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

Лабораторна робота №7
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 «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.

Теоретичні відомості

Шаблон «Mediator»

Шаблон «Mediator» призначений для інкапсуляції складної логіки взаємодії між багатьма об'єктами в одному спеціалізованому об'єкті – медіаторі, щоб компоненти не містили численних прямих посилань один на одного. Коли інтерфейс чи форма має велику кількість взаємопов'язаних контролів і зв'язки між ними стають заплутаними (наприклад: зміна стану одного контрола впливає на видимість і валідність інших), логіку координації переносять у медіатор; компоненти лише повідомляють медіатор про події або запитують у нього дії. Це зменшує зв'язаність, спрощує тестування і зміну взаємодій (можна підмінити медіатор тестовим), але ризик – медіатор із часом може сконцентрувати забагато логіки і перетворитися на важко підтримуваний «God Object».

Шаблон «Facade»

Шаблон «Facade» надає простий, уніфікований інтерфейс до складної підсистеми, ховаючи від клієнтів внутрішню структуру та деталі взаємодії класів. Якщо підсистема містить багато класів і варіантів конфігурації, фасад збирає часті або складні послідовності викликів у набір зрозумілих методів, спрощуючи використання і зменшуючи «спагеті» у клієнтському коді; це також дозволяє змінювати внутрішню реалізацію без масових модифікацій у користувачів. Фасад підвищує інкапсуляцію і зручність, але знижує гнучкість для клієнтів, яким може знадобитися доступ до більш тонкого функціоналу підсистеми.

Шаблон «Bridge»

Шаблон «Bridge» розділяє абстракцію і її реалізацію на дві незалежні ієрархії, дозволяючи змінювати їх окремо та уникнути експоненційного зростання комбінованих підкласів. Коли у вас є дві незалежні осі варіацій (наприклад: типи фігур і різні графічні драйвери), замість створення всіх комбінацій класів створюються абстракції, які делегують роботу реалізаторам (implementor). Це дає гнучкість у розширенні обох ієрархій без модифікації існуючих класів, але вводить додаткову індирекцію і архітектурну складність – більше класів та проміжних рівнів, які треба підтримувати.

Шаблон «Template method»

Шаблон «Template Method» визначає каркас алгоритму в базовому класі, делегуючи окремі кроки підкласам, які реалізують специфічну поведінку. Якщо кілька алгоритмів мають однакову загальну структуру, але відрізняються деталями виконання, базовий клас задає послідовність викликів, а підкласи перевизначають потрібні методи; це зменшує дублювання і робить додавання нових варіантів

простішим (достатньо створити новий підклас). Недоліком є жорсткий каркас алгоритму, ризик ускладнень при великій кількості hook-методів і потенційне порушення заміщення Лісков; якщо спільного коду мало, часто доцільніше використати Strategy.

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 реалізовали структурний патерн проектування «Фасад» (Facade) з метою спрощення взаємодії клієнтського коду з підсистемою автоматизації правил та інкапсуляції складної логіки внутрішніх компонентів. Патерн «Фасад» дозволяє створити єдиний точковий доступ до різноманітних служб, які забезпечують управління планувальником, виконання правил, логування та збереження історії виконаних дій, приховуючи від користувача деталі взаємодії між окремими сервісами. Центральним елементом архітектури є інтерфейс IAutomationFacade та його реалізація AutomationEngine, яка надає методи Start(), Stop(), GetHistory(), CreateRule(Rule) та CreateMacroRule(string, string), а також реалізує події RuleTriggered, RuleExecuted і RuleFailed. Клієнтський код (UI або MainForm) звертається саме до цього фасаду, викликаючи його методи та підписуючись на події, що дозволяє централізовано керувати всією системою автоматизації без необхідності знати деталі роботи окремих компонентів (рис. 1).

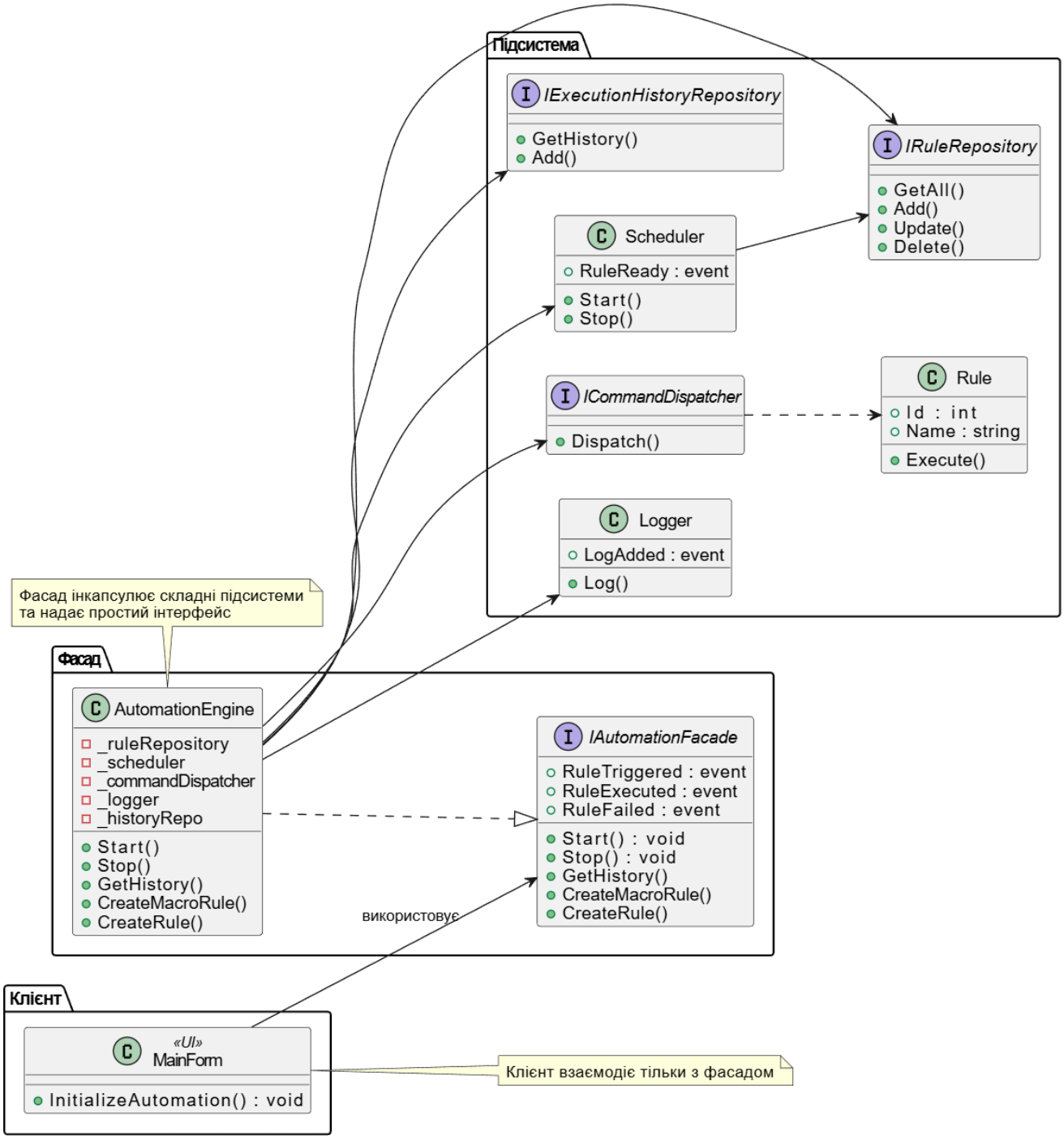


Рисунок 1. Діаграма класів паттерну Facade

AutomationEngine агрегує декілька внутрішніх сервісів: IRuleRepository для доступу до правил, Scheduler для планування та моніторингу готовності правил, ServiceManager для надання платформених сервісів, Logger для ведення логів, ICommandDispatcher для виконання правил, IInterpreter для інтерпретації дій та IExecutionHistoryRepository для збереження історії виконань. Фасад реагує на події Scheduler, перевіряє готовність правил, делегує виконання через CommandDispatcher та інжектує необхідні сервіси у об’єкти Action перед виконанням. При цьому всі внутрішні виклики компонентів, логування та оновлення історії відбуваються прозоро для користувача фасаду.

CommandDispatcher реалізує централізоване виконання правил, викликаючи метод Execute() об'єкта Rule, який у свою чергу делегує виконання конкретної дії класам-нащадкам ActionBase, таким як MessageBoxAction, RunProgramAction, OpenUrlAction, FileWriteAction чи MacroAction. Кожна дія інкапсулює власну логіку виконання та взаємодіє з платформеними сервісами, не потребуючи знань про виклик зі сторони фасаду.

Правила в системі представлені класом Rule, який містить властивості Id, Name, Description, Trigger, Action та IsActive, а також методи CheckTrigger() і Execute(). Тригери (TimeTrigger, EventTrigger, ManualTrigger) визначають умови спрацювання правила через метод ShouldExecute(), а виконувані дії інкапсулюються у спадкоємцях ActionBase. Додатково, AutomationEventHandler підписується на події фасаду та транслює їх у UI через StatusUpdated і LogEntryAdded, що забезпечує синхронізацію стану системи та інтерфейсу користувача. Подібна організація дозволяє клієнтському коду працювати з простим і зрозумілим інтерфейсом, не турбуючись про складну внутрішню взаємодію Scheduler, Logger, ICommandDispatcher, репозиторіїв та сервісів, що реалізують різні аспекти системи.

Використання патерну «Фасад» забезпечує високий рівень інкапсуляції внутрішньої підсистеми, спрощує інтерфейс взаємодії для клієнтського коду та зменшує залежності від деталей реалізації. Це дозволяє модифікувати внутрішні компоненти AutomationEngine, Scheduler, Logger чи репозиторії без потреби змінювати код клієнтів, а також знижує ймовірність виникнення «спагеті-коду». Водночас фасад концентрує управління підсистемою в одному місці, що полегшує підтримку та розширення функціональності, хоча дещо знижує гнучкість прямого налаштування окремих сервісів. Така структура робить систему FlexibleAutomationTool більш зрозумілою, надійною та готовою до масштабування складних сценаріїв автоматизації.

Фрагменти коду реалізації

Інтерфейс фасаду IAutomationFacade визначає контракт фасаду з методами та подіями:

```
namespace FlexibleAutomationTool.Core.Facades
{
    public interface IAutomationFacade
    {
        event Action<Rule>? RuleTriggered;
        event Action<Rule>? RuleExecuted;
        event Action<Rule, Exception>? RuleFailed;

        void Start();
        void Stop();

        IEnumerable<LogEntry> GetHistory();

        void CreateMacroRule(string name, string macroText);
        void CreateRule(Rule rule);
```

```
    }
}
```

Клас AutomationEngine, конструктор та ін'єкція залежностей: ініціалізація фасаду з усіма підсистемами:

```
private readonly IRuleRepository _repo;
private readonly Scheduler _scheduler;
private readonly ServiceManager _serviceManager;
private readonly Logger _logger;
private readonly IInterpreter _interpreter;
private readonly ICommandDispatcher _dispatcher;
private readonly Factories.IPlatformFactory _factory;
private readonly IExecutionHistoryRepository? _historyRepo;

public event Action<Rule>? RuleTriggered;
public event Action<Rule>? RuleExecuted;
public event Action<Rule, Exception>? RuleFailed;

public AutomationEngine(IRuleRepository repo, Scheduler scheduler, ServiceManager
svcManager, Logger logger, IInterpreter interpreter, ICommandDispatcher dispatcher,
FlexibleAutomationTool.Core.Factories.IPlatformFactory factory,
IExecutionHistoryRepository? historyRepo = null)
{
    _repo = repo;
    _scheduler = scheduler;
    _serviceManager = svcManager;
    _logger = logger;
    _interpreter = interpreter;
    _dispatcher = dispatcher;
    _factory = factory;
    _historyRepo = historyRepo;

    _scheduler.RuleReady += OnRuleReady;
}

private void InjectPlatformServicesIntoAction(Actions.ActionBase action)
{
    if (action == null) return;

    if (action is MessageBoxAction mba)
    {
        var svc = _factory.CreateMessageBoxService();
        if (svc != null)
            mba.MessageBoxService = svc;
    }

    if (action is MacroAction mac)
    {
        foreach (var child in mac.Actions)
        {
            InjectPlatformServicesIntoAction(child);
        }
    }
}
```

Обробник події OnRuleReady реалізує реакцію фасаду на подію від Scheduler та виклик CommandDispatcher:

```
private void OnRuleReady(Rule rule)
{
    try
    {
        RuleTriggered?.Invoke(rule);
        _logger.Log(rule.Name, "Rule triggered");
```

```
        _historyRepo?.Add(rule.Id, DateTime.UtcNow, "Triggered", "Rule triggered by
scheduler");

        InjectPlatformServicesIntoAction(rule.Action);
        _dispatcher.Dispatch(rule);

        RuleExecuted?.Invoke(rule);
        _logger.Log(rule.Name, "Rule executed");
        _historyRepo?.Add(rule.Id, DateTime.UtcNow, "Executed", "Rule executed
successfully");
    }
    catch (Exception ex)
    {
        RuleFailed?.Invoke(rule, ex);
        _logger.Log(rule.Name, $"Error executing rule: {ex}");
        _historyRepo?.Add(rule.Id, DateTime.UtcNow, "Failed", ex.ToString());
    }
}
```

Клієнтський код у MainForm використовує фасад без знання про внутрішні підсистеми:

```
public partial class MainForm : Form
{
    private readonly IAutomationFacade _facade;

    public MainForm(IAutomationFacade facade)
    {
        InitializeComponent();
        _facade = facade ?? throw new ArgumentNullException(nameof(facade));

        // Підписка на події фасаду — UI реагує на високорівневі події
        _facade.RuleTriggered += r => StatusLabel.Text = $"Triggered: {r.Name}";
        _facade.RuleExecuted  += r => StatusLabel.Text = $"Executed: {r.Name}";
        _facade.RuleFailed    += (r, ex) =>
        {
            StatusLabel.Text = $"Failed: {r.Name}";
            MessageBox.Show($"Rule {r.Name} failed: {ex.Message}", "Error");
        };

        // Запускаємо двигун через фасад
        _facade.Start();
    }

    // Виклик при додаванні правила з UI — фасад бере на себе збереження/логіку
    private void btnAddRule_Click(object sender, EventArgs e)
    {
        var newRule = new Rule
        {
            Name = "HelloRule",
            Trigger = new FlexibleAutomationTool.Core.Triggers.ManualTrigger(),
            Action = new
FlexibleAutomationTool.Core.Actions.InternalActions.MessageBoxAction()
            {
                Title = "Hi",
                Message = "Hello from facade"
            },
            IsActive = true
        };

        _facade.CreateRule(newRule); // клієнту не потрібно знати де і як зберігається
правило
    }

    // Показати історію через фасад (отримуємо тільки модель LogEntry)
```

```
    private void btnShowHistory_Click(object sender, EventArgs e)
    {
        var history = _facade.GetHistory();
        foreach (var entry in history)
        {
            listBoxHistory.Items.Add($"{entry.Timestamp:HH:mm} {entry.LoggedBy}:
{entry.Message}");
        }
    }

    protected override void OnFormClosing(FormClosingEventArgs e)
    {
        base.OnFormClosing(e);
        // Зупинка роботи через фасад
        _facade.Stop();
    }
}
```

Обґрунтування використання паттерну

Використання патерну «Facade» (фасад) у системі FlexibleAutomationTool є обґрунтованим з огляду на потребу надати клієнтам (зокрема UI) простий, стабільний і зрозумілий інтерфейс для управління складною підсистемою автоматизації. У архітектурі проєкту підсистема включає кілька взаємодіючих компонентів: сховище правил (IRuleRepository/SqliteRuleRepository), планувальник (Scheduler), стратегії тригерів, диспетчер команд (ICommandDispatcher/CommandDispatcher), інтерпретатор макросів, платформи сервісів (IPlatformFactory, IMessageBoxService тощо), логер та репозиторій історії виконань. Без фасаду клієнтський код повинен був би знати про конфігурацію і порядок взаємодії цих компонентів: створювати, підписуватися, кидати виклики і координувати роботу, що призводить до розрізнених точок залежності, дублювання логіки і «спагеті»-зв'язків між модулями.

Фасад реалізує єдину публічну точку доступу (впроваджений як інтерфейс IAutomationFacade і реалізований AutomationEngine), яка інкапсулює політику запуску/зупинки, створення правил, підписки на високорівневі події та отримання історії. Завдяки цьому UI-компоненти (наприклад, MainForm, CreateRuleForm, ViewHistoryForm) працюють виключно з набагато простішим API: викликають Start, Stop, CreateRule, CreateMacroRule, отримують GetHistory і підписуються на події RuleTriggered/RuleExecuted/RuleFailed. Такий підхід усуває потребу в знанні деталей реалізації підсистеми на стороні UI, що зменшує площу впливу змін у внутрішній реалізації та дозволяє розвивати внутрішні компоненти без руйнування клієнтського коду.

Патерн фасад також чітко розподіляє відповідальності: AutomationEngine координує взаємодію підсистем (перевірка тригерів, інжекція платформних сервісів у дії, делегування виконання диспетчеру, централізоване логування та запис в історію). Це гарантує, що політики, пов'язані з обробкою помилок, логуванням і життєвим циклом планувальника, реалізуються в одному місці, а не розпорошуються по UI або по різних службах. Така централізація підвищує

передбачуваність поведінки системи, полегшує відстеження причин помилок і скорочує дублювання коду, яке виникло б у випадку, коли б кожний клієнт самостійно координував низькорівневі служби.

З точки зору розширюваності і підтримуваності фасад добре поєднується з іншими застосованими патернами в проєкті, зокрема з патерном «Команда». Дії (ActionBase і його реалізації) залишаються інкапсульованими одиницями поведінки і можуть змінюватись, додаватись або комбінуватись (через MacroAction) без змін у клієнтському коді, оскільки фасад забезпечує лише координування їх виконання та інжекцію необхідних платформних сервісів. Інтерфейс фасаду дозволяє диспетчеру та логеру залишатися приватними деталями реалізації, а UI отримує тільки необхідні семантичні операції — це підтримує принципи інверсії залежностей і відкритості-закритості.

Практичні переваги застосування фасаду в цьому проєкті також стосуються конфігурації і тестування. Оскільки IAutomationFacade є інтерфейсом, його легко підмінити мок-реалізацією у юніт-тестах інтерфейсних компонентів UI; тестування UI не вимагає підняття реального шедулера або бази даних. У середовищі розгортання реалізацію фасаду можна конфігурувати через DI-контейнер (як в Program.cs), підключаючи SQLite або in-memory репозиторій, різні реалізації диспетчера чи фабрик сервісів без змін у коді викликачів. Це зменшує ризик регресій під час еволюції системи і спрощує процес CI/CD.

Водночас фасад має свої обмеження, які слід усвідомлювати та управляти ними. Наприклад, фасад може зменшувати гнучкість для клієнтів, яким потрібен тонкий контроль над внутрішніми механізмами (будь то спеціальна політика планування чи низькорівневі оптимізації виконання).

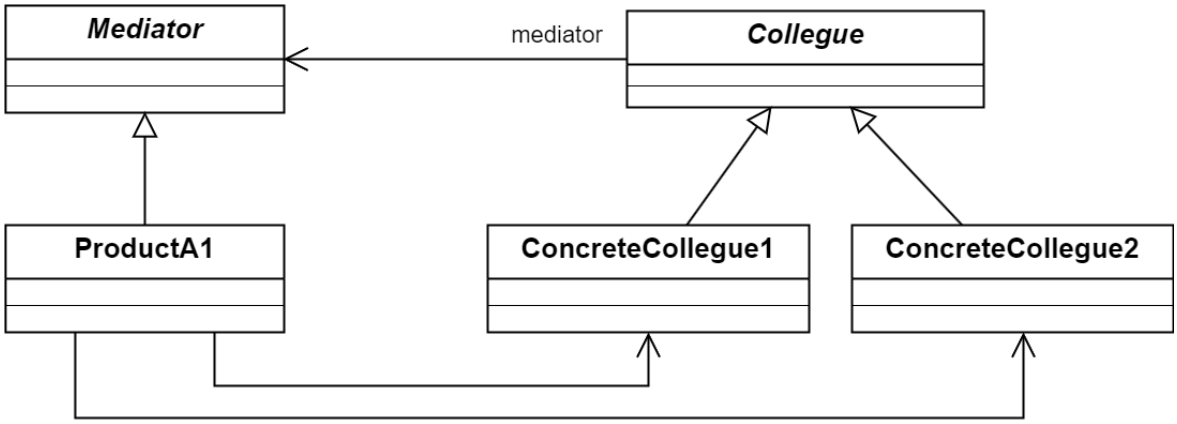
Підсумовуючи, застосування патерну «Facade» у FlexibleAutomationTool створює стабільну, зрозумілу і тестовану межу між UI-клієнтами та складною підсистемою автоматизації. Фасад зменшує зв’язаність, концентрує політики і логіку координації в одному місці, полегшує конфігурування через DI та сприяє ізольованому тестуванню. У поєднанні з уже використаними патернами (Команда, Стратегія, Абстрактна Фабрика) фасад підвищує модульність і підтримуваність системи, зберігаючи при цьому можливість еволюції внутрішньої реалізації без болісних змін у клієнтському коді.

Відповіді на контрольні запитання

1. Яке призначення шаблону «Посередник»?

Посередник централізує і інкапсулює логіку взаємодії між компонентами, зменшуючи прямі залежності між ними й підвищуючи зрозумілість коду.

2. Нарисуйте структуру шаблону «Посередник».



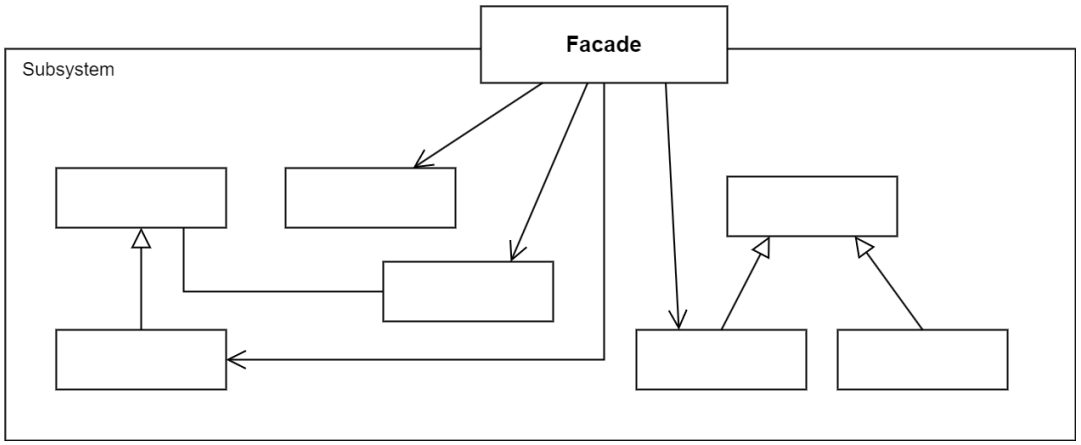
3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія?

Шаблон містить інтерфейс/абстракт Mediator, конкретний ConcreteMediator, абстракцію Colleague і конкретні ConcreteColleague; колеги повідомляють медіатору про події, а медіатор координує виклики між колегами.

4. Яке призначення шаблону «Фасад»?

Фасад дає простий уніфікований інтерфейс до складної підсистеми, приховуючи внутрішню реалізацію і спрощуючи використання.

5. Нарисуйте структуру шаблону «Фасад».



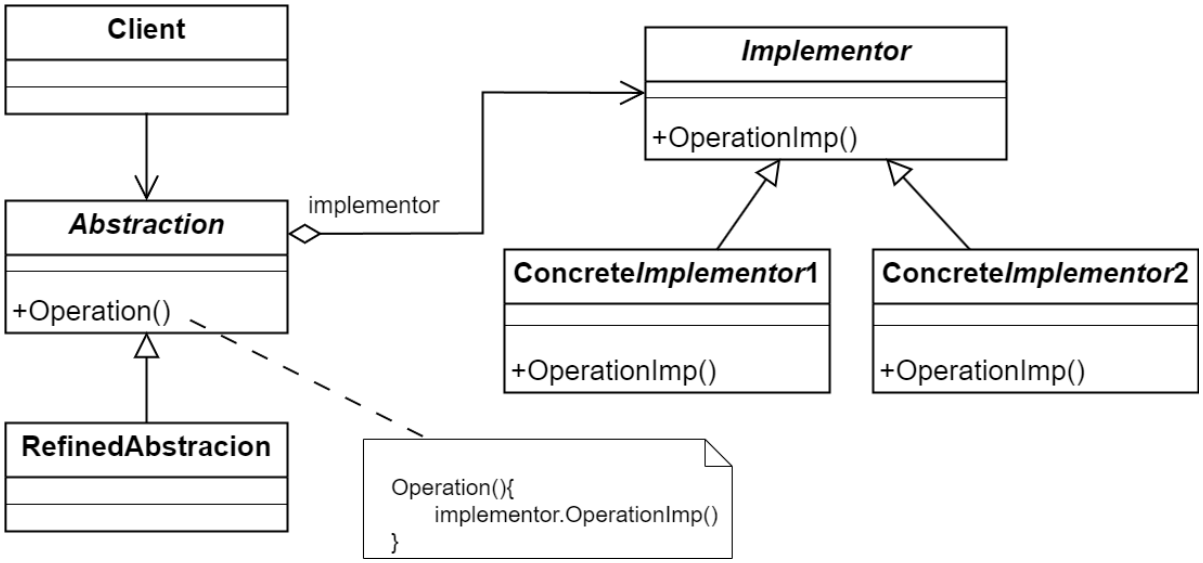
6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?

Є клас Facade і набір підсистемних класів (SubsystemA, SubsystemB тощо); клієнт викликає методи фасаду, а фасад координує виклики внутрішніх підсистем без розкриття їх деталей.

7. Яке призначення шаблону «Міст»?

Abstract Factory створює ціле сімейство пов’язаних об’єктів, а Factory Method створює один об’єкт певного типу.

8. Нарисуйте структуру шаблону «Міст».



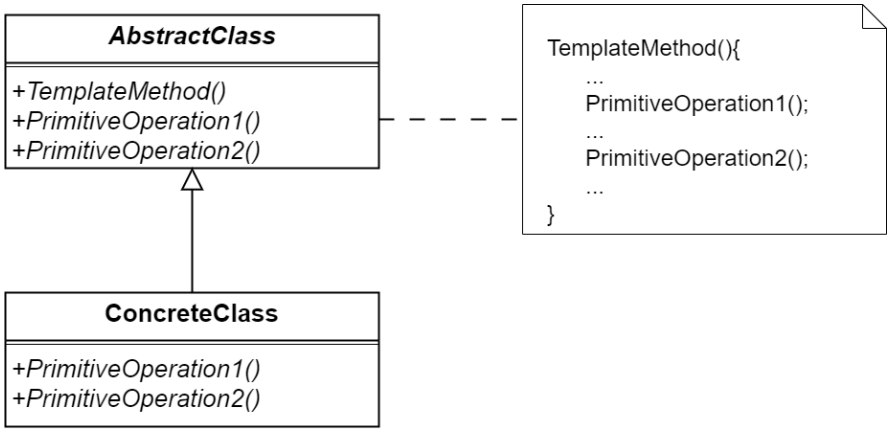
9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

Міст має Abstraction (і її розширення RefinedAbstraction), інтерфейс Implementor та конкретні ConcreteImplementor – абстракція делегує конкретну роботу реалізатору через інтерфейс, що дозволяє міняти обидві ієрархії незалежно.

10. Яке призначення шаблону «Шаблоний метод»?

Шаблонний метод визначає скелет алгоритму в базовому класі і делегує реалізацію окремих кроків підкласам, що дозволяє повторно використовувати загальну структуру алгоритму.

11. Нарисуйте структуру шаблону «Шаблоний метод»



12. Які класи входять в шаблон «Шаблоний метод», та яка між ними взаємодія?

Є AbstractClass, який реалізує templateMethod() і оголошує примітивні (віртуальні) кроки, та ConcreteClass(и), які перевизначають ці примітиви; при виклику templateMethod() базовий клас викликає перевизані кроки підкласів.

13. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

Шаблонний метод визначає послідовність кроків алгоритму і делегує деякі кроки підкласам, тоді як фабричний метод спеціалізується на делегуванні створення

об’єктів підкласам; Шаблонний метод про структуру алгоритму, фабричний метод – про створення екземплярів.

14. Яку функціональність додає шаблон «Міст»?

Міст дає можливість незалежно розширювати абстракцію і реалізацію, вибирати реалізатор в рантаймі та уникнути множення класів при комбінуванні різних абстракцій і реалізацій.

Висновки

У ході виконання лабораторної роботи реалізували та інтегрували структурний патерн проєктування «Фасад» у систему автоматизації FlexibleAutomationTool шляхом введення публічного інтерфейсу IAutomationFacade та його реалізації (центрального координатора) AutomationEngine. Запровадження фасаду дозволило надати клієнтським компонентам, зокрема інтерфейсу користувача (MainForm), простий та стабільний набір операцій (Start(), Stop(), GetHistory(), CreateRule(...) тощо), при цьому складна взаємодія між внутрішніми сервісами (IRuleRepository, Scheduler, CommandDispatcher, Logger, ServiceManager, репозиторій історії виконань та інтерпретатор макросів) була інкапсульована всередині фасаду.

Реалізація фасаду централізує управління життєвим циклом системи, підписки на події та процедурою виконання правил, включно з інжекцією платформених сервісів у об’єкти дій через ServiceManager і координованим логуванням через Logger. Такий підхід зменшує розпорошення логіки по проекту, усуває дублювання коду в UI і дозволяє забезпечити єдину політику обробки помилок і ведення історії виконань, що підвищує передбачуваність та надійність роботи підсистеми.

Водночас фасад концентрує координаційну логіку в одному місці, що іноді може зменшувати прямий контроль для споживачів, яким потрібен тонкий доступ до внутрішніх механізмів планування або виконання. Для пом’якшення цього обмеження доцільно передбачити розширювані точки конфігурації або додаткові інтерфейси для адміністративного або налагоджувального доступу, зберігаючи при цьому основну простоту API для більшості клієнтів.

Підсумовуючи, застосування патерну «Фасад» у FlexibleAutomationTool підвищило модульність, підтримуваність і тестованість системи, спростило взаємодію UI з підсистемою автоматизації та створило стійку основу для подальшого розширення функціональності при мінімальному впливі на клієнтський код.