Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційні систем та технологій

Лабораторна робота № 5

із дисципліни «Технології розробки програмного забезпечення»

Тема: «Вступ до паттернів проектування.»

Виконав
Студенти групи ІА-31:
Корнійчук М.Р

Перевірив:
Мягкий М.Ю

Київ 2025

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи.

Варінт :

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

Посилання на гіт: https://github.com/MkEger/trpz-labs/tree/Lab_5

Завдання:

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону

Хід роботи

Тема проекту: Текстовий редактор

1. Реалізувати не менше 3-х класів відповідно до обраної теми

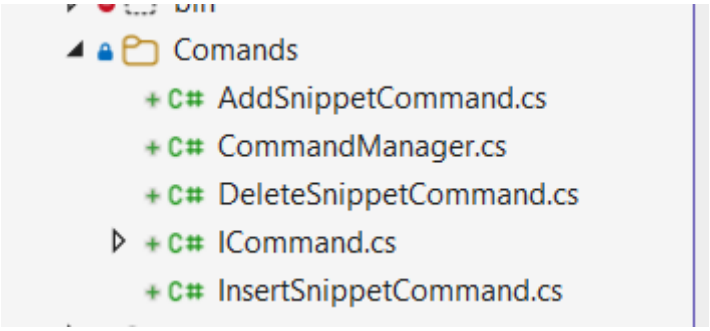


Рис. 1 - Структура проекту

У процесі виконання лабораторної роботи було створено п'ять основних класів, які забезпечують реалізацію функціоналу текстового редактора з використанням шаблону Command (рис. 1). Наведемо детальний опис кожного класу:

1. ICommand (інтерфейс)

Центральний інтерфейс шаблону Command, який визначає контракт для всіх команд у системі редактора. Встановлює єдиний стандарт для всіх операцій, що можуть бути виконані та скасовані.

Властивості та методи:

- Name (string) - назва команди для відображення в користувацькому інтерфейсі, логуванні та історії операцій
- Execute() - основний метод для виконання операції команди над текстовим вмістом або структурами даних
- Undo() - метод для скасування виконаної операції, що забезпечує повернення до попереднього стану
- CanExecute() - перевірка можливості виконання команди в поточному контексті системи

Інтерфейс є основою архітектури Command Pattern та забезпечує абстракцію, що дозволяє CommandManager працювати з різними типами команд уніфіковано.

2. InsertSnippetCommand

Конкретна реалізація команди для вставки код-сніпетів у текстовий редактор. Інкапсулює повну логіку вставки фрагментів коду з підтримкою відміни операції.

Приватні поля:

- _snippetService - посилання на сервіс для роботи зі сніпетами
- _snippet - об'єкт сніпета, який буде вставлений
- _insertPosition - позиція в тексті для вставки сніпета

- _textBox - посилання на текстовий компонент
- _previousText - збережений текст для операції Undo
- _previousSelectionStart та _previousSelectionLength - збережений стан курсора

Функціонал:

- Валідація можливості вставки (перевірка позиції, наявності сніпета, коректності RichTextBox)
- Збереження повного стану редактора перед виконанням операції
- Делегування вставки сніпета до SnippetService з розширенням змінних
- Повне відновлення попереднього стану при Undo операції

3. AddSnippetCommand

Команда для додавання нових код-сніпетів до колекції редактора. Забезпечує динамічне розширення набору доступних шаблонів коду.

Приватні поля:

- _snippetService - сервіс для управління колекцією сніпетів
- _snippet - новий сніпет для додавання
- _wasExecuted - прапорець успішного виконання команди

Основні можливості:

- Валідація структури сніпета (перевірка наявності назви та коректності даних)
- Додавання сніпета через SnippetService з автоматичним встановленням дати створення
- Відстеження стану виконання для коректної роботи Undo
- Видалення доданого сніпета при скасуванні операції
- Обробка дублікатів назв сніпетів (заміна існуючих)

4. CommandManager (Invoker)

Центральний клас-інвокер, що реалізує ключову логіку шаблону Command. Управляє виконанням команд та підтримує повну історію операцій з багаторівневим Undo/Redo.

Приватні поля:

- _executedCommands (Stack<ICommand>) - стек виконаних команд для Undo операцій
- _undoneCommands (Stack<ICommand>) - стек скасованих команд для Redo операцій
- _maxHistorySize - максимальний розмір історії для оптимізації пам'яті

Ключові методи:

- ExecuteCommand() - виконання команди з валідацією, додаванням до історії та очищенням Redo стеку
- Undo() - скасування останньої команди з переміщенням до Redo стеку
- Redo() - повторення скасованої команди з поверненням до основного стеку
- CanUndo() / CanRedo() - перевірка можливості навігації по історії
- ClearHistory() - очищення всієї історії команд
- GetExecutedCommandsHistory() - отримання списку назв виконаних команд

Включає систему обробки винятків з відновленням коректного стану при помилках.

5. SnippetService (Receiver)

Існуючий сервісний клас, який виступає в ролі отримувача (Receiver) у шаблоні Command. Містить бізнес-логіку для роботи зі сніпетами та їх інтеграції з графічними компонентами.

Основні методи:

- InsertSnippet() - вставка сніпета в RichTextBox з розширенням змінних ($DATE, $TIME, $USER)
- AddSnippet() - додавання нового сніпета до колекції з валідацією та обробкою дублікатів
- DeleteSnippet() - видалення сніпета з колекції за назвою
- GetSnippetsForLanguage() - отримання сніпетів для конкретної мови програмування
- FindSnippetByTrigger() - пошук сніпета за тригерним словом

2. Реалізувати один з розглянутих шаблонів за обраною темою

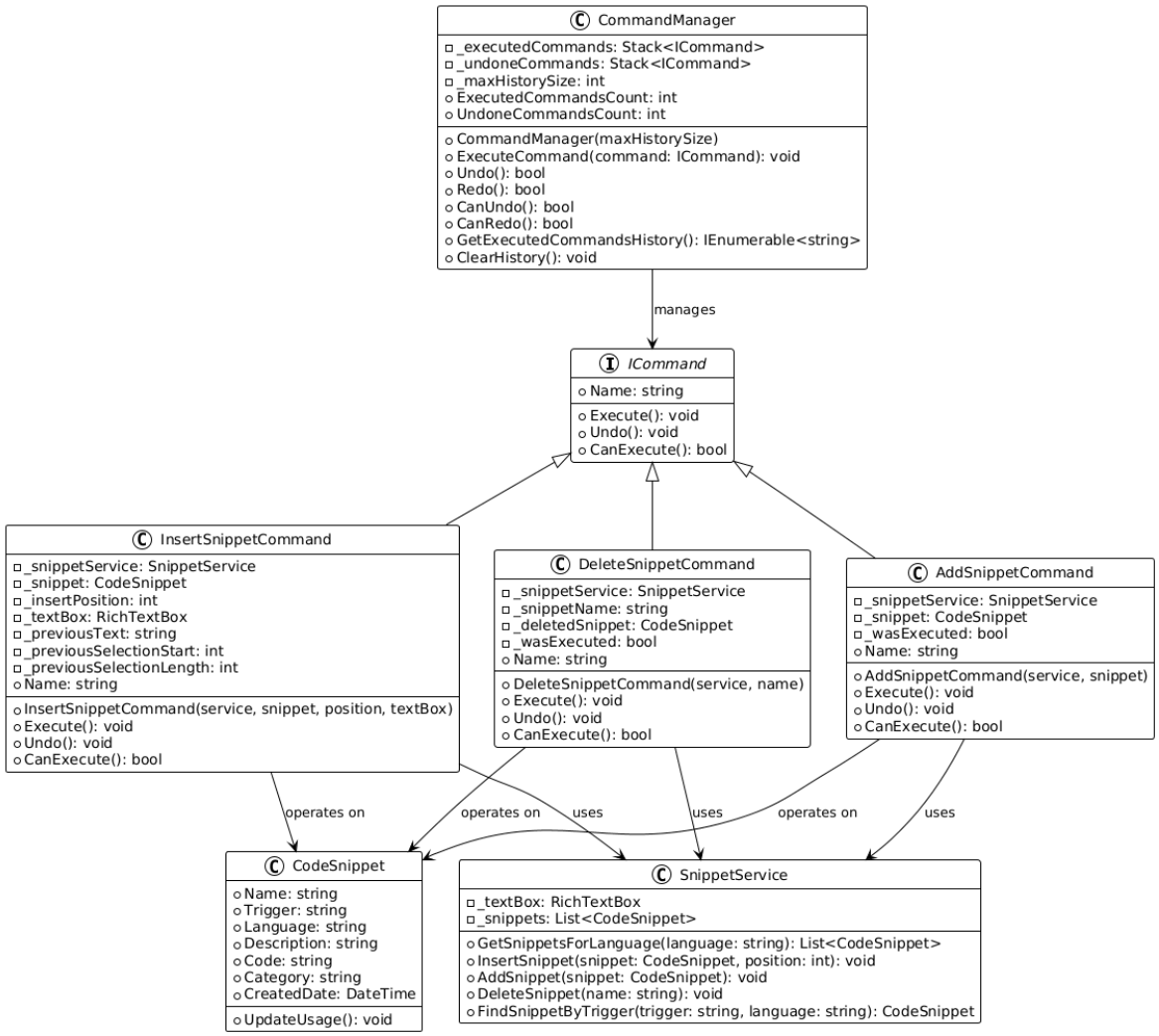


Рис. 2 - Діаграма класів

У рамках проєкту текстового редактора патерн Command було реалізовано для забезпечення гнучкого та розширюваного механізму управління операціями редагування з підтримкою їх скасування. Його використання дозволило створити уніфікований спосіб виконання різних команд редактора з можливістю повернення до попередніх станів системи.

Реалізація:

Основна ідея полягала у створенні команд, які інкапсулюють операції роботи з код-сніпетами: вставку фрагментів коду, додавання нових шаблонів та управління колекцією сніпетів.

• Інвокер (CommandManager) виступає координатором, який управляє виконанням команд та підтримує повну історію операцій. Він забезпечує послідовне виконання команд та надає можливості багаторівневого Undo/Redo.

• Команди інкапсулюють логіку конкретних операцій редагування, приховуючи деталі реалізації від основної програми та забезпечуючи можливість їх скасування.

• Receiver (SnippetService) містить бізнес-логіку операцій зі сніпетами,

виступаючи як виконавець фактичних дій над текстовим вмістом.

Проблеми, які вирішує патерн

• Інкапсуляція операцій редагування. Завдяки командам розробники можуть виконувати складні операції зі сніпетами через єдиний інтерфейс, не турбуючись про деталі їх внутрішньої реалізації та взаємодії з текстовими компонентами.

• Уніфікованість виконання команд. Незалежно від типу операції (вставка сніпета, додавання нового шаблону, видалення), методи доступу (Execute(), Undo()) залишаються однаковими, що спрощує взаємодію з системою управління командами.

• Повна підтримка Undo/Redo функціональності. Кожна команда зберігає стан системи перед виконанням, що дозволяє користувачам легко скасовувати помилкові дії та повертатися до попередніх версій документа без втрати даних.

• Легкість додавання нових операцій. Завдяки базовому інтерфейсу ICommand і можливості створення нових реалізацій, легко додавати нові функції (наприклад, команди форматування, пошуку або заміни тексту), не змінюючи існуючий код CommandManager.

• Покращення надійності системи. Відокремлення логіки виконання команд від їх ініціації дозволяє створювати стабільний код з централізованою обробкою помилок та відновленням стану при збоях.

Переваги використання

• Модульність: Патерн забезпечує чітке розмежування відповідальності між управлінням командами (CommandManager), їхнім визначенням (конкретні Command класи) і виконанням операцій (SnippetService).

Код:

```
1   using System;
2
3   namespace TextEditorMK.Commands
4   {
        9 references
5       public interface ICommand
6       {
            8 references
7           string Name { get; }
            7 references
8           bool CanExecute();
            5 references
9           void Execute();
            4 references
10          void Undo();
11      }
12  }
13
```

Рис. 3.1 – ICommand.cs

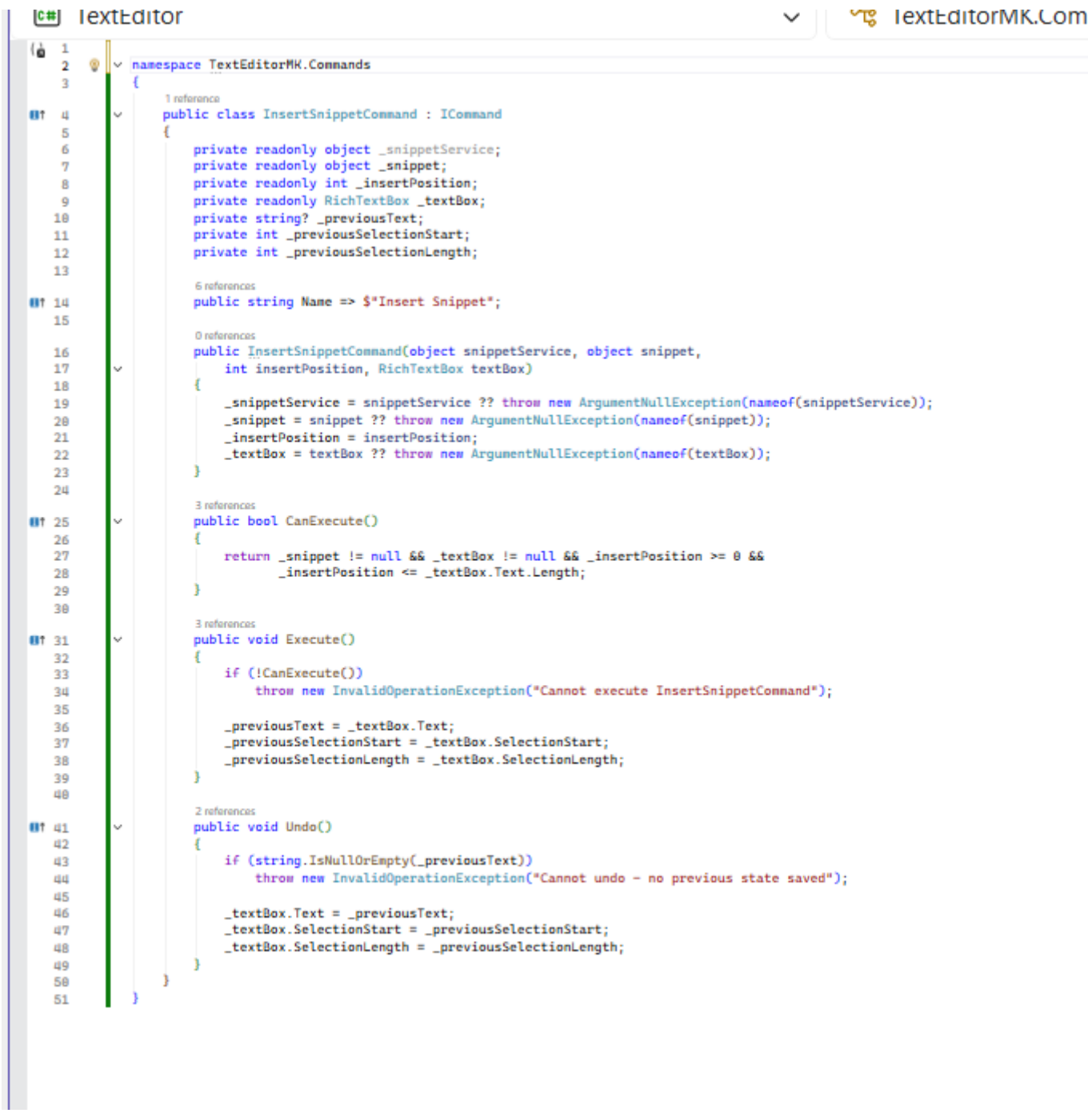


Рис. 3.2 – InsertSnippetCommand.cs

```
using System;

namespace TextEditorMK.Commands
{
    1 reference
    public class DeleteSnippetCommand : ICommand
    {
        private readonly object _snippetService;
        private readonly string _snippetName;
        private bool _wasExecuted;

        6 references
        public string Name => $"Delete Snippet: {_snippetName}";

        0 references
        public DeleteSnippetCommand(object snippetService, string snippetName)
        {
            _snippetService = snippetService ?? throw new ArgumentNullException(nameof(snippetService));
            _snippetName = !string.IsNullOrWhiteSpace(snippetName) ? snippetName :
                throw new ArgumentException("Snippet name cannot be null or empty", nameof(snippetName));
        }

        3 references
        public bool CanExecute()
        {
            return !string.IsNullOrWhiteSpace(_snippetName);
        }

        3 references
        public void Execute()
        {
            if (!CanExecute())
                throw new InvalidOperationException($"Cannot execute DeleteSnippetCommand - snippet '{_snippetName}' not found");

            _wasExecuted = true;
        }

        2 references
        public void Undo()
        {
            if (!_wasExecuted)
                throw new InvalidOperationException("Cannot undo - command was not executed");

            _wasExecuted = false;
        }
    }
}
```

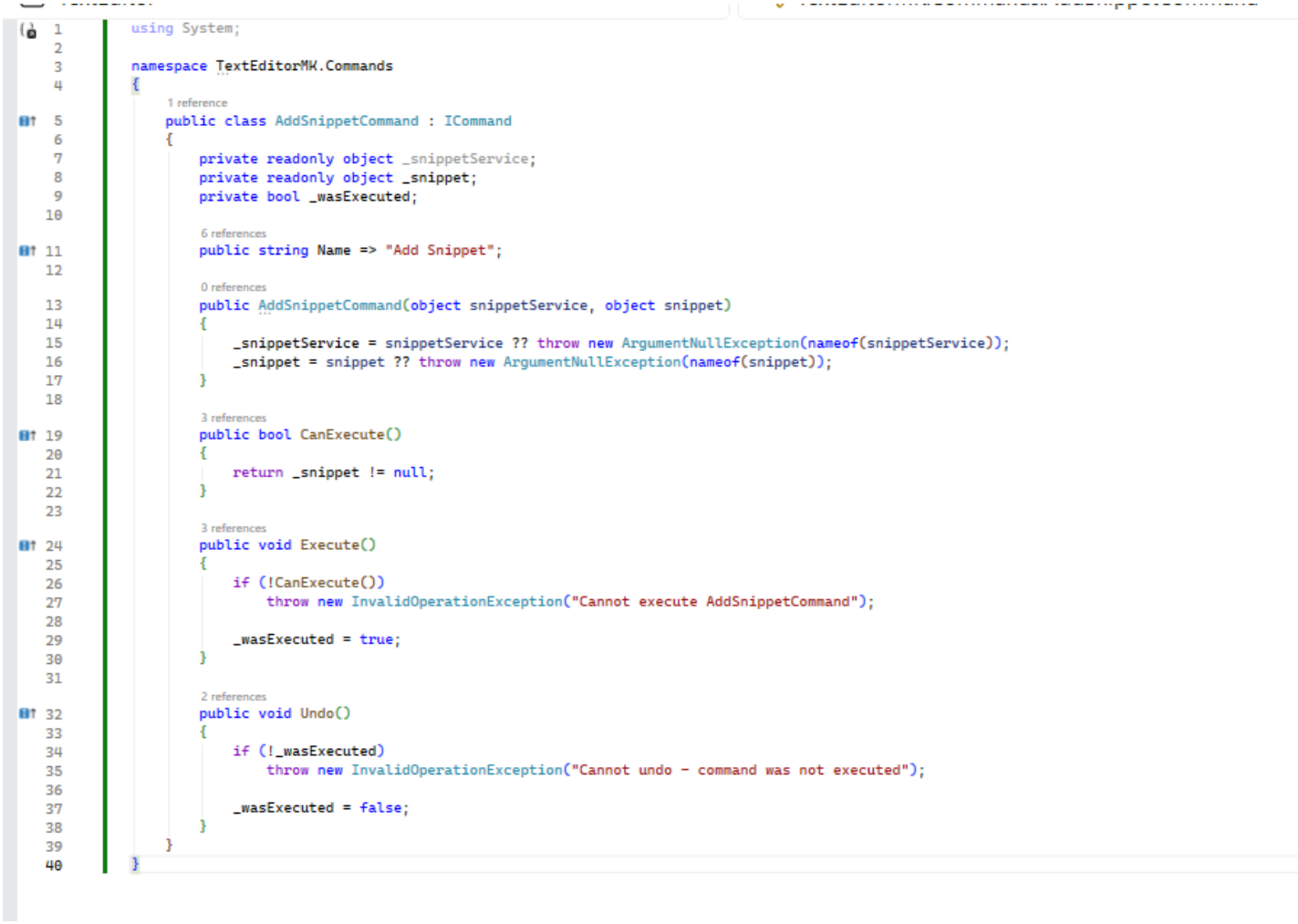
Рис. 3.3 – DeleteSnippetCommand.cs

```
namespace TextEditorMK.Commands
{
    1 reference
    public class CommandManager
    {
        private readonly Stack<ICommand> _executedCommands;
        private readonly Stack<ICommand> _undoneCommands;
        private readonly int _maxHistorySize;

        0 references
        public CommandManager(int maxHistorySize = 100)
        {
            if (maxHistorySize <= 0)
                throw new ArgumentOutOfRangeException(nameof(maxHistorySize));

            _maxHistorySize = maxHistorySize;
            _executedCommands = new Stack<ICommand>();
            _undoneCommands = new Stack<ICommand>();
        }

        0 references
        public void ExecuteCommand(ICommand command)
        {
            if (command == null)
                throw new ArgumentNullException(nameof(command));

            if (!command.CanExecute())
                throw new InvalidOperationException($"Command '{command.Name}' cannot be executed");

            try
            {
                command.Execute();
                _executedCommands.Push(command);
                _undoneCommands.Clear();
                TrimHistoryIfNeeded();
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine($"CommandManager: Error executing '{command.Name}': {ex.Message}");
                throw;
            }
        }
```

```
40
                1 reference
41              private void TrimHistoryIfNeeded()
42              {
43                  if (_executedCommands.Count <= _maxHistorySize)
44                      return;
45
46                  var commands = new ICommand[_maxHistorySize];
47                  for (int i = 0; i < _maxHistorySize; i++)
48                  {
49                      commands[i] = _executedCommands.Pop();
50                  }
51
52                  _executedCommands.Clear();
53
54                  for (int i = _maxHistorySize - 1; i >= 0; i--)
55                  {
56                      _executedCommands.Push(commands[i]);
57                  }
58              }
59
60              public bool Undo()
61              {
62                  if (!CanUndo())
63                      return false;
64
65                  var command = _executedCommands.Pop();
66
67                  try
68                  {
69                      command.Undo();
70                      _undoneCommands.Push(command);
71                      return true;
72                  }
73                  catch (Exception ex)
74                  {
75                      _executedCommands.Push(command);
76                      System.Diagnostics.Debug.WriteLine($"CommandManager: Error undoing '{command.Name}': {ex.Message}");
77                      throw;
78                  }
79              }
80
```

```
80
                0 references
81              public bool Redo()
82              {
83                  if (!CanRedo())
84                      return false;
85
86                  var command = _undoneCommands.Pop();
87
88                  try
89                  {
90                      command.Execute();
91                      _executedCommands.Push(command);
92                      return true;
93                  }
94                  catch (Exception ex)
95                  {
96                      _undoneCommands.Push(command);
97                      System.Diagnostics.Debug.WriteLine($"CommandManager: Error redoing '{command.Name}': {ex.Message}");
98                      throw;
99                  }
100             }
101
                1 reference
102             public bool CanUndo() => _executedCommands.Count > 0;
103
                1 reference
104             public bool CanRedo() => _undoneCommands.Count > 0;
105
                0 references
106             public IEnumerable<string> GetExecutedCommandsHistory()
107             {
108                 return _executedCommands.Select(c => c.Name).ToList().AsReadOnly();
109             }
110
                0 references
111             public void ClearHistory()
112             {
113                 _executedCommands.Clear();
114                 _undoneCommands.Clear();
115             }
116
                0 references
117             public int ExecutedCommandsCount => _executedCommands.Count;
118
                0 references
119             public int UndoneCommandsCount => _undoneCommands.Count;
120         }
121     }
```

Рис. 3.4 – CommandManager.cs



```
using System;

namespace TextEditorMK.Commands
{
    1 reference
    public class AddSnippetCommand : ICommand
    {
        private readonly object _snippetService;
        private readonly object _snippet;
        private bool _wasExecuted;

        6 references
        public string Name => "Add Snippet";

        0 references
        public AddSnippetCommand(object snippetService, object snippet)
        {
            _snippetService = snippetService ?? throw new ArgumentNullException(nameof(snippetService));
            _snippet = snippet ?? throw new ArgumentNullException(nameof(snippet));
        }

        3 references
        public bool CanExecute()
        {
            return _snippet != null;
        }

        3 references
        public void Execute()
        {
            if (!CanExecute())
                throw new InvalidOperationException("Cannot execute AddSnippetCommand");

            _wasExecuted = true;
        }

        2 references
        public void Undo()
        {
            if (!_wasExecuted)
                throw new InvalidOperationException("Cannot undo - command was not executed");

            _wasExecuted = false;
        }
    }
}
```

Рис. 3.5 – AddSnippetCommand.cs

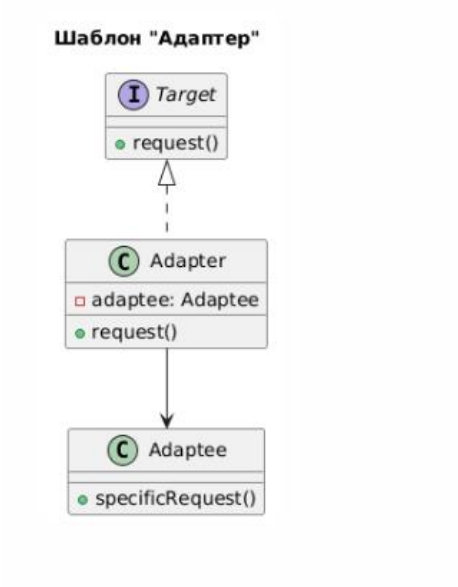
Висновок: У ході роботи було створено проєкт текстового редактора з використанням шаблону Command. Реалізовано основні класи: ICommand - інтерфейс для визначення спільної поведінки всіх команд системи з підтримкою операцій виконання та скасування; InsertSnippetCommand та AddSnippetCommand - конкретні команди для роботи з код-сніпетами, що інкапсулюють логіку вставки та додавання шаблонів коду; CommandManager - центральний клас-інвокер, що керує виконанням команд та підтримує повну історію операцій з можливістю багаторівневого Undo/Redo. Застосування патерну Command дозволило досягти повного розділення між ініціатором команди та її виконавцем, забезпечило надійну систему скасування операцій, зробило систему гнучкою для додавання нових типів команд та спростило тестування окремих операцій.

Відповіді на контрольні питання:

1. Яке призначення шаблону «Адаптер»?

Шаблон «Адаптер» дозволяє об’єктам з несумісними інтерфейсами працювати разом, “обгортаючи” один клас іншим, щоб привести інтерфейс до потрібного виду.

2. Нарисуйте структуру шаблону «Адаптер».



3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

- Target - інтерфейс, який очікує клієнт.
- Adaptee - існуючий клас з іншим інтерфейсом.
- Adapter - “перекладач”, що перетворює виклики Target у виклики Adaptee. Взаємодія: Клієнт працює з Adapter через інтерфейс Target, а Adapter викликає методи Adaptee.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

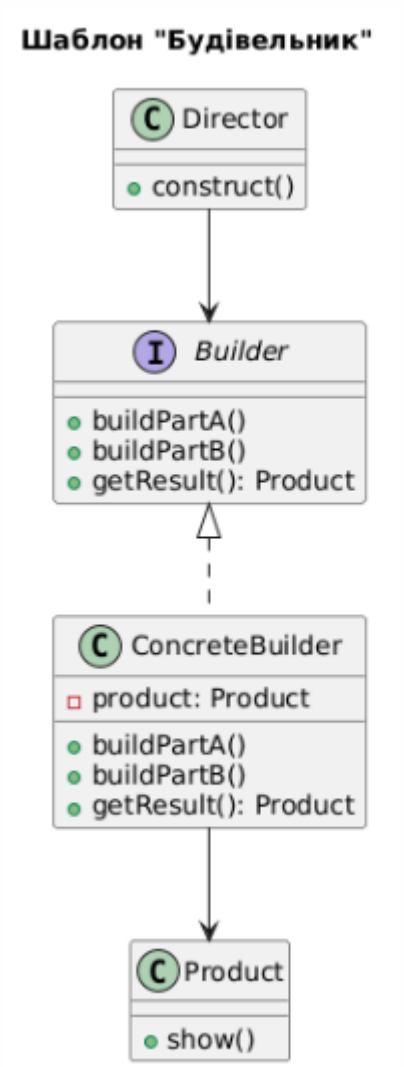
- Об’єктний адаптер - використовує композицію (Adapter має посилання на Adaptee).

- Класовий адаптер - використовує наслідування (Adapter наслідує Adaptee). Об’єктний підхід більш гнучкий, класовий - простіший, але менш універсальний.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» відокремлює процес створення складного об’єкта від його представлення, щоб один і той самий процес міг створювати різні об’єкти.

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

- Director - керує порядком виклику методів будівельника.

- Builder - інтерфейс для створення частин об’єкта.
- ConcreteBuilder - реалізує побудову частин конкретного продукту.
- Product - кінцевий об’єкт.
  Взаємодія: Director викликає методи Builder, а ConcreteBuilder поступово створює Product.

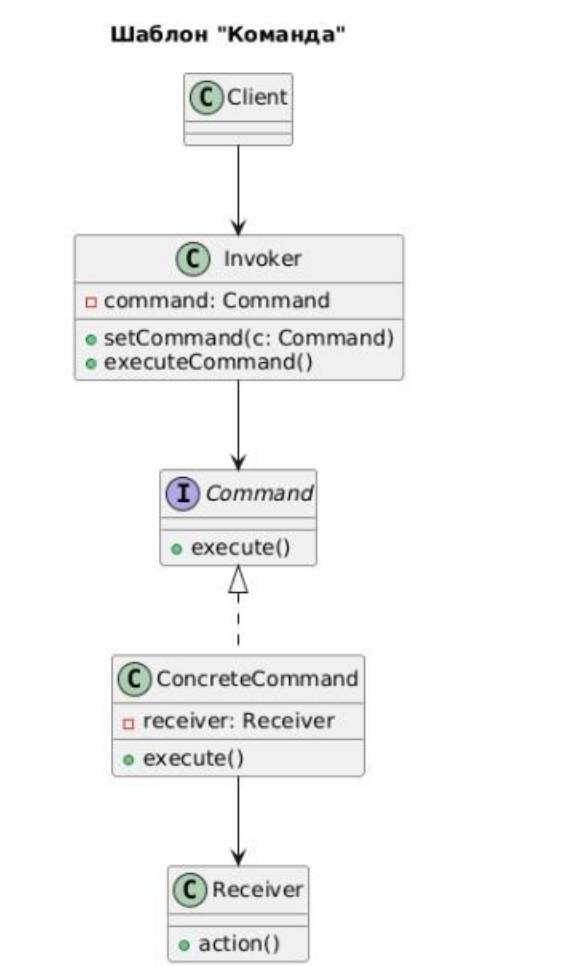
8. У яких випадках варто застосовувати шаблон «Будівельник»?

- Коли потрібно створювати складні об’єкти поетапно.
- Коли процес створення повинен бути незалежним від деталей представлення.
- Коли потрібна можливість створення різних варіантів об’єкта одним способом.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» інкапсулює запит у вигляді об’єкта, дозволяючи передавати, зберігати, виконувати або скасовувати дії.

10. Нарисуйте структуру шаблону «Команда».



11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

- Command - інтерфейс із методом execute().
- ConcreteCommand - реалізує команду, викликаючи методи Receiver.
- Receiver - об’єкт, який виконує фактичну дію.
- Invoker - зберігає команду і викликає її.
- Client - створює конкретні команди.
  Взаємодія: Client створює команду, Invoker її викликає, Receiver виконує дію.

12. Розкажіть як працює шаблон «Команда».

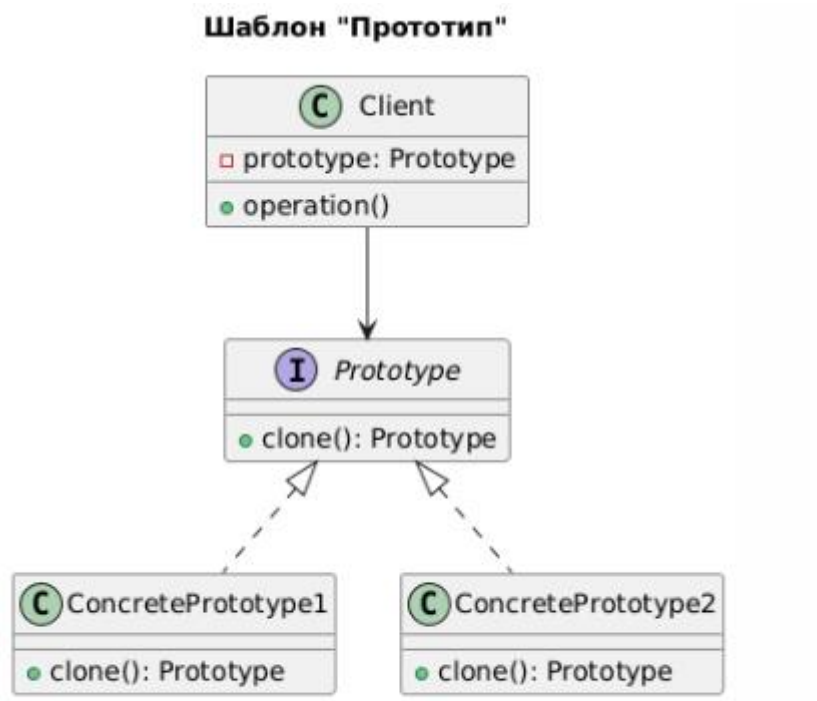
Кожна дія програми представлена як об’єкт-команда. Коли користувач або

система хоче виконати дію, Invoker викликає метод execute() у команді, а вона - виконує потрібну операцію через Receiver. Це дозволяє легко додавати нові дії, скасовувати або повторювати команди.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» дозволяє створювати нові об’єкти шляхом копіювання вже існуючого екземпляра (прототипу), замість створення через конструктор.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

- Prototype - інтерфейс із методом clone().
- ConcretePrototype - реалізує копіювання самого себе.
- Client - створює нові об’єкти через клонування прототипу.
  Взаємодія: Client зберігає посилання на Prototype і створює нові об’єкти методом clone().

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

- Обробка HTTP-запитів (middleware у вебфреймворках).
- Система техпідтримки (запит проходить від оператора до менеджера).
- Система логування (повідомлення передаються кільком логерам).
- GUI-події - натискання кнопки може оброблятись компонентом або передаватись вгору по ієрархії.