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

Лабораторна робота №4

# Технології розроблення програмного забезпечення

*«Шаблони «Singleton», «Iterator», «Proxy», «State», «Strategy»»*

Варіант 24

Виконав
студент групи ІА–33:
Вовчок М.М.

Перевірив
Мягкий М. Ю.

Київ — 2025

**Тема**

Реалізація шаблону проектування State у системі Flexible Automation Tool для керування станами правила автоматизації.

**Мета роботи**

Ознайомитися з шаблонами проектування та реалізувати один з них (State) на прикладі системи Flexible Automation Tool, продемонструвавши зміну поведінки об'єкта залежно від його стану.

**Короткі теоретичні відомості**

Шаблони проектування (Design Patterns) — це типові рішення поширених задач проектування програмного забезпечення. Вони описують структури класів та взаємодію об'єктів, що дозволяє підвищити гнучкість та підтримуваність коду.

Поведікові шаблони описують алгоритми та розподіл відповідальності між об'єктами. До цієї групи належить шаблон State (Стан), який дозволяє змінювати поведінку об'єкта під час виконання програми в залежності від його внутрішнього стану.

Шаблон State передбачає наявність контексту (об'єкта, поведінка якого змінюється) та набору класів-станів, що реалізують спільний інтерфейс. Контекст зберігає посилання на поточний стан і делегує йому виконання операцій. Кожен стан може ініціювати перехід до іншого стану.

**Обґрунтування вибору шаблону State**

У системі Flexible Automation Tool важливо керувати життєвим циклом правила автоматизації. Правило може перебувати у різних станах: чернетка (Draft), активне (Active), призупинене (Paused), виконане

(Completed), у стані помилки (Error) або завершене/відключене (Done). Поведінка правила (наприклад, можливість виконання) безпосередньо залежить від його поточного стану.

Якщо реалізовувати цю логіку за допомогою умовних операторів, код контексту (класу AutomationRule) буде перевантажений перевірками станів. Використання шаблону State дозволяє винести логіку у окремі класи-стани та спростити підтримку і розширення системи.

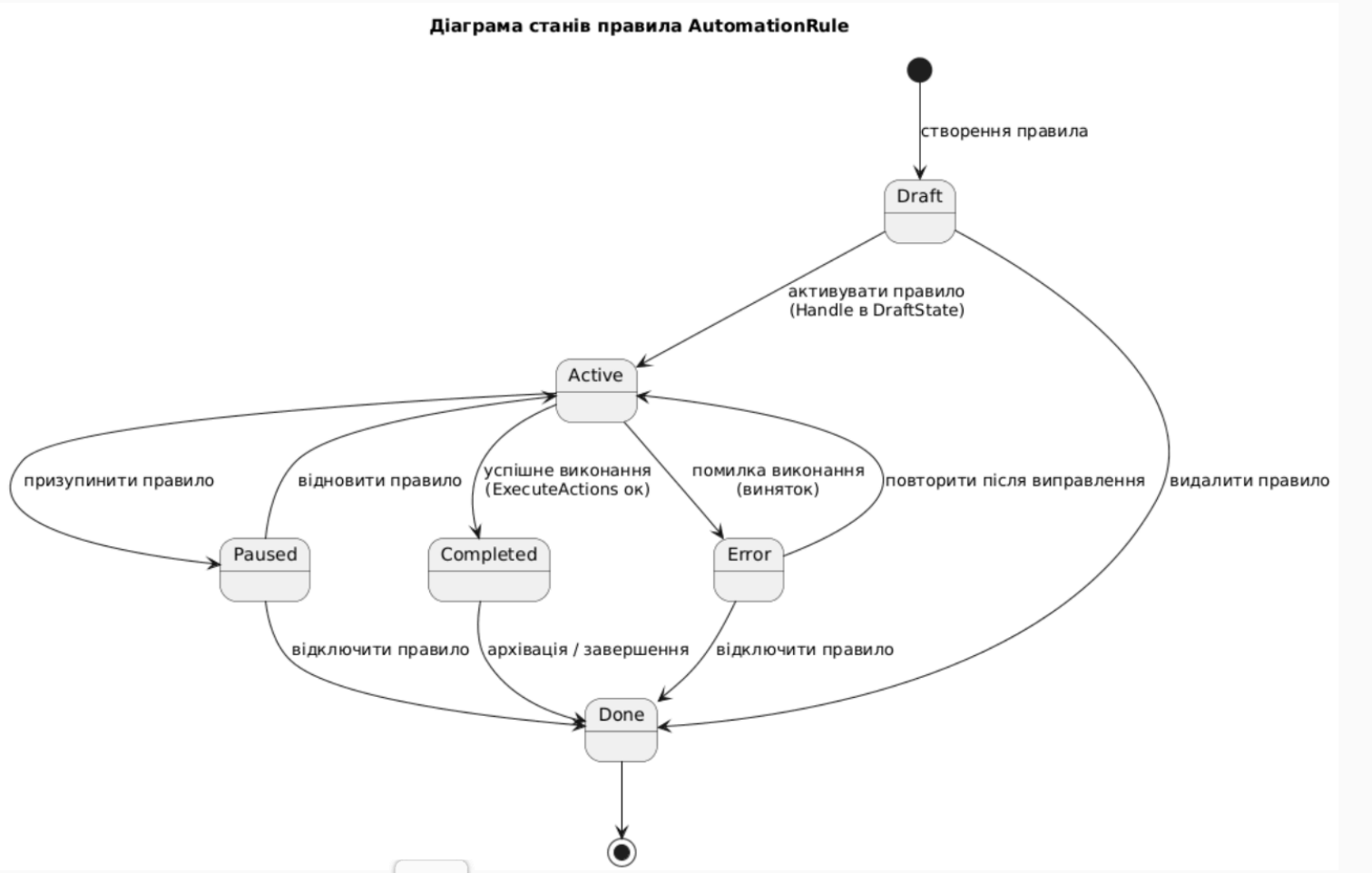
**Опис реалізації шаблону State**

У рамках лабораторної роботи було реалізовано такі основні елементи:

1. Інтерфейс IRuleState — описує спільний інтерфейс для всіх станів правила автоматизації (метод Handle та властивість Name).
2. Клас AutomationRule — контекст, який містить дані правила автоматизації та посилання на поточний стан. Містить методи TransitionTo для зміни стану та Handle для делегування обробки поточному стану.
3. Конкретні стани: DraftState, ActiveState, PausedState, CompletedState, ErrorState, DoneState, які реалізують інтерфейс IRuleState та містять специфічну логіку обробки.

Під час виконання системи об'єкт AutomationRule створюється у стані Draft. Після першого виклику Handle правило переводиться у стан Active, де виконується основна логіка та запуск дій. При успішному виконанні правило переходить у стан Completed, а далі — у фінальний стан Done. У разі виникнення помилки правило переходить у стан Error.

## Діаграма станів правила автоматизації




Діаграма станів (рис. 1)

Зображено основні стани об'єкта AutomationRule та можливі переходи між ними:

• Draft — початковий стан, правило перебуває у вигляді чернетки;

• Active — активний стан, правило може виконуватися за розкладом або подією;

• Paused — призупинений стан, виконання тимчасово заблоковано;

• Completed — правило успішно виконано (для одноразових правил);

• Error — стан помилки під час виконання правила;

• Done — фінальний стан, правило більше не використовується.

Переходи між станами ініціюються викликами методу Handle у відповідних класах-станах.

## Реалізація в коді

Нижче наведені основні фрагменти коду реалізації шаблону State.

### Інтерфейс IRuleState

```
public interface IRuleState
{
   void Handle(AutomationRule rule);
   string Name { get; }
}
```

### Клас AutomationRule (контекст)

```
public class AutomationRule
{
   public int Id { get; set; }
   public string Name { get; set; } = string.Empty;
   public string ScheduleExpression { get; set; } = string.Empty;

   private IRuleState _state;

   public AutomationRule(IRuleState initialState)
   {
```

```
        TransitionTo(initialState);
    }

    public void TransitionTo(IRuleState newState)
    {
        Console.WriteLine($"Rule [{Name}] transition to state: {newState.Name}");
        _state = newState;
    }

    public void Handle()
    {
        _state.Handle(this);
    }

    public void ExecuteActions()
    {
        Console.WriteLine($"Rule [{Name}] is executing actions...");
    }

    public void Log(string message)
    {
        Console.WriteLine($"[LOG][{Name}] {message}");
    }
}
```

### Клас DraftState

```
public class DraftState : IRuleState
{
    public string Name => "Draft";

    public void Handle(AutomationRule rule)
    {
        rule.Log("Rule is in draft state. Execution is not allowed.");
        rule.TransitionTo(new ActiveState());
    }
}
```

### Клас ActiveState

```
public class ActiveState : IRuleState
{
    public string Name => "Active";

    public void Handle(AutomationRule rule)
    {
        try
        {
            rule.Log("Rule is active. Starting execution...");
```

```
            rule.ExecuteActions();
            rule.TransitionTo(new CompletedState());
        }
        catch (Exception ex)
        {
            rule.Log($"Execution failed: {ex.Message}");
            rule.TransitionTo(new ErrorState(ex.Message));
        }
    }
}
```

### Клас CompletedState

```
public class CompletedState : IRuleState
{
    public string Name => "Completed";

    public void Handle(AutomationRule rule)
    {
        rule.Log("Rule is already completed. No execution needed.");
        rule.TransitionTo(new DoneState());
    }
}
```

### Клас ErrorState

```
public class ErrorState : IRuleState
{
    public string Name => "Error";

    private readonly string _errorMessage;

    public ErrorState(string errorMessage)
    {
        _errorMessage = errorMessage;
    }

    public void Handle(AutomationRule rule)
    {
        rule.Log($"Rule is in error state: {_errorMessage}");
    }
}
```

### Клас DoneState

```
public class DoneState : IRuleState
{
    public string Name => "Done";

    public void Handle(AutomationRule rule)
```

```
    {
        rule.Log("Rule is in final state. No further processing.");
    }
}
```

### Демонстраційний приклад (Program.cs)

```
static void Main(string[] args)
{
    var rule = new AutomationRule(new DraftState())
    {
        Id = 1,
        Name = "Download new episodes",
        ScheduleExpression = "0 18 * * 5"
    };

    Console.WriteLine("=== First handle (Draft -> Active) ===");
    rule.Handle();

    Console.WriteLine("\n=== Second handle (Active -> Completed) ===");
    rule.Handle();

    Console.WriteLine("\n=== Third handle (Completed -> Done) ===");
    rule.Handle();

    Console.WriteLine("\nPress any key to exit...");
    Console.ReadKey();
}
```

**Висновок**

У ході виконання лабораторної роботи було розглянуто шаблон проектування State та реалізовано його у системі Flexible Automation Tool для керування станами правила автоматизації. Використання цього шаблону дозволило винести логіку обробки станів у окремі класи та спростити код контексту. Продемонстровано послідовний перехід правила між станами Draft, Active, Completed та Done під час виконання програми.