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

Лабораторна робота №4
з дисципліни «Технології розроблення програмного забезпечення»
Тема: «Вступ до паттернів проектування»

Виконав:
студент групи ІА-34
Тунік О.

Перевірив:
асистент кафедри ІСТ
Мягкий М.Ю.

Київ 2025

Зміст

Теоретичні відомості 3
Singleton (Одинак) 3
Iterator (Ітератор) 3
Proxy (Проксі) 3
State (Стан) 4
Strategy (Стратегія) 4
Хід роботи 5
Поставлене завдання 5
Реалізація паттерну 5
Фрагменти коду реалізації 7
Обґрунтування використання паттерну 8
Відповіді на контрольні запитання 10
Висновки 13

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

Теоретичні відомості

Шаблони проєктування являють собою формалізовані та багаторазово перевірені в реальних проєктах рішення типових задач проєктування. Вони мають усталені назви та описують структуру рішення, взаємодію його складових і рекомендації щодо застосування. Застосування шаблонів дозволяє будувати моделі систем, у яких виокремлені суттєві елементи та зв'язки, завдяки чому архітектура стає простішою, наочнішою та гнучкішою. Вони підвищують стійкість систем до змін вимог, полегшують подальше доопрацювання та створюють єдиний «словник» для спілкування розробників.

Singleton (Одинак)

Патерн Singleton призначений для створення класу, що може мати лише один екземпляр або обмежену кількість екземплярів і надає глобальну точку доступу до цього об'єкта. Такий підхід необхідний у випадках, коли фізично існує лише один об'єкт, наприклад файл конфігурації, або коли потрібно контролювати всі операції через один центр, як у випадку єдиного сеансу зв'язку. Незважаючи на зручність, Singleton часто вважають анти-шаблоном, оскільки він створює глобальний стан, ускладнює тестування, порушує принцип єдиної відповідальності та може маскувати помилки в архітектурі.

Iterator (Ітератор)

Iterator забезпечує спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої реалізації. Колекції можуть мати різні структури — від простих списків до дерев чи графів, — але незалежно від цього користувач повинен мати можливість обійти їх елементи. Винесення логіки обходу в окремі класи дозволяє використовувати різні алгоритми переміщення по колекції, не змінюючи саму колекцію. Ітератор визначає операції переходу до першого та наступного елементів, визначення завершення обходу та отримання поточного елемента, що робить роботу з наборами даних універсальною. Цей підхід спрощує колекції, хоча в простих випадках може бути зайвим.

Proxy (Проксі)

Патерн Proxy створює об'єкт-замісник, що контролює доступ до реального об'єкта і може додавати додаткову логіку, не змінюючи клієнтський код. Він дозволяє оптимізувати взаємодію із зовнішніми ресурсами, обмежувати доступ, кешувати дані або виконувати інші додаткові дії. Наприклад, використання проксі між системою та сервісом підписання DocuSign дає змогу групувати запити й зменшувати їх кількість, значно знижуючи вартість використання сервісу. Клієнт працює через інтерфейс, тому присутність проксі для нього прозора. Хоча

використання замісника робить систему гнучкішою, воно може знижувати швидкодію та створювати ризики некоректної синхронізації відповідей із реальним об’єктом.

State (Стан)

State дозволяє змінювати поведінку об’єкта залежно від його внутрішнього стану шляхом винесення логіки кожного стану в окремі класи. Контекст містить посилання на поточний стан і делегує йому виконання дій, а зміна стану здійснюється простою заміною об’єкта стану. Це зручно у випадках, коли поведінка об’єкта суттєво різниться залежно від обставин, наприклад у роботі серверного Listener, який має три різні режими: ініціалізації, нормального функціонування та завершення. Патерн спрощує код контексту й полегшує додавання нових станів, хоча сам механізм їх перемикання може бути доволі складним.

Strategy (Стратегія)

Патерн Strategy дає змогу виділяти різні алгоритми в окремі класи та легко замінювати їх під час виконання програми. Контекст містить посилання на певну стратегію, а коли необхідно змінити алгоритм, достатньо підставити інший об’єкт. Такий підхід використовується там, де можливо досягти однакової мети різними способами, наприклад у різних алгоритмах сортування або у виборі політики обробки даних. На відміну від State, Стратегія не відображає стани об’єкта, а забезпечує гнучку зміну способу його роботи. Вона робить код контексту чистішим і зменшує кількість умовних конструкцій, але може ускладнювати систему, якщо алгоритмів мало або вони дуже прості.

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 реалізуємо поведінковий патерн проектування Стратегія для забезпечення гнучкості у виборі алгоритмів перевірки готовності правил до виконання. Цей патерн складається з трьох ключових елементів: контексту (Scheduler), інтерфейсу стратегії (ITriggerStrategy) та конкретної реалізації стратегії (PollingTriggerStrategy). Така архітектура дозволяє визначати родину алгоритмів, інкапсулювати кожен з них та робити їх взаємозамінними під час виконання програми.

Спроектували діаграму класів, які були реалізовані використовуючи патерн (рис.1).

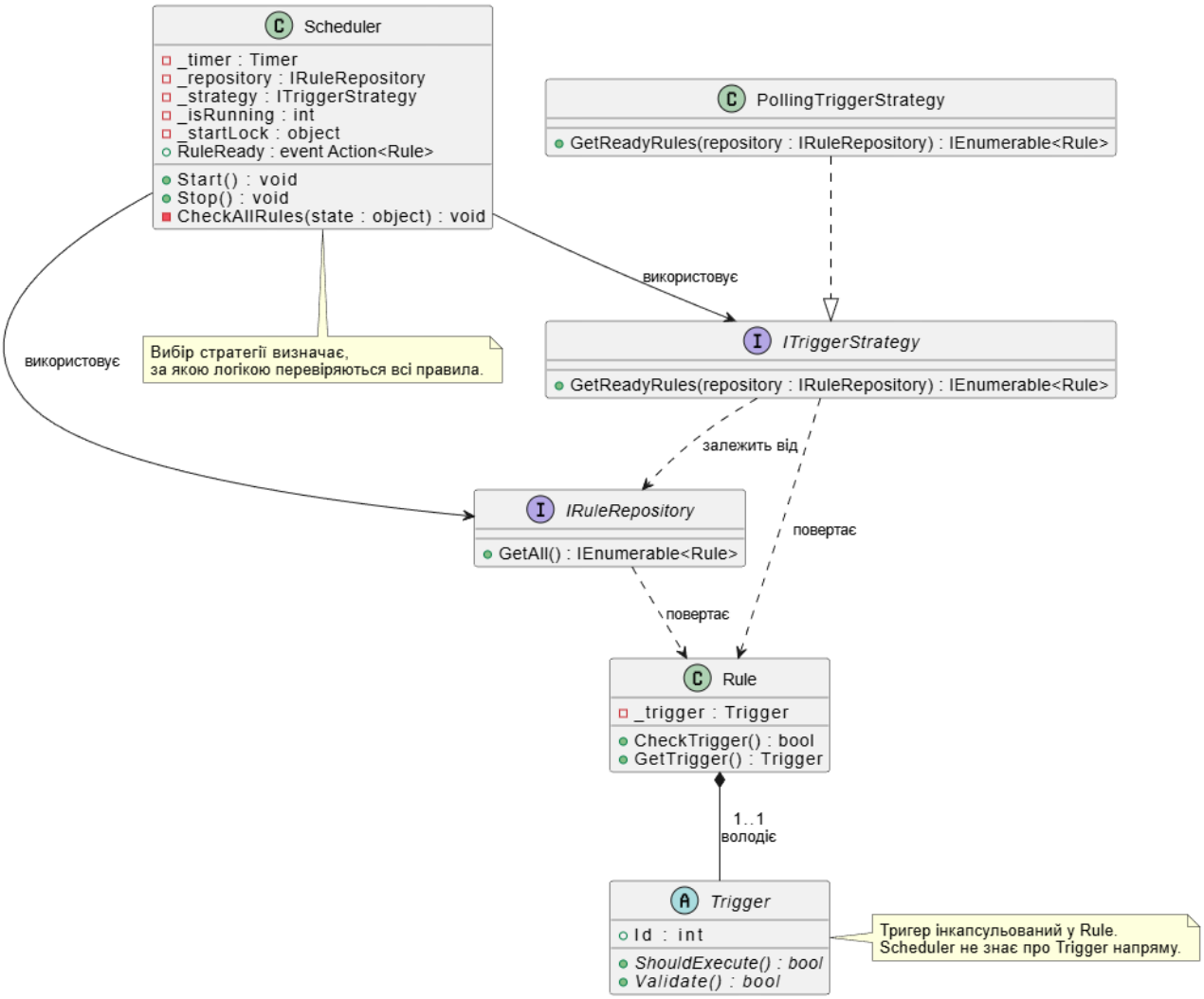


Рисунок 1. Діаграма класів паттерну Стратегія

Центральним елементом архітектури є інтерфейс ITriggerStrategy, який визначає контракт для всіх стратегій перевірки правил. Цей інтерфейс містить єдиний метод GetReadyRules, що приймає репозиторій правил та повертає колекцію правил, готових до виконання. Така абстракція дозволяє відокремити логіку визначення готовності правил від логіки їх виконання.

Центральним елементом архітектури є інтерфейс ITriggerStrategy, який виступає абстракцією стратегії та визначає контракт для всіх можливих алгоритмів перевірки правил. Цей інтерфейс містить єдиний метод GetReadyRules, що приймає репозиторій правил та повертає колекцію правил, готових до виконання. Така абстракція дозволяє відокремити логіку визначення готовності правил від логіки управління їх виконанням у контексті.

Конкретна реалізація стратегії представлена класом PollingTriggerStrategy, який використовує підхід періодичного опитування всіх правил у системі. У методі GetReadyRules відбувається ітерація по всіх правилах з репозиторію, де для кожного правила викликається метод CheckTrigger для визначення необхідності його виконання. Використання yield return забезпечує ледачу ініціалізацію та

ефективне використання пам'яті при роботі з великою кількістю правил. Важливо зазначити, що стратегія не має прямого доступу до об'єктів Trigger, оскільки тригери інкапсульовані всередині класу Rule, що забезпечує додатковий рівень абстракції та захисту даних.

Клас Scheduler виступає контекстом у патерні Стратегія та відповідає за керування процесом виконання правил автоматизації. Планувальник зберігає посилання на обрану стратегію у приватному полі _strategy, яке ініціалізується через конструктор відповідно до принципу впровадження залежностей. Scheduler також володіє таймером для періодичної активації процесу перевірки правил та використовує механізми синхронізації для запобігання одночасному виконанню декількох перевірок. Важливою особливістю є те, що Scheduler не знає про існування класу Trigger безпосередньо, працюючи лише з об'єктами Rule через обрану стратегію, що демонструє високий рівень інкапсуляції.

Клас Rule відіграє роль агрегатора, який володіє об'єктом Trigger через композицію. Кожне правило має власний тригер, що визначає умови його спрацювання. Метод CheckTrigger класу Rule делегує перевірку готовності внутрішньому тригеру, викликаючи його метод ShouldExecute. Така архітектура дозволяє змінювати логіку тригерів незалежно від логіки правил та стратегій перевірки, забезпечуючи гнучкість системи на кількох рівнях абстракції.

Метод CheckAllRules у класі Scheduler демонструє делегування відповідальності стратегії. Замість жорстко закодованої логіки перевірки правил, контекст просто викликає метод GetReadyRules обраної стратегії та для кожного отриманого правила генерує подію RuleReady. Такий підхід робить Scheduler незалежним від конкретної реалізації алгоритму перевірки та дозволяє легко розширювати систему новими стратегіями без модифікації існуючого коду контексту.

Фрагменти коду реалізації

Інтерфейс стратегії визначає контракт для всіх можливих алгоритмів перевірки правил:

```
namespace FlexibleAutomationTool.Core.Services
{
    public interface ITriggerStrategy
    {
        IEnumerable<Rule> GetReadyRules(IRuleRepository repository);
    }
}
```

Конкретна реалізація стратегії на основі polling демонструє алгоритм періодичної перевірки всіх правил у системі:

```
namespace FlexibleAutomationTool.Core.Services
{
    public class PollingTriggerStrategy : ITriggerStrategy
    {
        public IEnumerable<Rule> GetReadyRules(IRuleRepository repository)
        {
            foreach (var r in repository.GetAll())
```

```
            {
                if (r.CheckTrigger())
                    yield return r;
            }
        }
    }
}
```

Абстрактний клас Trigger представляє базову функціональність для всіх типів тригерів у системі:

```
namespace FlexibleAutomationTool.Core.Triggers
{
    public abstract class Trigger
    {
        public int Id { get; set; }
        public abstract bool ShouldExecute();
        public abstract bool Validate();
    }
}
```

Клас Rule інкапсулює тригер та надає метод для перевірки його готовності:

```
namespace FlexibleAutomationTool.Core.Models
{
    public class Rule
    {
        public int Id { get; set; }
        public string Name { get; set; } = string.Empty;
        public string? Description { get; set; }
        public FlexibleAutomationTool.Core.Triggers.Trigger Trigger { get; set; } =
null!;
        public FlexibleAutomationTool.Core.Actions.ActionBase Action { get; set; } =
null!;
        public bool IsActive { get; set; } = true;

        public bool CheckTrigger()
        {
            return Trigger?.ShouldExecute() ?? false;
        }

        public void Execute()
        {
            Action.Execute();
        }

        public override string ToString()
        {
            return string.IsNullOrWhiteSpace(Name) ? base.ToString() : Name;
        }
    }
}
```

Обґрунтування використання паттерну

Використання паттерну Стратегія в системі FlexibleAutomationTool є необхідним для підтримки різних підходів до визначення готовності правил автоматизації. У реальних сценаріях використання системи можуть знадобитися різні алгоритми перевірки, такі як періодичне опитування, реакція на події, отримання сповіщень від зовнішніх систем або гібридні підходи. Без застосування паттерну Стратегія доведеться використовувати умовні конструкції для вибору

потрібного алгоритму, що призведе до порушення принципу відкритості-закритості та ускладнить підтримку коду.

Патерн дозволяє дотримуватися принципу єдиної відповідальності, оскільки клас Scheduler відповідає лише за керування таймером та оркестрацію процесу перевірки правил, тоді як конкретні стратегії інкапсулюють логіку визначення готовності правил. Це робить код більш модульним та зрозумілим, адже кожен клас має чітко визначену область відповідальності.

Гнучкість системи також проявляється у можливості налаштування різних стратегій для різних середовищ виконання. У середовищі розробки може використовуватися стратегія з коротким інтервалом опитування для швидкого тестування, тоді як у продакшн-середовищі застосовується оптимізована стратегія з довшим інтервалом для зменшення навантаження на систему. При необхідності стратегію можна змінити навіть під час виконання програми без перезапуску системи.

Застосування паттерну Стратегія відповідає принципу відкритості-закритості з SOLID, оскільки система відкрита для розширення новими стратегіями, але закрита для модифікації існуючого коду контексту. Додавання нової стратегії вимагає лише створення класу, який реалізує інтерфейс ITriggerStrategy, без необхідності змінювати код Scheduler або інших компонентів системи. Це особливо важливо у контексті еволюції системи, коли з'являються нові вимоги до алгоритмів перевірки правил, наприклад, необхідність підтримки реактивних тригерів на основі подій або інтеграція з зовнішніми системами моніторингу.

Архітектурне рішення щодо інкапсуляції тригерів всередині класу Rule додатково підвищує гнучкість системи. Контекст Scheduler та стратегії ITriggerStrategy не мають прямого доступу до об'єктів Trigger, що дозволяє змінювати їх реалізацію незалежно від логіки перевірки правил. Це створює додатковий рівень абстракції, який захищає внутрішню структуру тригерів від зовнішніх змін та забезпечує стабільність інтерфейсів між компонентами системи.

Порівняно з альтернативними підходами, такими як використання switch-case для вибору алгоритму або жорстке кодування логіки безпосередньо у контексті Scheduler, патерн Стратегія забезпечує значно кращу архітектуру. Switch-case підхід вимагав би модифікації коду планувальника при додаванні кожної нової стратегії, а жорстке кодування унеможливлювало б гнучке налаштування системи під різні сценарії використання. Патерн Стратегія усуває ці недоліки, створюючи чисту архітектуру з чітким розділенням відповідальностей між компонентами системи, де кожен елемент патерну виконує свою специфічну роль: контекст координує виконання, інтерфейс стратегії визначає контракт, а конкретні реалізації інкапсулюють різні алгоритми.

Відповіді на контрольні запитання

1. Що таке шаблон проєктування?

Це формалізований опис часто повторюваної задачі проєктування та її вдалого рішення, яке можна багаторазово використовувати

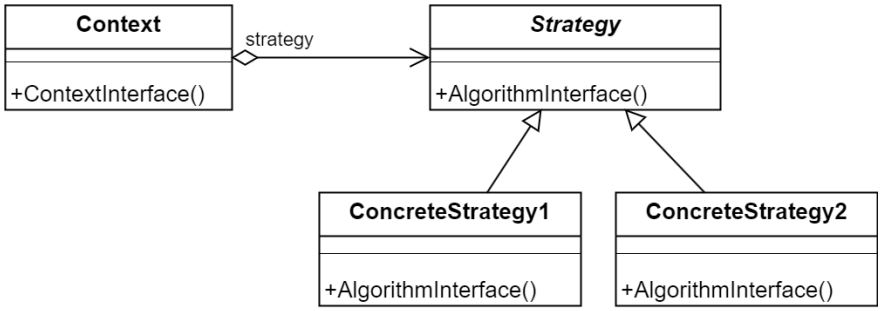
2. Навіщо використовувати шаблони проєктування?

Вони спрощують моделювання системи, покращують її архітектуру, підвищують стійкість до змін, полегшують доопрацювання й забезпечують єдиний «словник» для розробників.

3. Яке призначення шаблону «Стратегія»?

Відокремлювати взаємозамінні алгоритми в окремі класи та дозволяти підміняти їх під час виконання.

4. Нарисуйте структуру шаблону «Стратегія».



5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

Strategy – інтерфейс алгоритму.

ConcreteStrategyA/B – конкретні алгоритми.

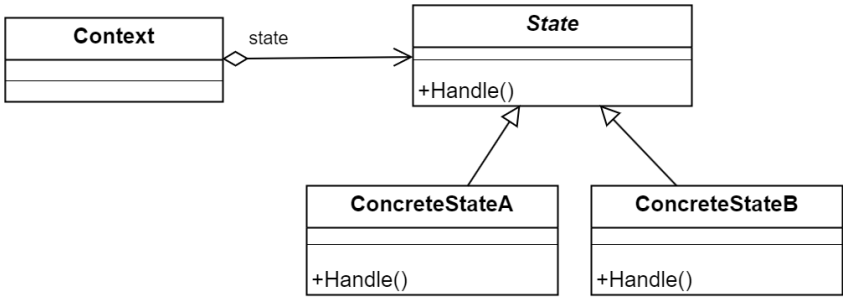
Context – містить посилання на стратегію та викликає її методи.

Context делегує виконання вибраній стратегії.

6. Яке призначення шаблону «Стан»?

Змінювати поведінку об’єкта при зміні його внутрішнього стану шляхом делегування роботи окремим класам-станам.

7. Нарисуйте структуру шаблону «Стан».



8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

State – інтерфейс стану.

ConcreteStateA/B – реалізації окремих станів.

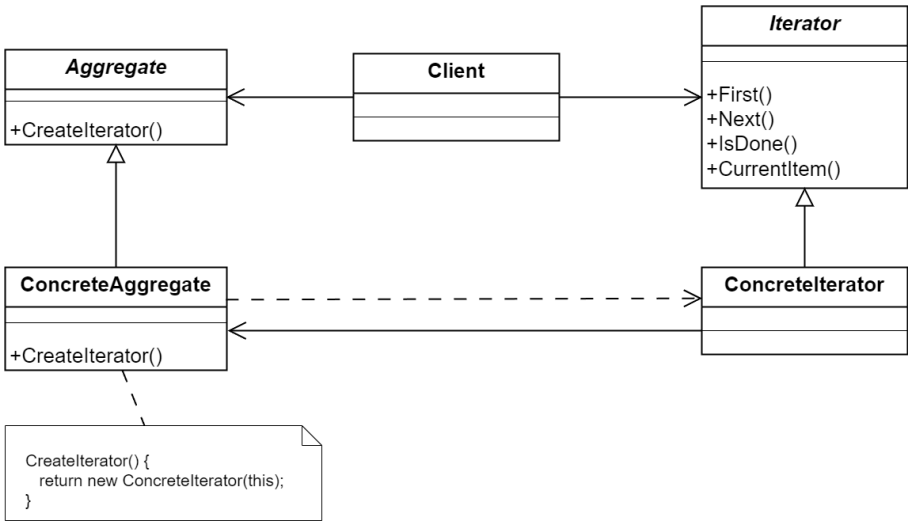
Context – містить об’єкт стану та делегує йому роботу.

Зміна стану відбувається заміною об’єкта в полі state.

9. Яке призначення шаблону «Ітератор»?

Забезпечити послідовний доступ до елементів колекції без розкриття її внутрішньої реалізації та винести логіку обходу в окремий клас.

10. Нарисуйте структуру шаблону «Ітератор».



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

Iterator – інтерфейс ітератора (First, Next, IsDone, CurrentItem).

ConcreteIterator – конкретна реалізація.

Aggregate – інтерфейс колекції.

ConcreteAggregate – колекція, що створює свій ітератор.

Iterator обходить елементи, не змінюючи код колекції.

12. В чому полягає ідея шаблона «Одинак»?

Забезпечити існування лише одного (або обмеженої кількості) екземпляра класу та надати глобальну точку доступу до нього.

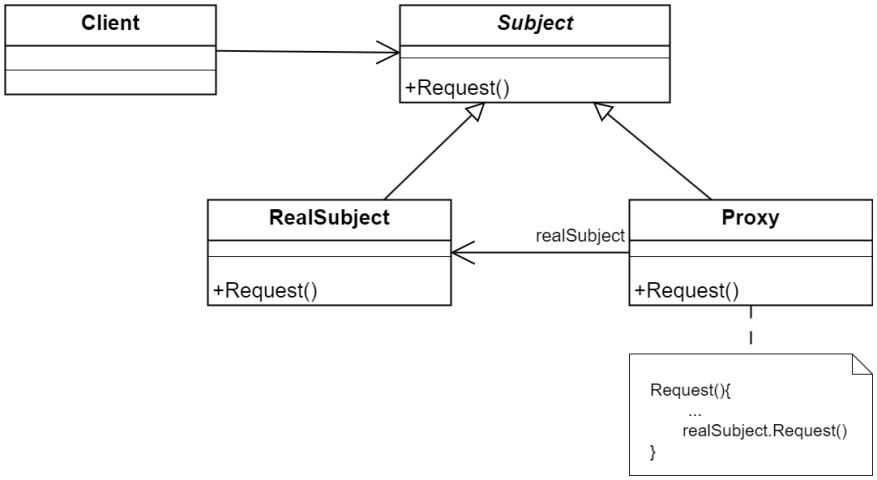
13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Бо він створює глобальні об’єкти зі станом, ускладнює тестування, порушує принцип єдиної відповідальності та маскує погану архітектуру.

14. Яке призначення шаблону «Проксі»?

Створити об’єкт-замісник, який контролює доступ до реального об’єкта та може додатково оптимізувати або обмежувати взаємодію з ним.

15. Нарисуйте структуру шаблону «Проксі».



16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

Subject – спільний інтерфейс.

RealSubject – реальний об’єкт.

Proxy – замісник, який містить посилання на RealSubject.

Client працює через Proxy, а Proxy керує доступом до RealSubject.

Висновки

У ході виконання лабораторної роботи розглянуди та реалізовали поведінковий патерн проєктування Стратегія в контексті системи автоматизації FlexibleAutomationTool. Застосування цього патерну дозволяє забезпечити гнучку й розширювану архітектуру, у якій алгоритми перевірки готовності правил ізольовані від логіки їх виконання. Реалізована структура з інтерфейсом ITriggerStrategy, конкретною стратегією PollingTriggerStrategy та контекстом Scheduler демонструє чітке розділення відповідальностей і можливість простого підключення нових алгоритмів без зміни існуючого коду.

Отримані результати підтверджують доцільність використання патерну Стратегія в системах, де необхідно підтримувати кілька варіантів алгоритмів і забезпечувати їхню взаємозамінність під час роботи програми. Архітектура, побудована на цьому підході, дотримується принципів SOLID, зокрема відкритості-закритості та єдиної відповідальності, що робить систему модульною, масштабованою й зручною в супроводі. Реалізація патерну в межах лабораторної роботи показує ефективність використання шаблонів проєктування для створення зручних, гнучких і надійних інструментів автоматизації.