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 №4

Технології розроблення програмного забезпечення

«Шаблони «Singleton», «Iterator», «Proxy», «State», «Strategy»»

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 теоритичні 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.

Шаблон Singleton призначений для забезпечення єдиного екземпляра класу з глобальною точкою доступу до нього. Це зручно в ситуаціях, коли необхідно централізовано зберігати стан, наприклад, для конфігурацій або доступу до бази даних. Таким чином, кожен раз, коли необхідний екземпляр класу, програма звертається до єдиного доступного об’єкта, що забезпечує унікальність і захищає від випадкових змін.

Шаблон Iterator створений для роботи з колекціями та дозволяє отримувати доступ до елементів послідовно, не розкриваючи внутрішньої структури колекції. Це особливо корисно, коли потрібно пройтися по елементах великого масиву або списку, адже ітератор надає єдиний стандартний інтерфейс для доступу, незалежно від того, яким чином колекція реалізована.

Proxy надає об'єкт-замісник, який контролює доступ до реального об'єкта. Це допомагає у випадках, коли необхідно обмежити або розширити функціональність об’єкта, наприклад, через додаткову логіку безпеки або кешування. Замісник перехоплює виклики до основного об'єкта, додає потрібні операції (такі як перевірка прав доступу або журналювання), а потім пересилає запит далі.

State призначений для управління поведінкою об'єкта залежно від його поточного стану. Цей шаблон корисний, коли об'єкт має різні стани, і кожен з них вимагає відмінної поведінки. Наприклад, в автоматі з продажу товарів зміна станів впливає на те, як автомат реагує на дії користувача. Об’єкт автоматично переходить у відповідний стан, і його поведінка змінюється відповідно.

Strategy надає можливість обирати різні алгоритми для вирішення певної задачі під час виконання програми. Це дозволяє об'єкту змінювати поведінку, коли змінюються зовнішні умови або потреби користувача, не змінюючи при цьому основний код. Кожен алгоритм інкапсулюється в окремому класі, і об'єкт просто підставляє потрібну стратегію в залежності від конкретної ситуації, що робить код гнучким і розширюваним.

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 State

Реалізація шаблону “State” у моєму проєкті полягає в управлінні поведінкою об'єкта HTTP-запиту (HttpRequest) залежно від його поточного стану. Шаблон “State” вимагає створення окремих класів для реалізації певної поведінки в залежності від стану об’єкту. З цією метою було

створено інтерфейс IState, що визначає метод ProcessingHandler для процесу обробки запиту незалежно від його стану.

```
1   using HttpServApp.Models;
2   using System.Net.Sockets;
3
4   namespace HttpServApp.State
5   {
6       // Інтерфейс, що відповідає за стан об'єкта HttpRequest
        7 references
7       interface IState
8       {
            6 references
9           public void ProcessingHandler(HttpRequest httpRequest, Socket socket);
10      }
11  }
```

Рис. 1 - Інтерфейс IState

У клас HttpRequest додано приватний член класу IState state, що містить дані про поточний стан об’єкта, а також метод TransitionTo, що дозволяє змінювати стан об'єкта під час виконання. Завдяки тому, що об'єкти станів мають загальний інтерфейс IState, HttpRequest зможе делегувати роботу стану, не прив'язуючись до його класу.

```
public void TransitionTo(IState state, Socket socket)
{
    Console.WriteLine($"HttpRequest state: Transition to {state.GetType().Name}.");
    this.state = state;
    // Виклик методу обробки нового стану запиту
    this.state.ProcessingHandler(this, socket);
}
```

Рис. 2 - Метод TransitionTo класу HttpRequest

Визначено основні стани об’єкта запиту та відповідні класи, що реалізують інтерфейс IState:

- стан: не валідний запит InvalidState;
- стан після валідації: валідний запит даних Web-сторінки: ValidatePageState;
- стан після валідації: валідний запит статистичних даних: ValidateStatisticState;

- стан: відповідь на запит сформована та відправлена: SendedState;
- кінцевий стан DoneState.

Схема переходу станів (стейт-машина) представлена на рисунку 3.

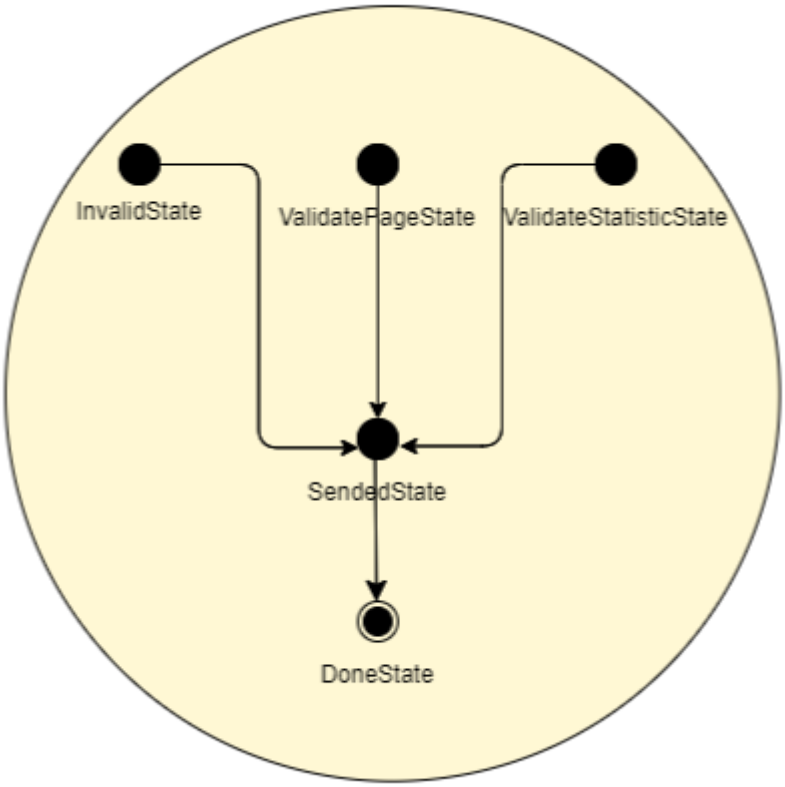

Рис. 3 - Схема переходу станів

Реалізація класів станів:

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

namespace HttpServApp.State
{
    // Стан після валідації: невалідний запит
    1 reference
    internal class InvalidState: IState
    {
        2 references
        public void ProcessingHandler(HttpRequest httpRequest, Socket socket)
        {
            // Формуємо відповідь: статус 500, запит не валідний
            httpRequest.CreateResponse();
            // Відсилаємо відповідь клієнту
            httpRequest.SendResponse(socket);
            Console.WriteLine($"HttpRequest state: InvalidState");

            // Перехід у новий стан: після відправки відповіді клієнту
            httpRequest.TransitionTo(new SendedState(), socket);
        }
    }
}
```

Рис. 4 - Клас InvalidState

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

namespace HttpServApp.State
{
    // Стан після валідації: валідний запит даних Web-сторінки
    1 reference
    internal class ValidatePageState: IState
    {
        2 references
        public void ProcessingHandler(HttpRequest httpRequest, Socket socket)
        {
            // Формуємо відповідь: метод віртуальний, повертає дані сторінки, що запитується
            httpRequest.CreateResponse();
            // Відсилаємо відповідь клієнту
            httpRequest.SendResponse(socket);
            Console.WriteLine($"HttpRequest state: ValidatePageState");

            // Перехід у новий стан: після відправки відповіді клієнту
            httpRequest.TransitionTo(new SendedState(), socket);
        }
    }
}
```

Рис. 5 - Клас ValidatePageState

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

namespace HttpServApp.State
{
    // Стан після валідації: валідний запит статистичних даних
    1 reference
    internal class ValidateStatisticState: IState
    {
        2 references
        public void ProcessingHandler(HttpRequest httpRequest, Socket socket)
        {
            // Формуємо відповідь: всередині віртуального метода запитуються дані статистики
            httpRequest.CreateResponse();
            // Відсилаємо відповідь клієнту
            httpRequest.SendResponse(socket);
            Console.WriteLine($"HttpRequest state: ValidateStatisticState");

            // Перехід у новий стан: після відправки відповіді клієнту
            httpRequest.TransitionTo(new SendedState(), socket);

        }
    }
}
```

Рис. 6 - Клас ValidateStatisticState

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


namespace HttpServApp.State
{
    // Стан після відправки даних клієнту: необхідно зберегти інформацію про запит в БД
    3 references
    internal class SendedState: IState
    {
        2 references
        public void ProcessingHandler(HttpRequest httpRequest, Socket socket)
        {
            // Викликаємо метод запису даних про запит до БД
            httpRequest.Repository.SaveToDB(httpRequest, '+');

            // Перехід у фінальний стан
            httpRequest.TransitionTo(new DoneState(), socket);
        }
    }
}
```

Рис. 7 - Клас SendedState

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

namespace HttpServApp.State
{
    // Фінальний стан об'єкта
    1 reference
    internal class DoneState: IState
    {
        2 references
        public void ProcessingHandler(HttpRequest httpRequest, Socket socket)
        {
            // Цей стан - останній, перехід не потрібен
            return;
        }
    }
}
```

Рис. 8 - Клас DoneState

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

namespace HttpServApp.Processing
{
    internal class ThreadProcessing
    {
        // Об'єкт для забезпечення обміну даними через мережу
        private readonly Socket socket;
        // Потік обробки даних
        private readonly Thread workThread;
        // Посилання на репозиторій
        private readonly Repository repository;

        public ThreadProcessing(Repository repository, Socket socket)
        {
            this.repository = repository;
            this.socket = socket;

            // Створюємо та запускаємо потік обробки даних запиту
            workThread = new Thread(DoWork) { Name = "requestThread" };
            workThread.Start();
        }

        /// <summary>
        /// Метод, що виконується при запуску потока обробки даних запиту
        /// </summary>
        public void DoWork()
```

```
        {
            Validator validator = new Validator(repository, socket);
            // Отримуємо строку запиту
            string strReceiveRequest = validator.GetStringRequest();
            // Визначаємо тип запиту
            string typeRequest = strReceiveRequest.Substring(10, 1);
            HttpRequest httpRequest;
            try
            {
                switch (typeRequest)
                {
                    // Запит сторінки
                    case "0":
                        {
                            // Запит сторінки валідний, інакше - Exception
                            httpRequest = validator.ParsePageRequest();
                            // Початковий стан запиту: валідний запит Web-
сторінки
                            // Далі викликаємо метод TransitionTo, що в процесі
виконання змінює стан запиту
                            // Початковий стан ValidatePageState
                            httpRequest.TransitionTo(new ValidatePageState(),
socket);
                            break;
                        }
                    // Запит статистики
                    case "1":
                        {
                            // Запит статистики валідний, інакше - Exception
                            httpRequest = validator.ParseStatisticRequest();
                            // Початковий стан запиту: валідний запит статистики
                            // Далі викликаємо метод TransitionTo, що в процесі
виконання змінює стан запиту,
                            // Початковий стан ValidateStatisticState
                            httpRequest.TransitionTo(new
ValidateStatisticState(), socket);
                            break;
                        }
                    default:
                        Console.WriteLine("Processing: Невизначений тип
запиту!");
                        throw new Exception();

                }
            }
            catch
            {
                // Початковий стан запиту: InvalidState
```

```
                httpRequest = new HttpRequest();
                httpRequest.TransitionTo(new InvalidState(), socket);
            }

            socket.Close();
            socket.Dispose();
        }
    }
}
```

Рис. 9 - Використання шаблону State в класі обробнику HTTP-запиту

Метод TransitionTo запускає ланцюжок зміни станів об’єкту HttpRequest у залежності від його початкового стану, тим самим реалізуючи послідовність обробки запиту починаючи з його валідації, закінчуючи записом результатів виконання і відправлення відповіді клієнту до бази даних та переходу до кінцевого стану.

Висновок: у ході виконання даної лабораторної роботи я ознайомилася зі структурою та принципами роботи декількох важливих шаблонів проектування: Singleton, Iterator, Proxy, State та Strategy. Кожен із них пропонує свій особливий підхід до вирішення конкретних задач у програмуванні, що допомагає робити код зрозумілим, структурованим та легким у підтримці. Було реалізовано шаблон State. Цей шаблон дозволяє легко додавати в майбутньому і обробляти нові стани, відокремлювати залежні від стану елементи об'єкта в інших об'єктах, і відкрито проводити заміну стану.

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