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 №6

Технології розроблення програмного забезпечення

«Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»»

Варіант 2

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

Перевірив
Мягкий М. Ю.

Київ 2025

ЗМІСТ

Тема 3
Короткі теоретичні відомості 3
Хід роботи 5
Завдання 5
Реалізація шаблону Factory 5
Висновок 11

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 з даної лабораторної роботи допомагають вирішувати типові задачі у розробці програмного забезпечення, забезпечуючи структурованість, масштабованість та легкість підтримки. Основою їх застосування є принципи SOLID:

- Single Responsibility Principle: клас має виконувати лише одну логічну задачу, спрощуючи зміну або розширення його функціональності.
- Open-Closed Principle: класи повинні бути відкритими для розширення, але закритими для змін, тобто нову поведінку додають через спадкування чи композицію.
- Liskov Substitution Principle: підкласи мають коректно працювати у будь-якому контексті, де використовується базовий клас.
- Interface Segregation Principle: інтерфейси слід розбивати на дрібні, вузько спрямовані частини, щоб уникати надлишкової реалізації.
- Dependency Inversion Principle: модулі вищого рівня не повинні залежати від модулів нижчого рівня, обидва мають залежати від абстракцій.

Розуміння цих принципів є основою правильного застосування шаблонів і створення якісного програмного забезпечення.

Шаблон Abstract Factory дозволяє створювати цілі сімейства взаємопов'язаних об'єктів без вказання їхніх конкретних класів. Його основною ідеєю є надання інтерфейсу, який приховує процес створення об'єктів і робить систему гнучкішою до змін. Наприклад, цей шаблон можна використовувати для розробки графічного інтерфейсу, який має підтримувати 4 різні операційні системи. У цьому випадку фабрика створює набори компонентів, таких як кнопки чи текстові поля, які виглядають і поводяться відповідно до конкретної платформи. Ця гнучкість дозволяє легко замінювати реалізацію без змін у бізнес-логіці програми.

Factory Method забезпечує можливість створювати об'єкти через виклики фабричних методів, які визначаються у підкласах. Це дозволяє класам делегувати створення об'єктів, не прив'язуючись до конкретних реалізацій. Наприклад, у додатку для роботи з повідомленнями можуть бути різні типи повідомлень, такі як SMS, електронна пошта або push-сповіщення. Factory Method дозволяє створювати ці повідомлення залежно від контексту, не змінюючи клієнтський код. Це значно полегшує додавання нових типів повідомлень і робить код більш масштабованим.

Шаблон Memento використовується для збереження стану об'єкта, щоб його можна було відновити без порушення інкапсуляції. Його основна ідея полягає в тому, щоб створити "знімок" поточного стану об'єкта, який може бути збережений і використаний пізніше для відновлення. Наприклад, у текстових редакторах реалізується функціональність скасування і повтору дій за допомогою цього шаблону. Стан документа зберігається перед виконанням змін, і якщо користувач вирішить скасувати дію, система повертає попередній стан. Цей підхід дозволяє легко управляти змінами без змішування логіки управління станом із бізнес-логікою.

Observer, або "Спостерігач", дозволяє об'єкту-спостерігачу автоматично отримувати повідомлення про зміну стану іншого об'єкта. Це забезпечує динамічну залежність між об'єктами, дозволяючи їм взаємодіяти без жорсткого зв'язування. Наприклад, цей шаблон використовується у додатках новин, де підписники автоматично отримують оновлення, коли з’являється новина. Суб'єкт, у даному випадку сервер новин, повідомляє всіх підписників про зміни. Такий підхід значно спрощує процес синхронізації та забезпечує 5 легке додавання або видалення спостерігачів.

Decorator дозволяє динамічно додавати нову поведінку об'єктам, зберігаючи їхню структуру незмінною. Це досягається шляхом обгортання об'єкта в інший клас, який реалізує додаткову функціональність. Наприклад, у графічних редакторах базовий об'єкт зображення може бути обгорнутий декоратором, який додає ефект тіні, рамки чи іншого візуального елементу. Decorator особливо корисний, коли потрібно розширити функціональність, уникаючи створення великої кількості підкласів для кожного можливого поєднання функцій.

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 Factory

Factory (фабрика) — це один з найпоширеніших патернів у програмуванні, основна мета якого - забезпечити створення об'єктів без явного вказування класу, що створюється.

Реалізація шаблону “Factory” у моєму проєкті полягає у створенні запиту відповідного типу від клієнта до Http-сервера. З цією метою мною було створено інтерфейс ICreator, який визначає “фабричний” метод FactoryMethod. Цей метод є зачіпкою для впровадження власного конструктора об'єктів. У моєму випадку він повертає “комбінований” об’єкт tuple, складовими частинами якого є об’єкти Http-запиту (HttpRequest) та початкового стану об’єкту запиту (IState).

```
1   using HttpServApp.Models;
2   using HttpServApp.Processing;
3   using HttpServApp.State;
4
5   namespace HttpServApp.Fabric
6   {
7       /// <summary>
8       /// Інтерфейс ICreator визначає методи
9       /// створення об'єкту запиту та його початкового стану
10      /// </summary>
        4 references
11      internal interface ICreator
12      {
13          /// <summary>
14          /// Створення об'єкту-кортежу (tuple) запиту та його початкового стану
15          /// </summary>
16          /// <returns></returns>
            4 references
17          public (HttpRequest, IState) FactoryMethod(Validator validator, Repository repository);
18
19      }
20  }
```

Рис. 1 - Інтерфейс ICreator

Для реалізації інтерфейса ICreator було створено наступні класи, що створюють відповідні класи дочірніх типів від HttpRequest та їх початкових станів IState:

- CreatorRequestPage - створює об’єкт запиту Web-сторінки та його початкового стану;

- CreatorRequestStat - створює об’єкт запиту статистики та його початкового стану;
- CreatorRequestInvalid - створює об’єкт невалідного запиту та його початкового стану.

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

namespace HttpServApp.Fabric
{
  internal class CreatorRequestPage : ICreator
  {
    /// <summary>
    /// Повертає об'єкт запиту Web-сторінки
    /// </summary>
    /// <param name="validator"></param>
    /// <param name="repository"></param>
    /// <returns></returns>
    public (HttpRequest, IState) FactoryMethod(Validator validator, Repository
repository)
    {
      try
      {
        HttpRequestPage httpRequest = new HttpRequestPage(
            repository, DateTime.Now,
            validator.GetVersionRequest(), validator.GetMethodRequest(),
            validator.GetRemoteEndPoint(), validator.GetContentTypeRequest(),
            Path.Combine(Configuration.ResourcePath ?? "C:\\",
validator.GetFileRequest()));

        Console.WriteLine($"Processing: запит сторінки {httpRequest.Path}!");

        return (httpRequest, new ValidatePageState());
      }
      catch (WebException webE)
      {
        Console.WriteLine($"CreatorRequestPage WebException: {webE.Message}
статус={webE.Status}");
        throw;
      }
      catch (Exception exc)
      {
        Console.WriteLine($"CreatorRequestPage Exception: {exc.Message}");
```

```
        throw;
      }
    }

  }
}
```

Рис. 2 - Клас CreatorRequestPage

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

namespace HttpServApp.Fabric
{
  internal class CreatorRequestStat : ICreator
  {

    /// <summary>
    /// Повертає об'єкт запиту статистики за період
    /// </summary>
    /// <param name="validator"></param>
    /// <param name="repository"></param>
    /// <returns></returns>
    public (HttpRequest, IState) FactoryMethod(Validator validator, Repository
repository)
    {
      try
      {
        HttpRequestStat httpRequest = new HttpRequestStat(
            repository, DateTime.Now,
            validator.GetVersionRequest(), validator.GetMethodRequest(),
            validator.GetRemoteEndPoint(), validator.GetContentTypeRequest(),
            validator.GetDateBegRequest(), validator.GetDateEndRequest(),
            validator.GetKeyAuthorization());
        Console.WriteLine($"Processing: запит статистики за період " +
             $"{httpRequest.DateBeg}-{httpRequest.DateEnd}!");

        return (httpRequest, new ValidateStatisticState());

      }
      catch (WebException webE)
      {
        Console.WriteLine($"CreateRequest WebException: {webE.Message}
статус={webE.Status}");
        throw;
      }
```

```
      catch (Exception exc)
      {
        Console.WriteLine($"CreateRequest Exception: {exc.Message}");
        throw;
      }
    }

  }
}
```

Рис. 3 - Клас CreatorRequestStat

```
using HttpServApp.Models;
using HttpServApp.Processing;
using HttpServApp.State;

namespace HttpServApp.Fabric
{
  internal class CreatorRequestInvalid : ICreator
  {
    /// <summary>
    /// Повертає об'єкт не валідного запиту
    /// </summary>
    /// <param name="validator"></param>
    /// <param name="repository"></param>
    /// <returns></returns>
    public (HttpRequest, IState) FactoryMethod(Validator validator, Repository
repository)
    {
      HttpRequestInvalid httpRequest = new HttpRequestInvalid(
        repository, DateTime.Now,
        validator.GetRemoteEndPoint(), "Невизначений тип запиту.");
      Console.WriteLine("Processing: Невизначений тип запиту!");

      return (httpRequest, new InvalidState());
    }
  }
}
```

Рис. 4 - Клас CreatorRequestInvalid

Шаблон Factory дозволив мені ізолювати код, який відповідає за створення об'єктів, від решти програмного коду. Замість безпосереднього створення відповідних дочірніх об’єктів Http-запитів, створюються об’єкти фабрики,

що імплементують загальний інтерфейс, потім викликається фабричний метод для створення об’єкту Http-запиту.

```
50              // Визначаємо тип запиту
51              string typeRequest = validator.GetTypeRequest();
52
53              switch (typeRequest)
54              {
55                  // Запит сторінки
56                  case "page":
57                      {
58                          creator = new CreatorRequestPage();
59                          break;
60                      }
61                  // Запит статистики
62                  case "stat":
63                      {
64                          creator = new CreatorRequestStat();
65                          break;
66                      }
67                  default:
68                      throw new WebException("Невизначений тип запиту", WebExceptionStatus.ProtocolError);
69
70              }
71          }
72          catch (WebException webExc)
73          {
74              // Запит favicon ігноруємо
75              if (webExc.Status == WebExceptionStatus.ReceiveFailure)
76                  return;
77              creator = new CreatorRequestInvalid();
78          }
79
80          if (creator != null)
81          {
82              (HttpRequest httpRequest, IState startState) = creator.FactoryMethod(validator, repository);
83              httpRequest.TransitionTo(startState, socket);
84          }
```

Рис. 5 - Приклад створення об’єктів дочірніх типів від HttpRequest за допомогою фабричного методу

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

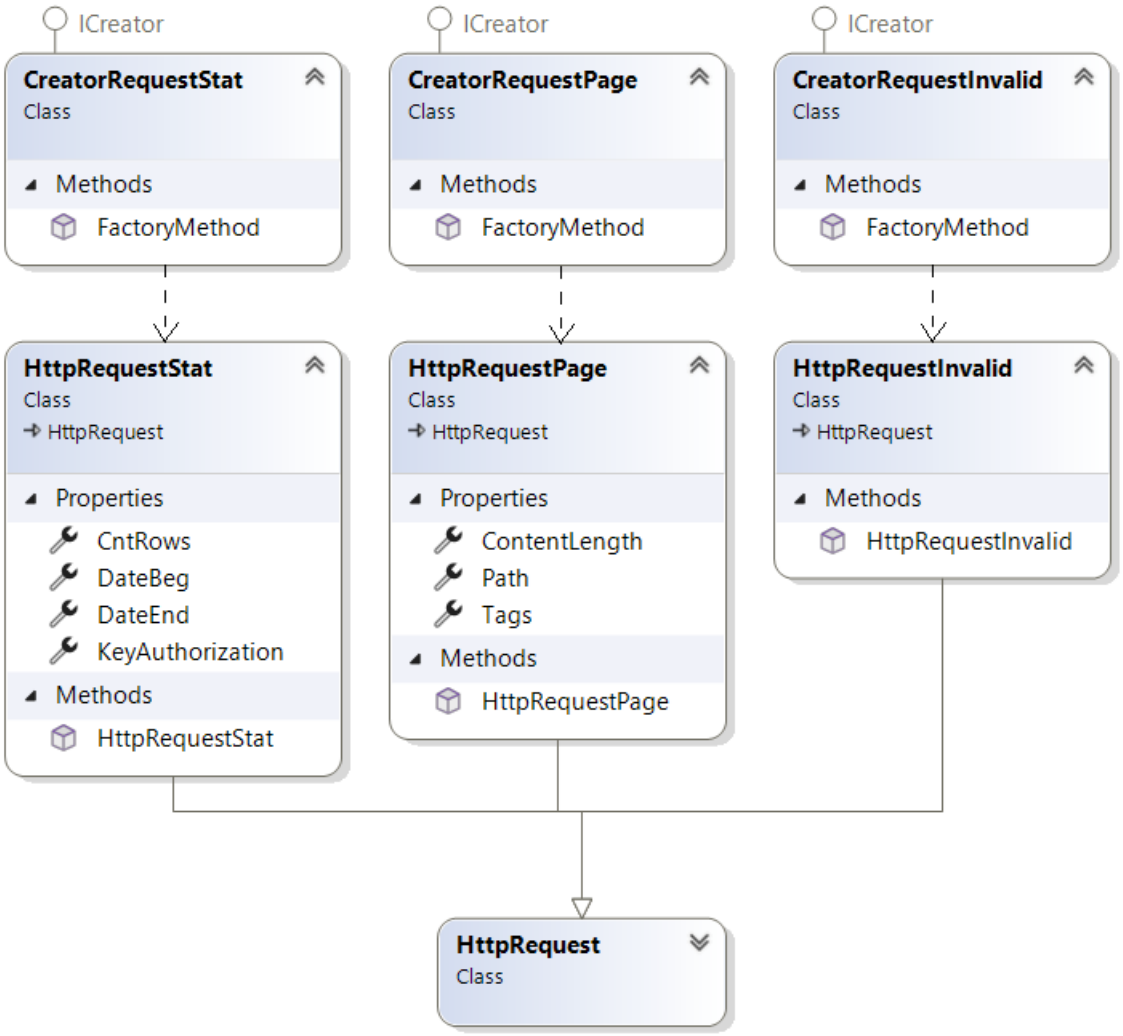

Рис. 6 - Діаграма класів

Висновок: у ході виконання даної лабораторної роботи я реалізував шаблон Factory, який визначає загальний інтерфейс для створення об’єктів, дозволяючи при цьому підкласам змінювати типи об’єктів, які створюються. Шаблон Factory дозволив мені ізолювати код, який відповідає за створення об'єктів, від решти програми. Я переконався, що використання цього паттерну дозволяє зменшити зв'язок між класами. Замість того, щоб один клас безпосередньо створював об'єкти інших класів (як це було реалізовано мною у попередніх роботах), фабрика бере на себе це завдання, що дозволяє

легко замінювати один об'єкт на інший або додавати нові класи чи типи об'єктів, не змінюючи код існуючих класів, що використовують фабрику.