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 №8

Технології розроблення програмного забезпечення

«Шаблони «Composite», «Flyweight», «Interpreter», «Visitor»»

Варіант 2

Виконала
студентка групи ІА–24:
Кійко А. О.

Перевірив
Мягкий М. Ю.

Київ 2024

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 «Composite», «Flyweight», «Interpreter» та «Visitor» належать до структурних і поведінкових патернів, які вирішують специфічні проблеми в організації коду. «Composite» та «Flyweight» зосереджуються на структурі об'єктів і оптимізації їх використання, тоді як «Interpreter» та «Visitor» забезпечують розширюваність та можливість виконання складних операцій у різних контекстах.

Composite використовується для роботи з ієрархіями об'єктів, де є відношення "частина-ціле". Завдяки цьому шаблону клієнтський код може однаково взаємодіяти як з окремими об'єктами, так і з групами об'єктів. Це досягається за рахунок створення спільного інтерфейсу для компонентів дерева, незалежно від того, чи це простий елемент, чи складна структура. Composite застосовується, наприклад, у файлових системах, де файли й папки мають однаковий набір операцій.

Flyweight зменшує споживання пам'яті та підвищує продуктивність шляхом повторного використання об'єктів. Основна ідея полягає в поділі стану об'єкта на внутрішній і зовнішній: внутрішній зберігається в спільному пулі, а зовнішній передається під час виконання операцій. Цей підхід особливо

ефективний у випадках, коли в системі є велика кількість об'єктів з повторюваним станом, наприклад, символів у текстових редакторах чи графічних елементах у програмі.

Interpreter забезпечує можливість інтерпретації виразів або команд певної мови. Він визначає граматику мови у вигляді набору класів, де кожен клас відповідає за окремий елемент граматики. Вирази інтерпретуються шляхом побудови дерева виконання, яке дозволяє розуміти й обробляти вхідні дані. Interpreter часто використовується в конфігураційних системах, SQL-запитах чи обробці математичних формул.

Visitor дозволяє відокремити логіку операцій від структури об'єктів, над якими ці операції виконуються. Це досягається через створення класів-відвідувачів, які "проходять" через об'єкти структури й виконують потрібні дії. Visitor корисний, коли необхідно додати нові операції до складних об'єктних структур, наприклад, для аналізу, модифікації чи рендерингу даних, зберігаючи при цьому стабільність основного коду.

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 COMPOSITE

COMPOSITE — це структурний патерн, який дозволяє створювати дерево об'єктів та працювати з ним так само, як і з одиничним об'єктом.
У попередніх версіях мого проєкту користувач міг виконати запит до Http-сервера, вказавши повний шлях до ресурсу, який він запитує (відносно

кореневої папки Http-серверу, що визначена у конфігурації застосунку). Для цього він мав точно знати, у якій саме вкладеній папці міститься той чи інший файл.

Реалізація шаблону “Composite” полягає у реалізації додаткової можливості зазначати при запиті сторінки просто ім’я файлу, який користувач хоче запросити, вказавши перед ним символ “*”. Система сама здійснить пошук заданого файлу і поверне один з можливих результатів:

1. Зміст сторінки, якщо знайдено єдиний файл у заданій чи будь-якій вкладеній папці відносно кореневої папки запиту;
2. Повідомлення про відсутність файлу у заданій папці та вкладених папках;
3. Повідомлення про перелік знайдених файлів (якщо їх декілька) з повідомленням про необхідність точніше зазначити розташування ресурсу, що запитується, або за посиланням перейти до необхідного ресурсу.

Для цього мною був створений абстрактний базовий **ResourceBase**, що визначає загальні методи як для простих (файли), так і для складених (папки) об'єктів дерева ресурсів файлового сховища.

```
namespace HttpServApp.Composite
{
  /// <summary>
  /// Базовий абстрактний клас
  /// </summary>
  internal abstract class ResourceBase
  {
  /// <summary>
    /// Метод, що дозволяє зрозуміти, чи може компонент мати дочірні об'єкти.
  /// </summary>
  /// <returns></returns>
    public virtual bool IsComposite() => true;

   /// <summary>
    /// Додавання дочірнього ресурсу
    /// </summary>
    /// <param name="resource"></param>
    /// <exception cref="NotImplementedException"></exception>
    public virtual void Add(ResourceBase resource)
```

```
    {
      throw new NotImplementedException();
    }

    /// <summary>
    /// Видалення дочірнього ресурсу
    /// </summary>
    /// <param name="resource"></param>
    /// <exception cref="NotImplementedException"></exception>
    public virtual void Remove(ResourceBase resource)
    {
      throw new NotImplementedException();
    }

    /// <summary>
  /// Абстрактний метод пошуку переліку ресурсів за заданим ім'ям resourceName
  /// Реалізація передбачена у класах-нащадках
    /// </summary>
    /// <param name="resourceName"></param>
    /// <returns></returns>
    public abstract List<string>? FindResources(string resourceName);
  }
}
```

Рис. 1 - Базовий абстрактний клас ResourceBase

Клас **ResourceFile** реалізує кінцеві об'єкти структури дерева - файли. Він не містить дочірніх елементів. Віртуальний метод **FindResource** виконує безпосередньо порівняння імені заданого у параметрі файла **resourceName** з іменем файла об’єкта класу. У випадку співпадіння повертає колекцію з одного елементу - повного імені файлу.

```
namespace HttpServApp.Composite
{
  /// <summary>
  /// Клас ResourceFile представляє собою файл - кінцевий вузол дерева
  /// </summary>
  internal class ResourceFile: ResourceBase
  {
    // Шлях до файлу
    private readonly string filePath;
    // ім'я файлу
    private readonly string fileName;
    public ResourceFile(string filePath, string fileName)
    {
      this.filePath = filePath;
      this.fileName = fileName;
    }
```

```
    public override bool IsComposite() => false;

    /// <summary>
    /// Метод пошуку файлу у дочірніх об'єктах
    /// </summary>
    /// <param name="resourceName"></param>
    /// <returns></returns>
    public override List<string>? FindResources(string resourceName)
    {
      // Якщо ім'я файлу співпадає із заданим resourceName (без врахування
регістру),
      // формуємо колекцію знайдених елементів у складі повного шляху до файла
      if (resourceName.Equals(fileName, StringComparison.CurrentCultureIgnoreCase))
        return new List<string>() { Path.Combine(filePath, fileName) };
      return null;
    }
  }
}
```

Рис. 2 - Клас ResourceFile

Клас **ResourceFolder** реалізує функціонал контейнеру, який має дочірні (вкладені) ресурси **childrenResource** (папки та файли). При створенні об’єкту даного класу рекурсивно будується дерево дочірніх об’єктів, причому завдяки наслідуванню від базового класу **ResourceBase** та віртуальності метода **Add** не важливо, об’єкт якого типу додається до загальної колекції. При виконанні віртуального метода **FindResource** виконується рекурсивний прохід через всіх дочірні об’єкти, у процесі чого збираються та підсумовуються у колекцію результати пошуку. Відповідно, дочірні об’єкти передають виклики своїм нащадкам. У результаті здійснюється повний обхід дерева об'єктів.

```
namespace HttpServApp.Composite
{
  /// <summary>
  /// Клас ResourceFolder представляє собою контейнер, що має
  /// дочірні елементи: вкладені файли та папки
  /// </summary>
  internal class ResourceFolder: ResourceBase
  {
    protected List<ResourceBase> childrenResource = new List<ResourceBase>();

    /// <summary>
    /// Конструктор класа: генерує дерево вкладених папок та файлів
```

```
    /// </summary>
    /// <param name="resourcePath"></param>
    public ResourceFolder(string resourcePath)
    {
      var directory = new DirectoryInfo(resourcePath);

      if (directory.Exists)
      {
        #region Вкладені папки
        DirectoryInfo[] dirs = directory.GetDirectories();
        foreach (DirectoryInfo dir in dirs)
        {
          childrenResource.Add(new ResourceFolder(dir.FullName));
        }
        #endregion

        #region Файли
        FileInfo[] files = directory.GetFiles();
        foreach (FileInfo file in files)
        {
          childrenResource.Add(new ResourceFile(directory.FullName, file.Name));
        }
        #endregion
      }
    }

    public override void Add(ResourceBase resource)
    {
      childrenResource.Add(resource);
    }

    public override void Remove(ResourceBase resource)
    {
      childrenResource.Remove(resource);
    }

    /// <summary>
    /// Метод пошуку файлу у дочірніх об'єктах
    /// </summary>
    /// <param name="resourceName"></param>
    /// <returns></returns>
    public override List<string>? FindResources(string resourceName)
    {
      List<string> resources = new List<string>();
      // Цикл по дочірнім об'єктам
      foreach (ResourceBase resource in childrenResource)
      {
        List<string>? findResources = resource.FindResources(resourceName);
        // Якщо знайдено хоча б 1 файл із заданим ім'ям resourceName, додаємо до
колекції
```

```
        if (findResources != null && findResources.Count > 0)
          resources.AddRange(findResources);
      }

      return resources;
    }
  }
}
```

Рис. 3 - Клас ResourceFolder

Приклад побудови дерева ресурсів та пошук файлу із заданим ім’ям наведено на рис. 4.

```
       // Якщо шлях до файлу містить символ '*',
      // це означає, що користувач хоче знайти сторінку у дереві репозиторію
(включаючи пошук у вкладених папках),
      // починаючи з якогось рівня
      else
      {
        // Формуємо шлях до файлу: частина строки від початку до першого символу "*"
        string folderPath = httpRequestPage.Path.Substring(0, indexOfStarSymbol);

        if (!Directory.Exists(folderPath))
        {
          httpRequestPage.Status = StatusEnum.NOT_FOUND;
          httpRequestPage.Message =
            $"Папка <b>'{folderPath}'</b> не знайдена на сервері";
        }

        // Формуємо ім'я файлу: частина строки від символу "*" до кінця
        string fileName = httpRequestPage.Path.Substring(indexOfStarSymbol + 1);

        // Будуємо дерево папок та файлів відносно folderPath
        ResourceBase resource = new ResourceFolder(
          Path.Combine(Configuration.ResourcePath ?? "C:", folderPath));
        // Шукаємо файли із заданою назвою
        List<string>? files = resource.FindResources(fileName);
        // Якщо відповідний файл не знайдено або знайдено декілька файлів
        if (files == null || files.Count != 1)
        {
          // Файл відсутній у дереві репозиторію => STATUS = NOT_FOUND
          if (files == null || files.Count == 0)
          {
            httpRequestPage.Status = StatusEnum.NOT_FOUND;
            httpRequestPage.Message =
              $"Файл сторінки <b>'{fileName}'</b> не знайдений у репозиторії
{folderPath} та вкладених папках";
          }

          // Знайдено декілька файлів у дереві репозиторію => STATUS = NOT_ALLOWED
          else
```

```
          {
            httpRequestPage.Status = StatusEnum.NOT_ALLOWED;
            httpRequestPage.Message =
              $"Знайдено <b>{files.Count} файла(ів)</b> у репозиторії {folderPath} з
іменем <b>{fileName}:</b>" +
              $"<br/><b>" +
                $"{string.Join("<br/>",
                  files.Select(file => {
                    // Формуємо посилання на знайдені файли
                    string localFile = file.Replace(Configuration.ResourcePath ??
"C:", "");
                    return $"<a href=\"{localFile}\">{localFile}</a>";

                  }).ToArray())}</b>" +
              $"<br/>Введіть повний шлях до потрібного ресурсу!";
          }
        }
        else
        {
          // Все добре. Файл знайдений в дереві репозиторію, і він єдиний
          httpRequestPage.Status = StatusEnum.OK;
          // Запам'ятовуємо повний шлях до файлу
          httpRequestPage.Path = files[0];
        }
      }
```

Рис. 4 - Приклад побудови дерева ресурсів та пошуку заданого файлу у методі BuilsStatus класуBuilderPage

Загальний вигляд структури класів та їх взаємозв’язків наведений на діаграмі класів (рис. 5).

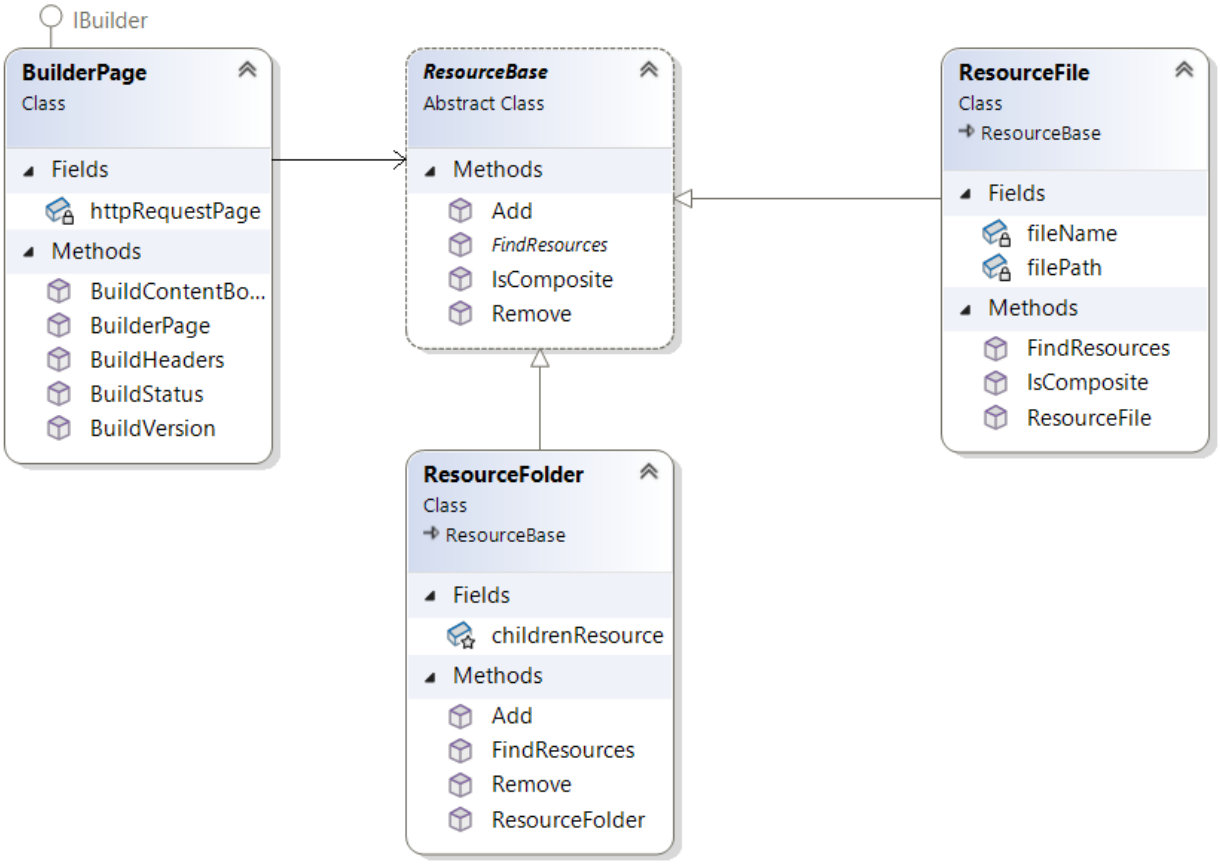

Рис. 5 - Діаграма класів

Приклади формування змісту сторінки та повідомлень наведені на рис. 6-8

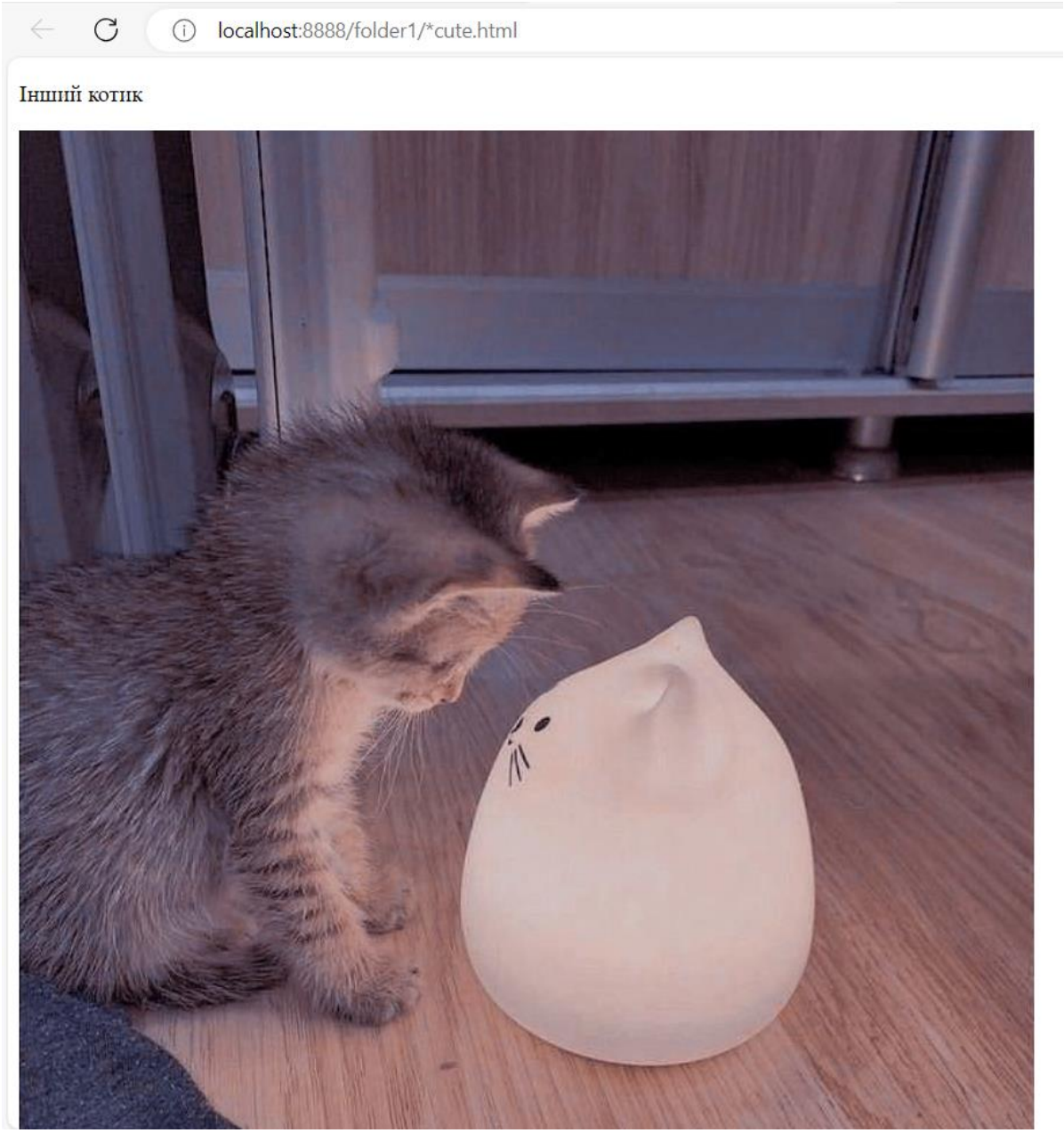

Рис.6 Зміст сторінки, що існує на сервері в єдиному екземплярі відносно кореневого ресурсу Http-сервера

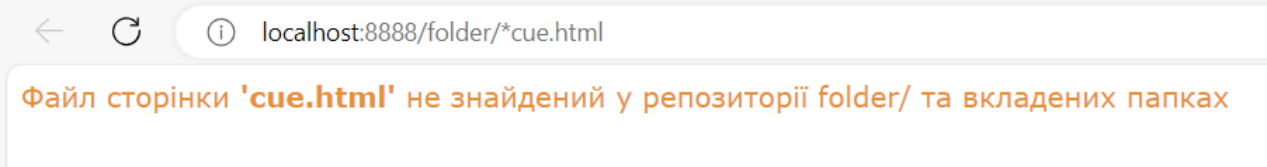

Рис.7 Повідомлення про відсутність ресурсу, що запитується, у заданій папці та вкладених папках

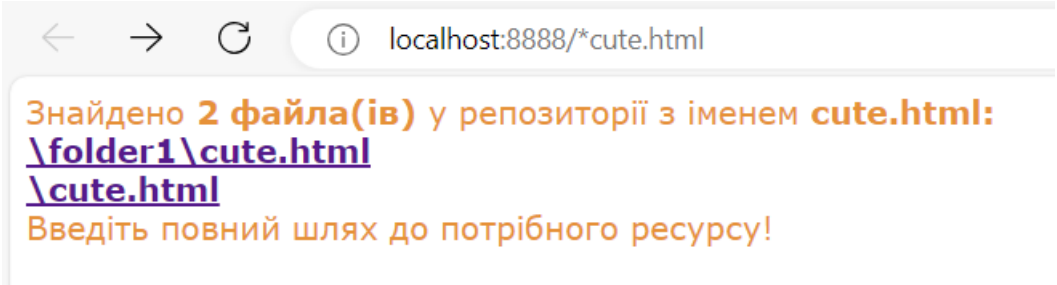

Рис.8 Повідомлення про існування декількох ресурсів з іменем файлу, що запитується, у заданій папці та вкладених папках

Висновок: у ході виконання даної лабораторної роботи я реалізувала шаблон Composite, який став синонімом усіх завдань, пов'язаних із побудовою дерева об'єктів (у моєму застосунку - дерева файлових ресурсів). Усі операції цього патерну засновані на рекурсії та «підсумовуванні» результатів на гілках дерева. Використання даного шаблону дозволило мені розширити функціональні можливості мого застосунку. Завдяки його використанню, прості та складені об'єкти наслідуються від єдиного абстрактного базового класу, тому при зверненні до них клієнтові байдуже, з яким саме об'єктом йому доводиться працювати, що значно спростило реалізацію функціоналу.

Посилання на репозиторій: [AnnaKiikoIA24/TRPZ_labs_Kiiko_AO_IA24 (github.com)]