МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Павло Чикунов, Артем Соловйов

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Навчально-методичний посібник
для здобувачів першого (бакалаврського) рівня вищої освіти галузей знань F «Інформаційні технології», A «Освіта» та G «Електроніка, автоматизація та електронні комунікації»

Одеса
2025

УДК 004.424:005.8(076)

Рекомендовано навчально-методичною радою Національного університету «Одеська юридична академія» (протокол №7 від 24 червня 2025 р.).

Автори:

Чикунов П. О. кандидат технічних наук, доцент, доцент кафедри інформаційних технологій Національного університету «Одеська юридична академія»; ORCID 0000-0003-4959-7744.

Соловйов А. С. асистент кафедри інформаційних технологій Національного університету «Одеська юридична академія»; ORCID 0009-0004-4023-2209.

Рецензенти:

Нечуйвітер Л. П. доктор фізико-математичних наук, професор, завідувач кафедрою інформаційних комп'ютерних технологій і математики Навчально-наукового інституту «Українська інженерно-педагогічна академія» Харківського національного університету ім. В.Н. Каразіна.

Соколов А. В. доктор технічних наук, професор, професор кафедри кібербезпеки Національного університету «Одеська юридична академія».

Чикунов П. О., Соловйов А. С.

Об'єктно-орієнтоване програмування : навч.-метод. посіб. для здобувачів першого (бакалаврського) рівня вищої освіти галузей знань F «Інформаційні технології», A «Освіта» та G «Електроніка, автоматизація та електронні комунікації» [Електронне видання] / П. О. Чикунов, А. С. Соловйов ; Нац. ун-т «Одес. юрид. академія». – Одеса : НУ «ОЮА», 2025. – 105 с.

Навчально-методичний посібник є навчальним виданням, змістом якого є методика вивчення дисципліни «Об'єктно-орієнтоване програмування». Складається з короткого змісту лекцій, рекомендованої літератури, методичних вказівок по виконанню лабораторного практикуму у комп'ютерному класі та організації самостійної роботи здобувачів вищої освіти.

Призначено для здобувачів вищої освіти першого (бакалаврського) рівня галузей знань F «Інформаційні технології», A «Освіта» та G «Електроніка, автоматизація та електронні комунікації».

УДК 004.424:005.8(076)

ISBN _____

ЗМІСТ

Передмова

В основі створення більшості сучасних програмних продуктів лежить парадигма об’єктно-орієнтованого програмування (ООП) (поєднання об’єктно-орієнтованого аналізу, об’єктно-орієнтованого проєктування та об’єктно-орієнтованого програмування). Увага ООП концентрується на побудові обґрунтованих ієрархій, складанні абстрактних інтерфейсів, універсальних сигнатур методів, використанні готових бібліотек класів.

Навчальна дисципліна формує необхідний теоретичний і практичний базис знань з об’єктно-орієнтованого програмування мовою C# у фахівців з інженерії програмного забезпечення.

Методологічну основу дисципліни складають компетентності та результати навчання, які здобувачі набули при вивченні дисциплін «Алгоритмізація та програмування», «Програмування: базові конструкції», «Алгоритми та структури даних» тощо. Дисципліна у є підґрунтям подальшого вивчення дисциплін «Аналіз та рефакторинг коду», «Конструювання програмного забезпечення», «Моделювання програмних продуктів», «Аналіз вимог до програмного забезпечення» тощо.

Метою навчальної дисципліни є вивчення основ об’єктно-орієнтованого програмування, базових концепцій ООП, мови програмування C#, що є необхідними для набуття навичок об’єктно-орієнтованого програмування.

Завданням навчальної дисципліни є опанування здобувачами концепцій об’єктно-орієнтованого програмування, набуття навичок розробки об’єктно-орієнтованих застосунків мовою C#, формування у здобувачів абстрактного мислення, необхідного для моделювання предметної області та розробки ієрархій класів.

Головними програмними результатами навчання дисципліни є:

– знати і застосовувати відповідні математичні поняття, методи доменного, системного і об’єктно-орієнтованого аналізу та математичного моделювання для розробки програмного забезпечення;

– знати і застосовувати на практиці фундаментальні концепції, парадигми і основні принципи функціонування мовних, інструментальних і обчислювальних засобів інженерії програмного забезпечення;

– вибирати вихідні дані для проєктування, керуючись формальними методами опису вимог та моделювання;

– застосовувати на практиці інструментальні програмні засоби доменного аналізу, проєктування, тестування, візуалізації, вимірювань та документування програмного забезпечення.

– вміти документувати та презентувати результати розробки програмного забезпечення.

У навчально-методичному посібнику міститься стислий лекційний матеріал, а також завдання для лабораторних та самостійних робіт. Запропоновано 5 лекцій, 10 лабораторних та 10 самостійних робіт, спрямованих

на закріплення знань із ООП.

Мета посібника – надати здобувачу вищої освіти методику вивчення дисципліни «Об'єктно-орієнтоване програмування», підвищити ефективність засвоєння теоретичних і практичних знань з дисципліни, а також організувати самостійну роботу.

Систематизований та логічно-структурований матеріал сприятиме покращенню якості підготовки здобувачів.

Перед виконанням лабораторної роботи здобувачу необхідно:

– вивчити відповідні розділи теоретичного курсу за лекціями та рекомендованою літературою;

– вибрати предметну область за індивідуальним варіантом;

– переглянути завдання лабораторної роботи, виконати моделювання предметної області у вигляді UML-діаграми класів, описати усі необхідні класи та їх атрибути.

До виконання лабораторної роботи допускаються лише здобувачі, які:

– ознайомилися з теоретичним матеріалом та завданням лабораторної роботи;

– виконали процес моделювання предметної області відповідно завданню роботи.

Після виконання лабораторної роботи в комп'ютерному класі здобувачі оформлюють звіт за наявним прикладом та захищають його перед викладачем.

Самостійна робота виконується здобувачами в позааудиторний час з оформленням звіту.

Посібник допоможе здобувачам ефективно засвоїти навчальний матеріал та підготуватися до подальшого навчання у сфері програмної інженерії.

У процесі викладання дисципліни застосовуються поєднання традиційних та інноваційних методів, спрямованих на формування теоретичних знань і практичних умінь. Лекції забезпечують систематизований виклад теоретичного матеріалу з акцентом на ключові поняття, принципи та приклади практичного застосування. Лабораторні роботи спрямовані на закріплення та поглиблення знань шляхом виконання практичних завдань, моделювання та експериментів у середовищі програмних і технічних засобів. Консультації дозволяють студентам отримати індивідуальну підтримку, уточнити складні питання та поглибити розуміння матеріалу. Самостійна робота сприяє розвитку навичок пошуку, аналізу та узагальнення інформації, формує вміння працювати з науковими та технічними джерелами. Бесіди та дискусії використовуються для обговорення проблемних питань, розвитку критичного мислення та формування вміння аргументовано відстоювати власну позицію.

Перелік лекцій:

Лекція №1. Вступ до ООП. Платформа Microsoft .NET Framework. Середовище розробки Visual Studio. Вимоги до оформлення коду.

Лекція №2. Поняття ООП: декомпозиція, абстракція, інкапсуляція. Сутності «клас» та «об'єкт». Елементи даних та функціональні елементи класів. Модифікатори доступу.

Лекція №3. Мова моделювання класів UML. Мова візуалізації класів PlantUML. Сервіс DRAW.io.

Лекція №4. Конструктори та деструктори класів. Наслідування та поліморфізм класів. Приховування елементів класів.

Лекція №5. Абстрактні, запечатані та статичні класи. Файлова структура проєктів. Перевизначення стандартних операторів в класах.

Перелік тем лабораторних робіт:

Лабораторна робота №1. Знайомство з середовищем Visual Studio та з вимогами до оформлення коду

Лабораторна робота №2. Класи, об'єкти, поля, властивості, модифікатори доступу.

Лабораторна робота №3. Елементи даних та функціональні елементи класів.

Лабораторна робота №4. Конструктори та деструктори класів.

Лабораторна робота №5. Наслідування класів.

Лабораторна робота №6. Приховування полів, властивостей та методів у похідному класі.

Лабораторна робота №7. Поліморфізм. Віртуальні методи та їх перевизначення.

Лабораторна робота №8. Абстрактні класи та абстрактні елементи класу. Запечатані та статичні класи.

Лабораторна робота №9. Робота з файловою структурою C#-проєкту.

Лабораторна робота №10. Перевизначення стандартних операторів в класах.

Перелік тем самостійної роботи:

Самостійна робота №1. Побудова UML-діаграм класів.

Самостійна робота №2. Розробка складних класів.

Самостійна робота №3. Розробка складних ієрархій класів.

ТЕМА 1. ОСНОВНІ ПОНЯТТЯ ООП. ПЛАТФОРМА .NET FRAMEWORK. СЕРЕДОВИЩЕ РОЗРОБКИ VISUAL STUDIO. КЛАСИ ТА ОБ'ЄКТИ. ЕЛЕМЕНТИ ДАНИХ ТА ФУНКЦІОНАЛЬНІ ЕЛЕМЕНТИ КЛАСІВ. МОДЕЛЮВАННЯ КЛАСІВ

Лекція 1.

Вступ до ООП. Платформа Microsoft .NET Framework. Середовище розробки Visual Studio. Вимоги до оформлення коду

Навчальні питання:

1. Вступ до ООП та платформи .NET:
– стисло про 4 принципи ООП;
– огляд Microsoft .NET Framework та сучасних .NET-технологій;
– роль Visual Studio у розробці програм на C#.
2. Вступ до стилю оформлення коду:
– значення стилю для читабельності та підтримки коду;
– вплив стилю на командну роботу та виявлення помилок.
3. Основні правила оформлення коду
– стилі іменування змінних, класів, методів;
– використання коментарів (однорядкові, багаторядкові, Doc Comments);
– спеціальні теги (TODO, FIXME).
4. Стилі іменування елементів коду:
– рекомендації для класів, методів, змінних, констант, масивів.
– правила іменування параметрів методів.
5. Структура коду:
– відступи та інтервали;
– оформлення умовних конструкцій;
– оголошення та ініціалізація масивів.
6. Висновки та рекомендації.

Стислий конспект лекції

Об'єктно-орієнтоване програмування (ООП) – це парадигма програмування, яка базується на концепції об'єктів, що представляють реальні сутності [13, 29]. Основна ідея ООП полягає в об'єднанні даних (полів) та методів їх обробки в єдині логічні блоки – класи [6, 23]. Цей підхід дозволяє

краще структурувати код, робить його більш зрозумілим і легшим для підтримки [15].

Ключові принципи ООП включають [4, 12]:

а) інкапсуляція – механізм приховування внутрішньої реалізації класу від зовнішнього коду;

б) наслідування – можливість створювати нові класи на основі існуючих;

в) поліморфізм – здатність об'єктів різних класів реагувати по-різному на однакові повідомлення;

г) абстракція – виділення важливих характеристик об'єкта ігноруючи незначні деталі.

Microsoft .NET Framework – це інтегрована платформа для розробки програмного забезпечення, яка включає [12, 24]:

а) середовище виконання CLR (Common Language Runtime) [13];

б) бібліотеку базових класів (BCL);

в) підтримку різних мов програмування (C#, VB.NET, F#);

г) механізми безпеки та керування пам'яттю.

Сучасні версії платформи (.NET Core, .NET 5+) пропонують [7, 19]:

а) кросплатформенність (Windows, Linux, macOS);

б) високу продуктивність;

в) покращену підтримку контейнеризації;

г) модульну архітектуру;

д) вдосконалені інструменти розробки.

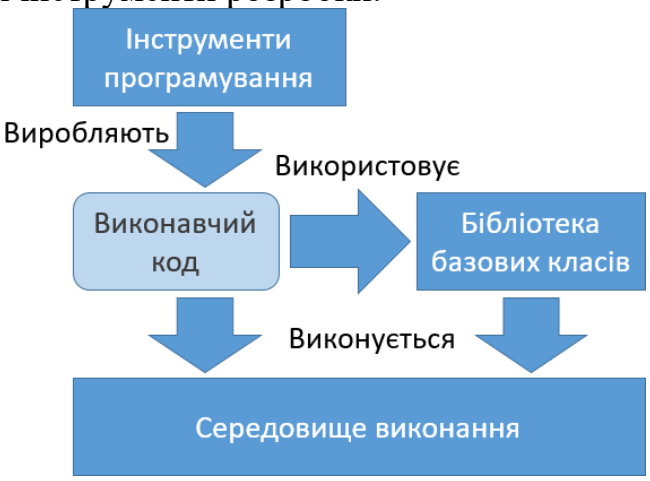

Рисунок 1 – Компоненти платформи .NET Framework [12]

Visual Studio є потужним інтегрованим середовищем розробки (IDE) від компанії Microsoft, яке поєднує в собі всі необхідні інструменти для професійної розробки програмного забезпечення [14, 20]. Воно пропонує розробникам зручний редактор коду з розширеним підсвітленням синтаксису, що значно прискорює процес написання програм.

Серед ключових можливостей Visual Studio варто відзначити передові інструменти рефакторингу, які допомагають покращувати структуру коду, та потужні засоби налагодження для ефективного усунення помилок [19]. Середовище має вбудовану підтримку систем контролю версій, зокрема Git, що спрощує роботу в команді та управління версіями проекту.

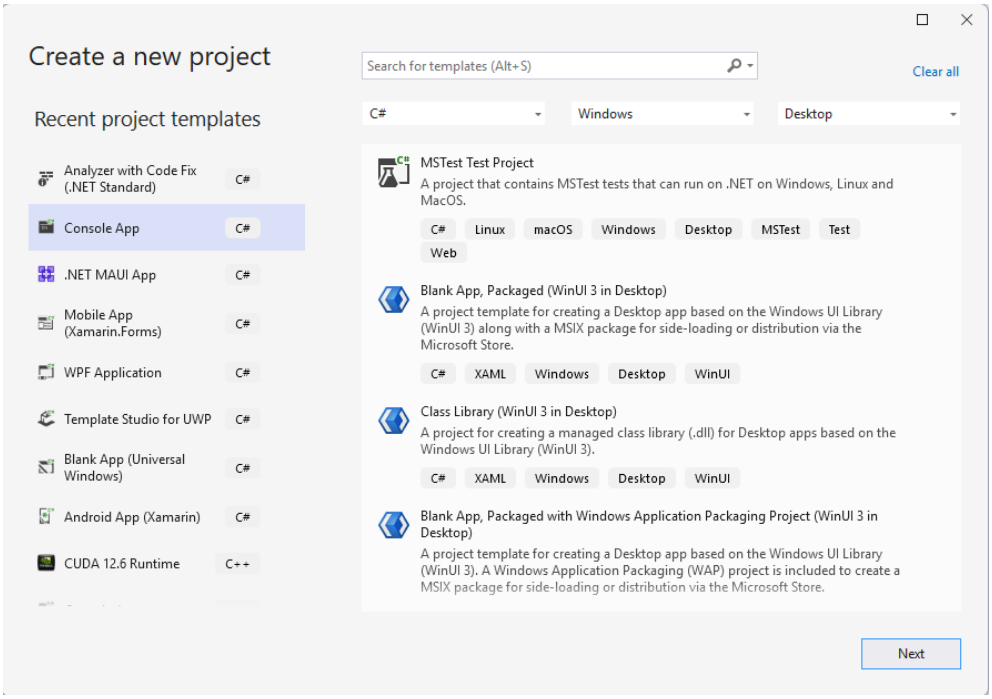

Рисунок 2 – Вибір типу проєкту у Visual Studio

Visual Studio також включає комплексні інструменти для тестування якості коду та профілювання продуктивності додатків. Воно забезпечує безперешкодну інтеграцію з хмарними сервісами Microsoft Azure, що відкриває широкі можливості для розгортання та масштабування додатків [6].

Важливою перевагою Visual Studio є його універсальність – воно підтримує розробку різноманітних типів проектів, включаючи консольні програми, веб-додатки, мобільні застосунки та багато іншого [12, 17].

Вимогами до оформлення коду є стиль оформлення коду, розбиття програми на структуровані одиниці, найменування змінних, класів та об'єктів, оформлення коментарів [15, 18].

Стиль оформлення коду – це набір правил і рекомендацій, які визначають, як має виглядати програмний код. Він охоплює такі аспекти, як іменування змінних, методів і класів, правила відступів, розміщення дужок, пробілів, коментарів, а також порядок розташування елементів у програмі. Основна мета стилю – зробити код зрозумілим, охайним і легким для читання [16].

Його використання дозволяє суттєво покращити читабельність програми [25]. Якщо код написаний з дотриманням єдиного стилю, то його легше зрозуміти як самому розробнику, так і будь-кому, хто працюватиме з ним у майбутньому [17]. Це особливо важливо при роботі в команді, адже однаковий підхід до написання коду зменшує непорозуміння та спрощує колективну роботу. Окрім того, правильно структурований код допомагає виявляти помилки ще до запуску програми, а також спрощує її подальше вдосконалення та підтримку [28].

Однорядкові коментарі пишуться з використанням подвійного символу //. Багаторядкові коментарі оформлюються через /* ... */. Коментарі документації (Doc Comments) оформлюються як /** ... */ або ///. Перед коментарем рекомендується залишати порожній рядок.

Спеціальні теги коментування:

– // TODO: – задачі, які ще потрібно реалізувати.

– // FIXME: – вказівка на баг, який слід виправити.

Загальні правила іменування змінних [29]:

– ім’я (ідентифікатор змінної) має бути інформативним та зрозумілим;

– не рекомендується використовувати непрозорі скорочення (spd, s, carminspd);

– однолітерні імена допустимі лише для короткоживучих змінних;

– заборонено починати імена змінних з цифри або символів;

– слід уникати трансліту та інших алфавітів, окрім латиниці (моя_переменная, DŽDžDzȯĂώÞЂ, viyskova_chast).

Таблиця 1.1 – Найбільш відомі стилі іменування змінних

Стиль	Опис	Приклад
Camel Case *(lowerCamelCase)*	Перше слово з маленької, решта – з великої літери. Використовується для змінних, параметрів.	maxSpeed, carModel
Pascal Case *(UpperCamelCase)*	Кожне слово з великої літери. Використовується для класів, методів.	FindMinimum(), DrawCircle()
Snake Case	Слова розділені нижнім підкресленням.	MAX_SPEED, min_value

Відступи першого рядка та інтервали між рядками:

– стандартний відступ складає 4 пробіли;

– код у тілі конструкцій має бути з відступами;

– фігурні дужки відкриваються та закриваються на тій же лінії;

– тіло конструкції (що складається навіть з одного рядка) повинне бути в фігурних дужках.

Написання імен методів та функцій:

– імена методів краще писати у стилі PascalCase;

– ім’я має починатися з дієслова, краще англійського;

– краще уникати символів _, $, скорочень;

– сигнатура методу:

[модифікатор доступу] [тип повернення] [ім’я]([аргументи]) { ... }.

Таблиця 1.2 – Рекомендовані стилі іменування елементів коду

Елемент коду	Рекомендований стиль	Приклад
Клас	PascalCase	UserProfile
Метод	PascalCase	CalculateSalary()
Змінна	camelCase	totalPrice
Константа	UPPER_SNAKE_CASE	MAX_RETRIES
Масив	camelCase	userIds
Параметр методу	camelCase	userId

Оформлення умовних конструкцій:

– тіло всіх циклів відокремлювати фігурними дужками {};

– слово *else* писати з нового рядка;

– тернарний оператор *?* використовувати лише для простих виразів.

Оголошення та ініціалізація масивів:

– при оголошенні масиву, типу елементів масиву вказувати ліворуч від дужок, наприклад int[];

– ініціалізацію масивів виконувати командою new: «int[] arr = new int[10]».

Висновки та рекомендації: дотримання єдиного стилю оформлення коду є критично важливим для створення якісного, підтримуваного програмного забезпечення [15, 23]. Узгоджений підхід до іменування змінних, структурування коду та коментування значно полегшує читання та розуміння програмного коду, особливо при роботі в команді. Використання стандартних конвенцій (PascalCase, camelCase, SNAKE_CASE), чітких відступів та фігурних дужок навіть для однорядкових блоків робить код більш структурованим, зменшує кількість помилок і прискорює процес налагодження. Крім того, правильно оформлений код з продуманими коментарями (включаючи спеціальні маркери TODO та FIXME) значно спрощує його подальшу підтримку та розширення.

Контрольні запитання

1. Що таке стиль оформлення коду та які його основні компоненти?
2. Чому важливо дотримуватись єдиного стилю коду?
3. Які правила іменування змінних та методів вважаються загальноприйнятими?
4. Як правильне форматування коду впливає на його читабельність?
5. Наведіть приклад коду з поганим і добрим стилем оформлення.
6. Яке значення мають коментарі у коді та якими мають бути хороші коментарі?
7. Що таке відступи в коді та чому їх правильне використання є важливим?
8. Які інструменти автоматичного контролю стилю коду ви знаєте?
9. Чи варто оформлювати код згідно зі стандартами мови програмування?
10. Як дотримання стилю оформлення коду допомагає уникати помилок?

Лабораторна робота 1.

Знайомство з середовищем Visual Studio та з вимогами до оформлення коду

Мета роботи: ознайомитися та набути навички практичного оформлення коду програм на мові C#.

Лабораторне завдання

1. Використовуючи теоретичні відомості та рекомендовану літературу, ознайомитись зі складовими вимог до оформлення коду на мові C#.

2. Дати відповіді на контрольні запитання.

3. Ознайомитися з прикладом коду на мові C#.

4. Обрати лістинг будь-якої програми з 1 курсу. Перекласти його на мову C#. Обов'язково повинні бути оголошення змінних, констант, масивів, наявність циклів, перевірки умов, оголошення та виклик функцій. Імена та значення повинні відрізнятися від наведеного прикладу.

5. У редакторі середовища Visual Studio Community оформити текст програми, написати коментарі.

6. Виконати компіляцію програми та виправити помилки.

7. У звіті надати текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Текст коду не повинен автоматично переноситися на новий рядок.

8. Продемонструвати працездатність програми скріншотами у звіті.

9. Захистити лабораторну роботу.

Приклад коду на мові C# з коментарями

```
/*
* Copyright (c) 2025. Username
*/

/*
* Це спеціальний коментар для документування (Doc comment).
* За допомогою спеціальної утиліти, такі коментарі можна перетворити на HTML-сторінки,
* які разом створюють документацію до вашої програми.
* Для зручності тут можна використовувати спеціальні посилання {@link Main} та <i>HTML-теги</i>
*/

class Program
{
    /*
    * Цей метод є "точкою входу" програми. У проекті може бути лише один метод із такою сигнатурою
    * під час запуску програми
    */

    static void Main(string[] args)
    {
        // Це однорядковий коментар (single line comment)

        /*
         * Це коментар у вигляді блоку (block comment)
         */

        // TODO: це спеціальний TODO-коментар. Тут можна описати, що потрібно ще доробити
        // FIXME: це теж TODO-коментар, зазвичай тут дрібні баги і що потрібно ще виправити

        // TODO:2023-09-01:Username: дописати виведення даних у файл

        int my_variable = 5;            // Коментарі після виразів повинні бути вирівняні
        int my_other_variable = 999;    // за допомогою табуляції
        int number = 924;
```

```
        // Якщо потрібно закоментувати код, то рядок коментується однорядковим коментарем
        //        int my_old_variable = 100;
        //        int my_other_old_variable = 200;

        // Перед коментарем прийнято залишати порожній рядок

        int spd = 25;           // ПОГАНО: Не рекомендується, назва не інформативна
        int carminspd = 25;     // ПОГАНО: Не заощаджуйте на назвах змінних!
        int carMinSpeed = 25;   // ДОБРЕ: Назва змінної говорить сама за себе

        int s = 0;              // МОЖНА: однолітерні допускаються, тільки якщо це
                                // короткоживучі змінні

        int speed = 150;        // ДОБРЕ: зрозуміло, що змінна відповідає за швидкість
        int Speed = 150;        // ПОГАНО: змінні не повинні починатися з капса
        int SPEED = 150;        // ПОГАНО: вкрай не рекомендується

        // Якщо назва змінної складається із двох слів
        int max_speed = 150;    // Можна: використовувати знак _ для відділення слів
        int maxSpeed = 150;     // ДОБРЕ: нормально, використовується стиль lowerCamelCase
        int MaxSpeed = 150;     // ПОГАНО: Не рекомендується, змінні не повинні починатися
                                // з капса

        int моя_переменная = 356;   // ПОГАНО: Теоретично можна, але дуже погано
        int DŽDžDzȯѦѽѢѣ = 145;       // ПОГАНО: Так можна, але, треба вбивати за таке
        int viyskova_chast = 29;    // ПОГАНО: Трансліт з кирилиці – це моветон!
        int _myvar = 100;           // ПОГАНО: Теоретично можна, але НЕ рекомендується

        /*
        int $myvar = 100;           // НЕ МОЖНА: заборонено починати з спец. символів
        int 2pac = 0;               // НЕ МОЖНА: з цифри починати не можна
        int % d = 5;                // НЕ МОЖНА: з інших знаків починати не можна
        int 'f' = 5;                // НЕ МОЖНА: з лапок починати не можна
        */

        const int Min_Speed = 25;   // ДОБРЕ: константи починаються з великої літери
        const int MIN_SPEED = 25;   // МОЖНА: стиль Java для констант іменування констант

    }

    // Приклади оголошення масивів
    private static void AnnounceArrays()
    {

        /*
         * ДОБРЕ: згідно з усіма угодами за кодом та різними рекомендаціями, квадратні
         * дужки ставлять ПІСЛЯ ТИПУ ДАНИХ
         */
        int[] goodArray;

        //  int badArray[];         // НЕ МОЖНА: компілятор видасть помилку
        //  int[5] anotherBadArray; // НЕ МОЖНА: при оголошенні не можна вказати його
// розмірність

        // Оголошення багатовимірних масивів

        int[,] twoDimensionalArray;       // МОЖНА: вказано 2 виміру
        int[,,] threeDimensionalArray;    // МОЖНА: вказано 3 виміру

        // Ініціалізація масивів

        goodArray = new int[10];    // Ініціалізуємо масив із 10 елементами
        goodArray[0] = 15;          // Привласнюємо значення першому елементу масиву
        goodArray[1] = 25;          // Привласнюємо значення другому елементу масиву

        twoDimensionalArray = new int[5, 4];
        // twoDimensionalArray = new int[5][4]; // НЕ МОЖНА: помилка компіляції
        twoDimensionalArray[0, 0] = 1;          // Привласнюємо значення
        twoDimensionalArray[1, 5] = 5;          // НЕ МОЖНА: Компілятор видасть помилку
```

```
        // Оголошення масиву з ініціалізацією
        int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 } }; // МОЖНА

        // Ініціалізація масиву після оголошення
        int[,] array5;
        array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // МОЖНА

        Console.WriteLine(twoDimensionalArray[1, 6]); // ПОГАНО: Компілятор не видасть
                                                      // помилку
    }

    // Приклади застосування циклів
    private void DesignLoops()
    {
        int progression = 0;
        // Так оформлюється for
        for (int i = 0; i < 5; i++)
        {
            progression += i;
        }

        // ПОГАНО: так оформляти цикли не рекомендується
        for (int i = 0; i < 5; i++) progression += i;

        // ПОГАНО: так оформляти цикли не рекомендується
        for (int i = 0; i < 5; i++)
            progression += i;

        // Порожній for
        for (int j = 0; j < 10; j++) ;
        // Так оформляється while
        int iterator = 0;
        while (iterator < 10)
        {
            // робимо щось у циклі
            iterator++;
        }
        // Так оформляється do-while
        int loops = 10;
        do
        {
            // щось робимо
            loops--;
        } while (loops > 0);

        // Так оформляється foreach
        var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
        foreach (int element in fibNumbers)
        {
            Console.Write($"{element} ");
        }
    }

    // Приклади застосування умовних конструкцій
    private void ifStatements()
    {
        int a = 5;
        int b = 4;
        int min;

        // Так потрібно оформляти звичайний if
        if (a >= b)
        {
            min = b;
        }

        // Так потрібно оформляти if-else
        if (a >= b)
```

```
    {
        min = b;
    }
    else
    {
        min = a;
    }

    // Так потрібно оформляти звичайний if-else  if-else
    if (a > b)
    {
        min = b;
    }
    else if (a < b)
    {
        min = a;
    }
    else
    {
        min = a;
    }

    // У C# використовується тернарний оператор ?
    min = (a >= b) ? b : a;

    // Це рівнозначно наступному виразу
    if (a >= b)
    {
        min = b;
    }
    else
    {
        min = a;
    }

    // Так оформляється switch
    switch (a)
    {
        case 1:
            // щось робимо
            break;
        case 2:
            // щось робимо
            break;
        default:
            // це виконується в тому випадку, якщо жоден з кейсів не здійснився
            break;
    }
}

/* Приклади оформлення функцій (методів)
* Синтаксис функції:
* [1-модифікатори доступу] [2-тип значення, що повертається] [3-ім'я]([4-аргументи])
* [5-список винятків] { 6 - тіло функції }
* 1 - модифікатори доступу: на даний можна нічого не писати або писати private
* 2 - тип значення, що повертається або void, якщо функція нічого не повертає
* 3 - обмеження як і на імена змінних, але є додаткові правила найменування
* 4 - список аргументів через кому, наприклад (int a, double b)
* Якщо немає аргументів - порожні дужки
* 5 - виняток поки не розглядаємо, якщо їх немає, то просто нічого не пишуть
* 6 - тіло функції, у ньому відбувається виконання функції.
* Якщо є тип даних, що повертається - повинен бути return
*/
private int FindMinimum(int a, int b)
{
    int min;

    min = (a < b) ? a : b;
    return min;
```

```
    }

    /*
     * Назва методу починається з великої літери, якщо кілька слів – використовується
     * lowerCamelCase.
     * Перше слово має бути дієсловом (т.к. метод, як правило, "щось робить"), інші слова
     * можуть бути прикметниками, іменниками тощо. Символ _ бажано не використовувати
     */

    // ДОБРЕ: з маленької літери, перше слово – дієслово
    private void DrawCircle() { }

    // ПОГАНО: Символи $ і _ не використовуємо
    // private void $er() { }

    // ПОГАНО: Символи _ не використовуємо
    private void Draw_circle() { }

    // ПОГАНО: перше слово з маленької літери
    private void draw() { }

    // ПОГАНО: перше слово має бути дієсловом
    private void circle() { }
}
```

Лекція 2.

Поняття ООП: декомпозиція, абстракція, інкапсуляція. Сутності «клас» та «об’єкт». Елементи даних та функціональні елементи класів

Навчальні питання:

1. Огляд теми лекції та її значення в ООП.
2. Декомпозиція в програмуванні:
– що таке декомпозиція?
– розбиття складних задач на простіші підзадачі;
– приклади декомпозиції у реальних проектах;
– як декомпозиція допомагає у проектуванні програм?
3. Абстракція як основа ООП:
– визначення абстракції: виділення суттєвих характеристик об’єкта;
– приклади абстракції у програмуванні.
4. Інкапсуляція (приховування реалізації):
– що таке інкапсуляція;
– об’єднання даних та методів у клас;
– приклади інкапсуляції в C#.
5. Класи та об’єкти:
– відмінність між класом (шаблоном) та об’єктом (екземпляром);
– створення класів у C#: поля, методи, конструктори.
6. Елементи даних класу з модифікаторами доступу:

– приховані (private) та публічні (public) поля;
– властивості (get, set) для доступу до полів.
7. Функціональні елементи класу (методи):
– базовий метод (без параметрів);
– перевантажені методи (з різними параметрами).
8. Висновки та рекомендації.

Стислий конспект лекції

1. Три фундаментальні концепції об'єктно-орієнтованого програмування, які формують основу сучасного підходу до розробки програмного забезпечення, це декомпозиція, абстракція та інкапсуляція [1, 2]. Це взаємопов'язані принципи, що дозволяють створювати ефективні, масштабовані та легкі у підтримці програмні системи. Важливо розуміти, що ці концепції не існують ізольовано – вони доповнюють одна одну, формуючи цілісну методологію проектування програм. Декомпозиція дає нам інструменти для аналізу складних систем, абстракція дозволяє виділяти суттєві аспекти цих систем, а інкапсуляція забезпечує захист і стабільність їх внутрішньої структури.

2. Декомпозиція являє собою систематичний підхід до розбиття складних програмних систем на менші, більш керовані та зрозумілі компоненти [3]. Цей процес можна порівняти зі складанням пазлу, де кожен окремий елемент виконує чітко визначену функцію. На практиці, наприклад при розробці інтернет-магазину, ми можемо виділити окремі модулі для роботи з користувацькими акаунтами, управління товарами, обробки замовлень та платіжними операціями. Кожен такий модуль, у свою чергу, може бути поділений на ще дрібніші компоненти. Основна перевага такого підходу полягає в тому, що він дозволяє розробникам концентруватися на конкретних задачах, спрощує процес тестування окремих компонентів, а також значно полегшує подальшу модернізацію та розширення системи [18].

3. Абстракція в об'єктно-орієнтованому програмуванні – це потужний інструмент, який дозволяє нам працювати зі складними системами, фокусуючись лише на їх істотних характеристиках та ігноруючи непотрібні деталі реалізації [4]. Цей принцип можна проілюструвати на прикладі роботи з базами даних: замість того, щоб мати справу з низькорівневими операціями читання/запису, ми працюємо з абстрактними поняттями на кшталт "Користувач" або "Замовлення". У мові C# абстракція реалізується через інтерфейси та абстрактні класи, які визначають контракт для похідних класів, але не надають конкретної реалізації [19]. Наприклад, інтерфейс "IStorage" може визначати методи для збереження та отримання даних, незалежно від того, чи реалізовано його для роботи з базою даних, файловою системою чи хмарним сховищем.

4. Інкапсуляція є одним з найважливіших принципів ООП, який забезпечує захист цілісності даних і стабільність програмних систем [5, 6]. Суть цього принципу полягає в об'єднанні даних і методів їх обробки в єдиний логічний

блок – клас, при цьому обмежуючи прямий доступ до внутрішньої реалізації. У мові C# інкапсуляція реалізується за допомогою модифікаторів доступу (private, protected, public), властивостей (properties) та методів доступу. Розглянемо приклад класу "Банківський рахунок": баланс рахунку має бути приватним полем (private), а для роботи з ним надаються публічні методи "Поповнити" та "Зняти", які включають додаткову логіку перевірки. Такий підхід запобігає прямому зміненню стану об'єкта, забезпечуючи контроль цілісності даних [20].

5. Розуміння відмінності між класами та об'єктами є критично важливим для ефективного використання ООП [7]. Клас можна розглядати як шаблон або креслення, яке визначає структуру та поведінку майбутніх об'єктів [21]. Він містить опис полів (даних), методів (операцій) та конструкторів (способів ініціалізації). Об'єкт ж є конкретним екземпляром класу, що має власний стан і може виконувати певні дії [22]. Наприклад, клас "Автомобіль" може містити такі поля як "Модель", "РікВипуску", "Колір", методи "ЗавестиДвигун", "Зупинитись", а також конструктори для різних способів створення об'єктів. Кожен конкретний автомобіль (наприклад, "BMW X5 2020 року чорного кольору") буде окремим об'єктом цього класу з власними значеннями полів.

```
class Автомобіль
{
    // Поля класу
    public string Модель;
    public int РікВипуску;
    public string Колір;

    // Конструктор з параметрами
    1 reference
    public Автомобіль(string модель, int рікВипуску, string колір)
    {
        Модель = модель;
        РікВипуску = рікВипуску;
        Колір = колір;
    }
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Створення об'єкта з використанням конструктора з параметрами
        Автомобіль авто1 = new Автомобіль("BMW X5", 2020, "чорний");
    }
}
```

Рисунок 1 – Приклад оголошення класу та об’єкту (екземпляру) класу

6. Розглядаючи структуру класів у C#, важливо розуміти фундаментальну різницю між полями та властивостями [8]. Поля класу, особливо позначені як private, слугують для внутрішнього зберігання стану об'єкта, забезпечуючи принцип інкапсуляції. На практиці це означає, що доступ до таких полів контролюється через спеціальні механізми – властивості (properties). Властивості з аксесорами get та set не лише надають контрольований доступ до даних, але й дозволяють впроваджувати додаткову логіку, таку як валідація

вхідних значень або обчислення похідних даних. Наприклад, у властивості для радіуса кола можна додати перевірку на від'ємні значення.

Властивість – це функціональний елемент класу, який забезпечує контрольований доступ до даних [23]. Вона схожа на поле у використанні, але містить виконавчий код. Властивість складається з двох методів доступу: метод *get* для читання значення; метод *set* для запису значення.

```
class Temperature
{
    private double _celsius;
    public double Celsius
    {
        get { return _celsius; }
        set
        {
            if (value >= -273.15) // Валідація
                _celsius = value;
            else
                _celsius = -273.15;
        }
    }
}
```

Таблиця 1 – Рекомендовані стилі іменування властивостей

Підхід	**Поле екземпляру**	**Локальна змінна**
Підхід 1	theData (camelCase)	TheData (PascalCase)
Підхід 2	_theData (з підкресленням)	TheData (PascalCase)

7. Метод представляє собою іменований фрагмент коду, який забезпечує функціональні можливості об'єктів [9]. Без методів об'єкт перетворюється на пасивну структуру даних, неспроможну виконувати операції над інформацією, що зберігається в ньому. Методи складають основу функціональності програми, доповнюючись іншими елементами класу, такими як властивості та оператори [24]. При створенні методів використовуються різноманітні типи даних та керуючі конструкції: умовні перевірки, цикли, переходи тощо.

Структура методу включає два основні компоненти: заголовок та тіло. Заголовок визначає характеристики методу – тип результату, назву, параметри для передачі та отримання даних. Тіло містить послідовність команд, які виконують потрібні дії.

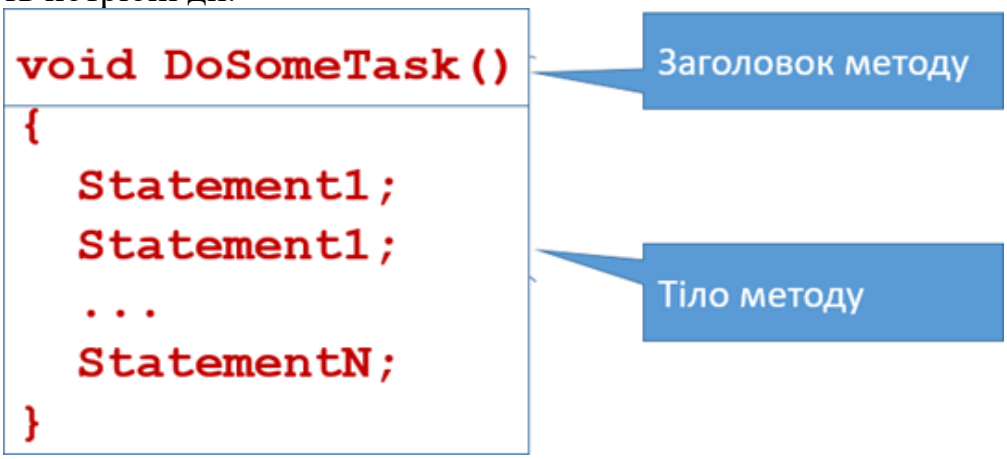

Рисунок 2 – Структура методу [12]

Найпростіша форма заголовку методу має такий вигляд:

```
MethodType MethodName(Param1Type param1, Param2Type param2);
```

Після заголовку розташовується тіло методу – програмний блок, обмежений фігурними дужками.

Локальні змінні є тимчасовими елементами, які функціонують лише в межах конкретного блоку коду. На відміну від полів класу, що зберігають дані об'єкта, локальні змінні призначені для проміжних обчислень.

Таблиця 2 – Порівняльна характеристика полів та локальних змінних

Характеристика	**Поле екземпляру**	**Локальна змінна**
Тривалість існування	Від створення до знищення екземпляру	Від оголошення до виходу з блоку
Автоматична ініціалізація	Значення за замовчуванням для типу	Відсутня, потребує явної ініціалізації

Компілятор C# дозволяє використовувати ключове слово var для автоматичного визначення типу змінної:

```
var counter = 1;           // int
var maxValue = 100.0D;     // double
var student = new Student(); // Student
```

Процес переходу до виконання методу називається викликом. Методи можуть викликатися з будь–якої точки програми в межах їх видимості. Метод, який ініціює виклик, називається викликаючим, а метод, який виконується, – викликаним.

Метод може повертати значення до викликаючого коду. Для цього при оголошенні вказується тип повернення, а в тілі використовується інструкція return. Якщо метод не повертає значення, вказується void.

Приклад методу з поверненням значення:

```
double CalculateCircleArea(double radius)
{
    const double PI = 3.14159;
    double area = PI * radius * radius;
    return area;
}
```

2. Параметри дозволяють передавати додаткові дані до методу та отримувати з нього більше інформації, ніж одне повернене значення.

Розрізняють формальні параметри (локальні змінні, оголошені у заголовку) та фактичні параметри (вирази або змінні, які використовуються при виклику).

Вихідні параметри з модифікатором out призначені для передачі даних з методу назовні:

```
static void CalculateSum(double a, double b, out double result)
{
    result = a + b;
}
static void Main(string[] args)
{
    double x = 10, y = 20, sum;
    CalculateSum(x, y, out sum);
    Console.WriteLine($"Сума {x} та {y} = {sum}");
}
```

Для методів, що складаються з єдиного виразу, можна використовувати спрощений синтаксис з лямбда-оператором:

```
// Традиційний синтаксис
public string GetAddress(string city, string street)
{
    return city + ", " + street;
}
// Синтаксис елементів-виразів
public string GetAddress(string city, string street) => city + ", " + street;
```

Клас може містити кілька методів з однаковою назвою, але різними сигнатурами. Сигнатура включає назву методу, кількість, типи і порядок параметрів, а також їх модифікатори.

```
class Calculator
{
    long CalculateSum(int a, int b) => a + b;
    long CalculateSum(int a, int b, int c) => a + b + c;
    long CalculateSum(float a, float b) => (long)(a + b);
    long CalculateSum(long a, long b) => a + b;
}
```

Висновки та рекомендації. Правильне застосування принципів декомпозиції, абстракції та інкапсуляції дозволяє створювати програмні системи, які відповідають критеріям якості, гнучкості та підтримуваності. Серед поширених помилок, яких слід уникати, можна відзначити надмірне розкриття деталей реалізації, порушення єдиного принципу відповідальності класів та неправильне використання модифікаторів доступу. Для поглибленого вивчення теми рекомендуються такі ресурси: посібник «Чистий код» Роберта Мартіна, «CLR via C#» Джефрі Ріхтера, офіційна документація Microsoft по C#, а також практичні курси на онлайн-платформах.

Контрольні запитання

1. Як визначити поняття "декомпозиція" в контексті об'єктно-орієнтованого програмування?
2. У чому полягає сутність принципу абстракції? Як він реалізується в мові C# через інтерфейси та абстрактні класи?
3. Поясніть концепцію інкапсуляції. Які механізми в C# дозволяють її реалізувати (модифікатори доступу, властивості тощо)?
4. Чим відрізняється клас від об'єкта? Наведіть приклад оголошення класу в C# з основними елементами (полями, методами, конструктором).
5. Які переваги дає використання інкапсуляції при проектуванні програмних систем?
6. Як принцип абстракції допомагає у спрощенні складних систем? Проілюструйте на прикладі роботи з базою даних.
7. Як пов'язані між собою поняття "клас" та "об'єкт"? Наведіть аналогію з реального життя для пояснення цього зв'язку.
8. Які дві основні ролі виконують методи класу в ООП?
9. Як принципи декомпозиції та абстракції допомагають у забезпеченні гнучкості та масштабованості програмного коду?
10. Як методи сприяють реалізації принципу інкапсуляції в ООП?

Лабораторна робота 2.

Класи, об'єкти, поля, властивості, модифікатори доступу

Мета роботи: здобути навички розробки структури класів з використанням елементів даних та модифікаторів доступу до них.

Лабораторне завдання

1. Визначити базовий клас у відповідності з індивідуальним варіантом опису предметної області. Механізми наслідування та поліморфізму не застосовувати, похідні класи не реалізовувати. Методи, конструктори та деструктори не реалізовувати.
2. Додати до класу усі необхідні поля, визначити їх області видимості як приватні, передбачити ініціалізацію значень, якщо це необхідно.
3. Створити публічні властивості Set і Get для доступу до приватних полів.
4. У структурі програми передбачити оголошення базового класу та реалізацію тіла класу, створити клас Program та головний метод Main(). Створити кілька об’єктів класу з різними значеннями полів.
5. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. На діаграмі обов’язково показати модифікатори доступу до полів та властивостей (знаками «+» та «–»). Клієнтський код (клас Program та метод Main) на діаграмі не показувати.
6. У звіті надати UML-діаграму з описом класу та його атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми базового класу

На рис. 1 наведена UML-діаграма, яка представляє клас графічного примітива. Поля name та area показано як приватні, а властивості – як публічні.

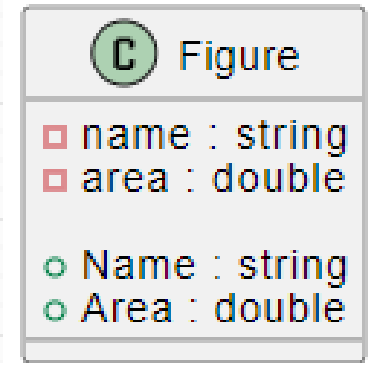

Рисунок 1 – UML-діаграма базового класу з полями, властивостями, методами

Приклад коду на мові C# з коментарями

```
using System.Text;
namespace Lab2
{
    // клас Figure - містить поля та властивості
    class Figure
    {
        // Приховані поля класу
        private string name="Фігура"; // Назва фігури
        private double area=0.0; // Площа фігури
        // Властивість для доступу до поля Name
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        // Властивість для доступу до поля Area
        public double Area
        {
            get
            {
                return area;
            }
            set
            {
                area = value;
            }
        }
    }

    class Program
    {
        // Головна точка входу в програму
        static void Main(string[] args)
        {
            // підтримка українських літер в консолі
            Console.OutputEncoding = Encoding.UTF8;

            // Оголошуємо об'єктну змінну тобто новий об'єкт circle1 класу Figure
            Figure circle1;
            // Виділяємо пам'ять для об'єкта тобто ініціалізуємо його
            circle1 = new Figure();
            // приклад скороченої форми оголошення та ініціалізації об'єкту
            Figure rectangle1 = new Figure();

            circle1.Name = "Коло1"; // звернення до властивості Figure.Name
            circle1.Area = 10.0;    // звернення до властивості Figure.Area

            rectangle1.Name = "Прямокутник1"; // звернення до властивості Figure.Name
            rectangle1.Area = 20.0;           // звернення до властивості Figure.Area

            Console.WriteLine("Фігура 1: назва = {0}, площа = {1}",
                circle1.Name, circle1.Area);
            Console.WriteLine("Фігура 2: назва = {0}, площа = {1}",
                rectangle1.Name, rectangle1.Area);
        }
    }
}
```

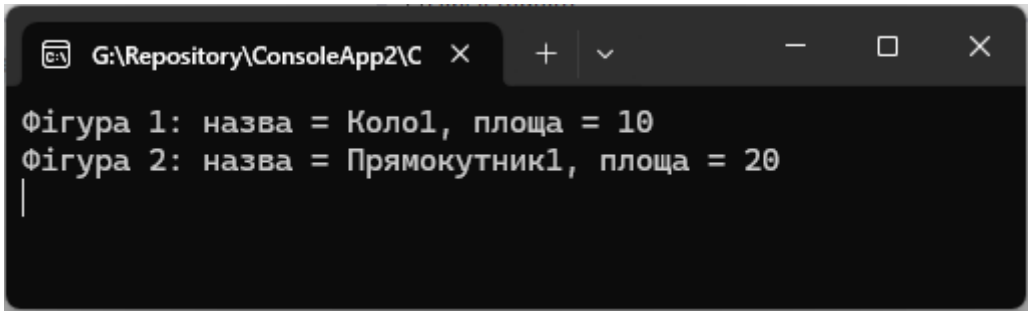

Рисунок 2 – Скріншот результатів виконання програми

Контрольні запитання

1. Чому поля класу Figure у прикладі коду оголошені як приватні?
2. Чому в даному завданні не використовуються механізми наслідування та поліморфізму?
3. Як називаються властивості для доступу до полів name та area?
4. Які модифікатори доступу використовуються для властивостей Name та Area?
5. Як ініціалізуються поля name та area за замовчуванням?
6. Як створюється новий об'єкт класу Figure у методі Main?
7. Чому властивості Set і Get є публічними, а поля – приватними?
8. Як на UML-діаграмі позначаються приватні поля та публічні властивості?
9. Як можна додати нове приватне поле до класу Figure?
10. Якщо поле area має бути додатнім числом, як можна модифікувати властивість Area для перевірки цього?

Лабораторна робота 3.

Елементи даних та функціональні елементи класів

Мета роботи: здобути навички розробки структури класів з використанням елементів даних та функціональних елементів на мові C#.

Лабораторне завдання

1. Використовуючи теоретичні відомості та рекомендовану літературу, ознайомитись з процесом розробки структури класів з елементами даних та функціональними елементами на мові C#.
2. Дати відповіді на контрольні запитання.
3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.
4. Модифікувати проєкт з лабораторної роботи №2 таким чином. Додати до базового класу (або до базових класів) методи для демонстрації

можливостей базового класу. Обов’язково вказати модифікатори для методів.

5. Виконати перевантаження (overload) мінімум одного з методів класу.

6. Створити клас Program та головний метод Main(). Створити кілька екземплярів класу (об’єктів). Продемонструвати за допомогою повідомлень у консолі, як працюють базові та перевантажені методи.

7. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод Main) на діаграмі не показувати.

8. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми базового класу

UML-діаграма в Draw.io за допомогою PlantUML дає змогу наочно представити структуру класу. Важливо правильно відображати модифікатори доступу (+ для public, – для private), що допомагає швидко зрозуміти архітектурні рішення. Для складніших класів можна додавати секції для конструкторів, подій та інших елементів.

На рис. 1 наведена UML-діаграма, яка представляє структуру класу Circle.

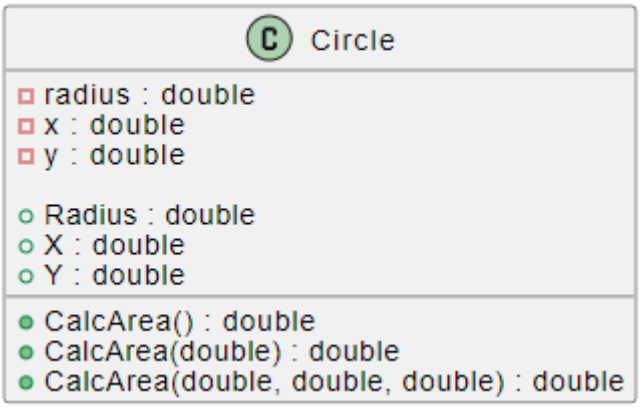

Рисунок 1 – UML-діаграма базового класу з полями, властивостями, методами

Діаграма складається з трьох основних секцій:

1. назва класу – верхня частина містить ім'я класу Circle, що вказує на те, що це модель геометричної фігури;

2. атрибути класу – у середній секції перелічені приватні поля та публічні властивості класу (позначені знаком мінус "-"):

– поле radius зберігає значення радіуса кола у вигляді числа з плаваючою точкою;

– поле x зберігає координату X центру кола;

– поле y зберігає координату Y центру кола;

– властивість Radius надає доступ до приватного поля radius;

– властивість X надає доступ до координати X;

– властивість Y надає для доступ до координати Y.

3. Методи класу – у середній частині перелічені публічні методи:

– CalcArea() – базовий метод для обчислення площі без параметрів;

– CalcArea(double) – перша перевантажена версія методу, що приймає значення радіусу як параметр;

– CalcArea(double, double, double) – друга перевантажена версія, що приймає радіус та координати центру.

Приклад коду на мові C# з коментарями

У демонстраційному прикладі створення об'єктів у методі Main показує, як інкапсуляція та перевантаження працюють у реальному коді. Кожен виклик методу CalcArea ілюструє різні сценарії використання: від простого випадку з радіусом до більш складного з додатковими координатами. Консольний вивід не лише демонструє результати обчислень, але й показує зміни стану об'єкта, що особливо важливо для розуміння роботи методів, які модифікують внутрішні поля.

```
using System; // для доступу до бібліотеки Math
namespace Lab3
{
    // клас Circle – містить поля, властивості та перевантажені методи CalcSumm
    class Circle
    {
        // Приховані поля класу
        private double radius=0.0d; // радіус
        private double x = 0.0d;    // координата x центру
        private double y = 0.0d;    // координата y центру

        // Властивість доступу до поля класу "radius"
        public double Radius
        {
            get
            {
                return radius;
            }
            set
            {
                radius = value;
            }
        }
        // Властивість доступу до поля класу "x"
        public double X
        {
            get
            {
                return x;
            }
            set
            {
                x = value;
            }
        }
        // Властивість доступу до поля класу "y"
        public double Y
        {
            get
            {
                return y;
```

```
            }
            set
            {
                y = value;
            }
        }
        // Метод обчислення площі кругу за замовчуванням
        public double CalcArea()
        {
            return 0.0d;
        }
        // Метод обчислення площі кругу з параметром "радіус"
        public double CalcArea(double newR)
        {
            Radius = newR;
            return 2 * Math.PI * Radius * Radius;
        }
        // Метод обчислення площі кругу з параметрами "радіус" та "координати центру"
        public double CalcArea(double newR, double newX, double newY)
        {
            Radius = newR;
            X = newX;
            Y = newY;

            return 2 * Math.PI * Radius * Radius;
        }
    }

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

            // /створюємо об'єкт класу Cicrcle
            Circle myCircle = new Circle();

            // виклик базового методу CalcArea()
            double area = myCircle.CalcArea();
            Console.WriteLine("Area = {0:f3}", area);

            // виклик перевантаженого методу CalcArea(newR)
            area = myCircle.CalcArea(10);
            Console.WriteLine("Area = {0:f3}, R = {1:f3}", area, myCircle.Radius);

            // виклик перевантаженого методу CalcArea(newR, newX, newY)
            area = myCircle.CalcArea(20, 15, 25);
            Console.WriteLine("Area = {0:f3}, R = {1:f3}, X = {2:f3}, Y = {3:f3}",
                              area, myCircle.Radius, myCircle.X, myCircle.Y);
        }
    }
}
```

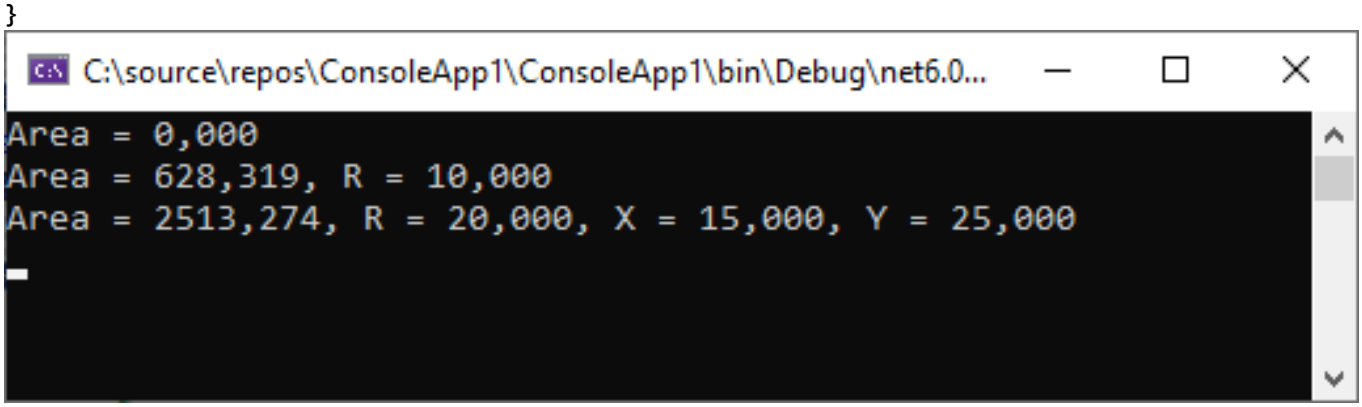

Рисунок 2 – Скріншот результатів виконання програми

Контрольні запитання

1. Що таке клас у C#? Наведіть приклад оголошення класу.
2. Які типи елементів даних можуть бути в класі? Поясніть різницю між полями (fields) та властивостями (properties).
3. Для чого використовуються модифікатори доступу (private, public)?
4. Що таке методи класу? Як вони оголошуються та використовуються?
5. Поясніть поняття "перевантаження методів" (method overloading).
6. Як викликаються перевантажені методи в програмі?
7. Чому поля класу часто роблять private, а методи public?
8. Як створити об’єкт класу в C#? Наведіть приклад ініціалізації об’єкта.
9. Які елементи класу відображаються на UML-діаграмі?
10. Які сервіси можна використовувати для створення UML-діаграм?

Лекція 3.

Уніфікована мова моделювання UML. Мова візуалізації діаграм PlantUML. Сервіс DRAW.io

Навчальні питання:

1. Вступ до UML:
– роль UML у розробці ПЗ;
– основні типи UML-діаграм.
2. Діаграма класів:
– поняття класу в UML;
– структура класу (назва, атрибути, операції);
– приклади зображення діаграм.
3. Типи зв'язків між класами:
– залежність (Dependency);
– асоціація (Association);
– агрегація (Aggregation);
– композиція (Composition);
– узагальнення (Inheritance);
– реалізація (Implementation).
4. Інструменти для створення UML-діаграм:
– огляд популярних інструментів;
5. Практичне застосування UML.
6. Висновки та рекомендації.

Стислий конспект лекції

UML-діаграма – це уніфікована мова моделювання, що використовується у парадигмі ООП для визначення, візуалізації, проєктування й документування

програмних систем [8, 10, 25]. UML є невід'ємною частиною уніфікованого процесу розробки ПЗ, і дозволяє розробникам з легкістю спілкуватися між собою про ПЗ. На цій діаграмі показують класи, інтерфейси, об'єкти й кооперації, а також їхні відносини (зв'язки) [26]. На діаграмі класи представлені у вигляді прямокутників, які містять три секції [27].

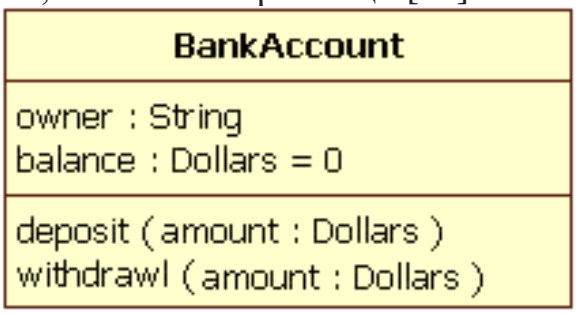

Рисунок 1 – Відображення UML-діаграми класу з трьома секціями [10]

Верхня секція містить назву класу. Вона надрукована напівжирним шрифтом і вирівняна по центру, починається з великої літери.

Середня секція містить атрибути класу. Вони вирівняні по лівому краю і перша літера мала.

Нижня секція містить операції, які може виконувати клас. Вони також вирівняні по лівому краю і перша літера мала.

Під час проектування системи визначають ряд класів і групують їх на діаграмі класів, яка допомагає визначити статичні зв'язки між ними [8].

Зв'язок (відношення, relationships) – це загальний термін, що охоплює конкретні типи логічних зв'язків, які зустрічаються на діаграмі класів та на діаграмі об'єктів. UML визначає наступні типи відношень (зв'язків) [28]:

– залежність;
– асоціація;
– агрегація;
– композиція;
– реалізація (впровадження);
– узагальнення (наслідування).

Залежність (dependency) – це тип асоціації, де існує зв'язок між залежним і незалежним елементами моделі [10, 29]. Вона існує між двома елементами, якщо зміни у визначенні одного елемента (сервера або цілі) можуть спричинити зміни в іншому (клієнті або джерелі). Цей зв'язок є односпрямованим. Залежність відображається у вигляді пунктирної лінії з відкритою стрілкою, яка вказує від клієнта до постачальника.

Залежність може бути слабшою формою зв'язку, яка вказує на те, що один клас залежить від іншого, оскільки він використовує його в певний момент часу. Один клас залежить від іншого, якщо незалежний клас є змінною-параметром або локальною змінною методу залежного класу. Іноді зв'язок між двома класами дуже слабкий. Вони взагалі не реалізуються за допомогою змінних-членів. Замість цього вони можуть бути реалізовані як аргументи функцій-членів.

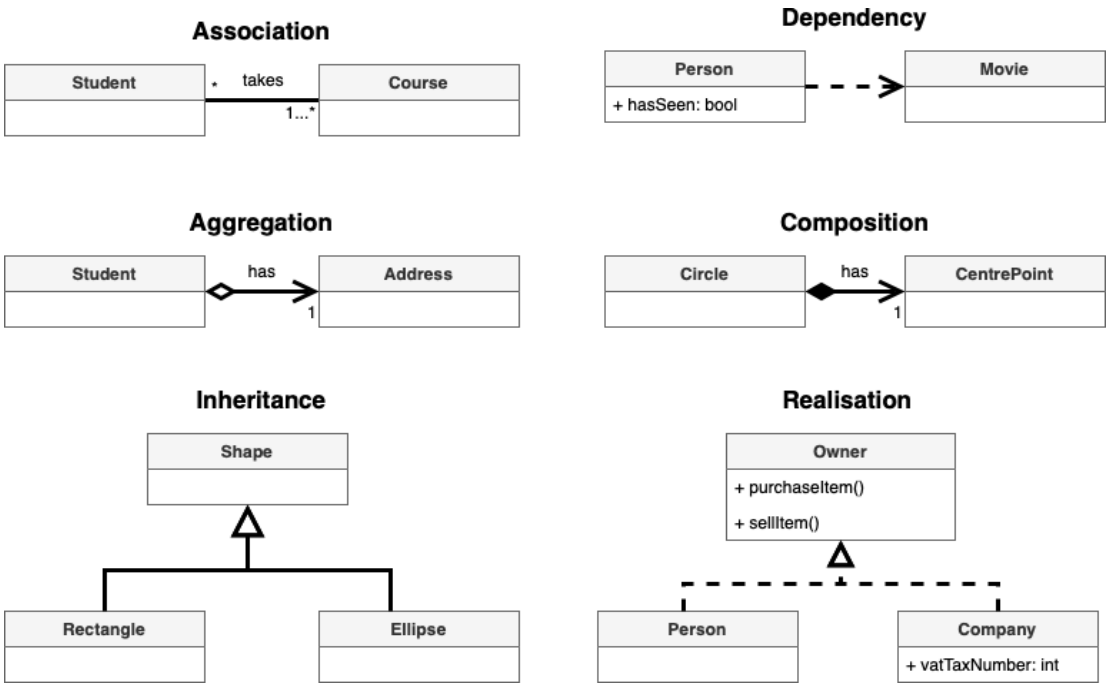

Рисунок 2 – Графічні позначки типів відношень [8]

На рис. 3 показана діаграма класів, що показує залежність між класами «Автомобіль» та «Колесо». Автомобіль залежить від Колеса», оскільки Автомобіль вже агрегує (а не просто використовує) Колесо.

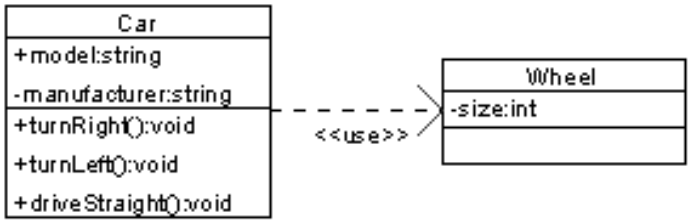

Рисунок 3 – Графічне зображення залежності двох класів [10]

Асоціація (association) показує, що об'єкти однієї сутності (класу) пов'язані з об'єктами іншої сутності [10, 13].

Якщо між двома класами визначена асоціація, то можна переміщатися від об'єктів одного класу до об'єктів іншого. Цілком припустимі випадки, коли обидва кінці асоціації відносяться до одного і того ж класу. Це означає, що з об'єктом деякого класу дозволено зв'язати інші об'єкти з того ж класу.

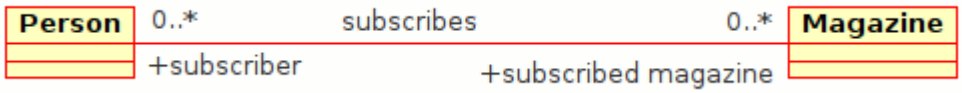

Рисунок 4 – Графічне зображення асоціації двох класів [10]

Клас, що бере участь в асоціації, грає в ній деяку роль. По суті, це «обличчя», яким клас, що знаходиться на одній стороні асоціації, звернений до класу з іншого її боку. Можна явно позначити роль, яку клас грає в асоціації.

Агрегація (aggregation) – проста асоціація між двома класами, де один з класів має вищий ранг (ціле) і складається з декількох менших за рангом (частин) [10, 14]. Приклад агрегації: клас *Професор* може бути пов'язаний з

одним або більше *Класів* (академічних груп), в той час кожен *Клас* пов'язаний лише з одним професором.

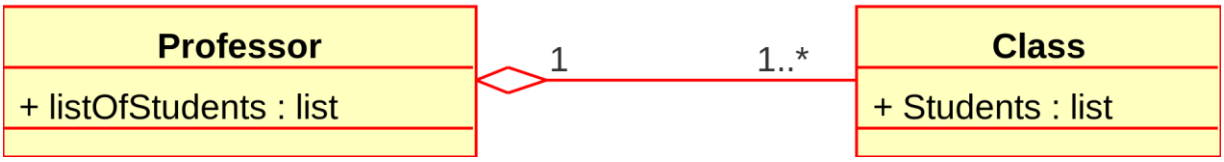

Рисунок 5 – Графічне зображення агрегації двох класів [10]

Агрегація є окремим випадком асоціації і її зображують як просту асоціацію з незафарбованим ромбом з боку «цілого». Агрегація не накладає жорстких умов на термін існування об'єктів. Наприклад, «частини» можуть існувати тоді, коли «ціле» зникає. Графічно агрегація представлена порожнім ромбом на блоці «цілого», і лінією, яка проведена від цього ромба до класу, що міститься в ньому («частин»).

Композиція (composition) – більш строгий варіант агрегації (strong aggregation) [10, 15]. Відома також як агрегація за значенням. Композиція має жорстку залежність часу існування екземплярів класу контейнера та екземплярів класів що містяться в ньому. Якщо контейнер буде знищений, то весь його вміст буде також знищено. Графічно представляється як і агрегація, але з зафарбованим ромбиком.

Приклад відношення композиції між *Університетом* та *Підрозділом*, яка чітко вказує, що об'єкти *Підрозділу* не існують, якщо не існує *Університет*, якому вони належать.

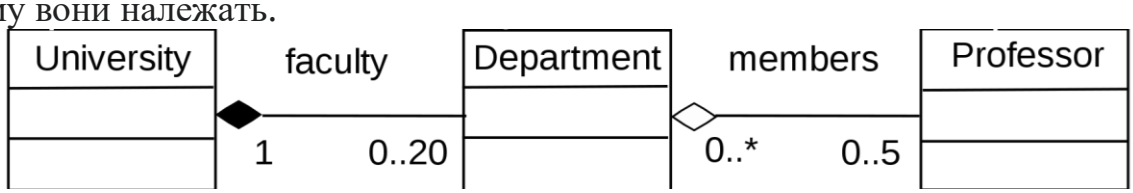

Рисунок 6 – Графічне зображення композиції трьох класів [10]

Узагальнення (наслідування, inheritance) означає, що один з двох споріднених класів (підклас, підтип, нащадок) вважається спеціалізованою формою іншого (суперклас, супертип, батько), а суперклас вважається узагальненням підкласу [10]. На практиці це означає, що будь-який екземпляр підкласу є також екземпляром суперкласу [16]. Відношення найлегше зрозуміти за допомогою фрази «А є В» (людина є ссавцем, ссавець є твариною).

Графічне представлення узагальнення в UML – це порожнистий трикутник на кінці лінії (або дерева ліній) суперкласу, який з'єднує його з одним або декількома підкласами.

Суперклас (базовий клас) у відношенні узагальнення також відомий як «батько», суперклас, базовий клас або базовий тип. Підтип у відношенні спеціалізації також відомий як «дочірній», підклас, похідний клас, похідний тип, успадкований клас або успадкований тип.

Реалізація (впровадження, implementation) – це зв'язок між двома елементами моделі, в якому один елемент моделі (клієнт) реалізує (впроваджує або виконує) поведінку, яку визначає інший елемент моделі (постачальник).

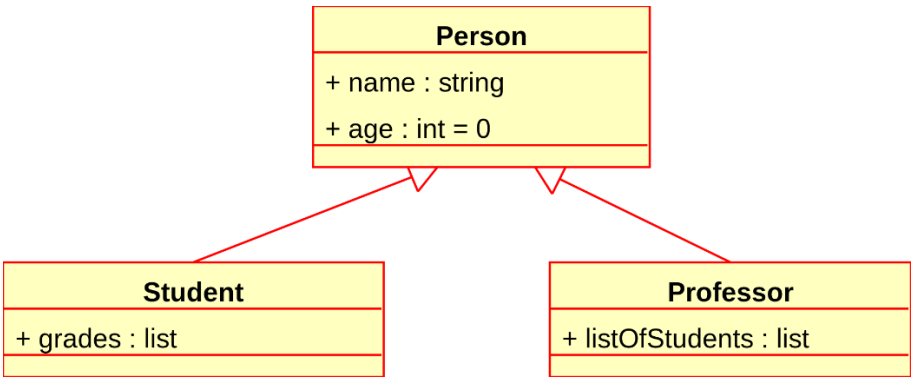

Рисунок 7 – Узагальнення між суперкласом *Особа* та двома підкласами *Студент* і *Професор* [10]

Реалізація – це зв'язок між класами, інтерфейсами, компонентами та пакетами, який з'єднує елемент-клієнт з елементом-постачальником. Зв'язок реалізації між класами/компонентами та інтерфейсами показує, що клас/компонент реалізує операції [17].

Графічне представлення реалізації – це порожнистий трикутник на інтерфейсному кінці пунктирної лінії (або дерева ліній), яка з'єднує її з одним або декількома реалізаторами.

Інструменти для створення UML-діаграм відіграють важливу роль у сучасному розробці та обслуговуванні програмного забезпечення. Вони оптимізують процес моделювання, підвищують продуктивність та сприяють покращенню якості розробки [18-20]. Розуміння їх функціоналу та впровадження їх у розробку може бути вирішальним кроком для розробницьких команд будь-якої складності.

Серед найпопулярніших сервісів та інструментів, які допомагають у створенні діаграм класової ієрархії UML, можна відзначити [12]:

1. Lucidchart: Веб-сервіс для створення різних типів діаграм, включаючи UML-діаграми. Цей інструмент полегшує простий створення ієрархій класів та зв'язків.

2. Visual Paradigm: Набір програмних рішень для візуального моделювання процесів, включаючи діаграми класової ієрархії UML.

3. StarUML: Безкоштовний інструмент для моделювання, який дозволяє створювати різні типи UML-діаграм, включаючи діаграми класів та об'єктні ієрархії.

4. Draw.io: інструмент для створення різних типів діаграм, зокрема UML-діаграми Зручний для розробників, системних архітекторів, керівників проектів та всіх, хто потребує інструмента для візуалізації та моделювання проектів. Draw.io підтримує створення таких діаграм, як ментальні карти, деревовидні діаграми, блок-схеми, мережеві діаграми, графіки Ганта, ER-діаграми, організаційні діаграми. Розробники можуть зберігати свої діаграми у хмарі (Dropbox, Google Drive, OneDrive), інтегрувати їх у вебсервіси та застосунки.

Draw.io надає розробникам інструмент для створення UML-діаграм за допомогою текстового опису ієрархії класів і об'єктів зі спеціалізованою мовою, яку називають PlantUML. Розробники можуть використовувати

PlantUML для генерації UML-діаграм, таких як діаграми класів, послідовностей, стану та багатьох інших, за допомогою текстового опису. Сам PlantUML є відкритим програмним забезпеченням, і існують плагіни для поширених програм, таких як Eclipse, NetBeans, Microsoft Word, LaTex і т.д. PlantUML також підтримується популярними IDE, такими як Visual Studio Code та IntelliJ IDEA. Опис мови можна знайти на офіційному сайті plantuml.com.

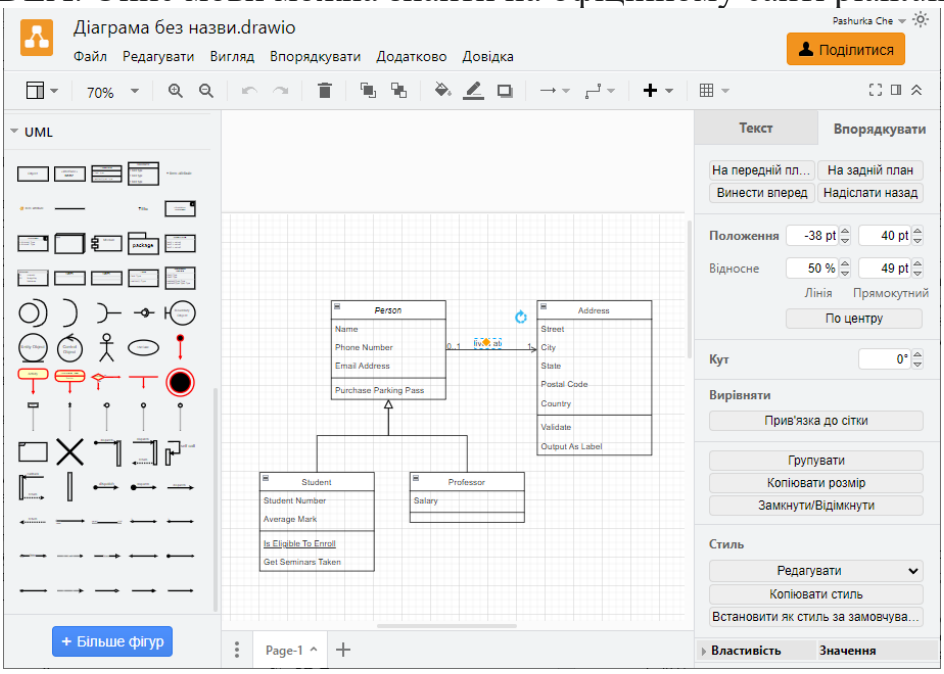

Рисунок 8 – Приклад побудови UML-діаграми ієрархії класів в Draw.io

Коли користувач вводить текст PlantUML в редактор Draw.io, вони отримують готову діаграму на основі текстового опису та обраного стилю виводу.

```
@startuml
skinparam shadowing false
Person <-- ClassDirector
Person <-- Employee

abstract class Person
{
+ {abstract} Surname : string
+ Name : string
+ Person(string, birthDate, gender)
- SetHomeTown(string)
+ {abstract} ShowLine()
~ Person()
}

class ClassDirector
{
+ Name : string
+ SealedClassDirector(string)
+ CorrectAdding()
}

static class StaticClassDeputy
{
+ {static} name : string
+ {static} DemonstrateDeputy()
}

class Employee
{
+ EmployeeNumber : int
+ Salary : double
+ WorkExperience : int
+ WorkedHours  : int
+ HourlyRate : double
- HomeTown : string
- AddHomeTown() : string
+ CorrectAdding()
~Employee()
}

@enduml
```

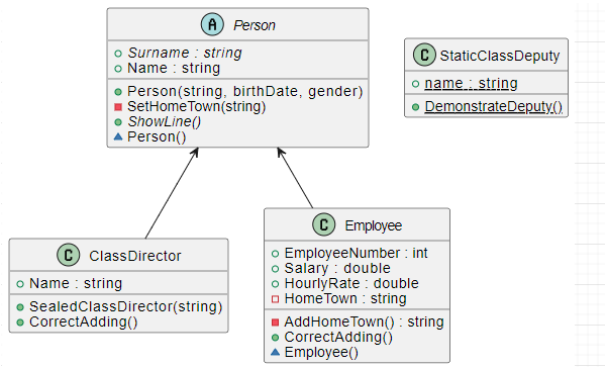

Рисунок 9 – Опис ієрархії класів на мові PlantUML та відповідна UML-діаграма

Висновки та рекомендації. UML-діаграми класів є потужним інструментом для візуалізації структури програмних систем у парадигмі ООП. Вони дозволяють чітко відображати класи, їх атрибути, методи та взаємозв'язки. Використання різних типів зв'язків (асоціація, агрегація, композиція, узагальнення тощо) допомагає точно моделювати реальні відношення між об'єктами, уникаючи логічних помилок на ранніх етапах розробки. Застосовуючи UML правильно, ви зможете ефективніше проектувати, аналізувати та документувати програмні рішення, що скоротить

час розробки та покращить якість коду. Для поглибленого вивчення варто звернутися до джерел [1, 2, 4, 8-11] та онлайн-курсів [19-22].

Контрольні запитання

1. Що таке UML і для чого він використовується в програмуванні?
2. Які три основні секції є в класі на UML-діаграмі?
3. Чим відрізняється асоціація від агрегації?
4. Як графічно позначається композиція на діаграмі класів?
5. У чому різниця між залежністю (dependency) і асоціацією (association)?
6. Як позначається узагальнення (наслідування) на UML-діаграмі?
7. Що таке реалізація (implementation) в UML?
8. Які інструменти можна використовувати для створення UML-діаграм?
9. Чи можуть об'єкти частини при агрегації існувати без об'єкта-контейнера?
10. Як саме мова PlantUML допомагає у створенні UML-діаграм?

Самостійна робота 1.

Побудова UML-діаграм класів

Мета роботи: опанувати принципи побудови UML-діаграм класів за допомогою інструментів візуалізації Draw.io, PlantUML. Навчитися відображати структуру класів, їх атрибути, методи та зв'язки між ними. Закріпити навички роботи з мовою опису PlantUML для автоматичної генерації діаграм.

Завдання до самостійної роботи

1. Ознайомитися з онлайн-сервісом побудови UML-діаграм Draw.io та мовою візуалізації PlantUML (стислу інформацію наведено у додатках). За бажанням переглянути сервіси Visual Paradigm / StarUML.
2. Ознайомитися з прикладом UML-діаграми.
3. Побудувати UML-діаграму класу, відповідно предметної області індивідуального варіанту. Відобразити поля, методи, модифікатори доступу (обов’язково public, private).
4. Оформити діаграму за допомогою:
– Draw.io (графічний редактор);
– PlantUML (текстовий опис).
5. Протестувати згенеровану діаграму:
– перевірити коректність відображення методів, полів;
– експортувати діаграму у формат PNG або SVG;
– додати пояснення до кожної частини діаграми.

Приклад 1: простий клас на мові C#

```
public class Student
{
    private string name;
    2 references
    public int Age { get; set; }

    0 references
    public Student(string name, int age) {
        this.name = name;
        Age = age;
    }

    0 references
    public void DisplayInfo() {
        Console.WriteLine($"Name: {name}, Age: {Age}");
    }
}
```

Відповідний опис UML-діаграми на мові PlantUML:

```
@startuml
class Student {
  - name : string
  + Age : int
  + Student(name: string, age: int)
  + DisplayInfo() : void
}
@enduml
```

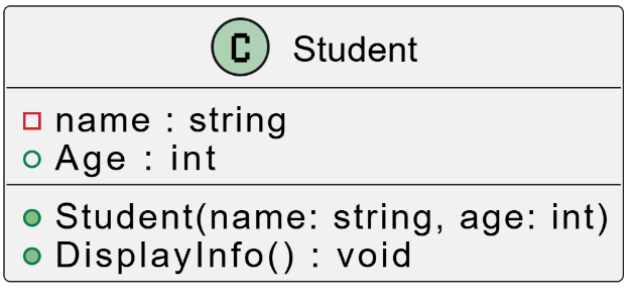

Рисунок 1 – UML-діаграма класу

Приклад 2: клас з успадкуванням та асоціацією на мові C#

```
public class Person
{
    0 references
    public string Name
    { get; set; }
    1 reference
    public virtual void Greet() {
        Console.WriteLine("Hello!");
    }
}
```

```
public class Employee : Person
{
    1 reference
    public string Department
    { get; set; }
    1 reference
    public override void Greet()
    {
        Console.WriteLine($"Hello, I work in {Department}!");
    }
}

0 references
public class Company
{
    private List<Employee> employees = new List<Employee>();
    0 references
    public void AddEmployee(Employee emp)
    {
        employees.Add(emp);
    }
}
```

Відповідний опис UML-діаграми на мові PlantUML:

```
@startuml
class Person {
  + Name : string
  + Greet() : void
}

class Employee {
  + Department : string
  + Greet() : void
}

class Company {
  - employees : List<Employee>
  + AddEmployee(emp: Employee) : void
}

Person <|-- Employee
Company "1" *-- "many" Employee
@enduml
```

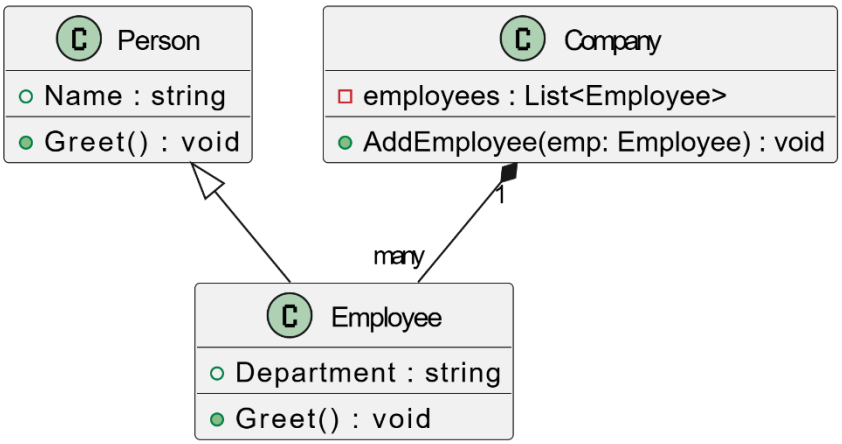

Рисунок 2 – UML-діаграма класу

Контрольні запитання

1. Які основні елементи UML-діаграми класів?
2. Як позначаються модифікатори доступу (public, private, protected) на діаграмі?
3. Які типи зв'язків між класами існують у UML?
4. Чим відрізняються агрегація та композиція?
5. Як позначити перевантаження методів (overloading) на діаграмі?
6. Які інструменти можна використовувати для створення UML-діаграм?
7. Як описати інтерфейс у PlantUML?
8. Навіщо потрібні UML-діаграми у розробці ПЗ?
9. Як позначається статичний метод або поле в UML-діаграмі класу?
10. Як відобразити інтерфейс на UML-діаграмі та реалізацію його класом?

ТЕМА 2. КОНСТРУКТОРИ ТА ДЕСТРУКТОРИ КЛАСІВ. НАСЛІДУВАННЯ ТА ПОЛІМОРФІЗМ КЛАСІВ. ПРИХОВУВАННЯ ЕЛЕМЕНТІВ КЛАСІВ

Лекція 4.

Конструктори та деструктори класів. Наслідування та поліморфізм класів. Приховування елементів класів

Навчальні питання:

1. Конструктори класів:
– призначення та особливості конструкторів;
– конструктор за замовчуванням;
– перевантаження конструкторів.
2. Деструктори класів:
– призначення та синтаксис деструкторів;
– автоматичний виклик при знищенні об'єкта;
3. Наслідування класів:
– ключові поняття;
– особливості наслідування;
4. Поліморфізм:
– віртуальні методи;
– перевизначення методів.
5. Приховування елементів даних та функціональних елементів:
– явне приховання елементів класу;
– доступ до прихованих елементів;
– схована реалізація.
6. Висновки та рекомендації.

Стислий конспект лекції

1. Конструктор – це спеціальний метод, який виконується при створенні нового екземпляру класу для ініціалізації об'єкта [8]. Основні характеристики конструктора [13-14]:

– назва збігається з назвою класу;
– не повертає значення;
– може мати параметри;
– може бути перевантаженим.

```
class Point
{
    public double X, Y;

    public Point() // Конструктор за замовчуванням
    {
        X = 0;
        Y = 0;
    }

    public Point(double x, double y) // Параметризований конструктор
    {
        X = x;
        Y = y;
    }
}
```

2. Деструктор використовується для звільнення ресурсів при знищенні екземпляру [8]. Він автоматично викликається системою збирання сміття. Особливості деструктора [15]:

– назва збігається з назвою класу з префіксом ~;

– не має параметрів та модифікаторів доступу;

– не викликається явно з програми.

```
class ResourceManager
{
    ~ResourceManager() // Деструктор
    {
        // Звільнення зовнішніх ресурсів
    }
}
```

3. Наслідування класів – це один з головних механізмів ООП, що дозволяє створювати нові класи на основі існуючих, наслідуючи їх властивості та методи [10, 12]. Ключові поняття наслідування:

– базовий клас (або предок), це наявний клас, який далі унаслідується;

– похідний клас (нащадок), це новий клас, що розширює функціонал базового;

– ієрархія класів, це дерево взаємозв’язків між класами.

```
class Animal {
    public string Name { get; set; }
    public void Eat() => Console.WriteLine($"{Name} їсть.");
}

class Dog : Animal {  // Наслідування від Animal
    public void Bark() => Console.WriteLine("Гав!");
}

// використання наслідування
Dog dog = new Dog { Name = "Рекс" };
dog.Eat();  // Успадкований метод
dog.Bark(); // Власний метод
```

Особливості наслідування: похідний клас може мати додаткові поля/методи, конструктор базового класу завжди викликається першим (через base()), усі класи неявно успадковуються від object [8].

У C# не підтримується множинне наслідування класів (тобто один клас не може безпосередньо успадковувати від двох або більше базових класів) [9]. Це обмеження введено для уникнення проблем, пов'язаних із неоднозначністю (наприклад, якщо два базові класи мають методи з однаковими іменами) [10].

Альтернативою для множинного наслідування в C# може бути використання інтерфейсів (клас може реалізовувати нескінченну кількість інтерфейсів), або композиції (можна створювати поля з посиланнями на інші класи та делегувати їм функціонал) [8].

4. Поліморфізм (від грецьких слів "полі" багато, "морфе" форма) – це принцип ООП, що дозволяє об'єктам різних класів використовувати однакові інтерфейси для різної реалізації [10–12]. У C# поліморфізм реалізується через віртуальні методи (virtual) та перевизначення методів (override).

Віртуальний метод оголошується в базовому класі з ключовим словом virtual. Він може бути перевизначеним у похідних класах з позначкою override і зміною поведінки базового методу. Якщо віртуальний метод у похідному класі не перевизначений, викликається його версія з базового класу [8].

```
class Shape
{
    public virtual void Draw() => Console.WriteLine("Малюю фігуру.");
}

class Circle : Shape
{
    public override void Draw() => Console.WriteLine("Малюю коло.");
}

// Використання:
Shape shape = new Circle();
shape.Draw();  // "Малюю коло" (викликається метод Circle, не Shape)
```

5. У похідних класах можна приховувати базові поля, властивості та методи. Якщо похідний клас оголошує поле, властивість або метод з тими самими іменами, що й базовий клас, компілятор видає попередження [8]. Щоб явно вказати на намір приховати елемент базового класу [25], використовується ключове слово *new*:

```
class BaseClass
{
    public int Value = 10;
    public void Show() => Console.WriteLine("Base");
}

class DerivedClass : BaseClass
{
    public new int Value = 20; // Приховує поле базового класу
    public new void Show() => Console.WriteLine("Derived"); // Приховує метод
}
```

При цьому є відмінність від перевизначення (override), тому що команда *new* створює нове оголошення елементу класу, яке не пов’язане з базовим, на відміну від *override*, що змінює поведінку методу базового класу.

Якщо треба, можна отримати доступ до прихованих членів базового класу через приведення до базового типу:

```
DerivedClass obj = new DerivedClass();
Console.WriteLine(((BaseClass)obj).Value); // базове поле
((BaseClass)obj).Show(); // базовий метод
```

Аналогічно виконується приховування властивостей:

```
class Base { public string Name { get; set; } = "Base"; }
class Derived : Base { public new string Name { get; set; } = "Derived"; }
```

У C# існує два основних способи приховування методів базового класу в похідному:

– за допомогою ключового слова new (схована реалізація);

– за допомогою перевизначення (override) (розглянуто у п. 4).

Якщо похідний клас оголошує метод з тим самим ім'ям, що й у базовому, але без *virtual* та *override*, це називається схованою реалізацією:

```
class BaseClass
{
    public void Show() => Console.WriteLine("Base");
}
class DerivedClass : BaseClass
{
    public new void Show() => Console.WriteLine("Derived");
}

BaseClass obj1 = new BaseClass();
obj1.Show(); // "Base"

DerivedClass obj2 = new DerivedClass();
obj2.Show(); // "Derived"

BaseClass obj3 = new DerivedClass();
obj3.Show(); // "Base" (не поліморфний виклик!)
```

6. **Висновки та рекомендації**. Конструктори, деструктори, наслідування, поліморфізм та механізми приховування є фундаментальними концепціями ООП у C#, які забезпечують гнучкість, структурованість та ефективне управління об'єктами. Конструктори дозволяють ініціалізувати стан об'єктів, деструктори – коректно звільняти ресурси, а наслідування спрощує повторне використання коду через ієрархії класів. Поліморфізм, реалізований через віртуальні методи та перевизначення, надає можливість об'єктам різних класів використовувати спільні інтерфейси для різної поведінки. Приховування членів класу за допомогою *new* корисне для локальних змін, але вимагає обережного застосування, щоб уникнути плутанини. Для побудови якісного коду рекомендується віддавати перевагу поліморфізму перед приховуванням, уникати надмірно глибоких ієрархій наслідування та чітко документувати використання нестандартних рішень. Оптимальне поєднання цих механізмів дозволяє створювати масштабовані, зручні для підтримки та інтуїтивно зрозумілі програми.

Контрольні запитання

1. Яке призначення конструкторів і чим вони відрізняються від звичайних методів?
2. Коли і як використовуються деструктори в C#?
3. Чи може конструктор чи деструктор повертати значення?
4. Чи можна викликати деструктор явно?
5. Що таке наслідування в ООП? Які його основні переваги?
6. Чи можна успадковувати від кількох класів у C#?
7. Яка різниця між virtual і override?
8. Що таке поліморфізм? Як він реалізується в C#?
9. Як приховати метод базового класу в похідному?
10. Чи можна перевизначити невіртуальний метод?

Лабораторна робота 4.

Конструктори та деструктори класів

Мета роботи: опанувати навички розробки та використання конструкторів і деструкторів класів у мові програмування C#.

Лабораторне завдання

1. Модифікувати проєкт з лабораторної роботи №3 шляхом додавання до класу різних типів конструкторів.
2. Реалізувати конструктор за замовчуванням, який ініціалізує поля класу базовими значеннями.
3. Створити конструктор з параметрами для ініціалізації об'єктів зі специфічними значеннями.
4. Імплементувати конструктор копіювання для створення копій об'єктів.
5. Визначити деструктор класу для демонстрації знищення об'єктів.
6. Продемонструвати за допомогою виведення повідомлень у консоль процес створення та знищення об'єктів.
7. Побудувати UML-діаграму класу за допомогою онлайн-сервісу Draw.io з використанням синтаксису PlantUML.
8. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми базового класу

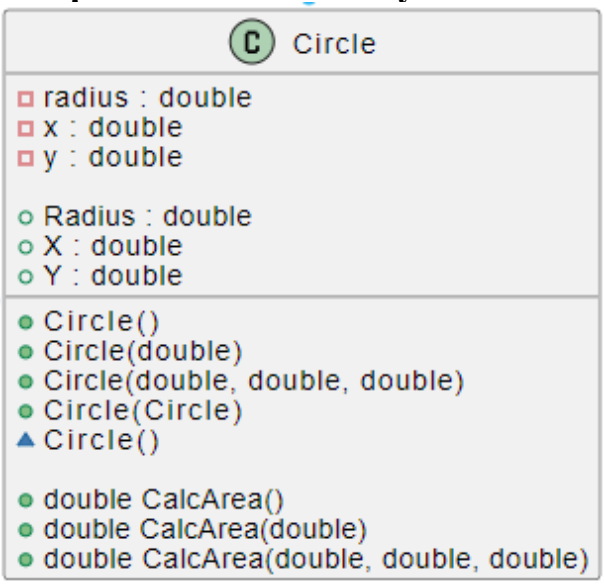

Рисунок 1 – UML-діаграма класу, що містить назву класу, поля, властивості, методи, конструктори та деструктори

Клас Circle представляє коло на площині та містить такі характеристики. Поля даних:

– radius, радіус кола (тип double);

– x, y, координати центру кола (тип double).

Властивості:

– Radius, X, Y, дозволяють отримувати або встановлювати значення радіусу та координат.

Конструктори:

– конструктор за замовчуванням Circle() ініціалізує коло з нульовими значеннями;

– Circle(double) створює коло з заданим радіусом (координати за замовчуванням);

– Circle(double, double, double), ініціалізує коло з радіусом та координатами центру;

– Circle(Circle), конструктор копіювання, який створює нове коло на основі існуючого.

Методи класу:

– CalcArea() обчислює площу кола з поточним радіусом;

– CalcArea(double) обчислює площу для заданого радіусу;

– CalcArea(double, double, double) обчислює площу кола з радіусом та координатами центру.

Приклад коду на мові C# з коментарями

```
using System.Text;
namespace Lab4
{
    // Клас Circle містить поля, властивості та різні типи конструкторів
    class Circle
    {
        // Приватні поля класу
        private double radius;    // радіус кола
        private double x;         // координата x центру кола
        private double y;         // координата y центру кола

        // Властивість для доступу до поля radius
        public double Radius
        {
            get { return radius; }
            set { radius = value; }
        }

        // Властивість для доступу до поля x
        public double X
        {
            get { return x; }
            set { x = value; }
        }

        // Властивість для доступу до поля y
        public double Y
        {
            get { return y; }
            set { y = value; }
        }
```

```
    // Конструктор за замовчуванням – ініціалізує об'єкт базовими значеннями
    public Circle()
    {
        radius = 0.0;       // встановлюємо радіус у 0
        x = 0.0;            // встановлюємо координату x у 0
        y = 0.0;            // встановлюємо координату y у 0
        Console.WriteLine("Конструктор за замовчуванням викликано");
    }

    // Конструктор з одним параметром – ініціалізує тільки радіус
    public Circle(double radius)
    {
        x = 0.0;           // координати центру за замовчуванням
        y = 0.0;
        Radius = radius;   // використовуємо властивість для встановлення радіуса
        Console.WriteLine("Конструктор з параметром 'radius' викликано");
    }

    // Конструктор з трьома параметрами – повна ініціалізація об'єкта
    public Circle(double radius, double x, double y)
    {
        Radius = radius;   // ініціалізуємо всі поля через властивості
        X = x;
        Y = y;
        Console.WriteLine("Конструктор з параметрами 'radius', 'x', 'y' викликано");
    }

    // Конструктор копіювання – створює копію існуючого об'єкта
    public Circle(Circle other)
    {
        Radius = other.Radius;        // копіюємо значення полів з іншого об'єкта
        X = other.X;
        Y = other.Y;
        Console.WriteLine("Конструктор копіювання об'єкта викликано");
    }

    // Деструктор – викликається при знищенні об'єкта
    ~Circle()
    {
        Console.WriteLine("Деструктор викликано");
    }

    // Метод обчислення площі кола за замовчуванням
    public double CalcArea()
    {
        return Math.PI * Radius * Radius;
    }

    // Перевантажений метод обчислення площі з новим радіусом
    public double CalcArea(double newRadius)
    {
        Radius = newRadius;                    // оновлюємо радіус
        return Math.PI * Radius * Radius;      // обчислюємо площу
    }

    // Перевантажений метод обчислення площі з новими координатами та радіусом
    public double CalcArea(double newRadius, double newX, double newY)
    {
        Radius = newRadius;                    // оновлюємо всі поля
        X = newX;
        Y = newY;
        return Math.PI * Radius * Radius;      // обчислюємо площу
    }
}
```

```
    class Program
    {
        static void Main(string[] args)
        {
            // Підтримка української мови в консолі
            Console.OutputEncoding = Encoding.UTF8;

            // Демонстрація роботи конструктора за замовчуванням
            Circle circle1 = new Circle();
            double area = circle1.CalcArea();
            Console.WriteLine("circle1: площа = {0:F3}", area);

            // Демонстрація роботи конструктора з одним параметром
            Circle circle2 = new Circle(10.0);
            area = circle2.CalcArea();
            Console.WriteLine("circle2: площа = {0:F3}, радіус = {1:F3}",
                              area, circle2.Radius);

            // Демонстрація роботи конструктора з трьома параметрами
            Circle circle3 = new Circle(20.0, 15.0, 25.0);
            area = circle3.CalcArea();
            Console.WriteLine("circle3: площа = {0:F3}, радіус = {1:F3}, " +
                "X = {2:F3}, Y = {3:F3}",
                area, circle3.Radius, circle3.X, circle3.Y);

            // Демонстрація роботи конструктора копіювання
            Circle circle4 = new Circle(circle3);
            area = circle4.CalcArea();
            Console.WriteLine("circle4: площа = {0:F3}, радіус = {1:F3}, " +
                "X = {2:F3}, Y = {3:F3}",
                area, circle4.Radius, circle4.X, circle4.Y);
            // Демонстрація роботи деструкторів
            // Встановлюємо посилання на null для ініціювання процесу збирання сміття
            circle1 = null;
            circle2 = null;
            circle3 = null;
            circle4 = null;
            // Примусово викликаємо збирач сміття для демонстрації деструкторів
            GC.Collect();
            GC.WaitForPendingFinalizers();

            Console.WriteLine("Натисніть будь-яку клавішу для завершення програми...");
            Console.ReadKey();
        }
    }
}
```

```
G:\Repository\ConsoleApp2\C
Конструктор за замовчуванням викликано
circle1: площа = 0,000
Конструктор з параметром 'radius' викликано
circle2: площа = 314,159, радіус = 10,000
Конструктор з параметрами 'radius', 'x', 'y' викликано
circle3: площа = 1256,637, радіус = 20,000, X = 15,000, Y = 25,000
Конструктор копіювання об'єкта викликано
circle4: площа = 1256,637, радіус = 20,000, X = 15,000, Y = 25,000
Натисніть будь-яку клавішу для завершення програми...
```

Рисунок 2 – Результат виконання програми

Контрольні запитання

1. Які види конструкторів реалізовано в класі Circle?
2. Як працює конструктор копіювання в цьому класі?
3. Для чого потрібен деструктор у класі Circle?
4. Які методи обчислення площі кола є в класі?
5. Чому поля класу ініціалізуються через властивості, а не напряму?
6. Що включає UML-діаграма класу Circle?
7. Як у програмі показується створення та знищення об’єктів?
8. Навіщо викликається GC.WaitForPendingFinalizers() у методі Main?
9. Які значення полів у *circle1* після виклику конструктора за замовчуванням?
10. Як перевірити, що конструктор копіювання створив копію об’єкта?

Лабораторна робота 5.

Наслідування класів

Мета роботи: опанувати навички розробки ієрархії класів з використанням механізму наслідування на мові C#.

Лабораторне завдання

1. Використовуючи теоретичні відомості та рекомендовану літературу, ознайомитись з процесом розробки ієрархії класів з використанням механізму наслідування на мові C#.
2. Дати відповіді на контрольні запитання.
3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.
4. Модифікувати проєкт з лабораторної роботи №4 таким чином:

– створити та додати до ієрархії всі необхідні класи, що є похідними від базового класу

– визначити поля, властивості та методи похідних класів, конструктори і деструктори;

– розробити головний метод в програму Main(), що демонструватиме роботу зі всіма класами за допомогою повідомлень у консолі

5. Побудувати UML-діаграму класу засобами онлайн–сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод Main) на діаграмі не показувати.
6. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми ієрархії класів

UML-діаграма відображає зв'язки між базовими та похідними класами. Стрілки наслідування показують, який клас є батьківським, а який – дочірнім. Важливо правильно відображати модифікатори доступу (+ для public, – для private, # для protected) та показувати перевизначені методи. Віртуальні методи позначаються курсивом або спеціальними мітками, що допомагає розуміти поліморфну поведінку системи. На рис. 1 наведена UML–діаграма, яка представляє ієрархію з базовим класом Circle та похідними Oval та Ellipse.

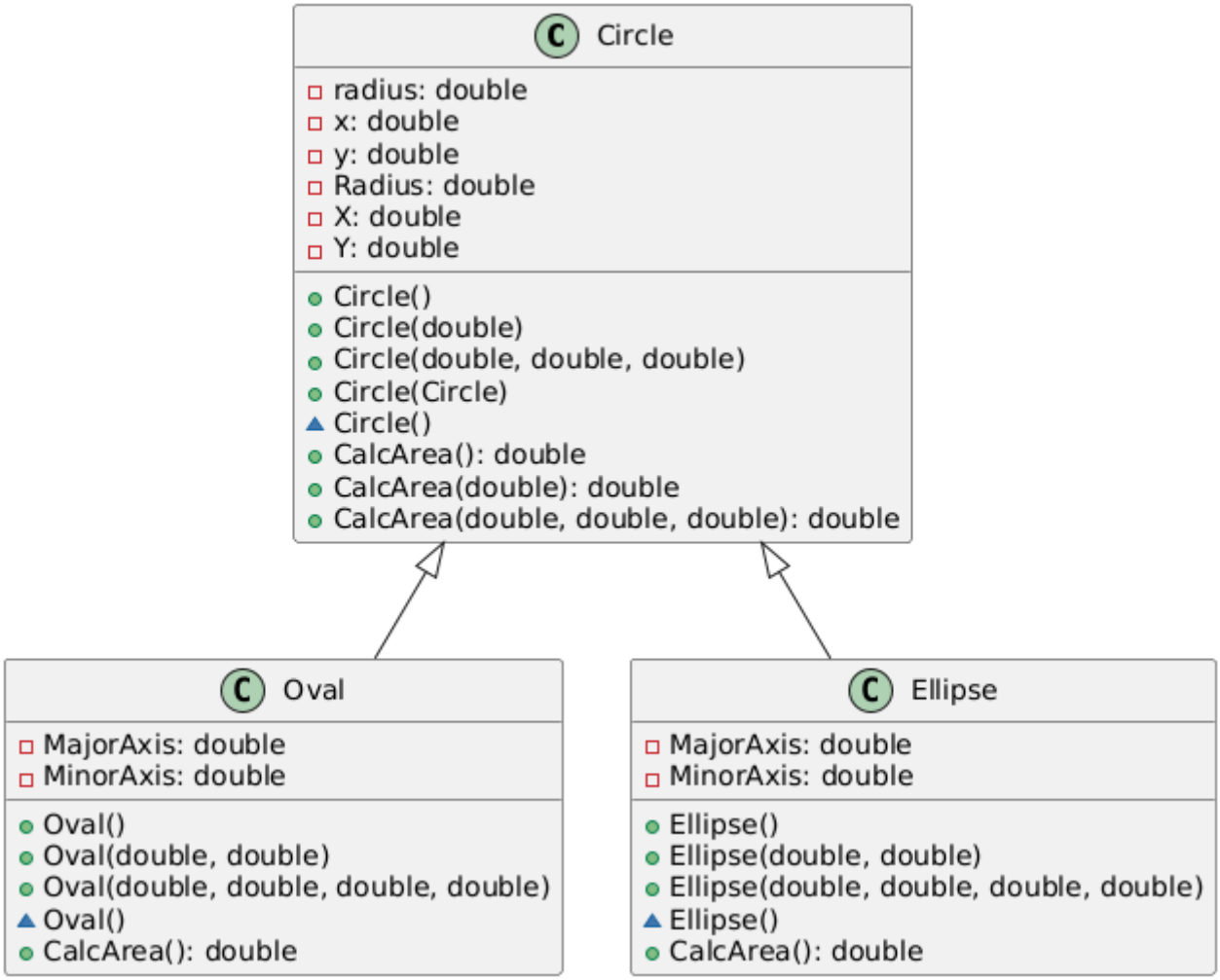

Рисунок 1 – UML-діаграма ієрархії класів, що містить назви класів, поля, властивості, методи, конструктори та деструктори

Базовий клас Circle містить:
– приватні поля radius, x, y для зберігання радіуса та координат центру;
– публічні властивості Radius, X, Y для доступу до приватних полів;
– декілька конструкторів для різних варіантів ініціалізації;
– віртуальний метод CalcArea() для обчислення площі;
– перевантажені методи CalcArea з різними параметрами;
– деструктор для очищення ресурсів.
Похідний клас Oval наслідує від Circle та додає новий функціонал:
– властивості MajorAxis та MinorAxis для довгої та короткої осей;

– конструктори, що викликають базові конструктори через *base*;
– перевизначений метод CalcArea() для обчислення площі овалу;
– деструктор.

Похідний клас Ellipse також наслідує від класу Circle та має схожу структуру до Oval з власними реалізаціями методів.

Приклад коду на мові C# з коментарями

У демонстраційному прикладі створення об'єктів різних класів показує, як працює наслідування та поліморфізм. Кожен похідний клас розширює функціональність базового класу, додаючи власні специфічні поля та методи. Використання ключового слова *override* дозволяє перевизначити поведінку базових методів, а *base* – звертатися до конструкторів батьківського класу. Консольний вивід демонструє, як викликаються конструктори в ієрархії наслідування та як працюють перевизначені методи.

```
using System.Text;

namespace Lab5
{
    // Базовий клас Circle містить поля, властивості
    // та перевантажені методи CalcArea
    class Circle
    {
        // Приховані поля класу
        private double radius; // радіус
        private double x; // координата x центру
        private double y; // координата y центру
        // Властивість доступу до поля класу "radius"
        public double Radius
        {
            get { return radius; }
            set { radius = value; }
        }
        // Властивість доступу до поля класу "x"
        public double X
        {
            get { return x; }
            set { x = value; }
        }
        // Властивість доступу до поля класу "y"
        public double Y
        {
            get { return y; }
            set { y = value; }
        }

        // Конструктор за замовчуванням
        public Circle()
        {
            radius = 0.0d; // ініціалізація радіусу
            x = 0.0d; // ініціалізація координати x
            y = 0.0d; // ініціалізація координати y
            Console.WriteLine("Конструктор за замовчуванням викликано");
        }

        // Конструктор з параметром "радіус"
        public Circle(double radius)
        {
```

```
        x = 0.0d; // ініціалізація координати x
        y = 0.0d; // ініціалізація координати y
        Radius = radius;
        Console.WriteLine("Конструктор з параметром 'radius' викликано");
    }

    // Конструктор з параметрами "радіус", "x" та "y"
    public Circle(double radius, double x, double y)
    {
        Radius = radius;
        X = x;
        Y = y;
        Console.WriteLine("Конструктор з параметрами 'radius', 'x', 'y' викликано");
    }

    // Конструктор копіювання
    public Circle(Circle other)
    {
        Radius = other.Radius;
        X = other.X;
        Y = other.Y;
        Console.WriteLine("Конструктор копіювання об'єкта викликано");
    }

    // Деструктор
    ~Circle()
    {
        Console.WriteLine("Деструктор викликано");
    }

    // Віртуальний метод обчислення площі кругу за замовчуванням
    virtual public double CalcArea()
    {
        return Math.PI * Radius * Radius;
    }

    // Метод обчислення площі кругу з параметром "радіус"
    public double CalcArea(double newR)
    {
        Radius = newR;
        return Math.PI * Radius * Radius;
    }

    // Метод обчислення площі кругу з параметрами "радіус" та "координати центру"
    public double CalcArea(double newR, double newX, double newY)
    {
        Radius = newR;
        X = newX;
        Y = newY;
        return Math.PI * Radius * Radius;
    }
}
// Похідний клас Oval наслідує від Circle
class Oval : Circle
{
    // Автореалізовані властивості для довгої та короткої вісей
    public double MajorAxis { get; set; }
    public double MinorAxis { get; set; }
    // Конструктор за замовчуванням зі зверненням
    // до унаслідованого конструктору класу Circle
    public Oval() : base()
    {
        MajorAxis = 0.0;
        MinorAxis = 0.0;
        Console.WriteLine("Конструктор за замовчуванням для Oval викликано");
```

```
    }

    // Конструктор з параметрами
    public Oval(double majorAxis, double minorAxis) : base()
    {
        MajorAxis = majorAxis;
        MinorAxis = minorAxis;
        Console.WriteLine("Конструктор з параметрами 'majorAxis' та 'minorAxis'" +
                          " для Oval викликано");
    }

    // Конструктор з параметрами для центр координат
    public Oval(double majorAxis, double minorAxis, double x, double y) :
                base(0.0, x, y)
    {
        MajorAxis = majorAxis;
        MinorAxis = minorAxis;
        Console.WriteLine("Конструктор з параметрами 'majorAxis','minorAxis'" +
                          "'x', 'y' для Oval викликано");
    }
    // Перевизначений метод обчислення площі овалу
    public override double CalcArea()
    {
        return Math.PI * MajorAxis * MinorAxis / 4;
    }
    // Деструктор
    ~Oval()
    {
        Console.WriteLine("Деструктор Oval викликано");
    }
}

// Похідний клас Ellipse наслідує від Circle
class Ellipse : Circle
{
    // Додаткові поля для довгої та короткої осі
    public double MajorAxis { get; set; }
    public double MinorAxis { get; set; }
    // Конструктор за замовчуванням
    public Ellipse() : base()
    {
        MajorAxis = 0.0;
        MinorAxis = 0.0;
        Console.WriteLine("Конструктор за замовчуванням для Ellipse викликано");
    }

    // Конструктор з параметрами
    public Ellipse(double majorAxis, double minorAxis) : base()
    {
        MajorAxis = majorAxis;
        MinorAxis = minorAxis;
        Console.WriteLine("Конструктор з параметрами 'majorAxis' та 'minorAxis'" +
                          "для Ellipse викликано");
    }

    // Конструктор з параметрами для центр координат
    public Ellipse(double majorAxis, double minorAxis, double x, double y) :
                   base(0.0, x, y)
    {
        MajorAxis = majorAxis;
        MinorAxis = minorAxis;
        X = x;
        Y = y;
        Console.WriteLine("Конструктор з параметрами 'majorAxis', 'minorAxis'," +
                          " 'x', 'y' для Ellipse викликано");
```

```
        }

        // Перевизначений метод обчислення площі еліпсу
        public override double CalcArea()
        {
            return Math.PI * MajorAxis * MinorAxis / 4;
        }
        // Деструктор
        ~Ellipse()
        {
            Console.WriteLine("Деструктор Ellipse викликано");
        }
    }


class Program
    {
        static void Main(string[] args)
        {
            // підтримка українських літер
            Console.OutputEncoding = Encoding.UTF8;

            // створюємо об'єкт класу Circle за замовчуванням
            Circle circle1 = new Circle();
            double area = circle1.CalcArea();
            Console.WriteLine("circle1: площа = {0:f3}", area);

            // створюємо об'єкт класу Circle з параметром "радіус"
            Circle circle2 = new Circle(10);
            area = circle2.CalcArea();
            Console.WriteLine("circle2: площа = {0:f3}, радіус = {1:f3}",
                              area, circle2.Radius);

            // створюємо об'єкт класу Circle з параметрами "радіус", "x" та "y"
            Circle circle3 = new Circle(20, 15, 25);
            area = circle3.CalcArea();
            Console.WriteLine("circle3: площа = {0:f3}, радіус = {1:f3}, " +
                              "X = {2:f3}, Y = {3:f3}",
                              area, circle3.Radius, circle3.X, circle3.Y);

            // створюємо копію об'єкту circle3
            Circle circle4 = new Circle(circle3);
            area = circle4.CalcArea();
            Console.WriteLine("circle4: площа = {0:f3}, радіус = {1:f3}, " +
                              "X = {2:f3}, Y = {3:f3}",
                              area, circle4.Radius, circle4.X, circle4.Y);

            // створюємо об'єкт класу Oval з параметрами
            Oval oval1 = new Oval(10, 5);
            area = oval1.CalcArea();
            Console.WriteLine("oval1: площа = {0:f3}, MajorAxis = {1:f3}, " +
                              "MinorAxis = {2:f3}",
                              area, oval1.MajorAxis, oval1.MinorAxis);

            // створюємо об'єкт класу Ellipse з параметрами
            Ellipse ellipse1 = new Ellipse(8, 4, 10, 15);
            area = ellipse1.CalcArea();
            Console.WriteLine("ellipse1: площа = {0:f3}, MajorAxis = {1:f3}, " +
                              "MinorAxis = {2:f3}, X = {3:f3}, Y = {4:f3}",
                              area, ellipse1.MajorAxis, ellipse1.MinorAxis,
                              ellipse1.X, ellipse1.Y);
            Console.ReadKey();
        }
    }
}
```

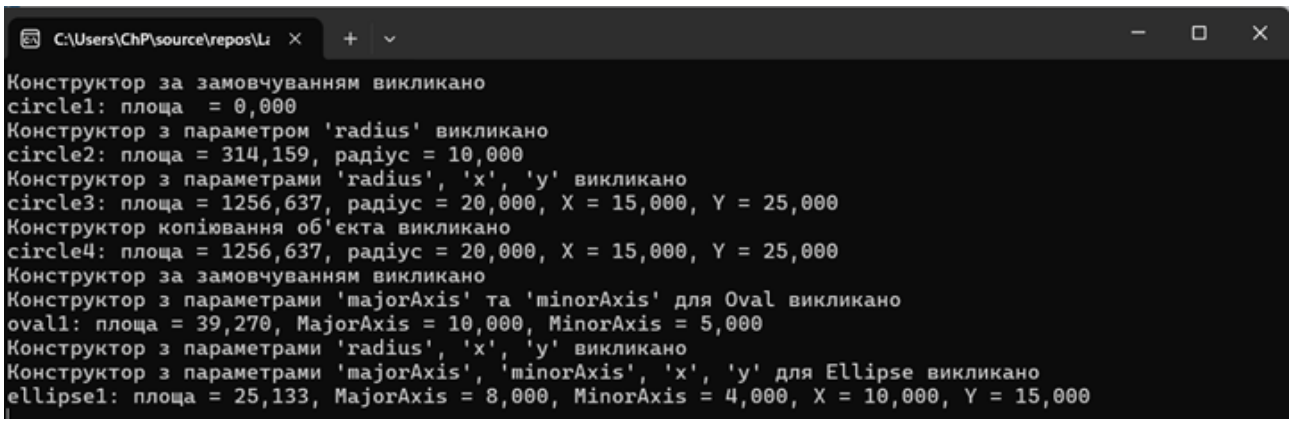

Рисунок 1 – Результат виконання програми

Контрольні запитання

1. Що таке наслідування класів у C#? Наведіть приклад оголошення похідного класу.
2. Які переваги дає використання наслідування в ООП?
3. Поясніть різницю між ключовими словами virtual та override. Для чого вони використовуються?
4. Що таке ключове слово base? Як воно використовується в конструкторах похідних класів?
5. Чи може похідний клас мати власні поля та методи, окрім успадкованих? Наведіть приклад.
6. Що таке поліморфізм? Як він реалізується через наслідування?
7. Які модифікатори доступу можуть мати члени класу? Як вони впливають на наслідування?
8. Чи можна перевизначити приватні методи базового класу?
9. Як правильно відобразити наслідування на UML-діаграмі?
10. Які особливості має виклик деструкторів в ієрархії наслідування?

Лабораторна робота 6.

Приховування полів, властивостей та методів у похідному класі

Мета роботи: опанувати навички приховування елементів даних та функціональних елементів класу в мові C# з використанням слова new.

Лабораторне завдання

1. Використовуючи теоретичні відомості та рекомендовану літературу, ознайомитись з механізмом приховування елементів класу в об'єктно–орієнтованому програмуванні на мові C#.
2. Дати відповіді на контрольні запитання.

3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.

4. Модифікувати проєкт з лабораторної роботи №5 таким чином. У похідному класі додати деякі додаткові поля та методи, що є специфічними для цього класу. Для спрощення коду можна прибрати зайві конструктори та методи базового і похідного класів. Замість приватних полів рекомендується створити автореалізовані властивості.

5. Виконати приховування у похідному класі публічних елементів базового класу за допомогою ключового слова new: одного поля та одного методу. У реалізації прихованого методу обов'язково звернутися до базового методу за допомогою ключового слова base.

6. Створити клас Program та головний метод void Main. Створити об'єкти базового та похідного класів. Продемонструвати звернення до публічного поля базового класу та до прихованого поля похідного класу. Показати виклик методу базового класу та прихованого методу похідного класу.

7. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод static void Main) на діаграмі не показувати.

8. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми ієрархії класів

UML-діаграма в Draw.io за допомогою PlantUML дає змогу наочно представити структуру ієрархії класів з урахуванням механізму приховування. Для позначення прихованих елементів можна використовувати спеціальні стереотипи або коментарі. На рис. 1 наведена UML-діаграма, яка представляє ієрархію класів Circle та Sector з прихованими елементами [16]. Базовий клас Circle містить основні атрибути та методи для роботи з колом:

– публічне поле type для визначення типу фігури;

– автореалізовані властивості Radius, X, Y для доступу до координат і радіуса;

– конструктори за замовчуванням та з параметрами;

– метод CalcArea() для обчислення площі кола;

– деструктор для коректного завершення роботи об'єкта.

Похідний клас Sector успадковує від Circle та додає специфічну функціональність:

– приховане поле type з новим значенням "sector";

– додаткова властивість Angle для зберігання кута сектора;

– конструктори, що викликають конструктори базового класу;

– прихований метод CalcArea(), що використовує базовий метод для обчислення площі сектора.

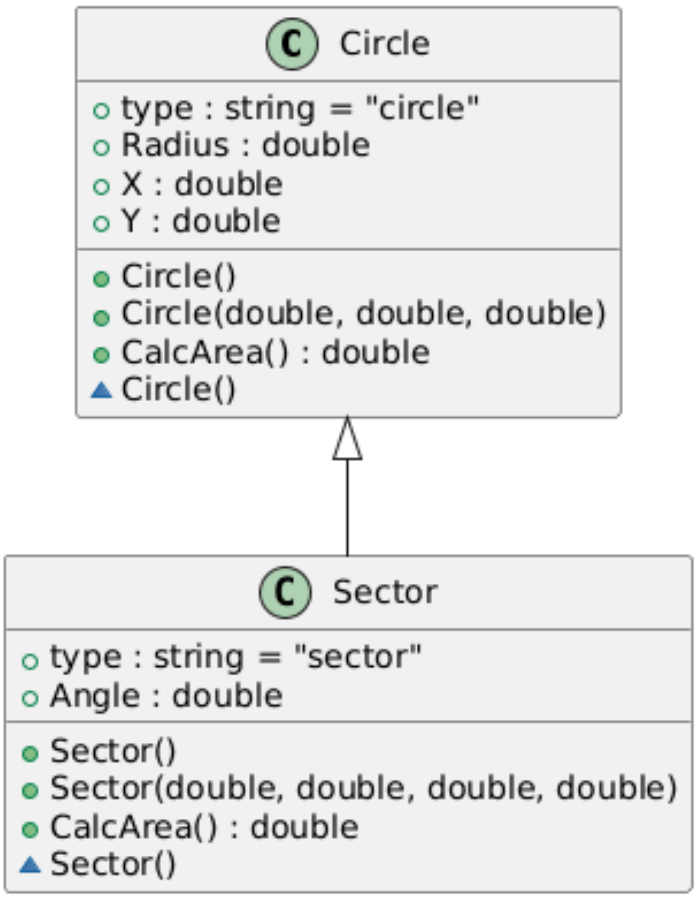

Рисунок 1 – UML-діаграма ієрархії класів, що містить назви класів, поля, властивості, методи, конструктори та деструктори

Приклад коду на мові C# з коментарями

У демонстраційному прикладі створення об'єктів у методі Main показує, як працює механізм приховування елементів класу. Кожен виклик методу та звернення до поля ілюструє різницю між базовими та прихованими елементами. Консольний вивід демонструє не лише результати обчислень, але й показує, які саме версії методів та полів використовуються в різних контекстах.

```
using System.Text;
namespace Lab6
{
    // базовий клас Circle
    class Circle
    {
        // публічне поле – тип фігури
        public readonly string type = "circle";
        // Автореалізовані властивості
        public double Radius { get; set; } // радіус
        public double X { get; set; } // координата x центру
        public double Y { get; set; } // координата y центру

        // Конструктор за замовчуванням
        public Circle()
        {
            Radius = 0.0d; // ініціалізація радіусу
            X = 0.0d; // ініціалізація координати x
            Y = 0.0d; // ініціалізація координати y
```

```
        Console.WriteLine("Конструктор за замовчуванням викликано");
    }

    // Конструктор зі всіма параметрами
    public Circle(double radius, double x, double y)
    {
        Radius = radius;
        X = x;
        Y = y;
        Console.WriteLine("Конструктор з параметрами 'radius', " +
                          "'x', 'y' викликано");
    }
    // Деструктор
    ~Circle()
    {
        Console.WriteLine("Деструктор викликано");
    }
    // Метод обчислення площі кругу
    public double CalcArea()
    {
        return Math.PI * Radius * Radius;
    }
}

class Sector : Circle
{
    // приховування публічного поля
    public new readonly string type = "sector";

    // Нова автореалізована властивість, величина кута сектора в градусах
    public double Angle { get; set; }

    // Конструктор за замовчуванням
    public Sector() : base()
    {
        Angle = 0.0;
        Console.WriteLine("Конструктор за замовчуванням для " +
                          "Sector викликано");
    }

    // Конструктор зі всіма параметрами
    public Sector(double radius, double angle, double x, double y)
        : base(radius, x, y)
    {
        Angle = angle;
        Console.WriteLine("Конструктор з параметрами 'radius', 'angle', " +
                          "'x', 'y' для Sector викликано");
    }

    // Приховування методу обчислення площі сектора
    public new double CalcArea()
    {
        return base.CalcArea() * (Angle / 360.0);
    }

    // Деструктор
    ~Sector()
    {
        // необхідно переконатися, що це повідомлення ми не побачимо
        Console.WriteLine("Деструктор викликано");
    }
}

class Program
{
```

```
        static void Main(string[] args)
        {
            // підтримка українських літер
            Console.OutputEncoding = Encoding.UTF8;

            // створюємо об'єкт класу Circle за замовчуванням
            Circle circle1 = new Circle();
            double area = circle1.CalcArea();
            Console.WriteLine("circle1: площа = {0:f3}", area);

            // створюємо об'єкт класу Circle з параметрами
            Circle circle2 = new Circle(20, 15, 25);
            Console.WriteLine("circle2: площа = {0:f3}, радіус = {1:f3}, " +
                              "X = {2:f3}, Y = {3:f3}, тип = {4}", circle2.CalcArea(),
                              circle2.Radius, circle2.X, circle2.Y, circle2.type);

            // створюємо об'єкт класу Sector з параметрами
            Sector sector1 = new Sector(20, 90, 100, 200);
            Console.WriteLine("sector1: площа = {0:f3}, радіус = {1:f3}, " +
                              "кут = {2:f3}, тип = {3}", sector1.CalcArea(),
                              sector1.Radius, sector1.Angle, sector1.type);
        }
    }
}
```

```
C:\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\net8.0\ConsoleApp2.exe
Конструктор за замовчуванням викликано
circle1: площа  = 0,000
Конструктор з параметрами 'radius', 'x', 'y' викликано
circle2: площа = 1256,637, радіус = 20,000, X = 15,000, Y = 25,000, тип = circle
Конструктор з параметрами 'radius', 'x', 'y' викликано
Конструктор з параметрами 'radius', 'angle', 'x', 'y' для Sector викликано
sector1: площа = 314,159, радіус = 20,000, кут = 90,000, тип = sector
```

Рисунок 1 – Скріншот результатів виконання програми

Контрольні запитання

1. Що таке приховування елементів класу в C#? Чим воно відрізняється від перевизначення?
2. Для чого використовується ключове слово new при оголошенні полів та методів у похідному класі?
3. Як працює ключове слово base при зверненні до прихованих елементів базового класу?
4. Які переваги та недоліки має механізм приховування порівняно з віртуальними методами?
5. Чи можна приховувати приватні елементи базового класу?
6. Як впливає приховування на поліморфізм та динамічне зв'язування?
7. Які рекомендації щодо використання приховування в реальних проектах?
8. Як відобразити приховані елементи на UML-діаграмі класів?
9. Чи можна приховувати конструктори та деструктори?
10. Як тестувати код, що використовує механізм приховування елементів?

Лабораторна робота 7.

Поліморфізм. Віртуальні методи та їх перевизначення

Мета виконання роботи: ознайомитися із концепцією поліморфізму в C#, вивчити використання віртуальних методів та їх перевизначення, розглянути різні модифікатори методів virtual та override, дослідити модифікатори доступу до елементів класу.

Лабораторне завдання

1. Використовуючи теоретичні відомості та рекомендовану літературу, ознайомитись з принципом поліморфізму класів в об'єктно-орієнтованому програмуванні на мові C#.

2. Дати відповіді на контрольні запитання.

3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.

4. Модифікувати проєкт з лабораторної роботи №5 таким чином:

– забезпечте можливість перевизначення методів базового класу в похідних класах за допомогою ключових слів "virtual" і "override";

– функціонал перевизначених методів повинен відрізнятися від базового методу;

– переконайтесь, що ви використовуєте модифікатори методів "virtual" та "override" правильно для забезпечення віртуальних методів у базовому класі та їх перевизначення в класах-нащадках;

– у базовому класі створити віртуальний метод;

– для кожного класу-нащадку реалізуйте віртуальний метод;

– застосуйте два рівні доступності класів: загальний (public) та внутрішній (internal);

– дослідить модифікатори доступу елементів класу, такі як public, protected, private, і застосувати їх у вашій програмі для демонстрації різних рівнів доступу;

– покажіть приклад виклику перевизначених методів в головній програмі в методі void main (). Продемонструйте виклик віртуального методу базової частки об'єкта похідного класу.

5. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод static void Main) на діаграмі не показувати.

6. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми ієрархії класів

UML-діаграма в Draw.io за допомогою PlantUML дає змогу наочно представити структуру класу. Важливо правильно відображати модифікатори доступу (+ для public, – для private), що допомагає швидко зрозуміти архітектурні рішення та принципи інкапсуляції. На рисунку 1 наведена UML-діаграма, яка представляє ієрархію класів з полями, властивостями та перевизначеним методом [16].

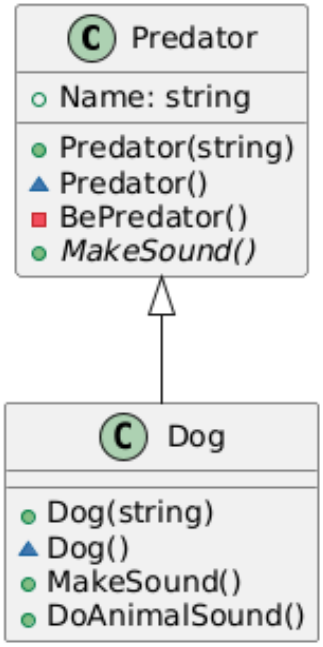

Рисунок 1 – UML–діаграма базового класу з полями, властивостями та перевантаженими методами

Базовий клас Animal (Тварина) має такі атрибути:
– публічна властивість Name (назва тварини);
– конструктор Predator(string);
– захищений метод BePredator();
– публічний віртуальний метод MakeSound().
Похідний клас Dog (спадкоємець класу Predator) має такі атрибути:
– унаслідувана властивість Name;
– унаслідуваний конструктор Predator();
– унаслідуваний метод BePredator();
– конструктор Dog(string);
– публічний перевизначений метод MakeSound();
– публічний метод DoAnimalSound().

Приклад коду на мові C# з коментарями

У демонстраційному прикладі створення об'єктів у методі Main показано, як працює принцип поліморфізму у реальному коді. Поліморфізм реалізується через віртуальний метод MakeSound у базовому класі Predator, який перевизначається в похідному класі Dog. Це дозволяє викликати різні реалізації

методу залежно від типу об’єкта (Predator видає тишу, а Dog – гавкіт). Метод DoAnimalSound у класі Dog демонструє виклик базової версії MakeSound через base, забезпечуючи доступ до поведінки базового класу. Таким чином, поліморфізм забезпечує гнучкість у поведінці об’єктів при спадкуванні.

```
using System;
using System.Text;

// Базовий клас "Хижак"
internal class Predator
{
    public string Name
    {
        get;
        set;
    }

    // конструктор класу
    public Predator(string name)
    {
        Name = name;
    }

    // деструктор класу за замовчуванням
    ~ Predator()
    {
    }

// Приватний метод не доступний у похідному класі,
// доступний лише власним методам
private void BePredator()
{
    Console.WriteLine($"Лише {Name} може жити у лісі");
}

// Віртуальний метод "Голос"
public virtual void MakeSound()
{
    Console.WriteLine($"{Name} говорить: (зловісна тиша)");
    BePredator();
}
}

// Клас "Собака" наслідує базовий клас "Тварина"
internal class Dog : Predator
{
    public Dog(string name) : base(name)
    {
        Console.WriteLine($"{Name} - домашня тварина!");
    }

    ~Dog()
    {
    }

    // Перевизначення методу "Голос" для собаки
    public override void MakeSound()
    {
        Console.WriteLine($"{Name} гавкає: Гав-гав!");
    }

    // Метод, який викликає віртуальний метод базового класу
    public void DoAnimalSound()
    {
        // Виклик віртуального методу базового класу за допомогою "base"
```

```
        Console.WriteLine($"{Name} говорить: я нащадок хижаків");
        base.MakeSound();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = UTF8Encoding.UTF8; // підтримка укр. літер

        // Створення об'єкта класу "Predator"
        Predator predator = new Predator("Хижак");
        predator.MakeSound();

        // Створення об'єкта класу "Собака"
        Dog myDog = new Dog("Барсик");

        // Виклик методу "MakeSound" класу "Собака"
        myDog.MakeSound();

        // Виклик методу "DoAnimalSound", який викликає метод класу "MakeSound"
        myDog.DoAnimalSound();
    }
}
```

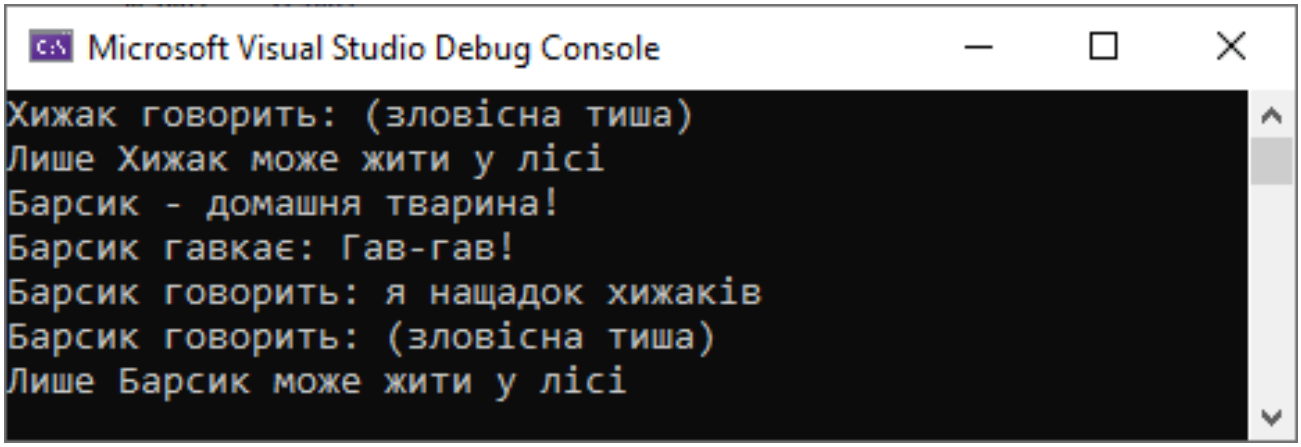

Рисунок 2 – Результат виконання програми

Контрольні запитання

1. Які модифікатори доступу відображаються в UML-діаграмах за допомогою PlantUML?
2. Як у коді на C# реалізується поліморфізм через віртуальний метод?
3. Що означає ключове слово virtual у методі MakeSound класу Predator?
4. Як працює ключове слово override у методі MakeSound класу Dog?
5. Чому метод BePredator є приватним і як це впливає на спадкування?
6. Як метод DoAnimalSound у класі Dog використовує базовий клас?
7. Яке значення має ключове слово base у методі DoAnimalSound?
8. Я ким чином деструктор у класах Predator і Dog впливає на їх поведінку?
9. Як на UML-діаграмі показати модифікатор доступу protected?
10. Як у UML-діаграмі позначити віртуальний метод MakeSound?

Самостійна робота 2.

Розробка складних класів

Мета роботи: навчитися розробляти класи підвищеної складності. Опанувати наслідування, віртуальні методи, поліморфізм на конктректних прикладах. Практично освоїти створення UML-діаграм складних ієрархій класів.

Завдання до самостійної роботи

1. Використовуючи теоретичні відомості та рекомендовану літературу, ознайомитись з принципом поліморфізму класів в об'єктно-орієнтованому програмуванні на мові C#.
2. Дати відповіді на контрольні запитання.
3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.
4. Створити ієрархію з 4 класів, відповідно предметної області індивідуального варіанту:
   – 2 батьківський класів (не абстрактні, не інтерфейси) з віртуальними методами;
   – 2 дочірні класи у кожного базового.
5. Використати основні можливості:
   – конструктори та деструктори класів;
   – унаслідування класів;
   – віртуальні методи;
   – перевизначення методів;
   – приховування полів, властивостей та методів у похідному класі (по можливості).
6. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод static void Main) на діаграмі не показувати.
7. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Надати звіт з самостійної роботи викладачу.

Побудова UML-діаграми ієрархії класів

UML-діаграма дає змогу наочно оцінити класи, їх атрибути та ієрархію зв’язків. На рисунку 1 наведена UML-діаграма, яка представляє ієрархію класів з полями, властивостями та методами.

Клас Pet описує домашню тварину з атрибутами ім’я (name), вік (age) і колір (color). Має віртуальні методи для звуку, гри та отримання інформації, а

також звичайний метод сну.

Клас WorkerAnimal представляє робочу тварину з атрибутами ім’я (name), роль (role) та сила (strength). Містить віртуальні методи для виконання роботи й отримання інформації.

Клас Dog наслідує Pet, додає атрибут породи (breed), перевизначає методи базового класу та вводить унікальний метод охорони.

Клас Cat також наслідує Pet, додає логічний атрибут isIndoor, перевизначає базові методи та реалізує метод полювання.

Клас Horse наслідує WorkerAnimal, має додатковий атрибут speed, перевизначає методи та додає метод галопування.

Клас Donkey наслідує WorkerAnimal, вводить атрибут вантажопідйомності (loadCapacity), перевизначає методи та реалізує поведінку впертої зупинки.

Клас PetManager керує колекцією тварин типу object, зберігає їх у списку, дозволяє додавати тварин, отримувати інформацію, запускати ігрові або робочі методи й визначати кількість тварин.

Рисунок 1 – UML-діаграма ієрархії класів

Приклад коду на мові C# з коментарями

У демонстраційному прикладі створення об'єктів у методі виконується створення об'єкта класу PetManager з додавання до нього різних тварин (собака, кіт, кінь, віслюк), а також виклик методів для виведення інформації, звуків, роботи та ігор для всіх доданих тварин.

```
using System.Collections.Generic;
using System.Text;

// Перший батьківський клас: Домашня тварина
public class Pet
{
    protected string name; // Ім'я тварини
    protected int age;     // Вік тварини
    protected string color; // Колір тварини

    // Конструктор
    public Pet(string name, int age, string color)
    {
        this.name = name;
        this.age = age;
        this.color = color;
        Console.WriteLine($"{name} (Домашня тварина) створено.");
    }

    // Деструктор
    ~Pet()
    {
        Console.WriteLine($"{name} (Домашня тарина) знищено.");
    }

    // Властивості
    public string Name => name;
    public int Age => age;
    public string Color => color;

    // Віртуальний метод для звуку
    public virtual void MakeSound()
    {
        Console.WriteLine($"{name} видає якийсь звук.");
    }

    // Віртуальний метод для інформації
    public virtual string GetInfo()
    {
        return $"Ім'я: {name}, Вік: {age}, Колір: {color}";
    }

    // Віртуальний метод для гри
    public virtual void Play()
    {
        Console.WriteLine($"{name} грається звичайним чином.");
    }

    // Звичайний метод для сну
    public void Sleep()
    {
        Console.WriteLine($"{name} спить...");
    }
     }
// Другий батьківський клас: Робоча Тварина
public class WorkerAnimal
{
    protected string name;   // Ім'я тварини
    protected string role;   // Роль тварини
    protected int strength;  // Сила тварини
```

```
    // Конструктор
    public WorkerAnimal(string name, string role, int strength)
    {
        this.name = name;
        this.role = role;
        this.strength = strength;
        Console.WriteLine($"{name} (Робоча тварина) створено.");
    }

    // Деструктор
    ~WorkerAnimal()
    {
        Console.WriteLine($"{name} (Робоча тварина) знищено.");
    }

    // Властивості
    public string Name => name;
    public string Role => role;
    public int Strength => strength;

    // Віртуальний метод для роботи
    public virtual void DoWork()
    {
        Console.WriteLine($"{name} виконує завдання {role}.");
    }

    // Віртуальний метод для інформації
    public virtual string GetInfo()
    {
        return $"Ім'я: {name}, Роль: {role}, Сила: {strength}";
    }
}

// Похідний клас 1 від Тварина: Собака
public class Dog : Pet
{
    private string breed; // Приховане поле: порода

    // Конструктор
    public Dog(string name, int age, string color, string breed)
        : base(name, age, color)
    {
        this.breed = breed;
        Console.WriteLine($"{name} (Собака) створено.");
    }

    // Деструктор
    ~Dog()
    {
        Console.WriteLine($"{name} (Собака) знищено.");
    }

    // Перевизначення віртуального методу
    public override void MakeSound()
    {
        Console.WriteLine($"{name} каже: Гав! Гав!");
    }

    // Перевизначення віртуального методу
    public override string GetInfo()
    {
        return base.GetInfo() + $", Порода: {breed}";
    }

    // Перевизначення методу гри
    public override void Play()
    {
        Console.WriteLine($"{name} грається в приношення м'яча!");
    }
```

```
    // Унікальний метод
    public void Guard()
    {
        Console.WriteLine($"{name} охороняє будинок!");
    }

    // Властивість для породи
    public string Breed => breed;

    // Приховування методу
    public new void Sleep()
    {
        Console.WriteLine($"{name} собака спить у своїй будці.");
    }

}
// Похідний клас 2 від Тварина: Кіт
public class Cat : Pet
{
    private bool isIndoor; // Приховане поле: чи живе в приміщенні

    // Конструктор
    public Cat(string name, int age, string color, bool isIndoor)
        : base(name, age, color)
    {
        this.isIndoor = isIndoor;
        Console.WriteLine($"{name} (Кіт) створено.");
    }

    // Деструктор
    ~Cat()
    {
        Console.WriteLine($"{name} (Кіт) знищено.");
    }

    // Перевизначення віртуального методу
    public override void MakeSound()
    {
        Console.WriteLine($"{name} каже: Няв! Няв!");
    }

    // Перевизначення віртуального методу
    public override string GetInfo()
    {
        return base.GetInfo() + $", У приміщенні: {(isIndoor ? "Так" : "Ні")}";
    }

    // Перевизначення методу гри
    public override void Play()
    {
        Console.WriteLine($"{name} грається з клубком пряжі!");
    }

    // Унікальний метод
    public void Hunt()
    {
        Console.WriteLine($"{name} полює на мишей!");
    }

    // Властивість для isIndoor
    public bool IsIndoor => isIndoor;

    // Приховування методу
    public new void Sleep()
    {
        Console.WriteLine($"{name} кіт спить на подушці.");
    }
}
```

```
// Похідний клас 1 від Робоча Тварина: Кінь
public class Horse : WorkerAnimal
{
    private double speed; // Приховане поле: швидкість

    // Конструктор
    public Horse(string name, string role, int strength, double speed)
        : base(name, role, strength)
    {
        this.speed = speed;
        Console.WriteLine($"{name} (Кінь) створено.");
    }
    // Деструктор
    ~Horse()
    {
        Console.WriteLine($"{name} (Кінь) знищено.");
    }
    // Перевизначення віртуального методу
    public override void DoWork()
    {
        Console.WriteLine($"{name} тягне віз зі швидкістю {speed} км/год.");
    }
    // Перевизначення віртуального методу
    public override string GetInfo()
    {
        return base.GetInfo() + $", Швидкість: {speed} км/год";
    }
    // Унікальний метод
    public void Gallop()
    {
        Console.WriteLine($"{name} галопує полем!");
    }
    // Властивість для швидкості
    public double Speed => speed;
    // Приховування властивості
    public new string Role => $"спеціалізована роль: {role}";
}

// Похідний клас 2 від Робоча Тварина: Віслюк
public class Donkey : WorkerAnimal
{
    private int loadCapacity; // Приховане поле: вантажопідйомність
    // Конструктор
    public Donkey(string name, string role, int strength, int loadCapacity)
        : base(name, role, strength)
    {
        this.loadCapacity = loadCapacity;
        Console.WriteLine($"{name} (Віслюк) створено.");
    }
    // Деструктор
    ~Donkey()
    {
        Console.WriteLine($"{name} (Віслюк) знищено.");
    }
    // Перевизначення віртуального методу
    public override void DoWork()
    {
        Console.WriteLine($"{name} несе вантаж вагою {loadCapacity} кг.");
    }
    // Перевизначення віртуального методу
    public override string GetInfo()
    {
        return base.GetInfo() + $", Вантажопідйомність: {loadCapacity} кг";
    }
    // Унікальний метод
    public void StubbornStop()
    {
        Console.WriteLine($"{name} вперто зупиняється і відмовляється рухатися!");
    }
```

```
    // Властивість для вантажопідйомності
    public int LoadCapacity => loadCapacity;

    // Приховування властивості
    public new string Role => $"спеціалізована роль: {role}";
}
// Клас для управління тваринами
public class PetManager
{
    private List<object> animals; // Список для зберігання всіх тварин

    // Конструктор
    public PetManager()
    {
        animals = new List<object>();
        Console.WriteLine("Менеджер тварин створено.");
    }

    // Деструктор
    ~PetManager()
    {
        Console.WriteLine("Менеджер тварин знищено.");
    }

    // Додавання тварини
    public void AddAnimal(object animal)
    {
        animals.Add(animal);
        string name = animal is Pet pet ? pet.Name : ((WorkerAnimal)animal).Name;
        Console.WriteLine($"Додано тварину: {name}");
    }

    // Показати інформацію про всіх тварин
    public void ShowAllInfo()
    {
        Console.WriteLine("\n=== Усі Тварини ===");
        foreach (var animal in animals)
        {
            Console.WriteLine(animal is Pet pet ? pet.GetInfo() :
                              ((WorkerAnimal)animal).GetInfo());
        }
    }

    // Виконати звуки всіх тварин (тільки для Pet)
    public void MakeAllSounds()
    {
        Console.WriteLine("\n=== Звуки Тварин ===");
        foreach (var animal in animals)
        {
            if (animal is Pet pet)
            {
                pet.MakeSound();
            }
        }
    }

    // Виконати роботу всіх робочих тварин
    public void DoAllWork()
    {
        Console.WriteLine("\n=== Робочі Дії ===");
        foreach (var animal in animals)
        {
            if (animal is WorkerAnimal worker)
            {
                worker.DoWork();
            }
        }
    }
    // Грати з усіма тваринами (тільки для Pet)
```

```
    public void PlayWithAll()
    {
        Console.WriteLine("\n=== Час Гри ===");
        foreach (var animal in animals)
        {
            if (animal is Pet pet)
            {
                pet.Play();
            }
        }
    }

    // Кількість тварин
    public int AnimalCount => animals.Count;
}
// Демонстрація роботи
public class Program
{
    public static void Main()
    {
        // підтримка українських літер в консолі
        Console.OutputEncoding = Encoding.UTF8;

        // Створення менеджера
        var manager = new PetManager();
        // Створення тварин
        var dog = new Dog("Рекс", 3, "Коричневий", "Золотистий Ретривер");
        var cat = new Cat("Мурзик", 2, "Сірий", true);
        var horse = new Horse("Гром", "Транспорт", 80, 40.5);
        var donkey = new Donkey("Мирко", "Вантажник", 60, 200);

        // Додавання тварин
        manager.AddAnimal(dog);
        manager.AddAnimal(cat);
        manager.AddAnimal(horse);
        manager.AddAnimal(donkey);

        // Показати інформацію
        manager.ShowAllInfo();
        // Виконати звуки (тільки для Pet)
        manager.MakeAllSounds();
        // Виконати роботу (тільки для WorkerAnimal)
        manager.DoAllWork();
        // Грати з тваринами
        manager.PlayWithAll();
        // Унікальні дії
        Console.WriteLine("\n=== Спеціальні Дії ===");
        dog.Guard();
        cat.Hunt();
        horse.Gallop();
        donkey.StubbornStop();

        // Демонстрація приховування методів і властивостей
        Console.WriteLine("\n=== Приховані Методи та Властивості ===");
        dog.Sleep(); // Викликає Sleep собаки
        cat.Sleep(); // Викликає Sleep кота
        ((Pet)dog).Sleep(); // Викликає Sleep батьківського класу
        ((Pet)cat).Sleep(); // Викликає Sleep батьківського класу
        Console.WriteLine($"Кінь, {horse.Role}"); // Викликає Role коня
        Console.WriteLine($"Віслюк, {donkey.Role}"); // Викликає Role віслюка

        // Загальна кількість тварин
        Console.WriteLine($"\nЗагальна кількість тварин: {manager.AnimalCount}");

        // Виклик збору сміття для демонстрації деструкторів
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

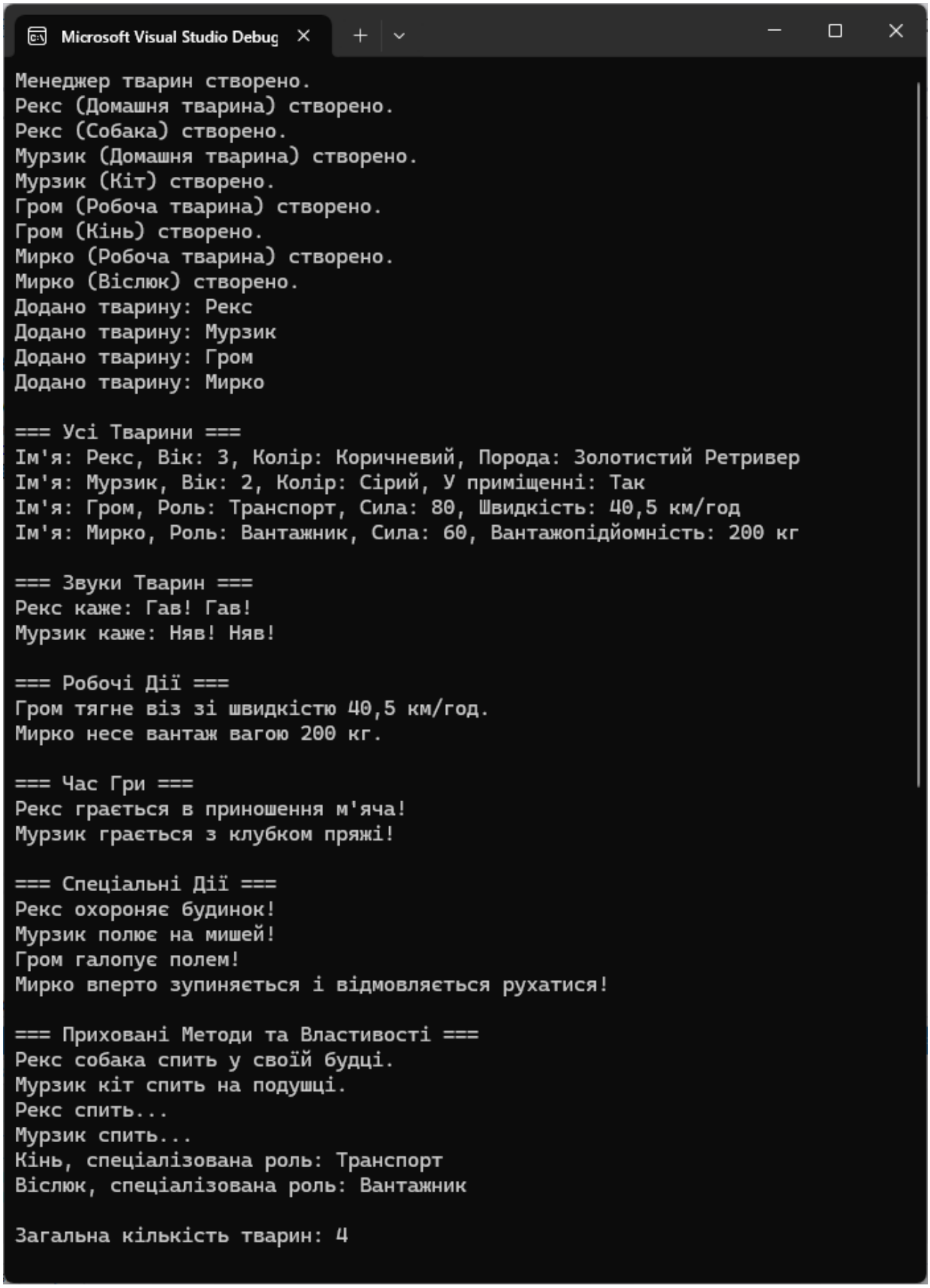

Рисунок 2 – Результат виконання програми

Контрольні запитання

1. Як працює поліморфізм у випадку класів Pet, Dog і Cat?
2. Чим відрізняється метод Sleep() у базовому класі Pet та похідних класах?
3. Чому поля name у класах Pet та WorkerAnimal не конфліктують?
4. Як PetManager демонструє принцип інкапсуляції?
5. Яким чином у класі PetManager визначено тип усіх тварин?
6. Як виконується перевірка типу тварини у методах PetManager?
7. Що таке метод-приховувач (new) і як він відрізняється від override?
8. Як у класах Dog і Cat реалізовано розширення поведінки методу GetInfo()?
9. Що станеться, якщо викликати метод Sleep() у змінній типу Pet, яка посилається на об'єкт Dog?
10. Як можна було б використати абстрактні класи для спрощення структури?

ТЕМА 3. АБСТРАКТНІ КЛАСИ ТА ЕЛЕМЕНТИ. ЗАПЕЧАТАНІ КЛАСИ. СТАТИЧНІ КЛАСИ. ФАЙЛОВА СТРУКТУРА ПРОЄКТІВ. ПЕРЕВИЗНАЧЕННЯ ОПЕРАТОРІВ

Лекція 5.

Абстрактні, запечатані та статичні класи. Файлова структура проєктів. Перевизначення стандартних операторів в класах

Навчальні питання:

1. Абстрактні класи та абстрактні елементи:
– поняття абстрактного класу;
– абстрактні методи, властивості, індексатори;
– реалізація в похідних класах.
2. Запечатаний клас:
– призначення та обмеження;
– запечатування методів;
– оптимізація продуктивності класів.
3. Статичні класи та статичні елементи класів.
– особливості статичних класів;
– статичні конструктори, поля, методи.
4. Файлова структура проєктів на мові C# у Visual Studio:
– основні папки;
– рекомендації з організації коду.
5. Перевизначення стандартних операторів в класах:
– синтаксис перевантаження
– обмеження та правила.
6. Висновки та рекомендації.

Стислий конспект лекції

1. У мові C# абстрактні класи оголошують шляхом розміщення ключового слова «abstract» перед ім’ям класу [8, 10]. Абстрактний клас є шаблоном для створення ієрархії класів і не дозволяє створювати власні екземпляри. Він визначає спільну поведінку та структуру для похідних класів, забезпечуючи поліморфізм [11]. Абстрактні елементи класів (методи, властивості, події, індексатори), позначені abstract, не мають реалізації та мусять бути реалізовані в похідних класах, якщо ті не є абстрактними. Звичайні члени (методи, поля, властивості) можуть мати реалізацію, яка успадковується. Абстрактні класи

використовуються для моделювання загальних концепцій. Вони підтримують віртуальні методи (virtual), які можуть бути перевизначені (override) у похідних класах, і дозволяють зберігати стан через поля та властивості [12]. Абстрактний клас не може бути запечатаним, а створення об’єктів абстрактних класів неможливе.

```
// Абстрактний клас, що представляє транспортний засіб
using System;

abstract class Vehicle
{
    // Властивість для зберігання моделі транспортного засобу
    public string Model { get; set; }

    // Абстрактний метод для запуску двигуна – має бути реалізований у похідному класі
    public abstract void StartEngine();

    // Віртуальний метод для зупинки транспортного засобу – можна перевизначити
    public virtual void Stop() => Console.WriteLine("Vehicle stopped.");
}

// Клас, що представляє автомобіль, успадковує клас Vehicle
class Car : Vehicle
{
    // Конструктор, що встановлює модель автомобіля
    public Car(string model) => Model = model;

    // Реалізація методу запуску двигуна
    public override void StartEngine() => Console.WriteLine($"{Model} engine started.");

    // Перевизначений метод зупинки автомобіля
    public override void Stop() => Console.WriteLine($"{Model} car stopped.");
}

// Клас з методом Main – точка входу в програму
class Program
{
    static void Main()
    {
        // Створення об'єкта типу Car, але через посилання на базовий тип Vehicle
        Vehicle vehicle = new Car("Toyota");

        // Виклик методу запуску двигуна
        vehicle.StartEngine(); // Виведе: Toyota engine started.

        // Виклик методу зупинки (перевизначена версія в Car)
        vehicle.Stop(); // Виведе: Toyota car stopped.
    }
}
```

2. Запечатаний клас, позначений ключовим словом sealed, забороняє спадкування від нього, що обмежує ієрархію класів [8]. Це корисно, коли клас є остаточною реалізацією, і подальше розширення може порушити його логіку або безпеку [9]. Запечатування оптимізує продуктивність, дозволяючи CLR замінювати віртуальні виклики на прямі, що зменшує накладні витрати. Окремі методи в похідних класах також можуть бути запечатані (sealed override), щоб запобігти їх подальшому перевизначенню [12]. Застосовується в бібліотеках, де потрібно захистити код, або в класах із фіксованою поведінкою, наприклад, утилітарних класах. Обмеження: запечатаний клас не може бути абстрактним, а надмірне використання може ускладнити тестування.

```
// Запечатаний клас Settings – від нього не можна успадковуватись
sealed class Settings
{
    // Властивість, що визначає тему оформлення
    public string Theme { get; set; }

    // Метод, що застосовує тему
    public void Apply() => Console.WriteLine($"Тема {Theme} застосована.");
}

// class CustomSettings : Settings { }
// Помилка: успадкування заборонено, оскільки клас Settings оголошено як sealed

// Базовий клас із віртуальним методом
class Base
{
    public virtual void Process() => Console.WriteLine("Обробка у Base.");
}

// Похідний клас, що перевизначає метод і забороняє подальше перевизначення
class Derived : Base
{
    public sealed override void Process() => Console.WriteLine("Обробка у Derived.");
    // sealed override – не дозволяє перевизначити цей метод у наступних похідних
класах
}

// Точка входу в програму
class Program
{
    static void Main()
    {
        // Створення об'єкта Settings і встановлення теми
        Settings settings = new Settings { Theme = "Темна" };

        // Застосування теми
        settings.Apply(); // Виведе: Тема Темна застосована.
    }
}
```

3. Статичний клас, позначений ключовим словом static, не дозволяє створювати екземпляри та містить лише статичні члени (поля, методи, властивості, події) [8, 10]. Статичні члени належать класу, а не об’єкту, і доступні через ім’я класу. Вони ініціалізуються один раз перед першим використанням, а статичний конструктор (static) виконується автоматично для ініціалізації. Використовуються для групування утилітарних функцій, констант або допоміжних методів, які не залежать від стану об’єкта [9]. Наприклад, клас Math у .NET є статичним. Обмеження: статичний клас не може реалізовувати інтерфейси або бути успадкованим [12]. Застосовується для логування, математичних обчислень, роботи з конфігурацією або кешування.

Статичні елементи належать класу в цілому, а не окремому екземпляру. Вони позначаються модифікатором static і існують навіть без створення об'єктів. Статичні методи можуть звертатися лише до інших статичних елементів класу.

```
// Статичний клас, який надає допоміжні методи
static class Utility
{
    // Статичне поле для підрахунку операцій
    private static int operationCount;
```

```
    // Статичний конструктор – виконується один раз перед першим використанням класу
    static Utility() => operationCount = 0;

    // Статичний метод для форматування повідомлення з номером операції
    public static string FormatMessage(string input)
    {
        operationCount++;
        return $"[{operationCount}] {input}";
    }
}

class Program
{
    static void Main()
    {
        // Виклик FormatMessage – інкрементує лічильник і додає його до повідомлення
        // Виведе: [1] Операцію розпочато
        Console.WriteLine(Utility.FormatMessage("Операцію розпочато"));
        // Виведе: [2] Операцію завершено
        Console.WriteLine(Utility.FormatMessage("Операцію завершено"));
    }
}
```

4. Файлова структура в C# забезпечує логічну організацію коду для полегшення розробки, підтримки та масштабування [13-14]. Основний файл Program.cs містить точку входу (Main). Код групується в папки за функціоналом, якщо є потреба, наприклад Models для класів даних, Services для бізнес-логіки, Controllers для обробки запитів у вебпроєктах, Data для роботи з базами даних, Utils для утиліт. Папка Properties містить конфігураційні файли, наприклад, appsettings.json для налаштувань. Файл .csproj визначає залежності, версію .NET і параметри збирання. Простори імен (namespace) відповідають структурі папок, наприклад, MyApp.Models. Рішення (.sln) об'єднують кілька проєктів. Рекомендації: використовувати шаблони проєктів (наприклад, ASP.NET Core), дотримуватися стилей іменування (PascalCase для класів), уникати надмірної вкладеності папок. Для великих проєктів застосовуються принципи розподілу за модулями (Feature-based structure) або шарами (Layered architecture) [15].

```
MySolution.sln
└── MyApp/
    ├── Program.cs
    ├── Models/
    │   └── Product.cs
    ├── Services/
    │   └── ProductService.cs
    ├── Controllers/
    │   └── ProductController.cs
    ├── Data/
    │   └── AppDbContext.cs
    ├── Utils/
    │   └── Logger.cs
    ├── Properties/
    │   └── appsettings.json
    └── MyApp.csproj
```

Рисунок 1 – Приклад файлової структури проєкту

5. Перевизначення операцій в C# дозволяє змінювати поведінку стандартних операторів для конкретних класів або структур даних [8, 11].

Зазвичай так діють, коли потрібно виконувати операції над об'єктами користувацьких типів даних, наприклад, додавання двох векторів або порівняння двох об'єктів. Це дозволяє використовувати перевизначені оператори над користувацькими типами так само природно, як і для вбудованих типів (int, string).

Щоб перевизначити оператор, потрібно:

1. використати ключове слово *public static*;

2. вказати тип повернення результату обчислень перевизначеним оператором;

3. використати ключове слово *operator* з оператором, який потрібно перевизначити (наприклад, +, –, ==, != тощо);

4. визначити вхідні параметри та виконати програмну реалізацію оператора.

Не всі операції можуть бути перевантажені. Існують певні правила та обмеження на перевантаження операцій, які показано у табл. 1

Перевантаження дозволяє інтуїтивно використовувати звичні оператори, застосовуючи їх до об'єктів класів, і при цьому розширювати логіку цих об'єктів безпосередньо через оператори, роблячи код більш зрозумілим і зручним для користування [12].

Таблиця 1 – Перелік операторів, які підлягають перевантаженню [16]

Операція	**Можливості та особливості перевантаження**
+, –, !, ~, + +, – –, true, false	Унарні символи операцій допускають навантаження. Значення true та false також є операціями
+, –, *, /, %, &, |, ^, <<, >>	Бінарні символи операцій, що допускають перевантаження
==, !=, <, >, <=, >=	Операції порівняння перевантажуються
&&, ||	Умовні логічні операції моделюються з допомогою раніше перевизначених операцій «&» та «|»
[]	Операції доступу до елементів масивів моделюються за рахунок індексаторів
+=, –=, *=, /=, %=, &=, =, ^=, <<=, >>=	Операції не перевантажуються через неможливість перевантаження операції присвоєння
=, ., ?:, –>, new, is, sizeof, typeof	Операції, що не підлягають перевантаженню

6. **Висновки та рекомендації.** Використовуйте абстрактні класи для створення ієрархій зі спільною логікою, яка має бути уточнена в похідних класах. Абстрактні методи забезпечують контракт для похідних класів, але не надають реалізації за замовчуванням. Обирайте між абстрактними класами та інтерфейсами, враховуючи потребу у спільному стані. Застосовуйте sealed для класів, які не повинні бути успадковані (наприклад, з міркувань безпеки або

продуктивності). Використовуйте sealed override для методів, які не повинні змінюватися в подальших похідних класах. Уникайте надмірного використання sealed і static – це може ускладнити модульне тестування. Дотримуйтесь стандартних шаблонів файлових проєктів (наприклад, MVC, Clean Architecture) для полегшення командного розвитку. Групуйте файли за функціональністю (Features) або за рівнями доступу (Layers). Завжди коментуйте перевизначені оператори та нестандартні рішення.

Лабораторна робота 8.

Абстрактні класи та абстрактні елементи класу. Запечатані та статичні класи

Мета роботи: опанувати концепцію абстрактних класів та їх елементів у мові C#; здобути навички використання абстрактних методів та властивостей для формування шаблонів класів; зрозуміти призначення запечатаних класів та особливості їх застосування; вивчити статичні класи та методи.

Лабораторне завдання

1. Використовуючи теоретичні відомості та літературу, ознайомитись з особливостями реалізації абстрактних, запечатаних та статичних класів у C#.
2. Дати відповіді на контрольні запитання.
3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.
4. Створити базовий абстрактний клас відповідно до індивідуального варіанту предметної області. Включити до класу неабстрактні поле та властивість, абстрактні властивість та метод, неабстрактний метод та конструктор. За необхідності додати інші елементи класу.
5. Реалізувати запечатаний клас-спадкоємець абстрактного класу з обов'язковою реалізацією всіх абстрактних елементів базового класу.
6. Створити статичний клас з статичними полем, властивістю та методом для доповнення функціональності системи.
7. Встановити відповідні рівні доступності класів: загальний (public) та внутрішній (internal).
8. У головній точці входу програми void Main() продемонструвати створення всіх можливих об'єктів, виклик перевизначених методів, звернення до статичного поля та виклик статичного методу.
9. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод static void Main) на діаграмі не показувати.
10. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору

тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми ієрархії класів

На рис. 1 представлено PlantUML-код UML-діаграми, яка демонструє ієрархію класів з абстрактним базовим класом Animal, запечатаним класом Kitten та статичним класом Predator.

Зверніть увагу, яким синтаксисом позначено абстрактні та статичні елементи!

```
@startuml

abstract class Animal
{
    - type : string = "Тварина"
    + Type : string { get; set }
    + {abstract} NickName: string { get; set }
    + Animal(string)
    + {abstract} MakeSound()
    + BePredator()
}

static class God
{
    + {static} name: string = "Бог"
    + {static} Gender: string { get }
    + {static} BeGod()
}


class Kitten << final >>
{
    + numOfPaws : int
    - nickName : string = "кіт"
    + NickName : string
    + Kitten(string, string, int)
    + MakeSound()
}

Animal <|-- Kitten

@enduml
```

Рисунок 1 – PlantUML-код UML-діаграми ієрархії класів [16]

Діаграма показує взаємозв'язки між класами, їх поля, властивості, методи та конструктори з відповідними модифікаторами доступу.

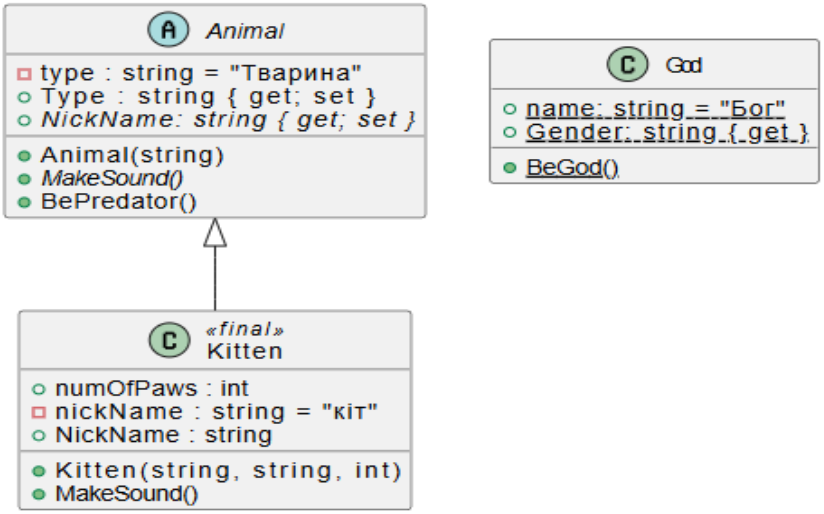

Рисунок 2 – UML-діаграма ієрархії класів [16]

Абстрактний клас Animal містить поле type, яке використовується для зберігання типу тварини. Доступ до цього поля надається через властивість Type. Клас також оголошує абстрактну властивість NickName, яка призначена для роботи з прізвиськом тварини, а її реалізація передбачається у похідних класах. Для ініціалізації типу тварини передбачено конструктор. Крім того, клас має абстрактний метод MakeSound(), який реалізується в спадкоємцях, та неабстрактний метод BePredator(), що демонструє поведінку хижака.

Статичний клас God включає статичне поле name, яке зберігає ім’я. Для отримання гендерної інформації в класі визначено статичну властивість Gender. Крім того, метод BeGod() виводить відповідне повідомлення, демонструючи функціональність божественної сутності.

Запечатаний клас Kitten має поле numOfPaws, яке вказує кількість лап, та приватне поле nickName, що використовується для зберігання прізвиська. Властивість NickName реалізується з валідацією згідно з вимогами абстрактної бази. У класі є конструктор, що ініціалізує всі поля. Також реалізується метод MakeSound(), який конкретизує звукову поведінку кошеняти.

Приклад коду на мові C# з коментарями

```
using System;
using System.Text;

// Абстрактний клас "Тварина"
internal abstract class Animal
{
    // Неабстрактне поле
    private string type = "Тварина";

    // Неабстрактна властивість
    public string Type
    {
        get { return type; }
        set { type = value; }
    }

    // Абстрактна властивість "прізвисько"
    abstract public string NickName { get; set; }

    // Неабстрактний конструктор
    public Animal(string type)
    {
        Type = type;
    }

    // Абстрактний метод
    public abstract void MakeSound();

    // Неабстрактний метод
    public void BePredator()
    {
        Console.WriteLine("Хижаки завжди харчуються м'ясом!");
    }
}
```

```
// Статичний клас "Бог"
internal static class God
{
    // Статичне поле
    public static string name = "Бог";

    // Статична властивість
    public static string Gender
    {
        get { return "невизначений"; }
    }

    // Статичний метод
    public static void BeGod()
    {
        Console.WriteLine("Бог нас всіх любить!");
    }
}

// Запечатаний клас "Кошеня"
internal sealed class Kitten : Animal
{
    // Власне поле - кількість лап
    public int numOfPaws;

    // Власне поле - прізвисько
    public string nickName = "кіт";

    // Реалізація базової абстрактної властивості
    public override string NickName
    {
        get { return nickName; }
        set
        {
            // Контроль коректності значення
            if (value == "Васька" || value == "Аліса")
            {
                nickName = "Барсік";
            }
            else
            {
                nickName = value;
            }
        }
    }

    // Конструктор класу Kitten
    public Kitten(string type, string nickname, int numofpaws)
        : base(type)
    {
        numOfPaws = numofpaws;
        NickName = nickname;
        Console.WriteLine($"{NickName} - це маленьке {type}, " +
                          $"у нього {numOfPaws} лапи та немає нащадків!");
    }

    // Реалізація абстрактного методу "MakeSound"
    public override void MakeSound()
    {
        Console.WriteLine($"{NickName} муркоче: Мяу-мяу!");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = UTF8Encoding.UTF8;

        // Створення об'єкта класу "Kitten"
        Kitten myKitten = new Kitten("Кошеня", "Аліса", 4);

        // Виклик реалізованого методу "MakeSound"
        myKitten.MakeSound();

        // Виклик успадкованого методу "BePredator"
        Console.Write($"{myKitten.NickName} - нащадок хижаків, тому ...");
        myKitten.BePredator();

        // Звернення до статичного поля
        Console.WriteLine($"Верховна духовна Сутність - це {God.name}.");
        Console.WriteLine($"Його гендер - {God.Gender}.");

        // Виклик статичного методу
        God.BeGod();
    }
}
```

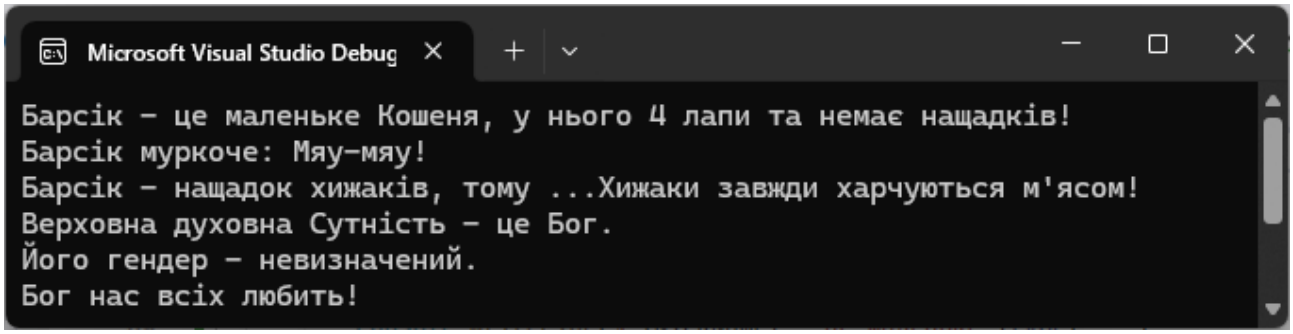

Рисунок 3 – Результат виконання програми

Контрольні запитання

1. Чому абстрактний клас Animal не може бути інстанційований безпосередньо?
2. Які елементи обов'язково повинен реалізувати клас-спадкоємець абстрактного класу?
3. Чому клас Kitten оголошено як запечатаний?
4. Як здійснюється звернення до статичних елементів класу God?
5. Яка різниця між абстрактними та неабстрактними методами в базовому класі?
6. Чому в абстрактному класі можна оголошувати неабстрактні методи?
7. Як реалізується контроль валідності в властивості NickName?
8. Яке призначення ключового слова override в методі MakeSound()?
9. Чому статичний клас може містити лише статичні елементи?
10. Які переваги надає використання запечатаних класів у системі?

Лабораторна робота 9.

Робота з файловою структурою C#-проєкту

Мета роботи: засвоїти принципи структурування консольних проєктів на мові C# через розділення класів по окремих файлах для забезпечення модульності та можливості повторного використання коду.

Лабораторне завдання

1. Використовуючи теоретичні відомості та літературу, ознайомитись з особливостями реалізації файлової структури проєктів на мові C# у Visual Studio та рекомендаціями.
2. Дати відповіді на контрольні запитання.
3. Ознайомитися з наведеним прикладом.
4. Здійснити реструктуризацію проєкту з лабораторної роботи №8 шляхом винесення всіх класів в окремі cs-файли.
5. Розмістити базові та похідні класи у відповідних cs-файлах з дотриманням правил іменування. Головний метод Main() залишити у файлі Program.cs для демонстрації функціональності класів.
6. Забезпечити коректне використання простору імен (namespace) для всіх файлів проєкту.
7. UML-діаграма ієрархії класів *не потрібна*.
8. У звіті надати лістинги усіх файлів проєкту. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Додати у звіт скріншоти файлової структури проєкту та результатів виконання програми. Захистити лабораторну роботу викладачу.

Приклад файлової структури проєкту

На рис. 1 показано приклад додавання до проєкту файлу з класом Animal.

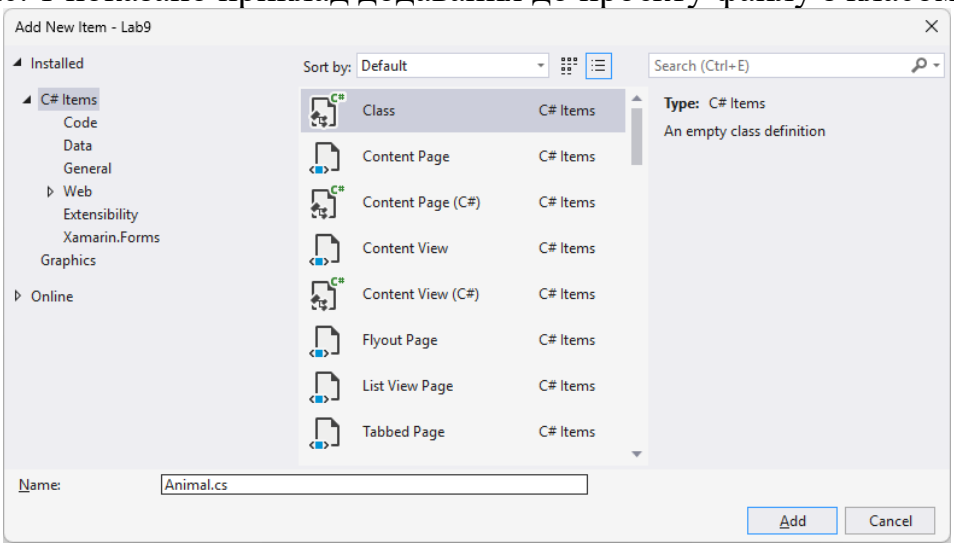

Рисунок 1 – Додавання файлу Animal.cs до проєкту

На рис. 2 показана повна файлова структуру C#-проєкту, де кожен клас розміщений в окремому файлі для забезпечення модульності та підтримки коду.

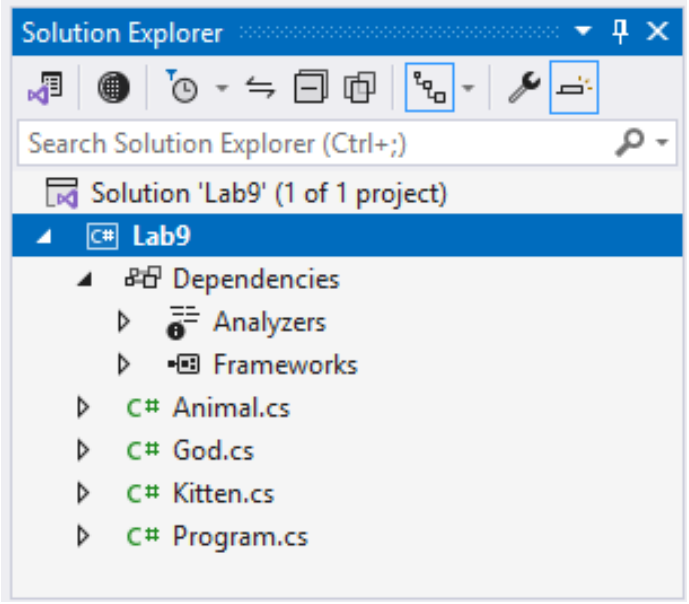

Рисунок 2 – Файлова структура C#-проєкту

Реалізація класів у окремих файлах

Файл Animal.cs, базовий абстрактний клас:

```
namespace Lab9
{
    internal abstract class Animal
    {
    // Неабстрактне поле
    private string type = "Тварина";

    // Неабстрактна властивість
    public string Type
    {
        get { return type; }
        set { type = value; }
    }

    // Абстрактна властивість "прізвисько"
    abstract public string NickName
        { get; set; }

    // Неабстрактний конструктор
    public Animal(string type)
    {
        Type = type;
    }

    // Абстрактний метод
    public abstract void MakeSound();

    // Неабстрактний метод
    public void BePredator()
    {
        Console.WriteLine("Хижаки завжди харчуються м'ясом!");
    }
    }
}
```

Файл God.cs, статичний клас:

```
namespace Lab9
{
    internal static class God
    {
    // Статичне поле
    public static string name = "Бог";

    // Статична властивість
    public static string Gender
    {
        get { return "невизначений"; }
    }

    // Статичний метод
    public static void BeGod()
    {
        Console.WriteLine("Бог нас всіх любить!");
    }

    }
}
```

Файл Kitten.cs, похідний клас:

```
namespace Lab9
{
    internal sealed class Kitten : Animal
{
    // Власне поле – кількість лап
    public int numOfPaws;

    // Власне поле – прізвисько
    public string nickName = "кіт";

    // Реалізація базової абстрактної властивості
    public override string NickName
    {
        get { return nickName; }
        set
        {
            // Контроль коректності значення
            if (value == "Васька" || value == "Аліса")
            {
                nickName = "Барсік";
            }
            else
            {
                nickName = value;
            }
        }
    }

    // Конструктор класу Kitten
    public Kitten(string type, string nickname, int numofpaws)
        : base(type)
    {
        numOfPaws = numofpaws;
        NickName = nickname;
        Console.WriteLine($"{NickName} – це маленьке {type}, " +
                          $"у нього {numOfPaws} лапи та немає нащадків!");
    }

    // Реалізація абстрактного методу "MakeSound"
    public override void MakeSound()
```

```
    {
        Console.WriteLine($"{NickName} муркоче: Мяу-мяу!");
    }
}
}
```

Файл Program.cs, головний клас програми:

```
using Lab9;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = UTF8Encoding.UTF8;

        // Створення об'єкта класу "Kitten"
        Kitten myKitten = new Kitten("Кошеня", "Аліса", 4);

        // Виклик реалізованого методу "MakeSound"
        myKitten.MakeSound();

        // Виклик успадкованого методу "BePredator"
        Console.Write($"{myKitten.NickName} - нащадок хижаків, тому ...");
        myKitten.BePredator();

        // Звернення до статичного поля
        Console.WriteLine($"Верховна духовна Сутність - це {God.name}.");
        Console.WriteLine($"Його гендер - {God.Gender}.");

        // Виклик статичного методу
        God.BeGod();
    }
}
```

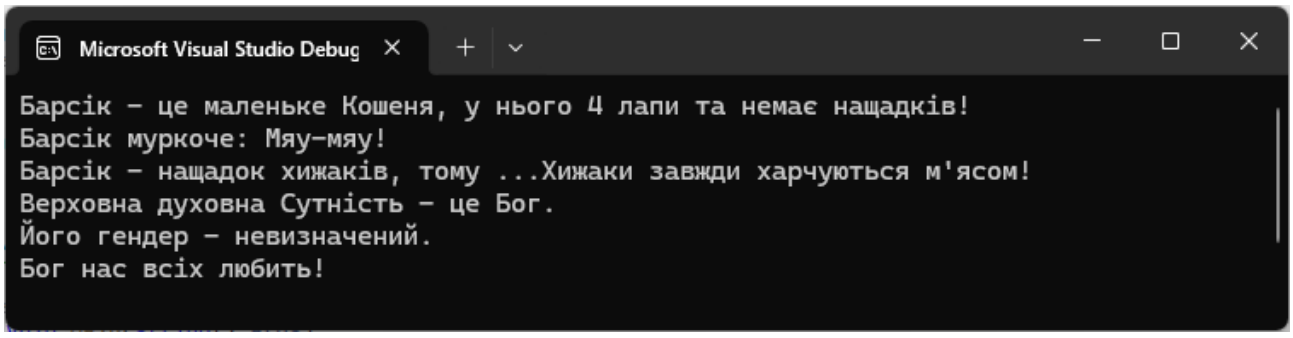

Рисунок 3 – Перевірка працездатності програми

Контрольні запитання

1. Які переваги дає розміщення класів у окремих файлах?
2. Чому важливо дотримуватися принципу «один клас – один файл»?
3. Як організовується простір імен (namespace) у структурованому проєкті?
4. Які правила іменування файлів слід дотримуватися при створенні класів?
5. Як забезпечується модульність та можливість повторного використання коду?
6. Чому головний метод Main() залишається у файлі Program.cs?
7. Як впливає файлова структура на читабельність та підтримку коду?

8. Які засоби Visual Studio допомагають у роботі з файловою структурою?

9. Як правильно організувати залежності між класами в різних файлах?

10. Які принципи SOLID реалізуються через правильну організацію файлової структури?

Лабораторна робота 10.

Перевизначення стандартних операторів в класах

Мета роботи: здобути навички модифікації поведінки стандартних операторів для класів чи структур даних користувача.

Лабораторне завдання

1. Використовуючи теоретичні відомості та літературу, ознайомитись з особливостями реалізації файлової структури проєктів на мові C# у Visual Studio та рекомендаціями.

2. Дати відповіді на контрольні запитання.

3. Ознайомитися з наведеним прикладом.

4. Модифікувати проєкт з лабораторної роботи №4 за допомогою перевизначення операторів «true, false, &, !, +, --».

5. У головному методі Main() показати процес створення екземплярів базового класу та застосування перевизначених операторів.

6. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод Main) на діаграмі не показувати.

7. У звіті надати UML-діаграму з описом класу та його атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Захистити лабораторну роботу викладачу.

Побудова UML-діаграми класу

Клас Dog: представляє модель собаки з властивостями Name (ім'я) та IsHappy (емоційний стан). Клас перевизначає стандартні оператори true, false, &, !, +, і --. Оператор «+» поєднує імена собак і встановлює позитивний стан, якщо принаймні одна собака щаслива. Оператор «-» змінює емоційний стан на негативний і модифікує ім'я префіксом. Оператор «!» змінює емоційний стан на протилежний і модифікує ім'я.

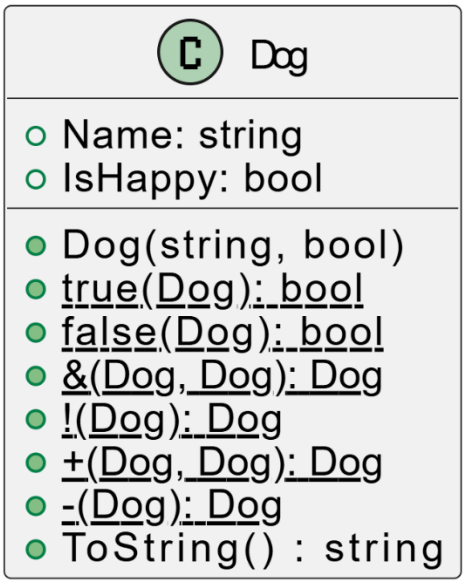

Рисунок 1 – UML-діаграма класу Dog

Приклад коду на мові C# з коментарями [16]

```
using System;
using System.Text;

class Dog
{
    public string Name { get; set; }
    public bool IsHappy { get; set; }

    public Dog(string name, bool isHappy)
    {
        Name = name;
        IsHappy = isHappy;
    }
    // Перевизначення оператора true: повертає істину, якщо собака в позитивному стані
    public static bool operator true(Dog dog)
    {
        return dog.IsHappy;
    }
    // Перевизначення оператора false: повертає істину, якщо собака в негативному стані
    public static bool operator false(Dog dog)
    {
        return !dog.IsHappy;
    }
    // Перевизначення оператора &: поєднує двох собак з урахуванням їх емоційного стану
    public static Dog operator &(Dog dog1, Dog dog2)
    {
        bool result = dog1.IsHappy && dog2.IsHappy;
        return new Dog($"Об'єднаний_{dog1.Name}_{dog2.Name}", result);
    }
    // Перевизначення оператора !: створює нову собаку з протилежним емоційним станом
    public static Dog operator !(Dog dog)
    {
        return new Dog($"Не_{dog.Name}", !dog.IsHappy);
    }
    // Перевизначення оператора +: поєднує імена собак
    // і зберігає позитивний стан, якщо принаймні одна в позитивному стані
    public static Dog operator +(Dog dog1, Dog dog2)
    {
        return new Dog($"{dog1.Name}_{dog2.Name}", dog1.IsHappy || dog2.IsHappy);
    }
```

```
    // Перевизначення оператора --: створює нову собаку з модифікованим ім'ям
    // та встановлює негативний емоційний стан
    public static Dog operator --(Dog dog)
    {
        return new Dog($"Нещасний_{dog.Name}", false);
    }
    // Метод Tostring для текстового представлення емоційного стану собаки
    public override string Tostring()
    {
        return IsHappy ? "Щасливий" : "Нещасний";
    }
}

class Program
{
    static void Main()
    {
        Dog dog1 = new Dog("Бадді", true);
        Dog dog2 = new Dog("Макс", false);
        Console.OutputEncoding = UTF8Encoding.UTF8; // підтримка українських літер

        // Демонстрація перевизначених операторів
        Console.WriteLine($"{dog1.Name} є щасливим: {dog1}");
        Console.WriteLine($"{dog2.Name} є щасливим: {dog2}");

        Dog combinedDog = dog1 & dog2;
        Console.WriteLine($"Об'єднаний пес: {combinedDog.Name} " +
                          $"є щасливим: {combinedDog}");

        Dog notHappyDog = !dog1;
        Console.WriteLine($"Не {dog1.Name}: {notHappyDog.Name} " +
                          $"є щасливим: {notHappyDog}");

        Dog addedDogs = dog1 + dog2;
        Console.WriteLine($"Додані пси: {addedDogs.Name} " +
                          $"є щасливим: {addedDogs}");

        Dog unhappyDog = dog1--;
        Console.WriteLine($"Нещасний {dog1.Name}: {unhappyDog.Name} " +
                          $"є щасливим: {unhappyDog}");
    }
}
```

Контрольні запитання

1. Які модифікатори доступу необхідні для перевизначення операторів у C#?
2. Чому оператори «true» та «false» повинні перевизначатися разом?
3. Яка різниця між унарними та бінарними операторами при перевизначенні?
4. Які оператори не можуть бути перевизначені та чому?
5. Як працює оператор «&» у контексті класу Dog?
6. Що повертає метод Tostring() у класі Dog?
7. Яким чином оператор «+» поєднує два об'єкти класу Dog?
8. Чому оператор «–» змінює емоційний стан собаки?
9. Як на UML-діаграмі позначаються статичні методи?
10. Які переваги надає перевизначення операторів для користувацьких класів?

Самостійна робота 3.

Розробка складних ієрархій класів

Мета роботи: опанувати принципи створення складних ієрархій класів, реалізуючи концепції конструкторів, деструкторів, успадкування, поліморфізму, приховування елементів класів, абстрактних класів, запечатаних класів, статичних класів та перевизначення стандартних операторів.

Завдання до самостійної роботи

1. Використовуючи теоретичні відомості та рекомендовану літературу, оновити знання щодо конструкторів, деструкторів, успадкування, поліморфізму, приховування елементів класів, абстрактних класів, запечатаних класів, статичних класів та перевизначення операторів на мові C#.
2. Дати відповіді на контрольні запитання.
3. Ознайомитися з прикладом UML-діаграми та коду на мові C#.
4. Створити ієрархію мінімум з 4 класів, відповідно предметної області індивідуального варіанту:
   - **абстрактний клас:**
     - поля: protected string, protected int;
     - конструктор, що ініціалізує поля;
     - абстрактний метод;
     - віртуальний метод.
   - **запечатаний клас, що унаслідується від абстрактного**:
     - поля: private string, private int;
     - конструктор: ініціалізує поля базового класу та власні;
     - метод, що перевизначає абстрактний;
     - метод, що перевизначає віртуальний;
     - перевизначення стандартного оператора «==» (наприклад, що порівнює об’єкти за значенням одного з полів);
     - деструктор: виводить повідомлення про видалення об’єкта;
   - **звичайний клас, що унаслідується від абстрактного:**
     - поле private int;
     - конструктор, що ініціалізує поля базового класу та власні;
     - метод, що перевизначає абстрактний;
     - метод, що перевизначає віртуальний.
   - **статичний клас**:
   - статичне поле private static int;.
   - статичний метод, що повертає значення статичного поля;
   - статичний метод, що збільшує на 1 значення статичного поля;
5. Побудувати UML-діаграму класу засобами онлайн-сервісу «Draw.io» з використанням мови PlantUML. Клієнтський код (клас Program та метод Main) на діаграмі не показувати.

6. У звіті надати UML-діаграму з описом класів та їх атрибутів, та текст програми на мові C#, оформлений належним чином. Код програми треба скопіювати з редактора Visual Studio зі збереженням відступів та кольору тексту. Продемонструвати працездатність програми скріншотами у звіті. Надати звіт з самостійної роботи викладачу.

Побудова UML-діаграми класу

На рис. 1 показана UML-діаграма ієрархії класі, що реалізує систему управління бібліотекою для опрацювання різних типів бібліотечних матеріалів (книги, журнали) та їх інвентаризації.

Абстрактний базовий клас LibraryItem містить загальні властивості для всіх матеріалів: назву (_title) та рік видання (_year). Цей клас також визначає абстрактний метод CalculateValue() для розрахунку вартості та віртуальний метод GetDescription() для отримання опису.

Клас Book успадковує LibraryItem і додає специфічні атрибути: автора (_author) та кількість сторінок (_pages). Він реалізує методи CalculateValue() та GetDescription(), а також включає оператори порівняння (== і !=). Клас позначений як sealed, що забороняє подальше успадкування.

Клас Magazine також успадковує LibraryItem і містить додатковий атрибут – номер випуску (_issueNumber). Він перевизначає методи CalculateValue() та GetDescription() для обліку особливостей журналів.

Для управління інвентарем використовується клас LibraryManager, який відповідає за підрахунок бібліотечних матеріалів. Він містить статичний лічильник _itemCount та статичні методи AddItem() (додавання нового матеріалу) і GetItemCount() (отримання загальної кількості).

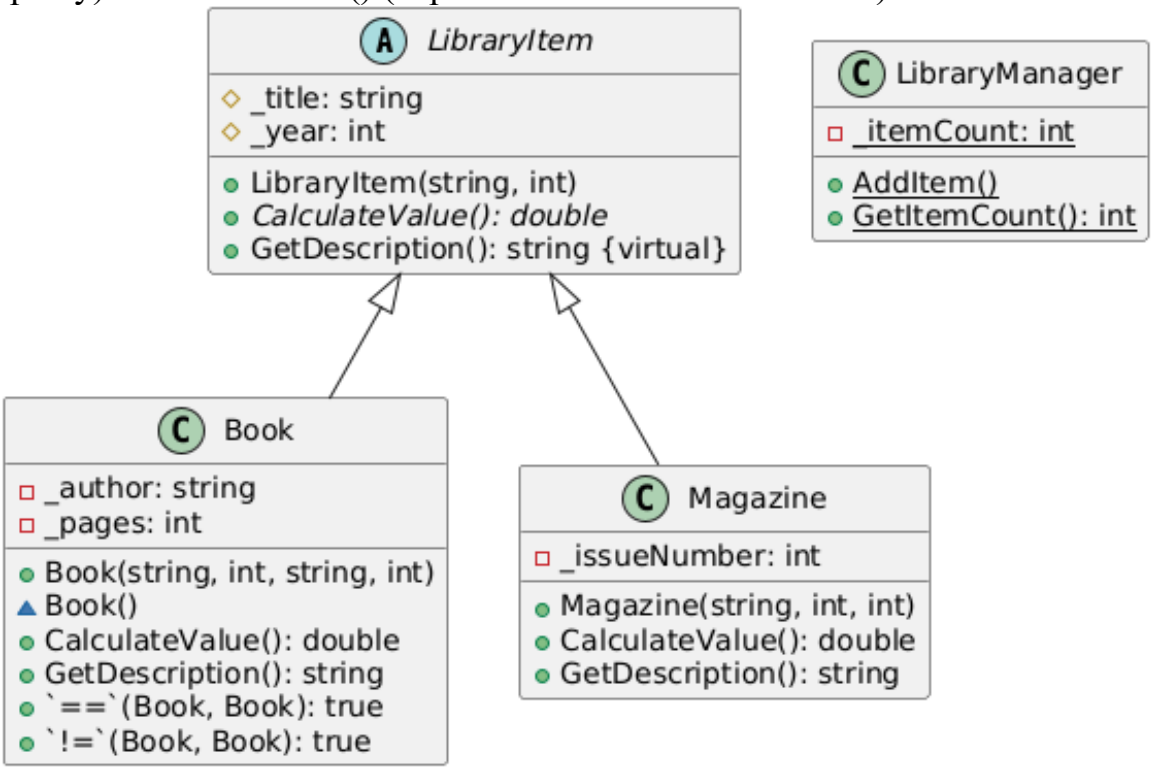

Рисунок 1 – UML-діаграма ієрархії класів

Приклад коду на мові C# з коментарями

```
using System;
using System.Text;

// Абстрактний клас, що представляє елемент бібліотеки (книга або журнал)
public abstract class LibraryItem
{
    // Назва елемента бібліотеки
    protected string _title;
    // Рік видання елемента
    protected int _year;

    // Конструктор для ініціалізації назви та року видання
    public LibraryItem(string title, int year)
    {
        _title = title;
        _year = year;
    }

    // Абстрактний метод для обчислення цінності елемента
    public abstract double CalculateValue();

    // Віртуальний метод для отримання опису елемента
    public virtual string GetDescription()
    {
        return $"Назва книжки: {_title}, рік видання: {_year}";
    }
}

// Клас, що представляє книгу, успадковує LibraryItem
public sealed class Book : LibraryItem
{
    // Автор книги
    private string _author;
    // Кількість сторінок у книзі
    private int _pages;

    // Конструктор для ініціалізації книги з назвою, роком, автором, сторінками
    public Book(string title, int year, string author, int pages) : base(title, year)
    {
        _author = author;
        _pages = pages;
    }

    // Обчислює цінність книги на основі кількості сторінок
    public override double CalculateValue()
    {
        return _pages * 10; // Кожна сторінка коштує 10 одиниць
    }

    // Повертає детальний опис книги
    public override string GetDescription()
    {
        return $"Книжка: {_title}, автор: {_author}, сторінок: {_pages}, " +
            $"рік видання: {_year}";

    }

    // Перевантаження оператора == для порівняння книг за назвою та автором
    public static bool operator ==(Book b1, Book b2)
    {
        if (ReferenceEquals(b1, null) || ReferenceEquals(b2, null))
            return false;
        return b1._title == b2._title && b1._author == b2._author;
    }
```

```
    // Перевантаження оператора != для порівняння книг
    public static bool operator !=(Book b1, Book b2)
    {
        return !(b1 == b2);
    }

    // Деструктор, що виводить повідомлення про видалення книги
    ~Book()
    {
        Console.WriteLine($"Книжка {_title} видалена.");
    }
}

// Клас, що представляє журнал, успадковує LibraryItem
public class Magazine : LibraryItem
{
    // Номер випуску журналу
    private int _issueNumber;

    // Конструктор для ініціалізації журналу з назвою, роком і номером випуску
    public Magazine(string title, int year, int issueNumber) : base(title, year)
    {
        _issueNumber = issueNumber;
    }

    // Обчислює цінність журналу залежно від року видання
    public override double CalculateValue()
    {
        return (_year >= 2020) ? 100.0 : 50.0; // Нові журнали (з 2020) цінніші
    }

    // Повертає детальний опис журналу
    public override string GetDescription()
    {
        return $"Журнал: {_title}, Випуск: {_issueNumber}, Рік випуску: {_year}";
    }
}

// Статичний клас для управління бібліотечними елементами
public static class LibraryManager
{
    // Лічильник загальної кількості елементів у бібліотеці
    private static int _itemCount = 0;

    // Додає елемент до бібліотеки, збільшуючи лічильник
    public static void AddItem()
    {
        _itemCount++;
    }

    // Повертає поточну кількість елементів у бібліотеці
    public static int GetItemCount()
    {
        return _itemCount;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // підтримка українських літер в консолі
        Console.OutputEncoding = Encoding.UTF8;

        // Створюємо приклади українських книг і журналів
```

```
        Book book1 = new Book("Енеїда", 1798, "Іван Котляревський", 150);
        Book book2 = new Book("Лісова пісня", 1911, "Леся Українка", 80);
        Magazine magazine1 = new Magazine("Локальна історія", 2023, 12);
        Magazine magazine2 = new Magazine("Український тиждень", 2018, 45);

        // Додаємо елементи до бібліотеки
        LibraryManager.AddItem();
        LibraryManager.AddItem();
        LibraryManager.AddItem();
        LibraryManager.AddItem();

        // Виводимо описи та цінність елементів
        Console.WriteLine(book1.GetDescription());
        Console.WriteLine($"Цінність: {book1.CalculateValue():F2}");
        Console.WriteLine();

        Console.WriteLine(book2.GetDescription());
        Console.WriteLine($"Цінність: {book2.CalculateValue():F2}");
        Console.WriteLine();

        Console.WriteLine(magazine1.GetDescription());
        Console.WriteLine($"Цінність: {magazine1.CalculateValue():F2}");
        Console.WriteLine();

        Console.WriteLine(magazine2.GetDescription());
        Console.WriteLine($"Цінність: {magazine2.CalculateValue():F2}");
        Console.WriteLine();

        // Перевіряємо кількість елементів у бібліотеці
        Console.WriteLine($"Загальна кількість елементів у бібліотеці: " +
            $"{LibraryManager.GetItemCount()}");


        // Перевіряємо оператори порівняння для книг
        Book book3 = new Book("Енеїда", 1798, "Іван Котляревський", 150);
        Console.WriteLine($"\nПеревірка порівняння книг:");
        Console.WriteLine($"book1 == book3: {book1 == book3}");
        Console.WriteLine($"book1 != book2: {book1 != book2}");
    }
}
```

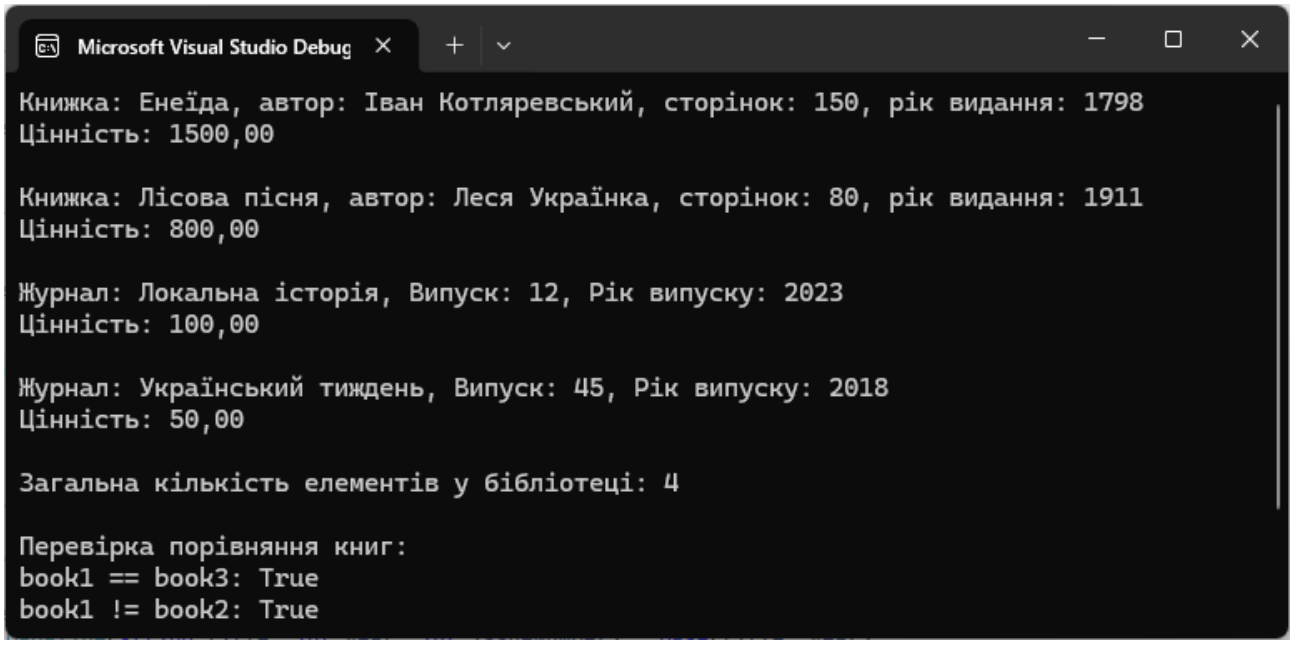

Рисунок 2 – Перевірка працездатності програми

Контрольні запитання

1. Яка роль конструкторів та деструкторів у класах бібліотеки?
2. Як реалізується успадкування між LibraryItem, Book та Magazine на UML-діаграмі?
3. Як поліморфізм відображається у методах CalculateValue та GetDescription?
4. Як позначити приховування елементів (private, protected) у PlantUML для класів Book та Magazine?
5. Як відобразити абстрактний клас LibraryItem у UML?
6. Які особливості запечатаного класу Book і як їх позначити на діаграмі?
7. Як відобразити статичний клас LibraryManager та його елементи в UML?
8. Як позначити перевизначення операторів == та != у класі Book?
9. Які інструменти використовуються для створення UML-діаграм із підтримкою зазначених концепцій?
10. Як пояснити зв’язок між базовим класом LibraryItem та похідними класами Book і Magazine у контексті поліморфізму?

Перелік індивідуальних варіантів опису предметної області

Написати програму – це більше, ніж правильно її оформити та змусити коректно виконуватись. Програми згодом неминуче модифікуються, повторно використовуються для побудови інших програм тощо. Тому велике значення має простота та зрозумілість їхньої внутрішньої структури. Інколи добре написану чужу програму набагато простіше зрозуміти, ніж власну, але написану погано.

Культура написання коду сприяє зменшенню помилок у програмах і полегшує їхню модифікацію та повторне використання. Під стилем програмування зазвичай розуміють набір прийомів і методів, що застосовуються з метою одержання правильних і ефективних програм, зручних для сприйняття, повторного застосування й модифікації.

Увесь інтерфейс та коментарі мають бути написані українською мовою.

Далі наведено загальноприйняті рекомендації з написання коду на C#:

– завжди вказуйте модифікатор доступу private, навіть якщо дозволено його опускати;

– не використовуйте публічних полів, замість цього використовуйте властивості;

– використовуйте максимальний рівень захисту для класів та їх елементів;

– використовуйте автоматичні властивості;

– уникайте файлів з більш ніж 500 рядками коду;

– уникайте методів з більш ніж 200 рядками коду;

– уникайте методів з більш ніж 5 аргументами, використовуйте структури для передачі великого числа параметрів;

– один рядок коду не може бути більше 120 символів;

– при оголошенні змінних кожну з них треба писати в окремому рядку;

– завжди ініціалізуйте змінні, навіть коли існує автоматична ініціалізація;

– не використовуйте так «магічних чисел», замість них оголошуйте константи і статичні змінні з коментарями;

– використовуйте порожній рядок для відокремлення логічних блоків класах та методах.

Далі у переліку наведено опис предметних областей, що підлягають моделюванню та програмної реалізації під час виконання здобувачами лабораторних та самостійних робіт. За вказівками викладача здобувач отримує одну предметну область для всіх робіт. Опис предметної області не є вичерпним чи абсолютно точним. Здобувач має можливість розширювати або звужувати предметну область, вносити зміни та доповнювати предметну область у відповідності до завдання роботи.

1. В аптеці продаються ліки різних виробників. Всі вони мають назву, групу ліків, до якої вони належать (антибіотики, протизапальні, шлункові тощо), ціну, термін зберігання. Однак, імпортні ліки повинні мати додатковий сертифікат про проходження препаратом лабораторного контролю в Україні.

Крім того, у випадку замовлення покупцем імпортного препарату, фармацевт (якщо звернеться до бази даних ліків в аптеці) повинен побачити не тільки інформацію про наявність ліків в аптеці та їх ціну, а й дані про вітчизняні аналоги препарату, які як правило, є дешевшими. Розробити структуру класів, які можна використовувати для комп'ютерної обробки даних про ліки (як вітчизняні, так й імпортні) в аптеці.

2. В офісі працюють люди різних професій, які займають відповідно різні посади: наприклад, директор, офіс-менеджер, системний адміністратор. Всі вони мають спільні характеристики: ПІБ, робоче місце (кімнату, стіл), заробітну плату, а також спільні функції: прийти на роботу/піти з роботи, отримати зарплатню, звільнитися тощо. Але крім того кожен робітник має свої власні професійні функції, а також інформацію, що його характеризує. Створити загальний базовий клас «Офісний працівник» і на його основі класи, що описують об'єкти «Директор», «Офіс-менеджер», «Системний адміністратор»; продемонструвати їх роботу.

3. Всі користувачі одного комп'ютера працюють з одним і тим самим з'єднанням з Інтернетом. Це з'єднання зберігається одна з характеристик кожного користувача. Забезпечити створення нових користувачів комп'ютера шляхом клонування базового екземпляра профілю користувача, при чому Інтернет-з'єднання повинне залишатися єдиним об'єктом, без копій.

4. Всі тварини сімейства котячих мають спільні риси: зріст, вагу, середовище проживання, вік, породу. Ті тварини, які мешкають у зоопарку, цирку, дома у людей, мають також ім'я. Кожна порода, окрім загальних дій, притаманних всім котячим (їсти, спати тощо), має свої власні дії (великі хижаки – рись, пума – полюють на здобич, домашні коти – граються із сонячним зайчиком тощо). Створити базовий клас «Тварина сімейства котячі», а також похідні від нього класи, які наслідують загальні риси та дії від нього, а також реалізують додаткові дії, притаманні конкретній тварині. Продемонструвати реалізацію структури класів.

5. Допоможіть Святому Миколаю сформувати подарунковий набір з іграшок, цукерок тощо та його розтиражувати. Всі складові набору повинні бути реалізовані як самостійні об'єкти та скопійовані до кожного наступного повністю.

6. Забезпечити друк квитків на концерти різними фірмами. Білети можуть бути просто вхідними (без місця), звичайними (з місцем) та вкладеними у спеціальне запрошення. Білети на театральні вистави друкують на державних підприємствах, а білети на гастролі та приватних театрів – в комерційних фірмах. Поліграфічні фірми відрізняються кількістю кольорів, які використовуються при друці білетів, та відповідно ціною на послуги. В залежності від типу білетів перенаправляти їх для друку на ту чи іншу фірму.

7. Імітувати процес випуску автомобілів відповідно до заданих прототипів. Кожне авто має модель, колір, тип кузова. Номер машині присвоюється тоді, коли екземпляр є готовим.

8. Магазин «Все для дому» реалізує господарські товари. Всі товари мають спільні характеристики: наприклад, назва, ціна, відділ, де вони

продаються, термін придатності чи гарантійний термін, а також оригінальні характеристики, які залежать від виду товару. Одні товари продаються тільки вроздріб, а інші можуть бути продані й оптом (при цьому ціна товару стає на 10% меншою). Створити базовий клас «Товар» та похідні класи, що міститимуть додаткову інформацію про товари, які можуть бути проданими як вроздріб, так і оптом.

9. На виставці собак всі учасники мають такі спільні характеристики як зріст, вага, вік, порода, ім’я хазяїна. Однак, деякі породи собак можуть мати додаткові обов’язкові характеристики: наприклад, куповані хвіст та вушка, додаткові щеплення тощо. Крім того, різняться також і функції, природні для собак: одні є мисливськими, інші – бійцівськими. Отже, крім загальних команд, вони мають вміти виконувати і ті, що притаманні лише їх породі. Створити базовий клас «Собака» та декілька класів, що характеризують певні породи. Продемонструвати програмну реалізацію даних класів.

10. Організувати випуск пакетів соку з різних фруктів та ягід декількома заводами. Заводи відрізняються сортами сировини, упаковкою та ціною продукції. В залежності від пори року клієнти замовляють сік у різних заводів.

11. Організувати виробництво взуття декількома виробниками. Види взуття – жіночі туфлі, чоловічі туфлі та кросівки. Перший виробник працює тільки з натуральною шкірою, другий – із шкірозамінником, третій використовую шкіру та хутро для оздоблення виробів. Забезпечити поставку до магазину продукції всіх трьох виробників.

12. Організувати виробництво грамот для лауреатів міжшкільного конкурсу, дипломів для дипломантів та грамот для всіх інших учасників. Список учасників конкурсу містить такі дані: школа та клас учня, а також місце яке він посів (або примітка про присудження диплому конкурсу). Всі створені грамоти повинні бути незалежними об’єктами. Поліграфічні фірми, які займаються виготовленням продукції такого типу, відрізняються кількістю кольорів, що використовуються при друці, якістю паперу та відповідно ціною на послуги. Забезпечити у програмі вибір фірми з виготовлення грамот у залежності від наявної суми грошей.

13. Організувати виробництво пластикових читацьких квитків до бібліотеки. Читацькі квитки можуть бути студентські, шкільні, наукові, особливі (для академіків тощо). Різні апарати можуть виготовляти пластикові картки з фото або без фото, з голограмою або без. В залежності від складності виробництва змінюється і вартість продукції. Забезпечити у програмі вибір апарата з виготовлення квитків у залежності від наявної суми грошей.

14. Організувати процес формування порцій у столовій. Кожна порція повинна містити м’ясне блюдо та гарнір. Крім того, порція може бути двох типів: дієтична або звичайна.

15. Побудувати генератор комплектів пам’яток для студентів, які можуть роздаватися у приймальній комісії. Наприклад, для студентів-першокурсників до комплекту включають карту університету, список необхідних телефонів служб університету та факультету, інформацію щодо програми навчання до ступеня «бакалавра», основні правила навчання в вузі (за що відраховують, за

що видають червоний диплом тощо). Для студентів-магістрів до комплекту включають програму навчання до ступеня «магістр», загальні поради щодо написання наукових статей, магістерської дисертації тощо.

16. Програмний текстовий чат підтримує виведення тексту повідомлення у віконці діалогу з іншим користувачем у звичайному вигляді (шрифт 14 кегль, Times New Roman, чорний). На основі базового повідомлення реалізувати виведення тексту заданим кольором, типом шрифта та розміром. Також на початку повідомлення додати час його надходження.

17. Реалізувати виведення інформації про користувачів, зареєстрованих у програмі, у різних форматах в залежності від типа користувача. Наприклад, повна інформація (звичайний користувач), обмежена інформація (адміністратор системи), мінімум інформації (vip- користувач).

18. Реалізувати виведення системних повідомлень програми (попередження про помилки, інформаційні вікна, діалогові вікна тощо) мовою, зрозумілою користувачу, який працює з даною системою. Мова є параметром профілю користувача та встановлюється для кожного користувача окремо. Нехай українська мова буде мовою, що встановлюється за замовчанням.

19. Реалізувати віртуальний автомат, який виробляє (або просто розливає) напої, наприклад: чай, каву, сік. Напої характеризуються не тільки типом, а й ціновою категорією. Наприклад, чай може бути заварним або в пакетиках; кава розчиненою або меленою; сік 100% або нектар. В залежності від обраної цінової політики налаштувати автомат на випуск напоїв певної якості.

20. Реалізувати віртуальний автомат, який виробляє напої. В залежності від того, які параметри задані покупцем (обрано чай/каву/какао, вказано «додатковий цукор», «лимон» і інші) згенерувати напій. Забезпечити виробництво як мінімум трьох різних напоїв.

21. Реалізувати віртуальні меблеві фабрики. Кожна фабрика виготовляє вітальні, м’які меблі, кухні. Характеризуються фабрики матеріалами, які використовуються для виготовлення меблів, та вартістю продукції. В залежності від побажань та фінансових можливостей замовника обирати ту фабрику, яка буде створювати для нього меблі в даний момент часу.

22. Реалізувати друк перепусток різних видів (постійних, тимчасових, одноразових тощо), а також ордерів на внесення-винесення техніки на певну територію в різних поліграфічних фірмах. Фірми відрізняються між собою якістю паперу, на якому здійснюється друк (звичайний або щільний), ступенем захисту (наявністю водяних знаків), вартістю партії виробів. В залежності від побажань клієнта перенаправляти замовлення на ту чи іншу фірму.

23. Реалізувати друк перепусток різних видів (постійних, тимчасових, одноразових тощо), а також ордерів на внесення-винесення техніки на території компанії. В залежності від замовника (одні фірми завжди замовляють перепустки, а інші – бланки ордерів) здійснювати друк тих чи інших документів.

24. Реалізувати модель бібліотеки. Бібліотека містить відділи, відділи – стелажі з різної тематики (наприклад, література 19 ст., детективи тощо), стелажі складаються з полиць, які в свою чергу містять книжки. Кожна книга

має свій унікальний код та шифр. Книг з однаковим кодом не може бути. А екземпляри однієї книжки можуть мати однаковий шифр. Забезпечити виведення на екран інформації про об’єкти будь-якого типу (книга, полиця тощо). Крім того, реалізувати виведення кількості найменувань книг на полиці, стелажі, відділі, а також загальну кількість книг на полиці, стелажі, у відділі.

25. Реалізувати модель магазину з медіа-продукцією. Магазин має відділи (наприклад: ігри, кіно, музика тощо), відділи складаються зі стелажів (наприклад: Ігри.Стратегії, Ігри.Квести, Кіно.Детективи тощо), на яких розміщено саму продукцію. Кожний товар має декілька копій (екземплярів). Забезпечити виведення на екран інформації про товари (причому виводити дані тільки про сам товар, а не про кількість його екземплярів). Крім того, реалізувати виведення даних про кількість товару на стелажі, у відділі, та в магазині в цілому.

26. Реалізувати модель мікрорайону: мікрорайон повинен складатися з вулиць, вулиці містять будинки, будинки складаються з під’їздів та квартир. Забезпечити виведення на екран інформації про кожний тип об’єктів (кількість населення вулиці, будинку тощо). Також забезпечити можливість додавання та видалення об’єктів всіх типів. Крім того, реалізувати метод «Переписати населення», який передбачає підсумовування населення мікрорайону шляхом обчислення населення вкладених в нього об’єктів.

27. Реалізувати обробку даних про покупки в інтернаціональному магазині. Організувати при виконанні команди «Купити товар» видачу покупцеві чеку про оплату товару у форматі, притаманному регіону проживання покупця (формат чека впливає на форму відображення часу та дати покупки, валюти, найменування товару тощо).

28. Реалізувати програму, яка демонструє складання тесту об’єктами класу «віртуальний студент». Відомо, що такий студент може бути декількох типів: відмінник, такий, що навчається добре, задовільно та незадовільно. Умовно встановимо, що відмінник завжди відповідає на всі поставлені запитання; студент, що навчається добре відповідає на 80% запитань; задовільно – на 60% запитань та незадовільно – на 40% запитань.

29. Реалізувати процес випуску книжок за заготовленими раніше матрицями. Кожна книга характеризується іменем автора, видавництвом, де вона була надрукована, серією та номером видання. Масив сторінок є тим об’єктом, який необхідно розтиражувати для кожної книги.

30. Реалізувати процес випуску меблів за прототипами. Наприклад, типами меблів можуть бути «кухня», «вітальня», «спальня» тощо. Кожний гарнітур характеризується типом, матеріалом та складається з модулів (шухляди, полиці і т.д.). Забезпечити реалізацію складових частин меблів як окремих об’єктів, а також організувати «глибоке копіювання» об’єктів при створенні нового екземпляра гарнітура.

31. Реалізувати процес віртуального збирання комп’ютерів та телевізорів у різних країнах. Кожна збірка відрізняється деталями, які вбудовуються до прибору, комплектацією (є базова, є vip-моделі тощо) та відповідно вартістю. В залежності від ціни послуги забезпечити запуск виготовлення продукції у тій чи

іншій країні.

32. Реалізувати структуру класів для підтримки гри «Морський бій», яка дозволяє розміщувати на гральному полі одно-, дво- та трипалубні кораблі. Також в залежності від типу корабля забезпечити визначення його стану (цілий, поранений, вбитий).

33. Розробити базові класи StudentT (група, прізвище, оцінка) та Teacher (прізвище та ініціали викладача, посада викладача). Визначити конструктори, деструктор та методи встановлення і виведення значень полів даних. Використовуючи множинне успадкування, визначити похідний клас Class з додатковими полями даних: дисципліна, вид заняття (лекція, практичне, лабораторне), дата, час, місце проведення. Визначити конструктори, деструктор, методи встановлення та визначення значень полів даних. Розробити клас Schedule занять, який містить масив об’єктів класу Class. Вивести розклад занять на екран.

34. Розробити клас Date (рік, місяць, день). Визначити конструктори ініціалізації, копіювання, деструктори та методи для зміни і читання значень полів цього класу. Створити похідний клас Book (прізвище автора, назва, видавництво, рік видання, кількість сторінок). Визначити необхідні дані, методи, конструктори та деструктори, методи або операторні функції введення-виведення. Розробити клас Library, що містить масив об’єктів класу Book. Виконати пошук книг за прізвищем автора, назвою, видавництвом, роком видання.

35. Розробити клас Goods, який містить назву товару, дату виготовлення, свідоцтво якості, вартість. Визначити конструктори, деструктор та методи встановлення і виведення значень полів даних. Визначити похідний клас GoodsInShop з додатковими полями даних: націнка, термін придатності. Визначити конструктори, деструктор, методи роботи з полями даних. Розробити клас Shop, що містить масив об’єктів класу GoodsInShop. Визначити прибуток магазину від продажу товарів.

36. Розробити клас Goods, який містить назву товару, кількість одиниць товару, інструкцію застосування, вартість. Визначити конструктори, деструктор та методи встановлення і читання значень полів даних. Визначити похідний клас Medicine з додатковими даними: концентрація, термін придатності. Визначити операторні методи введення, виведення, корегування полів даних класу. Розробити клас Pharmacy, який містить масив об’єктів класу Medicine. Визначити вартість ліків за рецептом.

37. Розробити клас Organisation, який містить дані про назву організації, телефон, адресу та ін. Визначити конструктори ініціалізації, копіювання, деструктори та методи для зміни і читання значень полів цього класу. Створити похідний клас Department з додатковими полями: спеціальність, кількість бакалаврів, спеціалістів і магістрів. Визначити необхідні дані, методи, конструктори та деструктори, методи або операторні функції введення-виведення. Розробити клас Faculty, у закритій частині якого розміщено масив об’єктів з даними про кафедри. Визначити загальну кількість студентів.

38. Розробити клас Produce, який містить назву продукту та ціну одиниці

продукту. Визначити конструктори, деструктор та методи встановлення і читання значень полів даних. Визначити похідний клас Ingredient з додатковими даними про дозування продукту. Визначити операторні методи введення, виведення, корегування полів даних класу. Розробити клас Recipe кулінарного виробу, який містить масив об'єктів класу Ingredient. Визначити вартість виготовлення кулінарного виробу.

39. Розробити класи Date (день, місяць, рік) та Time (година, хвилина, секунда). Визначити конструктори ініціалізації, копіювання, деструктори та методи для зміни і читання значень полів розроблених класів. Перевіряти коректність значень полів даних. Створити похідний від Date і Time клас File, який містить дані про файл: назва, розмір, дата та час створення, атрибути. Визначити необхідні дані, методи, конструктори та деструктори, методи або операторні функції введення-виведення. Розробити клас Dir, що містить масив об'єктів класу File. Визначити необхідні конструктори, деструктори, методи роботи з файлами та каталогами. Вивести дані про файли, що відповідають заданій масці пошуку.

40. Розробити класи Date (рік, місяць, день) і Time(година, хвилина, секунда). Визначити конструктори ініціалізації, копіювання, деструктори та методи для зміни і читання значень полів цього класу. На основі множинного успадкування класів створити похідний клас TVShow (дата, час трансляції, вид та назва передачі). Визначити необхідні дані, методи, конструктори та деструктори, операторні функції введення- виведення. Розробити клас Schedule, що містить масив об'єктів класу TVShow. Виконати пошук передачі за її назвою.

41. Розробити код, який буде створювати складене меню в ресторані швидкого харчування. До складу однієї порції повинні входити м'ясне блюдо, гарнір, напій, десерт. Створити меню для мереж ресторанів швидкого харчування МакДональдс, Швидко та Пузата хата.

42. Розробити механізм клонування групи студентів, яка складається з окремих об'єктів класу «Студент». Дане клонування необхідне для обробки масиву даних про студентів різними службами університету: поліклінікою, деканатом, студентською радою.

43. Створити будівельника будинку, який працює в такі етапи: закладення фундаменту, побудова стін та перекриттів, встановлення даху, внутрішні роботи. Будинки можуть бути панельними, цегляними, монолітно-каркасними. Панельні та цегляні будинки не перевищують 9 поверхів. Монолітно-каркасні можуть бути вище. В панельних та цегляних будинках встановлюються дерев'яні вікна, а в монолітних – пластикові.

44. Створити декілька видів клієнтських карток у мережі ресторанів. Карткою першого виду передбачається надання 5% знижки на вартість замовлення. Картка другого виду надає можливість отримати 10% знижки на вартість замовлення, а також 2 безалкогольні напої безкоштовно (у разі замовлення не менше, ніж на суму 100 грн.). Картка третього виду передбачає можливість отримання 20% знижки на замовлення та пляшки вина у подарунок від ресторану (у разі замовлення, незалежно від суми замовлення).

Продемонструвати, як працює метод «Обчислити суму замовлення», за допомогою створення відповідних абстракції та її реалізації. Передбачити випадок, коли клієнт ресторану не має картки взагалі.

45. Створити клас Person (прізвище, ім’я, по батькові, дата народження). Визначити конструктори ініціалізації, копіювання, деструктори та методи для зміни і читання значень полів даного класу. Створити похідний клас Subscriber з додатковим полем – номер телефону. Визначити необхідні дані, методи, конструктори та деструктори, методи або операторні функції введення-виведення. Розробити клас PhoneBook, у закритій частині якого розміщено масив об’єктів з даними про абонентів. Виконати пошук номера телефону за прізвищем абонента.

46. Створити клас Person (прізвище, ім’я, по батькові, дата народження, стать та ін.). Визначити конструктори ініціалізації, копіювання, деструктори та методи для зміни і читання значень полів даного класу. Створити похідний клас Emloyee з додатковими полями: табельний номер, оклад, стаж роботи, кількість відпрацьованих годин, зарплата за годину роботи. Визначити необхідні дані, методи, конструктори та деструктори, методи або операторні функції введення-виведення. Розробити клас Payment, у закритій частині якого розміщено список об’єктів з даними про співробітників, для яких розраховується зарплата. Обчислити зарплату співробітників.

47. Створити структуру класів, до якої будуть входити класи, які відповідають таким об’єктам як Лікар, Відділення та Поліклініка. Кожний такий об’єкт має своє ім’я. Поліклініка та відділення характеризуються кількістю лікарів та хворих, а також мають власних завідувачів. У програмі передбачити реалізацію спеціального методу – «Пройти медичний огляд» у одного, декількох або всіх лікарів.

48. У банку вакансій кожна позиція характеризується назвою, компанією, що її пропонує, категорією вакансії та регіоном. Забезпечити виведення на екран інформації про вакансії будь-якого типу групування. Також реалізувати виведення тільки кількості вакансій з різними типами групування (за компанією, за категорією, за регіоном тощо).

49. У вищому навчальному закладі студент є частиною навчальної групи. Група входить до складу потоку на кафедрі та курсу на факультеті в цілому. І студенти, і група, і факультет мають ім’я або назву. Також студент характеризується масивом оцінок, які він отримує протягом сесії. Потік студентів складається з масиву груп, курс – з масиву потоків, факультет – з масиву курсів. Організувати виведення оцінок студентів, згрупувавши їх по групам, потокам, курсам. Також реалізувати метод обчислення середнього балу для студента, групи тощо.

50. У місті існує три Інтернет провайдери. Перший забезпечує з’єднання із всесвітньою мережею за допомогою телефонної лінії, другий – телевізійного кабелю, третій підтримує безпровідний зв’язок. При цьому кожний з провайдерів пропонує користувачам пакети, що відрізняються по швидкості передачі даних, об’єму трафіку та відповідно вартості. Деякий сервісний центр з підключення населення міста до Інтернету заключив договори з усіма

вищезазначеними провайдерами. Реалізувати механізм «підключення до інтернету» з підтримкою можливості вибору за певними критеріями провайдера, послугами якого хоче скористуватися клієнт.

51. У різні пори року постачальник продуктів до ресторану замовляє їх у різних регіонах. Влітку та восени він отримує продукти з України; взимку та навесні – з Туреччини, Африки тощо. Реалізувати механізм постачання продуктів та продемонструвати, що в залежності від пори року місце поставки змінюється.

52. У різні пори року ресторан замовляє продукти у різних постачальників. Кожний постачальник працює з виробниками з різних регіонів. Одні постачальники орієнтовані на співпрацю з вітчизняними виробниками продуктів, інші – із зарубіжними виробниками. Виробники продуктів відрізняються сортами та ґатунком продукції.

53. У штаті вищого навчального закладу є багато різних типів працівників, зокрема науково-педагогічні працівники, адміністративні працівники та методисти. Всі вони мають спільні характеристики: наприклад, ПІБ, корпус та кімната, де знаходиться робоче місце, факультет тощо. Але крім того всі вони мають і свої власні обов’язки: наприклад, читання лекцій, проведення іспитів, складання розкладу занять, моніторинг успішності навчального процесу тощо. Розробити базовий клас «Працівник вузу», а також похідні від нього класи, що реалізують характерні функції працівників та містять додаткову інформацію про них.

54. Учні класу пішли до бібліотеки за підручниками. Кожний підручник має свій унікальний бібліотечний шифр. Сформувати прототип набору підручників; створити набори підручників для учнів. Забезпечити виведення на екран списку учнів, а також списку підручників, які дісталися кожному учню (указати шифр та назву кожної книги, яка видана учню).

55. Шкільна бібліотека. Створити формувач наборів підручників для різних класів. Підручники діляться на групи: математичні (математика – молодші класи; алгебра і геометрія – середні класи; тригонометрія, стереометрія – старші), природничі (природознавство – молодші, географія – середні, фізика-хімія – старші), лінгвістичні (різні види літератури та мов). Для кожного класу викликається свій комплектатор, який формує набори підручників кожного типу у комплект.

Рекомендований перелік джерел

1. Bell D. The UML 2 class diagram. An introduction to structure diagrams in UML 2. IBM Developer. Dostęp, 2023, 27: 2024.
2. Chykunov P. Services for creating UML class diagrams // P. Chykunov, I. Chykunov/ *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні*: Матеріали VI Всеукраїнської науково-практичної інтернет-конференції (м. Харків, 06-07 грудня 2023 р.). – Харків: ННІПІ УІПА, 2023. – С. 42-43.
3. Yuliia Prokop, Natalia Nestoruk, Pavlo Chykunov, Olena Trofymenko. Impact of a problem-based approach on IT student learning. *2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT),* Lviv, Ukraine, 2024, pp. 1-4, doi: 10.1109/CSIT65290.2024.10982647.
4. Fowler M. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional, 2018.
5. Hendrawan E., Meisel M., Sari D. Analysis and implementation of computer network systems using software Draw.io. *Asia Information System Journal*, 2023.
6. Microsoft Learn. Object-Oriented programming (C#). URL: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop
7. Microsoft Learn. Overview of classes, structs, and records in C#. URL: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/ URL: https://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058602.pdf
8. UML-concept. URL: https://wiki.ashish.me/software-development/uml/uml-class.html
9. Гудзовата О. О. Використання уніфікованої мови візуального моделювання UML (Unified Modeling Language) як інструменту підтримки проектування інформаційних систем / О. О. Гудзовата, В. І. Костирко, І. В. Артищук / *Підприємництво і торгівля*. – 2019. – Вип. 24. – С. 108-114. URL: https://plantuml.com/class-diagram
10. Діаграма класів. URL: https://uk.m.wikipedia.org/wiki/Діаграма_класів
11. Кіпоренко І. Є., Ліщинська Л. Б. Автоматизована побудова uml-діаграм у процесі розробки програмного забезпечення: методи, інструменти та перспективи // Матеріали LIV Всеукраїнської науково-технічної конференції підрозділів ВНТУ, Вінниця, 24-27 березня 2025 р. 2025.
12. Коноваленко І. В. Платформа .NET та мова програмування C# 8.0 : навч. посіб. / І. В. Коноваленко, П. О. Марущак. – Тернопіль : В. А. Паляниця, 2020. – 320 с.
13. Куліков В.М., Рябцев В.В., Паршуков С.С. Об’єктно-орієнтоване програмування для фахівців з кібербезпеки: навч. посіб. Київ: КПІ ім. Ігоря Сікорського, 2023. 365 с.
14. Ланевич Т. Принципи SOLID у розробці програмного забезпечення. *Природничі та гуманітарні науки. Актуальні питання*: матер. VII міжнар. студ. наук.-техн. конф. 2024. – С. 89-90.
15. Муляр В. П. Об’єктно-орієнтоване програмування: лабораторний практикум. Луцьк: Вежа-Друк, 2022. 112 с. URL: https://evnuir.vnu.edu.ua/handle/123456789/21291
16. Настарчук Д.В. Професійна підготовка фахівців з цифрових технологій для викладання освітнього модулю «Реалізація поліморфізму мовою C#» у закладах вищої освіти: кваліфікаційна робота магістра / Д. В. Настарчук;

наук. кер. П. О. Чикунов ; Харків. нац. ун-т ім. В. Н. Каразіна. – Харків, 2024. – 80 с.
17. Об’єктно-орієнтоване програмування: електрон. метод. посіб. / уклад.: А. Л. Рачинська, О. А. Недєва, К. С. Палій. Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2024. 338 с.
18. Омельчук Л. Л. Об’єктно-орієнтоване програмування. Лабораторний практикум: навчальний посібник. Київ, 2021.
19. Онлайн-курс Coursera. Object Oriented Development using C#. URL: https://www.coursera.org/learn/oo-development-using-c-sharp
20. Онлайн-курс Udemy. C# Basics for Beginners: Learn C# Fundamentals by Coding. URL: https://www.udemy.com/course/csharp-tutorial-for-beginners/
21. Онлайн-курс Udemy. C# Intermediate: Classes, Interfaces and OOP. URL: https://www.udemy.com/course/csharp-intermediate-classes-interfaces-and-oop/
22. Онлайн-курс Udemy. UML and Object-Oriented Design Foundations. URL: https://www.udemy.com/course/uml-and-object-oriented-design-foundations/?couponCode=LETSLEARNNOW
23. Основи об’єктно-орієнтованого програмування: навч. посіб. / Гришанович Т. О., Глинчук Л. Я. Луцьк : ВНУ імені Лесі Українки, 2022. 120 с. URL: https://evnuir.vnu.edu.ua/handle/123456789/20320
24. Основи програмної інженерії: навч.-метод. посібник / О.Г. Трофименко, С.Ю. Манаков, Д.Г. Ларін. Одеса: Фенікс, 2022. 197 с. ISBN 978-966-928- 808-0.
25. Прокоп Ю. В., Трофименко О. Г., Толокнов А. А., Дубовой Я. В. Комплексний аналіз підходів до викладання курсів CS1 і CS2 в університетах світу та України. Вісник Черкаського державного технологічного університету. Технічні науки. 2021. № 2. С. 18-30. DOI: 10.24025/2306-4412.1.2021.229502.
26. Чикунов П.О. Модернізація лабораторного практикуму з дисципліни «Об’єктно-орієнтоване програмування» / П.О. Чикунов, І.П. Чикунов // *Актуальні тенденції розвитку освіти, науки та технологій* : Матеріали VII Міжн. наук.-практ. конф. (м. Бахмут, м. Харків, 15 травня 2024 р.). : у 3-х ч. / За заг. ред. Г. Г. Михальченко. Бахмут – Харків: ННПІ УІПА, 2024. Ч 2. – С. 97-99.
27. Чикунов П.О. Постановка лабораторного практикуму «Реалізація поліморфізму мовою C#» / П.О. Чикунов, Д.В. Настарчук // *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні*: Матеріали VII Всеукраїнської науково-практичної інтернет-конференції (м. Харків, 05-06 грудня 2024 р.). [Електронний ресурс]. – Харків : БННППІ ХНУ ім. В. Н. Каразіна, 2024. – С. 41-43.
28. Чикунов П.О. Розробка лабораторного практикуму «Об’єктно-орієнтоване програмування» / П.О. Чикунов, В.С. Лизунов // *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні*: Матеріали VI Всеукраїнської науково-практичної інтернет-конференції (м. Харків, 06-07 грудня 2023 р.). – Харків: ННППІ УІПА, 2023. – С. 38-39.
29. Щербаков О. В., Парфьонов Ю. Е., Федорченко В. М. Основи об'єктно-орієнтованого програмування: навч. посіб. Харків: ХНЕУ ім. С. Кузнеця, 2019.

Навчально-методичне видання

Павло Олександрович Чикунов
Артем Сергійович Соловйов

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Навчально-методичний посібник
для здобувачів першого (бакалаврського) рівня вищої освіти галузей знань F «Інформаційні технології», A «Освіта» та G «Електроніка, автоматизація та електронні комунікації»

Електронне видання

В авторській редакції

Ум-друк. арк. 5,25.