Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №5
з дисципліни «Технології розроблення програмного забезпечення»
Тема: «Патерни проектування»

Виконав:
студент групи ІА-34
Тунік О.

Перевірив:
асистент кафедри ІСТ
Мягкий М.Ю.

Київ 2025

Зміст

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

Теоретичні відомості

Шаблон «Adapter»

Патерн Adapter призначений для узгодження інтерфейсів класів, які не можуть взаємодіяти напряму через несумісність своїх інтерфейсів. Він дозволяє обгорнути об’єкт з існуючим інтерфейсом у спеціальний клас-адаптер, який надає очікуваний клієнтом інтерфейс. Це дає змогу використовувати сторонні або застарілі компоненти без зміни їх коду та зменшує залежність клієнтського коду від конкретних реалізацій.

Шаблон «Builder»

Патерн Builder використовується для відокремлення процесу створення складного об’єкта від його кінцевого представлення. Він дозволяє покроково конструювати об’єкт, використовуючи один і той самий алгоритм побудови для створення різних варіантів продукту. Такий підхід підвищує гнучкість процесу створення об’єктів і спрощує підтримку коду при зміні структури або способу ініціалізації складних об’єктів.

Шаблон «Command»

Патерн Command інкапсулює запит на виконання дії в окремий об’єкт, що дозволяє відокремити ініціатора дії від її безпосереднього виконавця. Команди можуть містити параметри виконання та додаткову інформацію, необхідну для підтримки таких механізмів, як логування, відміна або повторне виконання дій. Патерн забезпечує гнучке розширення системи шляхом додавання нових команд без зміни існуючого коду та покращує тестованість логіки виконання.

Шаблон «Chain of Responsibility»

Патерн Chain of Responsibility дозволяє передавати запит послідовно через ланцюжок обробників, поки один із них не візьме його на обробку. Клієнт при цьому не знає, який саме обробник виконає запит, що зменшує зв’язність між компонентами системи. Такий підхід забезпечує гнучке додавання, видалення або перестановку обробників у ланцюжку без зміни клієнтського коду.

Шаблон «Prototype»

Патерн Prototype використовується для створення нових об’єктів шляхом клонування вже існуючих екземплярів, які виступають у ролі прототипів. Це дозволяє уникнути складної логіки створення об’єктів і зменшити залежність клієнтського коду від конкретних класів. Патерн є особливо корисним у випадках,

коли створення об’єктів є ресурсомістким або коли необхідно динамічно змінювати структуру створюваних об’єктів під час виконання програми.

Хід роботи

Поставлене завдання

Flexible automation tool (strategy, command, abstract factory, facade, interpreter, SOA)

Інструмент автоматизації повинен забезпечувати найпростіші автоматичні дії для зручності користувача: завантаження нових фільмів / книг / файлів при випуску (наприклад, щоп'ятниці з'являються нові серії улюблених серіалів); встановити статуси в комунікаторах (skype – away при нульовій активності на тривалий час) і т.д. Автоматизація забезпечується шляхом введення правил (на зразок IFTTT.com сервісу), запису макросів (натискання клавіш, дії миші), планувальника завдань (о 5 ранку – початок роздачі торрент-файлів).

Реалізація паттерну

У системі автоматизації FlexibleAutomationTool реалізовано поведінковий патерн проектування «Команда» (Command) з метою інкапсуляції дій автоматизації у вигляді окремих об’єктів та відокремлення логіки виконання дій від логіки їх ініціації. Використання даного патерну дозволяє зробити систему гнучкою, розширюваною та придатною до реалізації додаткових можливостей, таких як валідація, логування, керування доступністю дій, а також потенційна підтримка скасування (Undo).

Патерн «Команда» у даній архітектурі складається з таких ключових елементів: інтерфейсу/абстракції команди (ActionBase), конкретних команд (MessageBoxAction, RunProgramAction, OpenUrlAction, FileWriteAction, MacroAction), одержувачів (platform services) та ініціаторів виконання (AutomationEngine, UI-компоненти). Спроектовану діаграму класів, що ілюструє використання патерну «Команда», наведено на рис. 1.

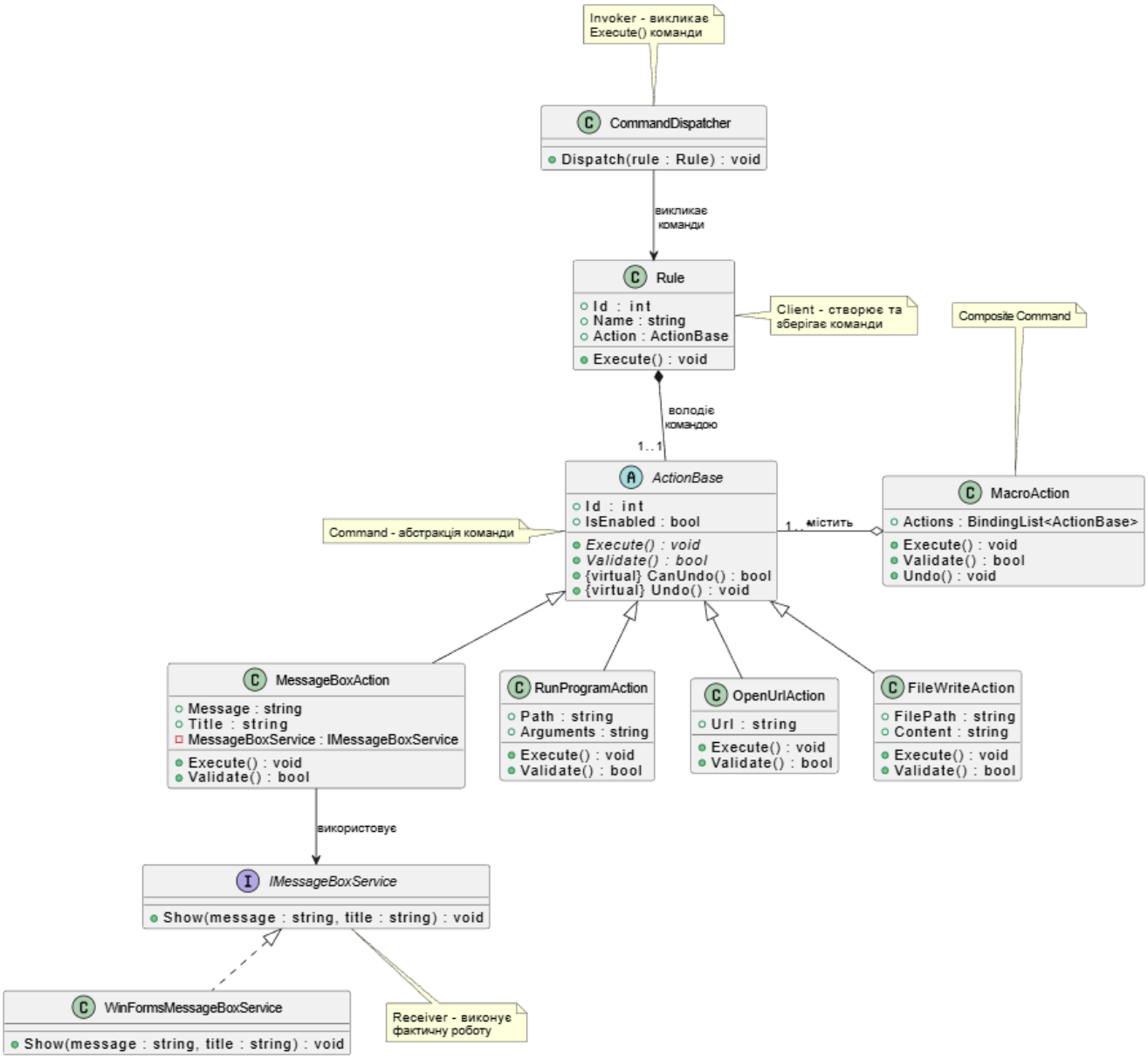


Рисунок 1. Діаграма класів паттерну Команда

Центральним елементом архітектури є абстрактний клас ActionBase, який виконує роль базової команди та визначає єдиний контракт для всіх дій автоматизації. Даний клас містить метод Execute(), який інкапсулює виконання дії, а також метод Validate(), що відповідає за перевірку коректності параметрів команди перед виконанням. Додатково в ActionBase визначені віртуальні методи CanUndo() та Undo(), які створюють основу для підтримки скасування дій у майбутньому. Наявність властивості IsEnabled дозволяє керувати доступністю команд з боку користувацького інтерфейсу або планувальника.

Конкретні команди реалізують клас ActionBase та інкапсулюють конкретні дії системи. Наприклад, MessageBoxAction відповідає за відображення повідомлення користувачеві, RunProgramAction – за запуск зовнішнього процесу, OpenUrlAction – за відкриття веб-посилання, а FileWriteAction – за запис даних у файл. Кожна команда містить лише ті дані, які необхідні для виконання відповідної дії, та не

залежить від способу її ініціації. Фактичне виконання дій делегується спеціалізованим сервісам платформи, наприклад IMessageBoxService, що виступають у ролі одержувачів (Receiver) у патерні «Команда».

Окрему роль у системі відіграє клас MacroAction, який реалізує композицію команд та дозволяє об'єднувати декілька об'єктів ActionBase у єдину складену команду. При виклику методу Execute() MacroAction послідовно виконує всі вкладені дії, що дозволяє створювати сценарії автоматизації довільної складності. Такий підхід поєднує патерни Command і Composite, забезпечуючи підтримку ланцюжків команд та повторного використання дій.

Ініціатором виконання команд у системі виступає AutomationEngine разом із компонентом CommandDispatcher. AutomationEngine реагує на події готовності правил до виконання, інжектує необхідні платформені сервіси у відповідні команди та передає керування диспетчеру. CommandDispatcher викликає метод Execute() відповідної команди, відповідає за логування результатів виконання та обробку помилок. При цьому ініціатор не має інформації про внутрішню реалізацію конкретної дії, що відповідає принципу слабкого зв'язування.

Клас Rule виступає зв'язувальною ланкою між умовою спрацювання та дією. Він містить об'єкт Trigger, який визначає момент виконання, і об'єкт ActionBase, що представляє команду. Метод Execute() класу Rule делегує виконання інкапсульованій команді, не знаючи деталей її реалізації. Така структура дозволяє незалежно змінювати як логіку тригерів, так і набір доступних команд.

Фрагменти коду реалізації

Абстрактний клас ActionBase визначає контракт для всіх команд автоматизації:

```
namespace FlexibleAutomationTool.Core.Actions
{

    [TypeConverter(typeof(ExpandableObjectConverter))]
    public abstract class ActionBase
    {
        [Browsable(false)]
        public int Id { get; set; }

        public bool IsEnabled { get; set; } = true;

        public abstract void Execute();

        public abstract bool Validate();

        [Browsable(false)]
        public virtual bool CanUndo => false;

        public virtual void Undo()
        {
            throw new NotSupportedException("Undo is not supported for this action.");
        }

        public override string ToString() => GetType().Name;
```

```
    }
}
```

Клас MessageBoxAction реалізує конкретну команду та інкапсулює дію відображення повідомлення:

```
namespace FlexibleAutomationTool.Core.Actions.InternalActions
{
    public class MessageBoxAction : ActionBase
    {
        public string Message { get; set; } = "";
        public string Title { get; set; } = "";

        [Browsable(false)]
        public IMessageBoxService? MessageBoxService { get; set; }

        public MessageBoxAction()
        {
        }

        public MessageBoxAction(IMessageBoxService service)
        {
            MessageBoxService = service;
        }

        public override void Execute()
        {
            try
            {
                if (MessageBoxService == null)
                    throw new System.InvalidOperationException("No IMessageBoxService
available to show message. Set MessageBoxService before executing this action.");

                MessageBoxService.Show(Message, Title);
            }
            catch (System.Exception ex)
            {
                throw new System.InvalidOperationException($"Failed to show message box:
{ex.Message}", ex);
            }
        }

        public override bool Validate() => !string.IsNullOrWhiteSpace(Message);
    }
}
```

Інтерфейс сервісу, який виконує фактичну роботу для команди:

```
namespace FlexibleAutomationTool.Core.Interfaces
{
    public interface IMessageBoxService
    {
        void Show(string message, string? title = null);
    }
}
```

Платформна реалізація одержувача:

```
namespace FlexibleAutomationTool.UI.Services
{
    public class WinFormsMessageBoxService : IMessageBoxService
    {
        public void Show(string message, string? title = null)
        {
            MessageBox.Show(message, title ?? "Info");
        }
```

```
    }
}
```

Клас MacroAction дозволяє об’єднувати декілька команд в одну:

```
namespace FlexibleAutomationTool.Core.Actions
{
    [TypeConverter(typeof(ExpandableObjectConverter))]
    public class MacroAction : ActionBase
    {
        [Browsable(true)]
        [DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]
        public BindingList<ActionBase> Actions { get; set; } = new
BindingList<ActionBase>();

        public override void Execute()
        {
            List<Exception> errors = null;

            foreach (var a in Actions)
            {
                try
                {
                    a.Execute();
                }
                catch (Exception ex)
                {
                    errors ??= new List<Exception>();
                    errors.Add(ex);
                }
            }

            if (errors != null && errors.Count > 0)
            {
                throw new AggregateException("One or more actions in the macro failed.",
errors);
            }
        }

        public override bool Validate() => Actions.Count > 0;

        public override string ToString() => $"{GetType().Name} ({Actions?.Count ?? 0}
actions)";
    }
}
```

Диспетчер команд інкапсулює виклик Execute та додаткову логіку:

```
namespace FlexibleAutomationTool.Core.Services
{
    public class CommandDispatcher : ICommandDispatcher
    {
        private readonly Logger _logger;

        public CommandDispatcher(Logger logger)
        {
            _logger = logger;
        }

        public void Dispatch(Rule rule)
        {
            try
            {
                rule.Execute();
```

```
                _logger.Log(rule.Name, "executed via dispatcher");
            }
            catch (Exception ex)
            {
                _logger.Log(rule.Name, $"dispatch failed: {ex}");
                throw;
            }
        }
    }
}
```

Обґрунтування використання паттерну

Використання патерну «Команда» (Command) у системі FlexibleAutomationTool є обґрунтованим з огляду на необхідність гнучкого та розширюваного керування діями автоматизації. У реальних сценаріях використання системи, набір можливих дій постійно змінюється та розширюється: від простих операцій, таких як відображення повідомлення або відкриття URL, до складніших дій, пов'язаних із запуском програм, модифікацією файлів або виконанням послідовностей команд. Без застосування патерну «Команда» логіка виконання таких дій була б жорстко прив'язана до місць їх виклику, що призвело б до дублювання коду, зростання зв'язаності між компонентами та ускладнення підтримки системи.

Патерн «Команда» дозволяє інкапсулювати кожну дію автоматизації в окремий об'єкт, що робить дії повноправними елементами моделі предметної області. У даній реалізації цю роль виконує абстрактний клас ActionBase, який визначає єдиний контракт для всіх команд. Завдяки цьому ініціатори виконання (AutomationEngine, CommandDispatcher, UI-компоненти) не залежать від конкретних реалізацій дій і взаємодіють з ними через спільну абстракцію. Такий підхід відповідає принципу інверсії залежностей та зменшує зв'язність між компонентами системи.

Застосування патерну «Команда» також дозволяє чітко розділити відповідальності між елементами архітектури. Команди відповідають лише за опис та ініціацію дії, тоді як фактичне виконання делегується спеціалізованим одержувачам (Receiver), реалізованим у вигляді платформених сервісів. Наприклад, MessageBoxAction не містить залежності від конкретної UI-технології, а взаємодіє з інтерфейсом IMessageBoxService, що дозволяє легко адаптувати систему до різних середовищ виконання або замінювати реалізації сервісів без змін у коді команд.

Важливою перевагою використання патерну «Команда» є можливість реалізації додаткових нефункціональних вимог без зміни існуючого коду дій. У системі FlexibleAutomationTool це проявляється у підтримці валідації команд, логування виконання, керування доступністю дій через властивість IsEnabled. Без патерну «Команда» впровадження таких можливостей вимагало б значних змін у багатьох частинах системи.

Окремо слід відзначити використання класу MacroAction, який дозволяє об’єднувати декілька команд у єдину складену дію. Це забезпечує підтримку ланцюжків команд та сценаріїв автоматизації довільної складності, що повністю відповідає призначенню патерну «Команда». При цьому складені команди можуть використовуватися так само, як і звичайні, що демонструє дотримання принципу підстановки Лісков та підвищує узгодженість архітектури.

Застосування патерну «Команда» також значно спрощує тестування системи. Оскільки логіка виконання дій інкапсульована в окремих класах і не залежить від користувацького інтерфейсу, команди можуть бути протестовані за допомогою модульних тестів без необхідності емулювати взаємодію з UI або події користувача. Це підвищує якість коду та зменшує вартість супроводу системи.

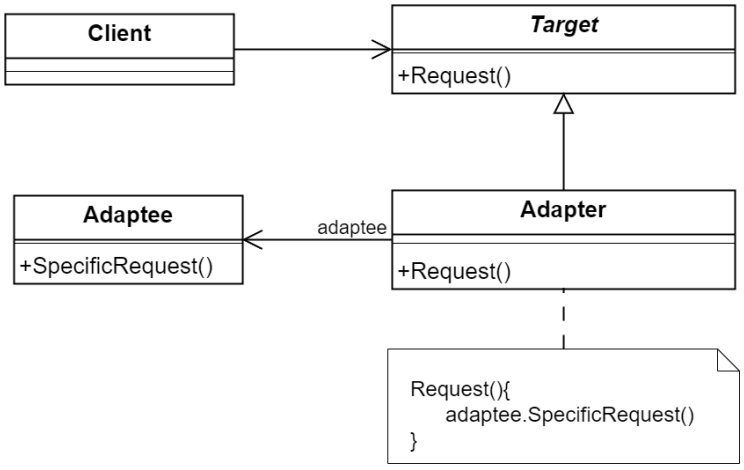
Порівняно з альтернативними підходами, такими як безпосереднє розміщення логіки дій у класах правил, планувальника або обробниках UI-подій, патерн «Команда» забезпечує значно кращу архітектурну структуру. Жорстке кодування дій у місцях виклику призводило б до порушення принципу відкритості-закритості та унеможливлювало б незалежну еволюцію системи. Використання ж патерну «Команда» дозволяє розширювати систему шляхом додавання нових команд без модифікації існуючого коду, зберігаючи чисту, модульну та масштабовану архітектуру.

Відповіді на контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Шаблон «Адаптер» дозволяє узгодити несумісні інтерфейси класів, щоб вони могли працювати разом.

2. Нарисуйте структуру шаблону «Адаптер»



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

У шаблон входять Client, Target, Adapter та Adaptee, де Adapter реалізує Target і делегує виклики Adaptee.

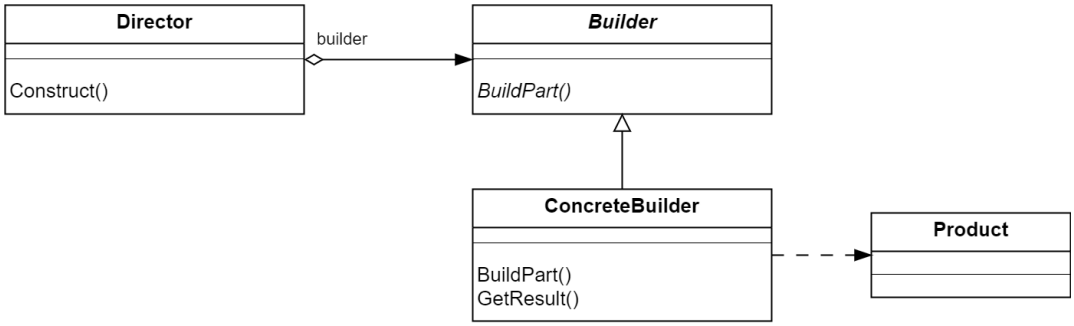
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

Об’єктний адаптер використовує композицію, а класовий – наслідування.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» відокремлює процес створення складного об’єкта від його представлення.

6. Нарисуйте структуру шаблону «Будівельник»



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

У шаблон входять Director, Builder, ConcreteBuilder і Product, де Director керує побудовою через Builder.

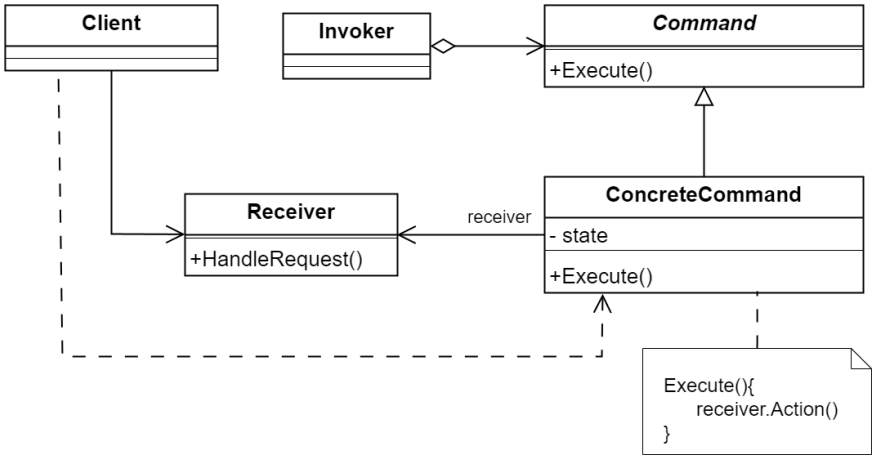
8. У яких випадках варто застосовувати шаблон «Будівельник»?

Будівельник застосовують, коли об’єкт має складну структуру або багато варіантів конфігурації.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» інкапсулює запит на виконання дії в окремий об’єкт.

10. Нарисуйте структуру шаблону «Команда».



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

У шаблон входять Invoker, Command, ConcreteCommand та Receiver, де команда викликає методи отримувача.

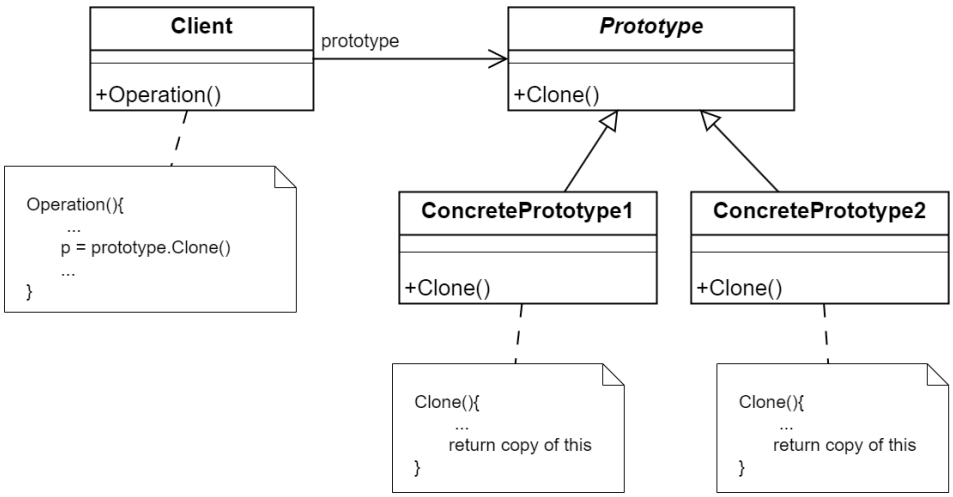
12. Розкажіть як працює шаблон «Команда».

Invoker викликає команду, яка делегує виконання конкретному Receiver.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» дозволяє створювати нові об’єкти шляхом клонування існуючих.

14. Нарисуйте структуру шаблону «Прототип».



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

У шаблон входять Client, Prototype та ConcretePrototype, де клієнт клонує об’єкти через інтерфейс Prototype.

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

Обробка подій у GUI, фільтрація HTTP-запитів, система логування або перевірка прав доступу.

Висновки

У ході виконання лабораторної роботи розглянули та реалізовали поведінковий патерн проєктування «Команда» (Command) у контексті системи автоматизації FlexibleAutomationTool. Застосування даного патерну дозволило інкапсулювати дії автоматизації у вигляді окремих об’єктів та відокремити логіку їх виконання від логіки ініціації, що забезпечує гнучку й розширювану архітектуру системи. Реалізована структура з базовим абстрактним класом ActionBase, набором конкретних команд, ініціатором виконання AutomationEngine та одержувачами у вигляді платформених сервісів демонструє чітке розділення відповідальностей між компонентами.

Отримані результати підтверджують доцільність використання патерну «Команда» у системах автоматизації, де набір можливих дій є динамічним і потребує постійного розширення. Реалізована архітектура спрощує додавання нових команд без зміни існуючого коду, забезпечує можливість впровадження додаткових нефункціональних вимог, таких як валідація, логування та керування доступністю дій, а також створює основу для підтримки скасування операцій у майбутньому. Використання складених команд у вигляді MacroAction демонструє ефективність поєднання патернів Command і Composite для побудови сценаріїв автоматизації довільної складності. Загалом реалізація патерну в межах лабораторної роботи підтверджує ефективність використання шаблонів проєктування для створення модульних, масштабованих і зручних у супроводі програмних систем.