконання

Для використання ШІ можна обрати будь-яку систему з підтримкою текстового спілкування. Для лабораторної роботи буде використано *OpenAI*. Для цього потрібно

1) перейти за посиланням https://platform.openai.com.
2) Можна створити окремий акаунт або використати свій акаунт з *Google*.
3) Перейти на вкладку “*Playground*” на боковій панелі:

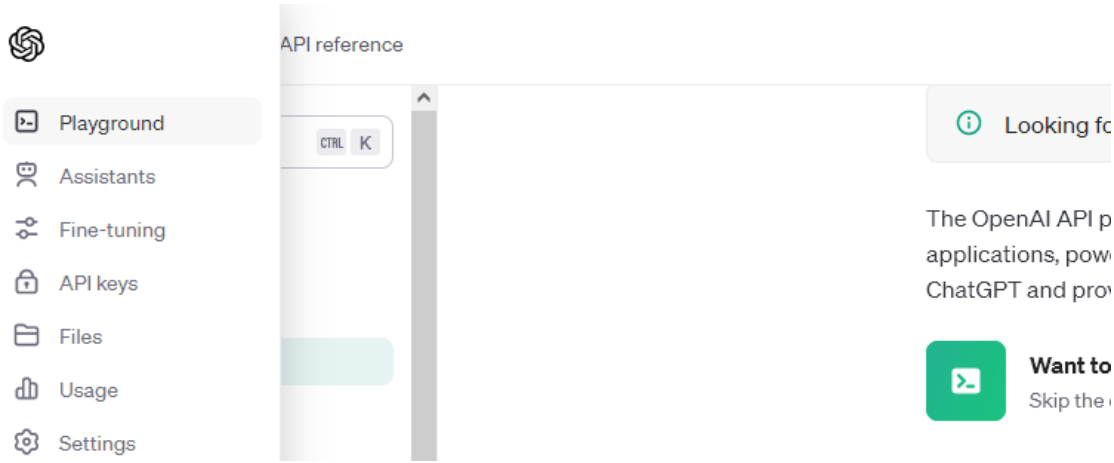

Рисунок 1.1 – Вікно *OpenAI*.

4) У вікні, що відкриється вводити запити.

Для тестування, використайте прості запити. Наприклад, *«calculate the sum 5+2»*.

Подальші дії з ШІ.

1) Попросити написати код програми підсумовування двох чисел. Мову програмування вказуйте ту, яку ви вивчали та знаєте. Протестувати код.
2) Скласти запит на створення більш складних програм. Наприклад, сортування, пошуку, обчислення факторіалу. Протестувати код.
3) Задати завдання, що потребують знання синтаксису та логіки.

Наприклад:

«Щоб нагодувати 4 коня і 12 корів, потрібно 120 кг сіна на один день. А щоб нагодувати 3 коня і 20 корів, потрібно 167 кг сіна на один день. Знайти, скільки потрібно сіна на один день, щоб нагодувати 1 коня і скільки потрібно сіна, щоб нагодувати 1 корову.»

Після відповіді ШІ, попросити написати код для цих обчислень. Протестувати код.

Оформити звіт, що містить запити і відповіді ШІ та фрагменти коду.

Контрольні запитання

1. Які системи ШІ ви знаєте?
2. Які функції може виконувати *ChatGPT*?
3. Ваше враження від спілкування з *OpenAI*.
4. Наскільки коректний та оптимальний код ви отримали?

ЛАБОРОТОРНА РОБОТА 2
МЕТОДИ КОРИСТУВАЧА

Мета роботи: Отримати практичні навички створення і використання методів користувача.

2.1 Загальні відомості

Загальне визначення методів має такий вигляд:

[модифікатори] тип_поверненого_значення назва_методу ([параметри])

{…// тіло методу }

Для виклику методу вказується його ім'я, після якого в дужках йдуть значення для його параметрів (якщо метод приймає параметри).

назва_методу (значення_для_параметрів_методу);

Параметри дозволяють передати в метод деякі вхідні дані. Параметри визначаються через заятую в дужках після назви методу у вигляді:

тип_методу ім'я_методу (тип_параметра1 параметр1, тип_параметра2 параметр2, ...)

{…// дії методу }

Необов'язкові параметри.

За замовчуванням при виклику методу необхідно надати значення для всіх його параметрів. Але C# також дозволяє використовувати необов'язкові параметри. Для таких параметрів нам необхідно оголосити значення за замов-

чуванням. Також слід враховувати, що після необов'язкових параметрів усі наступні параметри також мають бути необов'язковими:

```
    void PrintPerson(string name, int age = 1, string company = "Undefined")
    {
        Console.WriteLine($"Ім'я: {ім'я} Вік: {стаж} Компанія: {компанія}");
    }
```

Повернення значення та оператор return

Для цього застосовується оператор return, після якого йде значення, що повертається:

```
return значення, що повертається
```

Перевантаження методів.

Іноді виникає необхідність створити один і той самий метод, але з різним набором параметрів. І залежно від наявних параметрів застосовувати певну версію методу. Така можливість ще називається перевантаженням методів (method overloading).

В мові C# ми можемо створювати в класі кілька методів з одним і тим самим ім'ям, але різною сигнатурою. Сигнатура складається з таких аспектів:

- Ім'я методу
- Кількість параметрів
- Типи параметрів
- Порядок параметрів
- Модифікатори параметрів

Але назви параметрів у сигнатуру НЕ входять. Наприклад, візьмемо такий метод:

```
public int Sum(int x, int y)
{    повернути x + y;}
```

У цього методу сигнатура матиме такий вигляд: `Sum(int, int)`

І перевантаження методу якраз полягає в тому, що методи мають різну сигнатуру, в якій збігається тільки назва методу. Тобто методи мають відрізнятися за:

- Кількості параметрів
- Типу параметрів
- Порядку параметрів
- Модифікаторам параметрів

2.2 Порядок виконання

Для виконання роботи. необхідно створити програму, що містить декілька методів користувача.

У якості завдання можна обрати власні приклади або приклад зазначений нижче. При вільному виборі, необхідно продемонструвати визначення та виклик методів, передавання параметрів та отримання результату роботи, перевантаження методів.

Наприклад, створити програму, що містить методи:

- для форматованого виводу результату обчислень у консоль,
- для обчислення площі трикутника за різними вхідними даними (через перевантаження),
- для обчислення периметру трикутника.

Контрольні запитання

1. Що таке метод? Який його синтаксис?
2. Як викликаютться методи з тіла програми?
3. Що таке параметри і аргументи?

4. Що таке «перевантаження» методів?
5. Як можна використовувати оператор return?

ЛАБОРОТОРНА РОБОТА 3
ДЕЛЕГАТИ

Мета роботи: навчитись створювати та використиовувати делегати у програмах, закріпити теоретичні і практичні навички використання делегатів.

3.1 Загальні відомості

Для оголошення делегата використовується ключове слово *delegate*, після якого йде тип, що повертається, назва і параметри. Наприклад:

```
delegate void Message();
```

Делегат *Message* як тип, що повертається, має тип *void* (тобто нічого не повертає) і не приймає жодних параметрів. Це означає, що цей делегат може вказувати на будь-який метод, який не приймає жодних параметрів і нічого не повертає. Виклик делегата здійснюється подібно до виклику методу.

Розглянемо застосування цього делегата:

```
delegate void Message(); // 1. Оголошуємо делегат
Message mes; // 2. Створюємо змінну делегата
mes = Hello; // 3. Присвоюємо цій змінній адресу методу
mes(); // 4. Викликаємо метод

void Hello() => Console.WriteLine("Hello !!!");
```

Відповідність методів делегату

Як було написано вище, методи відповідають делегату, якщо вони мають один і той самий тип, що повертається, і один і той самий набір параметрів. Але треба враховувати, що до уваги також беруться модифікатори *ref*, *in і out*. Наприклад, нехай у нас є делегат:

```
delegate void SomeDel(int a, double b);
```

Цьому делегату відповідає, наприклад, такий метод:

```
void SomeMethod1(int g, double n) { }
```

А такі методи НЕ відповідають. Проаналізуйте чому?

```
double SomeMethod2(int g, double n) { return g + n; }
void SomeMethod3(double n, int g) { }
void SomeMethod4(ref int g, double n) { }
void SomeMethod5(out int g, double n) { g = 6;
```

Додавання та віднімання делегатів.

Під час додавання делегатів (оператор +=) слід враховувати, що ми можемо додати посилання на один і той самий метод кілька разів, і в списку виклику делегата тоді буде кілька посилань на один і той самий метод. Відповідно під час виклику делегата доданий метод буде викликатися стільки разів, скільки його було додано:

```
Message message = Hello;
message += HowAreYou;
message += Hello;
```

```
message += Hello;

message();
```

Подібним чином ми можемо видаляти методи з делегата за допомогою операцій -=:

```
Message? message = Hello;
message += HowAreYou;
message(); // викликаються всі методи з message
message -= HowAreYou; // видаляємо метод HowAreYou

if (message != null) message(); // викликається метод Hello
```

3.2 Порядок виконання

Проаналізуйте приклад. У обраному середовищі програмування створити код за нижче наведеним прикладом. За необхідності, адаптуйте до обраної мови. Відлагодити та пояснити роботу програми.

```
Operation operation = SelectOpera-
tion(OperationType.Add);
Console.WriteLine(operation(10, 4)); // 14

operation = SelectOperation(OperationType.Subtract);
Console.WriteLine(operation(10, 4)); // 6

operation = SelectOperation(OperationType.Multiply);
Console.WriteLine(operation(10, 4)); // 40
```

```
operation SelectOperation(OperationType opType)
{
    switch (opType)
    {
        case OperationType.Add: return Add;
        case OperationType.Subtract: return Subtract;
        за замовчуванням: повернути Multiply;
    }
}

int Add(int x, int y) => x + y;
int Subtract(int x, int y) => x - y;
int Multiply(int x, int y) => x * y;

enum OperationType
{
    Додавання, віднімання, множення
}
делегат int Operation(int x, int y);
```

Контрольні запитання

1. Відмінність делегатів від методів
2. Принципи використання делегатів.
3. Як поєднати кілька методів у одному делегаті?
4. Як видалити методи з делегату?
5. Як використовують лямбда вирази?

ЛАБОРОТОРНА РОБОТА 4
СТВОРЕННЯ ТА ВИКОРИСТАННЯ КЛАСІВ

Мета роботи: закріпити знання і отримати практичні навички у створенні та використанні класів.

4.1 Загальні відомості

Клас – базова сутність ООП. Клас представляє новий тип, що визначається користувачем. Клас визначається за допомогою ключового слова *class*:

```
class назва_класу
{
//Вміст класу
}
```

Після слова *class* йде ім'я класу і далі у фігурних дужках йде власне вміст класу. Наприклад, визначимо у файлі *Program.cs* клас *Person*, який представлятиме людину:

```
class Person
{      }
```

Поля та методи класу

Клас може зберігати деякі дані. Для зберігання даних у класі застосовуються поля. Насправді поля класу – це змінні, визначені лише на рівні класу.

Крім того, клас може визначати деяку поведінку або дії, що виконуються. Для визначення поведінки у класі застосовуються методи.

Наприклад, поля та методи класу Person :

```
classPerson
{
  publicstring name = "Undefined"; // ім'я
  publicint age; // вік

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

Створення об'єкта класу.

Після визначення класу ми можемо створювати об'єкти. Для створення об'єкту використовуються конструктори. По суті, конструктори представляють спеціальні методи, які називаються так само як і клас, і які викликаються при створенні нового об'єкта класу і виконують ініціалізацію об'єкта. Загальний синтаксис виклику конструктора:

new конструктор_класу (параметри_конструктора);

Спочатку йде оператор *new*, що виділяє пам'ять для об'єкта, а після нього йде виклик конструктора.

Конструктор за замовчуванням.

Якщо в класі не визначено жодного конструктора (як у випадку з нашим класом *Person*), для цього класу автоматично створюється порожній конструктор за замовчуванням, який не приймає жодних параметрів.

Створення конструкторів.

На рівні коду конструктор представляє метод, який називається на ім'я класу, який може мати параметри, але для нього не треба визначати тип, що повертається. Наприклад, визначимо у класі Person найпростіший конструктор:

Приклад:

```
Person tom = newPerson(); // Створення об'єкта класу Person

tom.Print(); // Ім'я: Tom Вік: 37

classPerson
{
  publicstring name;
  publicint age;
  publicPerson()
  {
     Console.WriteLine("Створення об'єкта Person");
     name = "Tom";
     age = 37;
  }
  publicvoid Print()
  {
  Console.WriteLine($"Ім'я: {name} Вік: {age}");
}
}
```

4.2 Порядок виконання

Проаналізувати та протестувати код, наведений нижче.

Приклад 1.

```
Person tom = new Person();
Person bob = new Person("Bob");
Person sam = new Person("Sam", 25);
tom.Print();
```

```
bob.Print();
sam.Print();

classPerson
{
  public string name;
  public int age;

  public Person() { name = "Невідомо"; age = 18; }
  public Person(string n) {name = n; age = 18; }
  public Person(string n, int a) {name = n; age =
a;}

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

Приклад 2.

```
Person sam = new ("Sam", 25);
sam.Print();

classPerson
{
  public string name;
  public int age;
  public Person() { name = "Невідомо"; age = 18; }
```

```
        public Person(string name) {this.name=name; age =
18; }
        public Person(string name, int age)
        {
          this.name = name;
          this.age = age;
        }
        public void Print() => Console.WriteLine($"Ім'я:
{name} Вік: {age}");
    }
```

Представити та пояснити результати виконання.

Контрольні запитання

1. Що таке «об’єкт»?
2. З яких структурних елементів складається клас?
3. Призначення конструкторів класу.
4. Чи є відмінність між методами і конструкторами?
5. Яку роль відіграє показчик *this ?*
6. Що таке область видимості змінних?

ЛАБОРОТОРНА РОБОТА 5
БІБЛІОТЕКИ КЛАСІВ

Мета роботи: навчитись створювати та використовувати бібліотеки класів, закріпити знання про модифікатори доступу.

5.1 Загальні відомості

Бібліотеки можуть бути створені двома шляхами. Перший – створення бібліотеки як окремої одиниці. Другий – додавання бібліотеки в проєкт, що розробляється. Порядок створення бібліотеки класів в проєкті наступний.

1) У структурі проекту натиснемо правою кнопкою на назву рішення і далі в контекстному меню виберемо *Add -> New Project...*

2) Далі у списку шаблонів проекту знайдемо пункт *Class Library*:

3) Потім дамо новому проекту якусь назву, наприклад, *MyLib*

Після створення цього проекту до рішення буде додано новий проект, у моєму випадку з назвою *MyLib*. За замовчуванням, новий проект має один порожній клас *Class1* у файлі *Class1.cs*. Ми можемо видалити або перейменувати цей файл, як нам більше подобається.

4) скомпілюємо бібліотеку класів. Для цього натиснемо правою кнопкою на проект бібліотеки класів та в контекстному меню виберемо пункт *Rebuild*.

Після компіляції бібліотеки класів у папці проекту у каталозі *bin/Debug/net6.0* ми зможемо знайти скомпільований файл *dll* (*MyLib.dll*).

5) Підключимо його до основного проекту. Для цього в основному проекті натиснемо правою кнопкою на вузол *Dependencies* і в контекстному меню виберемо пункт *Add Project Reference...:*

Далі нам відкриється вікно додавання бібліотек. У цьому вікні виберемо пункт *Solution*, який дозволяє побачити всі бібліотеки класів із проектів поточного рішення, поставимо позначку поряд із нашою бібліотекою та натиснемо на кнопку *OK*:

Якщо бібліотека була створена як окрема одиниця, то за допомогою кнопки *Browse* ми можемо знайти місце розташування файлу *dll* і також його підключити.

6) У секції *using*, прописуємо рядок: *using MyLib*;

Усі поля, способи та інші компоненти класу мають модифікатори доступу. Модифікатори доступу дозволяють встановити допустиму область видимості для компонентів класу. Тобто модифікатори доступу визначають контекст, у якому можна використовувати цю змінну чи спосіб.

У мові C# застосовуються такі модифікатори доступу:

- *private*: закритий або приватний компонент класу чи структури Приватний компонент доступний лише у межах свого класу чи структури.
- *private protected*: компонент класу доступний з будь-якого місця у своєму класі або похідних класах, які визначені в тій же збірці.
- *file*: доданий у версії C# 11 і застосовується до типів, наприклад, класів та структур. Клас або структура з таким модифікатором доступні тільки з поточного файлу коду.
- *protected*: такий компонент класу доступний з будь-якого місця у своєму класі або похідних класах. При цьому похідні класи можуть розміщуватись в інших зборках.
- *internal*: компоненти класу або структури доступні з будь-якого місця коду в тій самій збірці, однак він недоступний для інших програм та збірок.
- *protected internal*: поєднує функціонал двох модифікаторів protected та internal. Такий компонент класу доступний з будь-якого місця в поточній збірці та похідних класів, які можуть розташовуватися в інших збірках.
- *public*: публічний, загальнодоступний компонент класу чи структури. Такий компонент доступний з будь-якого місця в коді, а також інших програм і збірок.

5.2 Порядок виконання

1) Створити проєкт, що містить бібліотеку з класами описаними нижче.

```
class State
{
  // однаково, що private string defaultVar;
  string defaultVar = "default";
  // поле доступне лише з поточного класу
  private string privateVar = "private";
  // Доступно з поточного класу та похідних класів,
  які визначені в цьому ж проекті
  protected private string protectedPrivateVar =
  "protected private";
  // доступно з поточного класу та похідних класів
  protected string protectedVar = "protected";
  // Доступно в будь-якому місці поточного проекту
  internal string internalVar = "internal";
  // Доступно в будь-якому місці поточного проекту
  та з класів-спадкоємців в інших проектах
  protected internal string protectedInternalVar =
  "protected internal";
  // Доступно в будь-якому місці програми, а також
  для інших програм і збірок
  public string publicVar = "public";

  // За замовчуванням має модифікатор private
  voidPrint() => Console.WriteLine(defaultVar);

  // метод доступний лише з поточного класу
```

```
    private void PrintPrivate() =>
    Console.WriteLine(privateVar);

    // доступний з поточного класу та похідних класів,
    які визначені в цьому ж проекті
    protected private void PrintProtectedPrivate() =>
    Console.WriteLine(protectedPrivateVar);

    // доступний з поточного класу та похідних класів
    protected void PrintProtected() =>
    Console.WriteLine(protectedVar);

    // доступний у будь-якому місці поточного проекту
    internal void PrintInternal() =>
    Console.WriteLine(internalVar);

    // доступний у будь-якому місці поточного проекту
    та з класів-спадкоємців в інших проектах
    protected internal void PrintProtectedInternal()
    => Console.WriteLine(protectedInternalVar);

    // доступний у будь-якому місці програми, а також
    для інших програм та збірок
    public void PrintPublic() =>
    Console.WriteLine(publicVar);
}
```

2) В основній частині використати нижче наведений код. Надати аналіз роботи коду. Пояснити, як впливають модифікатори на взаємодію між класами.

Змінити модифікатори та надати аналіз результатів (або помилок) виконання. Які модифікації потрібно зробити для коректної роботи?

```
class StateConsumer
{
  public void PrintState()
  {
    State state = newState();

    // звернутися до змінної defaultVar у нас не
    вийде,
    // Так як вона має модифікатор private і клас
    StateConsumer її не бачить
    Console.WriteLine(state.defaultVar); //Помилка,
    отримати доступ не можна

    // Те ж саме стосується і змінної privatevar
    Console.WriteLine(state.privateVar); // Помилка,
    отримати доступ не можна

    // звернутися до змінної protectedPrivateVar не
    вийде,
    // Оскільки клас StateConsumer перестав бути
    класом-спадкоємцем класу State
    Console.WriteLine(state.protectedPrivateVar); //
    Помилка, отримати доступ не можна

    // звернутися до змінної protectedVar теж не
    вийде,
```

```
        // Оскільки клас StateConsumer перестав бути
        класом-спадкоємцем класу State
        Console.WriteLine(state.protectedVar); // Помил-
        ка, отримати доступ не можна

        // Змінна internalVar з модифікатором internal
        доступна з будь-якого місця поточного проекту
        // тому можна отримати чи змінити її значення
        Console.WriteLine(state.internalVar);

        // змінна protectedInternalVar також доступна з
        будь-якого місця поточного проекту
        Console.WriteLine(state.protectedInternalVar);

        // змінна publicVar загальнодоступна
        Console.WriteLine(state.publicVar);
    }
}
```

Контрольні запитання

1) Як створюється бібліотека класів?
2) Як отримати доступ до елементів бібліотеки?
3) За яких умов можна використати сторонні бібліотеки?
4) Які є модифікатори доступу до структурних елементів програми?
5) Які конфлікти є у наведених прикладах та як ви їх прибрали?

ЛАБОРОТОРНА РОБОТА 6
СПАДКУВАННЯ КЛАСІВ

Мета роботи: отримати практичні навички у використанні принципу спадкування

6.1 Загальні відомості

Спадкування (*inheritance*) є одним із ключових моментів ООП. Завдяки успадкування один клас може успадкувати функціональність іншого класу.

За умовчанням усі класи успадковуються від базового класу *Object*, навіть якщо ми явно не встановлюємо спадкування. Тому вище певні класи *Person* і *Employee* крім своїх методів, також матимуть і методи класу *Object*: *ToString*(), *Equals*(), *GetHashCode*() і *GetType*().

Усі класи за умовчанням можуть успадковуватись. Однак тут є низка обмежень:

- Якщо не підтримується множинне спадкування, клас може успадковуватися тільки від одного класу.
- При створенні похідного класу треба враховувати тип доступу до базового класу - тип доступу до похідного класу повинен бути таким самим, як і у базового класу, або більш суворим. Тобто, якщо базовий клас має тип доступу internal, то похідний клас може мати тип доступу internal чи private, але з public.

Проте слід враховувати, що й базовий і похідний клас перебувають у різних складаннях (проектах), то цьому випадку похідний клас може успадковувати лише від класу, що має модифікатор *public*.

- Якщо клас оголошений з модифікатором *sealed*, то цього класу не можна успадковувати і створювати похідні класи. Наприклад, наступний клас не допускає створення спадкоємців:

```
sealed class Admin
```

```
{
}
```

Приклад спадкування:

```
classPerson
{
privatestring _name = "";

publicstring Name
{
get{return_name; }
set{_name = value; }
}
publicvoid Print()
{
Console.WriteLine(Name);
}
}
classEmployee : Person
{
}
```

Для класу Employee базовим є Person, і тому клас Employee успадковує ті самі властивості, методи, поля, які є у класі Person. Єдине, що не передається під час наслідування, це конструктори базового класу з параметрами.

Таким чином, успадкування реалізує відношення is-a (є), об'єкт класу Employee також є об'єктом класу Person. І якщо в базовому класі не визначено конструктора за замовчуванням без параметрів, а лише конструктори з параметрами (як у випадку з базовим класом Person), то у похідному класі ми обов'язково повинні викликати один з цих конструкторів через ключове слово base.

```
classEmployee : Person
{
publicstring Company { get; set; } = "";
publicEmployee(string name, string company)
: base(name)
{
Company = company;
}
}
```

6.2 Порядок виконання

Записати, відлагодити та проаналізувати роботу елементів програми, що наведено нижче. Додати структурні елементи за необхідності.

1)

```
class Person
{
stringname;
intage;

publicPerson(string name)
{
this.name = name;
Console.WriteLine("Person(string name)");
}
publicPerson(string name, int age) : this(name)
{
this.age = age;
Console.WriteLine("Person(string name, int age)");
```

```
    }
    }
    class Employee : Person
    {
    string company;

    public Employee(string name, int age, string company)
: base(name, age)
    {
    this.company = company;
    Console.WriteLine("Employee(string name, int age,
string company)");
    }
    }
```

Контрольні запитання

1) Що дає механізм спадкування з огляду на розробку програм?
2) Як реалізується послідовність конструкторів у похідних класах?
3) Що дає ключове слово **base**?
4) Основні принципи створення похідних класів.

ЛАБОРОТОРНА РОБОТА 7
ІНКАПСУЛЯЦІЯ ТА ПОЛІМОРФІЗМ

Мета роботи: закріпити знання та отримати практичні навички з використання механізму інкапсуляції та поліморфізму.

7.1 Загальні відомості

Інкапсуляція – це один із основних принципів об'єктно-орієнтованого програмування (ООП). Вона є механізм, що дозволяє об'єднати дані та методи всередині об'єкта, приховуючи їх від прямого доступу ззовні. Це дозволяє створити своєрідну «підопераційну систему» для об'єкта, яка визначає, які дані об'єкт може містити та які операції він може виконувати.

Інкапсуляція безпосередньо пов’язана з двома іншими принципами – поліморфізмом та спадкуванням. Реалізація цього принципу відбувається в основному двома шляхами. Перший – це використання модифікаторів доступу. Другий – використання властивостей для компонентів класу.

Визначення властивостей

Стандартний опис властивості має наступний синтаксис:

```
[модифікатори] тип_властивості назва_властивості
{
get{ дії, що виконуються при отриманні значення властивості}
set{ дії, що виконуються при встановленні значення властивості}
}
```

У блоці set встановлюється значення поля. У цьому блоці за допомогою параметра value ми можемо отримати значення, яке передано властивості.

Блоки get і set ще називаються акссесорами чи методами доступу (до значення властивості), і навіть геттером і сеттером.

Властивості тільки для читання та запису

Блоки set і get не обов'язково одночасно мають бути присутніми у властивості. Якщо властивість визначає лише блок get, то така властивість доступна лише для читання – ми можемо отримати його значення, але не встановити.

І навпаки, якщо властивість має тільки блок set, тоді ця властивість доступна тільки для запису - можна встановити значення, але не можна отримати.

Обчислювані властивості.

Властивості необов'язково пов'язані з певною змінною. Вони можуть обчислюватися на основі різних виразів. Наприклад:

```
Person tom = new ("Tom", "Smith");
Console.WriteLine(tom.Name); // Tom Smith
classPerson
{
stringfirstName;
stringlastName;
publicstring Name
{
get{ return $"{firstName} {lastName}"; }
}
publicPerson(string firstName, string lastName)
{
this.firstName = firstName;
this.lastName = lastName;
}
}
```

Можна застосовувати модифікатори доступу не тільки до всієї властивості, але і до окремих блоків get і set

При використанні модифікаторів у властивостях слід враховувати низку обмежень:

1. Модифікатор для блоку set або get можна встановити, якщо властивість має обидва блоки (і set, і get)
2. Тільки один блок set або get може мати модифікатор доступу, але не обидва відразу
3. Модифікатор доступу блоку set або get повинен бути більшим, ніж модифікатор доступу властивості. Наприклад, якщо властивість має модифікатор public, то блок set/get може мати тільки модифікатори protected internal, internal, protected, private protected і private

Автоматичні властивості

Властивості керують доступом до полів класу. Однак якщо у нас з десяток і більше полів, то визначати кожне поле і писати для нього однотипну властивість було б стомлюючим. Тому в .NET було додано автоматичні властивості. Вони мають скорочене оголошення:

```
classPerson
{
publicstring Name { get; set; }
publicint Age {get; set; }

publicPerson(string name, int age)
{
Name = name;
Age = age;
}
}
```

Блок init

Починаючи з версії C# 9.0, сеттери у властивостях можуть визначатися за допомогою оператора init (від слова "ініціалізація" - це блок init покликаний ініціалізувати властивість). Для встановлення значень властивостей з init можна використовувати лише ініціалізатор, або конструктор, або при оголошенні вказати для нього значення. Після ініціалізації значень подібних властивостей їх значення не можна змінити - вони доступні тільки для читання. У цьому плані init-властивості зближуються із властивостями для читання. Різниця полягає в тому, що init-властивості ми можемо встановити в ініціалізаторі (властивості для читання встановити в ініціалізаторі не можна).

Поліморфізм (polymorphism) – можливість однієї й тієї ж змінної у різні моменти виконання програми позначати об'єкти різних типів (класів), які стосуються одному базовому типу (класу чи інтерфейсу).

Реалізація поліморфізму відбувається в основному через використання абстрактних і віртуальних методів та класів а також інтерфейсів.

Абстракція клас схожий на звичайний клас. Він може мати змінні, методи, конструктори, властивості. Єдине, що для визначення абстрактних класів використовується ключове слово abstract

```
abstractclass Transport
{
publicvoid Move()
{
Console.WriteLine("Транспортний засіб рухається");
}
}
```

Головна відмінність – при створенні екземпляру класу не можна використовувати конструктори з базового абстрактного класу але буде успадкований весь доступний функціонал. Крім цього, можна додавати свої елементи – поля і методи. Саме це дозволяє отримати кілька класів – нащадків з одного абстракт-

ного класу, що матимуть різний функціонал та різну реалізацію одних і тих методів. Наприклад.

```
Transport car = new Car("машина");
Transport ship = new Ship("корабель");
Transport aircraft = new Aircraft("літак");

car.Move(); // машина рухається
ship.Move(); // корабель рухається
aircraft.Move(); // Літак рухається
abstract class Transport
{
public string Name { get; }
// конструктор абстрактного класу Transport
public Transport(string name)
{
Name = name;
}
public void Move() =>Console.WriteLine($"{Name} рухається");
}
// клас корабля
class Ship : Transport
{
// Викликаємо конструктор базового класу
public Ship(string name) : base(name) { }
}
// клас літака
class Aircraft : Transport
{
```

```
public Aircraft(string name) : base(name) { }
}
// клас машини
class Car : Transport
{
public Car(string name) : base(name) { }
}
```

7.2 Порядок виконання

Записати та відлагодити наступний код. Пояснити принципи роботи програми. Доповнити своїми визначеннями для трикутника та ромба.

```
// абстрактний клас фігури
abstract class Shape
{
// абстрактний метод отримання периметра
public abstract double GetPerimeter();
// абстрактний метод отримання площі
public abstract double GetArea();
}
// Похідний клас прямокутника
class Rectangle : Shape
{
public float Width { get; set; }
public float Height {get; set; }

// Перевизначення отримання периметра
public override double GetPerimeter() => Width * 2 +
Height * 2;
// Переопреляння отримання площі
```

```
    public override double GetArea() => Width * Height;
    }
    // Похідний клас кола
    class Circle: Shape
    {
    public double Radius {get; set; }

    // Перевизначення отримання периметра
    public override double GetPerimeter() => Radius * 2 *
3.14;
    // Перевизначення отримання площі
    public override double GetArea() => Radius * Radius *
3.14;
    }
```

Контрольні запитання

1. Що таке інкапсуляція?
2. Надайте визначення поліморфізму.
3. Як реалізується інкапсуляція?
4. Що таке абстрактний клас?
5. Призначення команди *override*.

ЛАБОРОТОРНА РОБОТА 8
ВИКОРИСТАННЯ PATTERN MATCHING

Мета роботи: засвоїти поняття «pattern matching» та навчитись використовувати цей принцип при написанні програм.

8.1 Основні відомості про Pattern matching

Pattern matching – принцип за яким виконують співставлення деякого значення з певним шаблоном. За виконанням умови відбувається виконання послідовності дій. Мовою C# передбачено виконання різних типів співставлянь.

Відповідно, можна виділити окремі принципи:

- type pattern – паттерн типів;
- constant pattern - зіставлення з деякою константою;
- паттерн властивостей дає змогу порівнювати зі значеннями певних властивостей об'єкта;
- патерни кортежів;
- позиційний патерн застосовується до типу, у якого визначено метод деконструктора.

Наприклад, щоб перевірити відповідність одного класу іншому, можна використати type pattern. Нехай є два класи (пам’ятаємо про спадкування):

```
class Employee
{ public virtual void Work() => Console.WriteLine("Співробітник працює");}
class Manager : Employee
{  public override void Work() => Console.WriteLine("Менеджер працює");
   public bool IsOnVacation { get; set; }}
```

Щоб перевірити, чи представляє об'єкт Employee клас Manager, використаємо наступний код:

```
Employee tom = new Manager();
UseEmployee(tom); // Менеджер працює

void UseEmployee(Employee emp)
{
if (emp is Manager manager && manager.IsOnVacation == false)
   { manager.Work(); }
else
   { Console.WriteLine("Перетворення не допустиме"); }
}
```

Зверніть увагу на використання оператора «*is*».

Зіставлення з константою дуже подібне до звичайних умовних операторів але застосовується до об’єктів. Наприклад, відповідно до попереднього коду, перевірка на значення *null*:

```
Employee? bob = new Employee();
Employee? tom = null;

UseEmployee(bob);
UseEmployee(tom);

void UseEmployee(Employee? emp)
{
   if (emp is not null)
      emp.Work();
}
```

Увага! Пам’ятаємо, що оператор «?» дозволяє додати значення *null* до типів, які зазвичай не можуть приймати це значення.

Аналогічно можуть бути використані інші умовні конструкції – *switch-case* та *when*.

8.2. Порядок виконання роботи

Патерн властивостей дозволяє співставляти відповідність властивостей заданим.

1) Напишіть приклад на основі патерну властивостей для наступних умов.

Є клас, що містить три поля властивостей:

- назва виробника гаджету
- тип гаджету
- тип зарядного пристрою (або роз’єму)

Програма повинна написати, наприклад, за заданим типом гаджету, який зарядний пристрій потрібно використати.

Увага! Дотримуйтесь синтаксису. Порівняння властивостей можна описати наступним чином:

```
if(екземпляр класу is клас { властивість: "значення властивості" })
   { }
else
   { }
}
```

2) Додайте ще дві властивості на власний розсуд і напишіть програму, що порівнює одразу кілька властивостей. Наприклад, якщо це виробник «FirstDev» і модель «2005» то зарядний пристрій «USB1» а якщо виробник «FirstDev» і модель «2021», то пристрій «Lighting».

Контрольні запитання

1. Що таке Pattern matching?
2. Які принципи Pattern matching ви знаєте?
3. Як перевірити на відповідність властивості класу?
4. Як описується позиційний патерн?

ЛАБОРОТОРНА РОБОТА 9

Вирішення завдань проектування патернами, що породжують

Мета роботи: засвоїти принцип проєктування патерном, що породжує навчитись використовувати цей принцип при написанні програм.

9.1 Загальні відомості.

Прототип відносять до патернів, що породжують. Фактично, він реалізує механізм клонування об’єктів з об’єкту-прототипу. Цей патерн доцільно використовувати, коли сутність нового об’єкту визначається динамічно. Тобто під час виконання. На відміну від абстрактної фабрики, не буде створюватись нова окрема ієрархія класів з паралельної ієрархії класів. Також клонування доцільне, якщо відомо про обмежену кількість станів об’єкта. При цьому відпадає необхідність у створені нового об’єкта та його ініціалізації за допомогою конструктора. UML-діаграма Прототипу може бути представлена у вигляді як на рис. 9.1.

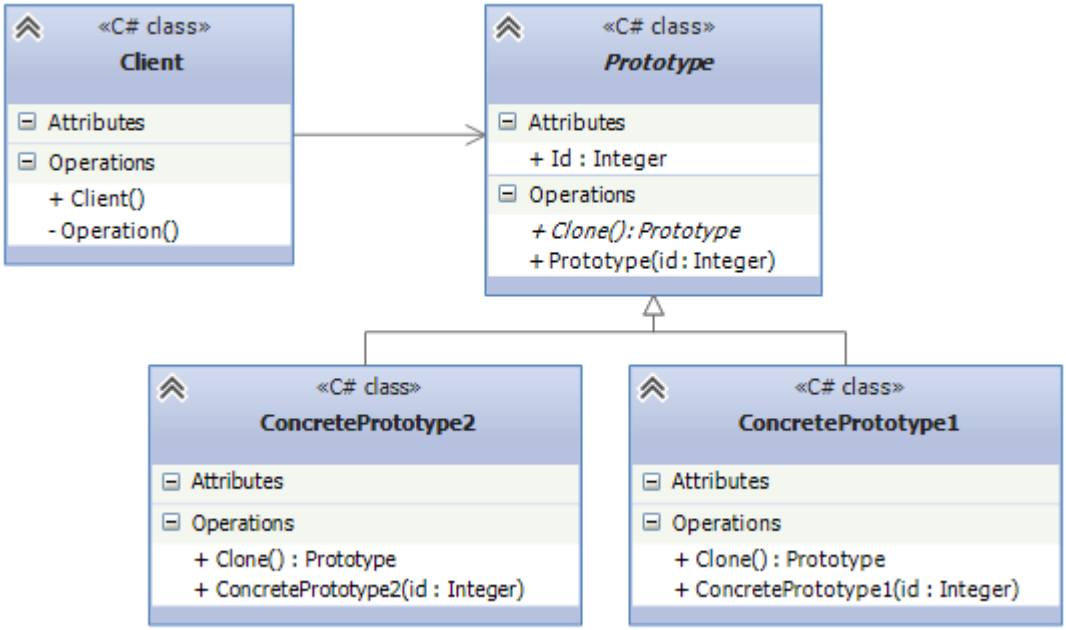

Рисунок 9.1 – UML-діаграма патерну «Прототип»

Як варіант, формальну структуру цього патерну можна описати наступним чином:

```
class Client
{
    void Operation()
    {
        Prototype prototype = new ConcretePrototype1(1);
        Prototype clone = prototype.Clone();
        prototype = new ConcretePrototype2(2);
        clone = prototype.Clone();
    }
}

abstract class Prototype
{
    public int Id {get; private set; }
```

```
    public Prototype(int id)
      {
        this.Id = id;
      }
    public abstract Prototype Clone();
}

class ConcretePrototype1 : Prototype
{
    public ConcretePrototype1(int id) : base(id)
    {}
    public override Prototype Clone()
    { return New ConcretePrototype1(Id); }
}

class ConcretePrototype2 : Prototype
{
    public ConcretePrototype2(int id) : base(id)
    {}
    public override Prototype Clone()
    { return New ConcretePrototype2(Id); }
}
```

Складові (учасники) патерну:

- *Prototype*: визначає інтерфейс для клонування самого себе, який зазвичай представляє метод *Clone*()
- *ConcretePrototype1* і *ConcretePrototype2*: конкретні реалізації прототипу. Реалізують метод *Clone*()
- *Client*: створює об'єкти прототипів за допомогою методу *Clone*()

9.2. Порядок виконання

На основі представленої інформації, реалізувати алгоритм та програму для клонування кіл і прямокутників. Для кола задають значення радіусу а для прямокутника – ширину і висоту.

Прототип реалізовано за допомогою інтерфейсу, наприклад, *IFigure:*

```
interfaceIFigure
{
    IFigure Clone();
    void GetInfo();
}
```

Контрольні запитання

1. В чому сутність патернів, що породжують?
2. Які типи патернів об’єднують в групу тих, що породжують?
3. Як функціонує патерн «Прототип»?
4. У чому відмінність між «Фабричним методом» та «Прототипом»?

ЛАБОРОТОРНА РОБОТА 10
ВИРІШЕННЯ ЗАВДАНЬ ПРОЕКТУВАННЯ ПАТЕРНАМИ ПОВЕДІНКИ

Мета роботи: засвоїти принцип проєктування патерном поведінки навчитись використовувати цей принцип при написанні програм.

10.1 Загальні відомості

Одним з патернів поведінки є «Стратегія». Паттерн Стратегія (Strategy) є шаблоном проектування, який визначає набір алгоритмів, інкапсулює кожен з них і забезпечує можливість їх взаємозамінності. Залежно від конкретної ситуації можна легко замінити один алгоритм іншим. При цьому заміна алгоритму відбувається незалежно від об'єкта, який використовує цей алгоритм. Доцільність використання зумовлена наступними чинниками:

- • Коли існує декілька родинних класів з відмінною поведінкою, можна визначити основний клас, а різні варіанти поведінки винести в окремі класи, які можуть бути легко використані за потреби.
- • У випадку потреби у виборі серед кількох варіантів алгоритмів, які можна легко змінювати залежно від умов, застосування паттерну Стратегія є відмінним варіантом.
- • Коли необхідно динамічно змінювати поведінку об'єктів під час виконання програми, використання паттерну Стратегія дозволяє здійснювати такі зміни без впливу на інші частини системи.
- • У випадку, коли клас, який використовує певну функціональність, повинен залишатися невідомим щодо її конкретної реалізації, паттерн Стратегія надає можливість відокремити інтерфейс від реалізації.

UML-діаграма «Стратегії» може бути представлена у вигляді як на рис. 10.1.

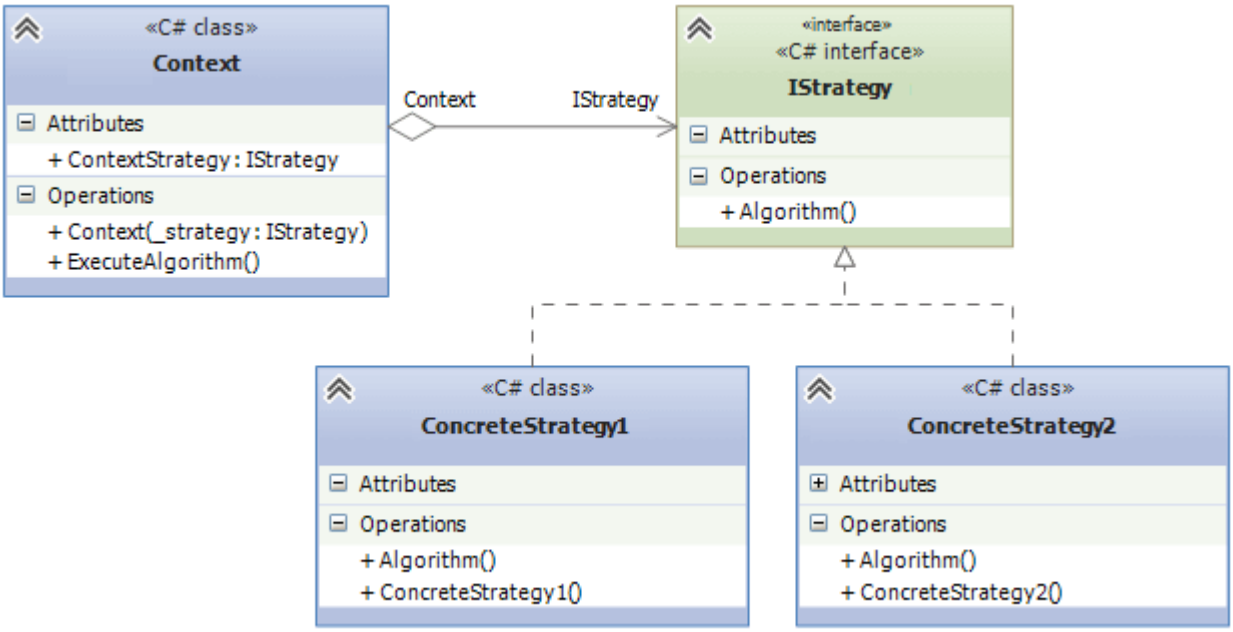

Рисунок 10.1 – UML-діаграма патерну «Стратегія»

Формальний опис можна представити у вигляді наступного коду.

```
public interface IStrategy
{ void Algorithm(); }

public class ConcreteStrategy1 : IStrategy
{
    publicvoid Algorithm()
    {}
}
public class ConcreteStrategy2 : IStrategy
{
    publicvoid Algorithm()
    {}
}
```

```
public class Context
{
public IStrategy ContextStrategy { get; set; }
public Context(IStrategy _strategy)
    {
        ContextStrategy = _strategy;
    }

publicvoid ExecuteAlgorithm()
    {
        ContextStrategy.Algorithm();
    }
}
```

Відповідно до діаграми та наведеного коду, учасники можуть бути такими:

- Абстрактний клас *IStrategy* (або інтерфейс *IStrategy*), який визначає метод `*Algorithm*()`. Цей абстрактний клас або інтерфейс є загальним для всіх алгоритмів, які його реалізують. В даному випадку використовується інтерфейс, але також можна було б використовувати абстрактний клас.
- Класи *ConcreteStrategy1* та *ConcreteStrategy2*, які реалізують інтерфейс *IStrategy*, надаючи свої власні версії методу `*Algorithm*()`. Кількість подібних класів-реалізацій може бути будь-якою.
- Клас *Context*, який містить поле для зберігання об'єкта типу *IStrategy* (назване, наприклад, `*contextStrategy*`). Взаємодія між *Context* і *IStrategy* відбувається через агрегацію. Для динамічної установки стратегії можна використовувати спеціальний метод, що дозволяє змінювати `*contextStrategy*` в будь-який момент.

У цьому випадку, об'єкт *IStrategy* міститься у властивості `*contextStrategy*` класу *Context*, хоча також можна було б використовувати приватну змінну, яку можна змінювати динамічно за допомогою відповідного методу.

10.2. Порядок виконання

Розробити алгоритм та код на основі наведеної вище інформації про патерн «Стратегія». Для прикладу використати класифікацію автомобілів за різними джерелами енергії, що дозволить ефективно управляти різними видами джерел енергії без необхідності модифікації основного автомобільного класу.

Використати наступні сутності:

- Інтерфейс або абстрактний клас *IStrategy*: Визначає загальний метод, наприклад, *drive*(), який викликається для руху автомобіля.
- Конкретні стратегії *ElectricStrategy*, *GasolineStrategy*, *HybridStrategy*, тощо: Кожен з цих класів реалізує інтерфейс або наслідує від абстрактного класу *IStrategy* і містить власну реалізацію методу *drive*(), враховуючи специфічні вимоги для кожного типу енергії.
- Клас *CarContext*: Утримує посилання на об'єкт стратегії, яке може бути динамічно змінене в будь-який момент. Коли водій обирає інше джерело енергії, *CarContext* просто змінює стратегію, не змінюючи основної логіки автомобіля.

Контрольні запитання

1. Що таке патерни поведінки?
2. Класифікація патернів поведінки.
3. Як працює патерн «Стратегія»?
4. Коли доцільно використати патерн «Стратегія»?

ЛАБОРОТОРНА РОБОТА 11
ВИРІШЕННЯ ЗАВДАНЬ ПРОЕКТУВАННЯ СТРУКТУРНИМИ ПАТЕРНАМИ

Мета роботи: засвоїти принцип проєктування на основі структурного патерна, навчитись використовувати цей принцип при написанні програм.

11.1 Загальні відомості

До структурних патернів відносять такі як:

- Декоратор (Decorator)
- Адаптер (Adapter)
- Фасад (Facade)
- Компонувальник (Composite)
- Заступник (Proxy)
- Міст (Bridge)
- Пристосуванець (Flyweight)

Загальною рисою цих шаблонів є утворення з окремих сутностей більш громіздких структур. Наприклад, використовуючи спадкування, з класів можна створити композиції з інтерфейсів та реалізацій а об’єкти отримують новий функціонал. Важливим моментом є висока гнучкість за рахунок зміни композицій в процесі виконання.

Розглянемо патерн «Адаптер». Його діаграма наведена на рис. 11.1. Він призначений для трансформації інтерфейсу одного класу у інтерфейс іншого. Використовуючи цей патерн, можна забезпечити сумісність класів з несумісними інтерфейсами та використовувати їх разом.

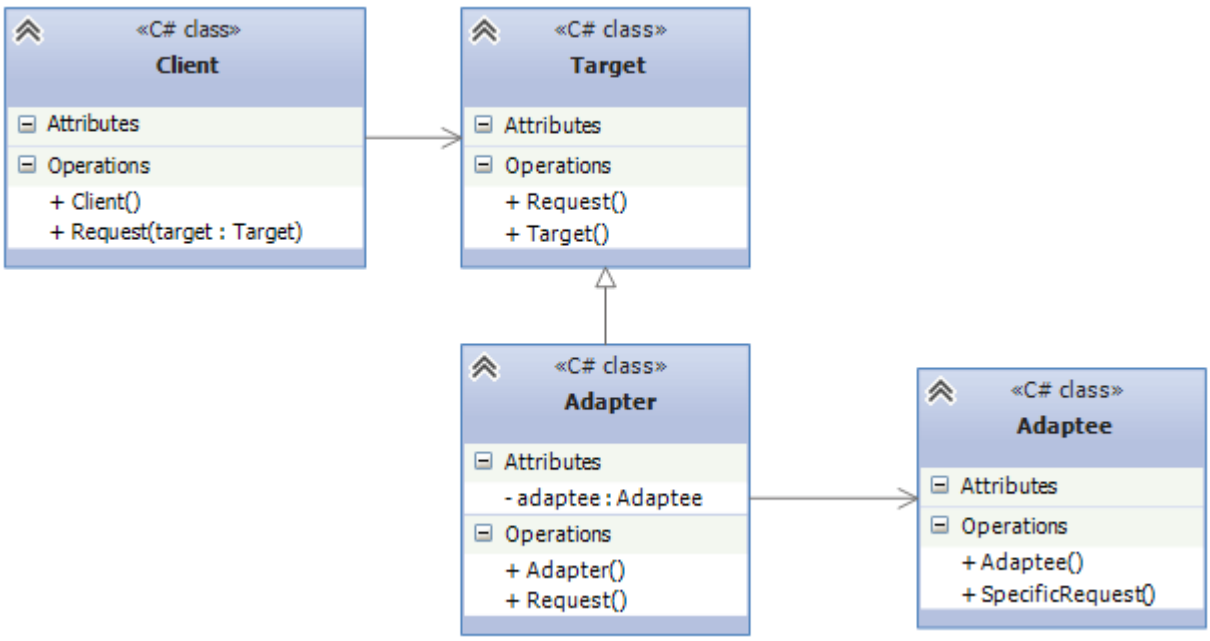

Рисунок 11.1 – UML-діаграма патерну «Адаптер»

Паттерн Адаптер доцільно використовувати у таких випадках:

- Коли необхідно використовувати клас, що вже існує, але його інтерфейс не відповідає поточним потребам реалізації.
- Коли потрібно використовувати клас, що вже існує разом з іншими класами, але інтерфейси цих класів не сумісні між собою.

Формальний опис можна представити у вигляді наступного коду.

```
class Client
{
    public void Request(Target target)
    { target.Request();}
}
// клас, до якого потрібно адаптувати інший клас
class Target
```

```
{ public virtual void Request() {} }

// Адаптер
class Adapter: Target
{
    private Adaptee adaptee = new Adaptee();
    publicoverride void Request()
    { adaptee.SpecificRequest(); }
}

// Адаптований клас
classAdaptee
{ publicvoid SpecificRequest() {} }
```

Учасники паттерну Адаптер, відповідно до діаграми та коду:

- *Target* (Ціль): Представляє об'єкти, які використовуються клієнтом.
- *Client* (Клієнт): Використовує об'єкти *Target* для досягнення своїх завдань.
- *Adaptee* (Адаптований): Представляє клас, який потрібно адаптувати, і який клієнт хотів би використовувати замість об'єктів *Target*.
- *Adapter* (Адаптер): Фактично адаптер, який дозволяє працювати з об'єктами *Adaptee* так, ніби це об'єкти *Target*.

Клієнт не має знати про *Adaptee*, він використовує лише об'єкти *Target*, а завдяки адаптеру можемо використовувати об'єкти *Adaptee* як об'єкти *Target*.

11.2. Порядок виконання

Розробит алгоритм та реалізацію для шаблону «Адаптер» за наступним описом.

Мандрівник представляє клас, який використовується для мандрівки машиною, але коли виникає необхідність пересуватися пісками пустелі, де маши-

на не придатна, мандрівник може використати верблюда. Проте, клас мандрівника не передбачає використання класу верблюда безпосередньо.

Тут може виникнути необхідність створити адаптер, який дозволить мандрівнику використовувати верблюда як засіб пересування. Адаптер буде між класом мандрівника і класом верблюда, конвертуючи інтерфейс верблюда в такий, який може бути використаний мандрівником. Таким чином, застосовуючи паттерн Адаптер, мандрівник зможе використовувати верблюда, не змінюючи свій власний інтерфейс.

Давайте дещо конкретизуємо назви класів та інтерфейсів:

- *Client* (Клієнт): Клас *Driver*, який використовує об'єкт *ITransport* для своїх потреб.
- *Adaptee* (Адаптований): Клас *Camel*, який представляє собою транспорт (в даному випадку, верблюда), який мандрівник може використовувати.
- *Target* (Ціль): Інтерфейс *ITransport*, який використовується класом *Driver*. Очікується, що будь-який транспорт, який використовується *Driver*, реалізує цей інтерфейс.
- *Adapter* (Адаптер): Клас *CamelToTransportAdapter*, який є адаптером між *Camel* і *ITransport*. Він перетворює інтерфейс верблюда в інтерфейс транспорту, щоб об'єкт *Camel* міг бути використаний класом *Driver* як *ITransport*.

Контрольні запитання

1. Що таке структурні патерни?
2. Класифікація структурних патернів.
3. Як працює патерн «Адаптер»?
4. Коли доцільно використати патерн «Адаптер»?

ЛАБОРОТОРНА РОБОТА 12
ВИРІШЕННЯ ЗАВДАНЬ ПРОЕКТУВАННЯ ЗА ПРИНЦИПАМИ SOLID

Мета роботи: засвоїти принципи проєктування SOLID, зокрема принцип «відкритості-закритості», навчитись використовувати ці принцип при написанні програм.

12.1 Загальні відомості

SOLID – це акронім утворений з перших літер принципів проєктування, що покладені в його основу:

- Single Responsibility Principle (Принцип єдиного обов'язку)
- Open/Closed Principle (Принцип відкритості/закритості)
- Liskov Substitution Principle (Принцип підстановки Лісків)
- Interface Segregation Principle (Принцип розподілу інтерфейсів)
- Dependency Inversion Principle (Принцип інверсії залежностей)

SOLID не є певним шаблоном (патерном) а скоріше набір правил написання коду, які полегшують його наступні модифікації та підтримку у майбутньому а також поліпшують сам код.

Так як одночасне використання усіх принципів можливе при створенні великих проєктів, у даній роботі буде розглянуто принцип «відкритості-закритості».

Принцип відкритості/закритості забезпечує, щоб елементи програми були відкритими для розширення своїх можливостей, але при цьому залишалися закритими для змін у своїй основі. Це означає, що архітектура системи повинна дозволяти легко впроваджувати новий функціонал шляхом додавання нового коду, не вимагаючи модифікацій у вже наявному коді. Такий підхід сприяє збереженню стабільності інших компонентів системи при внесенні змін, забезпечуючи високу гнучкість та легкість управління програмним кодом.

12.2. порядок виконання

Створимо клас Cook, що відповідає за приготування їжі за принципом «відкритості-закритості».

1) Створіть та проведіть відлагодження коду, наведеного нижче.

2) Проаналізуйте його роботу.

3) Надайте відповіді на контрольні питання.

```
class Cook
{
    public string Name { get; set; }

    public Cook(string name)
    { this.Name = name; }

    public void MakeDinner(IMeal meal)
    { meal.Make(); }
}

interface IMeal
{ void Make(); }

class PotatoMeal : IMeal
{
    public void Make()
    {
        Console.WriteLine("Чистимо картоплю");
        Console.WriteLine("Ставимо почищену картоплю на вогонь");
        Console.WriteLine("Зливаємо залишки води, розминаємо варену картоплю в пюре");
```

```
        Console.WriteLine("Посипаємо пюре спеціями та зеленню");
        Console.WriteLine("Картопляне пюре готове");
    }
}
class SaladMeal : IMeal
{
    public void Make()
    {
        Console.WriteLine("Нарізаємо помідори та огірки");
        Console.WriteLine("Посипаємо зеленню, сіллю та спеціями");
        Console.WriteLine("Поливаємо соняшниковою олією");
        Console.WriteLine("Салат готовий");
    }
}
```

Контрольні запитання

1. Яке функціональне навантаження інтерфейсу *IMeal*?
2. Як реалізовано принцип «відкритості-закритості»?
3. Як можна розширити функціонал *Cook ?*
4. Поясніть, як використовується інкапсуляція у даному прикладі?
5. Чи можна використати патерни для подібної реалізації?

ЗМІСТ

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.

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ

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

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

Укладачі: ПУГАНОВСЬКИЙ Олег Валентинович
ДЕМЕНКОВА Світлана Дмитрівна

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. 3,5. Наклад 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