Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №7

Технології розроблення програмного забезпечення

«Шаблони «Mediator», «Facade», «Bridge», «Template Method»»

Варіант 2

Виконав
студент групи ІА–33:
Перегуда П. О.

Перевірив
Мягкий М. Ю.

Київ 2025

ЗМІСТ

Тема: HTTP-сервер (state, builder, factory method, mediator, composite, p2p). Сервер повинен мати можливість розпізнавати вхідні запити і формувати коректні відповіді (згідно протоколу HTTP), надавати сторінки chtml (html сторінки з додаванням найпростіших C# конструкцій на розсуд студента), вести статистику вхідних запитів, обробку запитів у багатопотоковому/подієвому режимах.

Короткі теоретичні відомості

Шаблони проектування — це стандартизовані рішення для типових задач, які часто виникають під час розробки програмного забезпечення. Вони є своєрідними схемами, які допомагають ефективно вирішувати проблеми, роблячи код гнучкішим, розширюваним та зрозумілим.

Шаблони, розглянуті у цій лабораторній роботі, дозволяють ефективно вирішувати типові проблеми розробки програмного забезпечення, дотримуючись принципів, які забезпечують простоту, структурованість та гнучкість коду. Їх основою є ключові програмні філософії:

- Принцип DRY (Don’t Repeat Yourself) спрямований на усунення повторень у коді, що дозволяє уникати дублювання функціональності. Це сприяє легкості підтримки, знижує ризик помилок і полегшує масштабування, оскільки зміни в одному місці автоматично поширюються на всю систему. У контексті шаблонів цей принцип реалізується через повторне використання загальних структур, наприклад, фасаду чи посередника.
- Принцип KISS (Keep It Simple, Stupid!) наголошує на необхідності створення простих і зрозумілих конструкцій. У складних системах кожен компонент має виконувати лише одну конкретну функцію, що полегшує розуміння та підтримку. Шаблони, як-от "Міст" або "Фасад",

ілюструють це, дозволяючи приховати складність через спрощені інтерфейси або розділення абстракції та реалізації.

- Принцип YOLO (You Only Load It Once!) допомагає оптимізувати продуктивність системи, мінімізуючи повторні звернення до ресурсів. Наприклад, при використанні шаблону "Фасад" можна створити єдиний об'єкт, який відповідатиме за ініціалізацію даних, забезпечуючи ефективність виконання запитів.
- Принцип Парето (80/20) нагадує про важливість фокусування на ключових аспектах системи. Наприклад, при реалізації шаблону "Посередник" слід приділити основну увагу найбільш критичним комунікаціям між об'єктами, які забезпечують основну функціональність програми.
- Принцип YAGNI (You Ain’t Gonna Need It) закликає уникати зайвої 4 складності, створюючи лише те, що необхідно в поточний момент. Це проявляється в реалізації шаблону "Шаблонний метод", де забезпечується базова структура алгоритму, а додаткові деталі вводяться лише тоді, коли вони дійсно потрібні.

Шаблон "Посередник" (Mediator) вирішує проблему хаотичних зв'язків між об'єктами в програмній системі. Коли кожен компонент напряму взаємодіє з іншими, виникає сильна залежність між ними, що ускладнює підтримку та розширення системи. "Посередник" дозволяє уникнути цієї плутанини, вводячи єдиний центральний об'єкт, через який координується вся взаємодія. Об'єкти, які використовують посередника, називаються колегами; вони більше не звертаються напряму одне до одного, а передають запити через посередника. Таким чином, кожен компонент знає лише про існування цього посередника, що значно знижує зв'язаність системи. Проте, хоча це спрощує логіку взаємодії компонентів, складність може перенестися в самого посередника, якщо він починає координувати надто багато процесів.

Шаблон "Фасад" (Facade) створений для того, щоб приховувати складність програмних підсистем за простим і зрозумілим інтерфейсом. Уявіть систему з численними класами та складною структурою, які потрібно використовувати для виконання певного завдання. Фасад пропонує єдиний вхідний пункт для доступу до цієї системи, спрощуючи її використання. Він слугує своєрідним "обличчям" системи, яке відображає тільки те, що потрібно користувачам, приховуючи зайві деталі. Однак, важливо пам'ятати, що фасад сам по собі не додає нових функцій; він лише спрощує доступ до вже існуючої функціональності.

Шаблон "Міст" (Bridge) розв'язує проблему залежності між абстракцією та її реалізацією. Уявіть, що в системі є кілька абстракцій, кожна з яких може мати різні реалізації. Якщо ці аспекти не розділені, кожна нова абстракція або реалізація потребуватиме значної переробки існуючого коду. "Міст" розділяє ці два рівні, дозволяючи їм розвиватися незалежно. Реалізація цього шаблону включає два окремих ієрархічних дерева: одне для абстракції, а інше для реалізації. Абстракція делегує виклики своїй реалізації, що дозволяє легко змінювати реалізацію без необхідності переписувати 5 клієнтський код. Це забезпечує велику гнучкість, хоча й додає певної складності на етапі проектування.

Шаблон "Шаблонний метод" (Template Method) допомагає структурувати алгоритми таким чином, щоб загальна логіка залишалася незмінною, а конкретні деталі могли змінюватися підкласами. Він визначає основу алгоритму у вигляді кроків у базовому класі, залишаючи можливість підкласам уточнювати чи перевизначати ці кроки. Це дозволяє забезпечити єдність структури алгоритму для всіх підкласів, зберігаючи при цьому їхню гнучкість у реалізації окремих частин. Наприклад, уявіть алгоритм обробки документа: спільні кроки, як зчитування та збереження, визначаються в базовому класі, а специфічні деталі обробки документа — у підкласах.

Шаблонний метод гарантує послідовність виконання основних кроків, водночас дозволяючи змінювати їхню реалізацію там, де це необхідно.

Хід роботи

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.
4. Скласти звіт про виконану роботу.

Реалізація шаблону Mediator

Mediator (посередник) — це поведінковий патерн, який полегшує комунікацію між компонентами системи завдяки переміщенню зв'язків між різними класами-колегами до одного класу-посередника.

Реалізація шаблону “Mediator” у моєму проєкті полягає у створенні інтерфейсу **IMediator** з єдиним інтерфейсним методом **void Notify(object sender, object target)** та класу-посередника **MediatorProcessing**, що реалізує даний інтерфейс.

```
namespace HttpServApp.Mediator
{
  /// <summary>
  /// Інтерфейс посередника обробки
  /// </summary>
  internal interface IMediator
  {
    // Інтерфейс Посередника надає метод, що використовується компонентами для
    // сповіщення посередника про різні події. Посередник може реагувати
    // на ці події і передавати виконання іншим компонентам.
    public void Notify(object sender, object target);
  }
}
```

Рис. 1 - Інтерфейс IMediator

Призначення класу **MediatorProcessing** - обробка повідомлень (сповіщень) від компонентів, що є членами класу, і організація їх взаємодії. Якщо в одному з компонентів відбувається важлива подія, він сповіщає свого посередника, який вирішує — чи стосується подія інших компонентів та яких саме. При цьому компонент-відправник не знає, хто обробить його запит, а компонент-одержувач не знає, хто його надіслав.

Наприклад, при надходженні запиту від клієнта, об’єкт класу Listener сповіщає про це медіатор, на який зберігає посилання всередині. Для цього він викликає метод **Notify**, куди передає власне посилання на себе та відповідний сокет - об’єкт вхідного підключення.

```
while (isRunning)
{
  // Очікуємо спробу з'єднання,
  // після з'єднання створюється новий сокет для його обробки (вхідне підключення)
  Socket responseSocket = listenSocket.Accept();
  Console.WriteLine($"\n================ Адреса підключеного клієнта:
{responseSocket.RemoteEndPoint}");
  if (responseSocket.Connected)
  {
    // Відправка сповіщення медіатору про надходження нового запиту від клієнта
    Mediator?.Notify(this, responseSocket);
  }
}
```

Рис. 2 - Приклад надсилання повідомлення медіатору

Медіатор, отримуючи повідомлення, в методі **Notify** аналізує, хто з компонентів-колег надіслав повідомлення. У випадку, якщо це об’єкт класу Listener, медіатор зчитує конфігурацію застосунку в частині визначення режима обробки повідомлень: багатопотоковий чи подієвий. У залежності від цього викликається:

- метод **Process** класу **MultiThreadProcessing** (для багатопотокового режиму, в якому створюється окремий потік для виконання обробки запиту);
- метод **Process** класу **SingleThreadProcessing** (для обробки запиту в основному потоці, що забезпечує послідовну обробку запитів, що надходять від клієнтів).

Таким чином, завдяки використанню патерна програмування Mediator, вдалось позбавитись залежності між класом прийому повідомлень та базовим класом обробки повідомлень **RequestProcessing**, забезпечуючи вибір режиму обробки запитів всередині медіатора.

Крім того, у методі **Notify** відбувається також вивід на консоль (у подальшому, можливо застосування Logger для виводу інформації у файл) усіх повідомлень, що надходять від компонента Repository (дії з колекцією запитів).

Реалізацію класів MediatorProcessing, MultiThreadProcessing, SingleThreadProcessing наведено на рис. 3-5.

```
using HttpServApp.Models;
using HttpServApp.Processing;
using System.Net.Sockets;

namespace HttpServApp.Mediator
{
  /// <summary>
  /// Посередник обробки запиту від Http-клієнта
  /// </summary>
  internal class MediatorProcessing: IMediator
  {
    private readonly Listener listener;
    private readonly Repository repository;
    private readonly MultiThreadProcessing multiThreadProcessing;
    private readonly SingleThreadProcessing singleThreadProcessing;

    /// <summary>
    /// Конструктор об'єкта, у параметрах задані всі об'єкти-колеги, що взаємодіють
з медіатором
  /// </summary>
  /// <param name="listener"></param>
  /// <param name="repository"></param>
  /// <param name="multiThreadProcessing"></param>
  /// <param name="singleThreadProcessing"></param>
    public MediatorProcessing(Listener listener, Repository repository,
      MultiThreadProcessing multiThreadProcessing,
      SingleThreadProcessing singleThreadProcessing)
    {
      this.listener = listener;
      this.listener.Mediator = this;

      this.repository = repository;
      this.repository.Mediator = this;

      this.multiThreadProcessing = multiThreadProcessing;
      this.multiThreadProcessing.Mediator = this;

      this.singleThreadProcessing = singleThreadProcessing;
      this.singleThreadProcessing.Mediator = this;
    }

    /// <summary>
```

```
    /// Метод, що використовується компонентами для сповіщення посередника про різні
події.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="target"></param>
    public void Notify(object sender, object target)
    {
      // Якщо повідомлення надійшло від Listener
      if (sender is Listener)
      {
        // як цільовий об'єкт переданий Socket
        if (target is Socket clientSocket)
        {
          // Зчитуємо конфігурацію застосунку.
          // Якщо параметр багатопотоковості встановлений, то для обробки запиту
використовуємо об'єкт типу MultiThreadProcessing,
          // що запускає окремий потік обробки запиту
          if (Configuration.MultiThread)
            multiThreadProcessing.Process(repository, clientSocket);
          // Якщо параметр багатопотоковості НЕ встановлений, то для обробки запиту
використовуємо об'єкт типу SingleThreadProcessing,
          // що виконує обробку запиту в основному потоці
          else
            singleThreadProcessing.Process(repository, clientSocket);
        }
        else
        {
          Console.WriteLine(target.ToString());
        }

      }
      // Якщо сповіщення прийшло від Repository, просто виводимо цільове
повідомлення
      else if (sender is Repository)
      {
          Console.WriteLine(target.ToString());
      }
    }
  }
}
```

Рис. 3 - Клас MediatorProcessing

```
using HttpServApp.Models;
using System.Net.Sockets;

namespace HttpServApp.Processing
{
  /// <summary>
  /// Клас обробки запиту в багатопотоковому режимі
```

```
  /// </summary>
  internal class MultiThreadProcessing: RequestProcessing
  {
    public void Process(Repository repository, Socket clientSocket)
    {
      // Створення потоку обробки даних
      Thread workThread = new Thread(args => DoWork(args as ProcessingArgs));
      // Старт потоку з передачою параметрів
      workThread.Start(
        new ProcessingArgs()
        {
          Repository = repository,
          Socket = clientSocket
        });
    }
  }
}
```

Рис. 4 - Клас MultiThreadProcessing

```
using HttpServApp.Models;
using System.Net.Sockets;

namespace HttpServApp.Processing
{
  /// <summary>
  /// Клас обробки запиту в однопотоковому режимі (в основному потоці)
  /// </summary>
  internal class SingleThreadProcessing: RequestProcessing
  {
    public void Process(Repository repository, Socket clientSocket)
    {
      // Запуск методу обробки запиту
      DoWork(new ProcessingArgs()
      {
        Repository = repository,
        Socket = clientSocket
      });
    }
  }
}
```

Рис. 5 - Клас SingleThreadProcessing

Загальний вигляд структури класів та їх взаємозв’язків наведений на діаграмі класів (рис. 6).

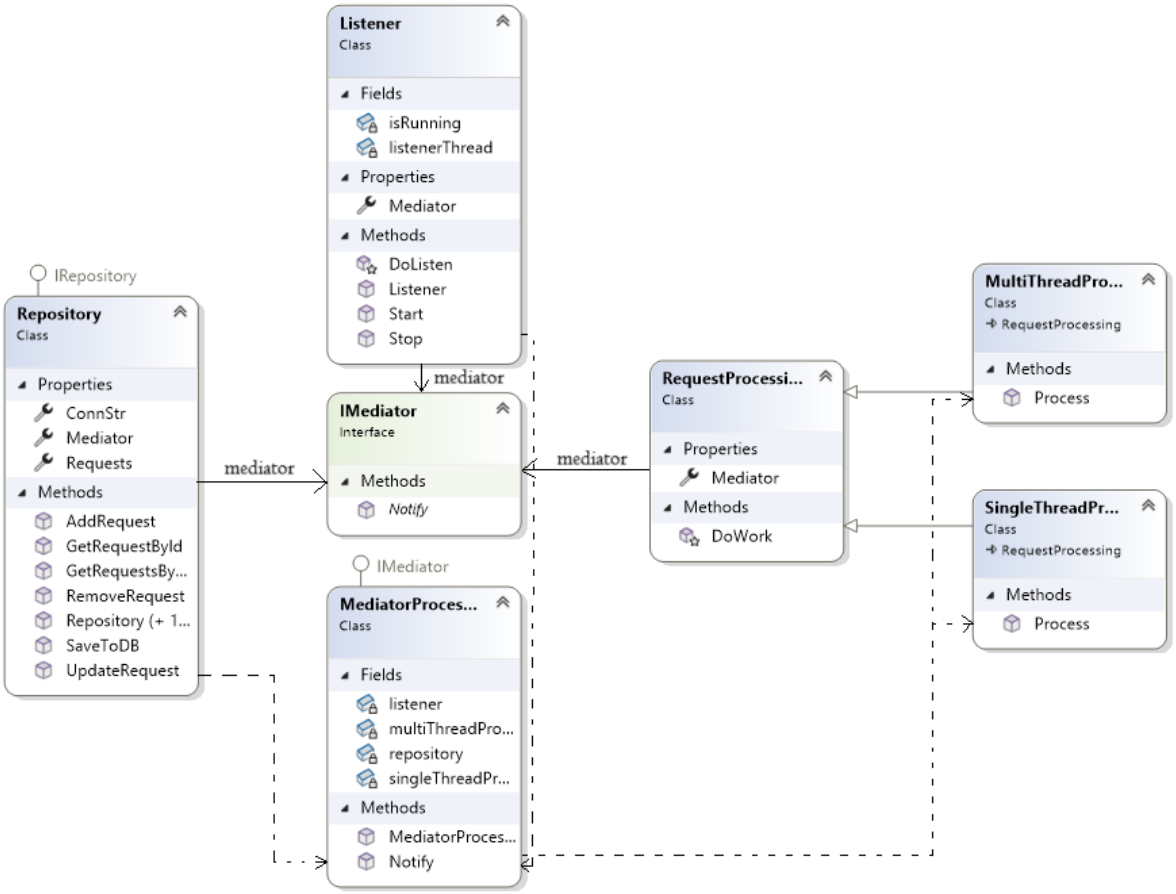

Рис. 6 - Діаграма класів

Приклад журналу послідовної обробки запитів в однопоточному (подієвому) режимі наведений на рис. 7

Рис.7 Журнал послідовної обробки запитів

Висновок: у ході виконання даної лабораторної роботи я реалізував шаблон Mediator - поведінковий патерн, який централізує спілкування між окремими компонентами системи. Використання даного шаблону дозволило мені усунути залежність між компонентами, спростивши взаємодію між ними та

збільшивши гнучкість системи, а також централізувати керування процесом обробки повідомлень в одному місці.