

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Практикум №3

з курсу «Основи розробки програмного забезпечення на платформі
Microsoft.NET»

на тему: «Шаблони проектування.

Породжуючі шаблони»

Викладач:

Крамар Ю.М.

Виконав:

студент 2 курсу

групи ПІ-15 ФІОТ

Костін В.А.

Київ-2023

Комп'ютерний практикум № 3.

Тема: шаблони проектування, породжуючі шаблони.

Мета: ознайомитися з основними шаблонами проектування, навчитися застосовувати їх при проектуванні і розробці ПЗ.

Постановка задачі комп'ютерного практикуму № 3

При виконанні комп'ютерного практикуму необхідно виконати наступні дії:

1) Вивчити породжуючі патерни. Знати загальну характеристику та призначення кожного з них, особливості реалізації кожного з породжуючих патернів та випадки їх застосування.

2) Реалізувати задачу згідно варіанту, запропонованого нижче у вигляді консольного застосування на мові C#. Розробити інтерфейси та класи з застосування одного або декількох патернів. Повністю реалізувати методи, пов'язані з реалізацією обраного патерну.

3) Повністю описати архітектуру проекту (призначення методів та класів), особливості реалізації обраного патерну. Для кожного патерну необхідно вказати основні класи та їх призначення,

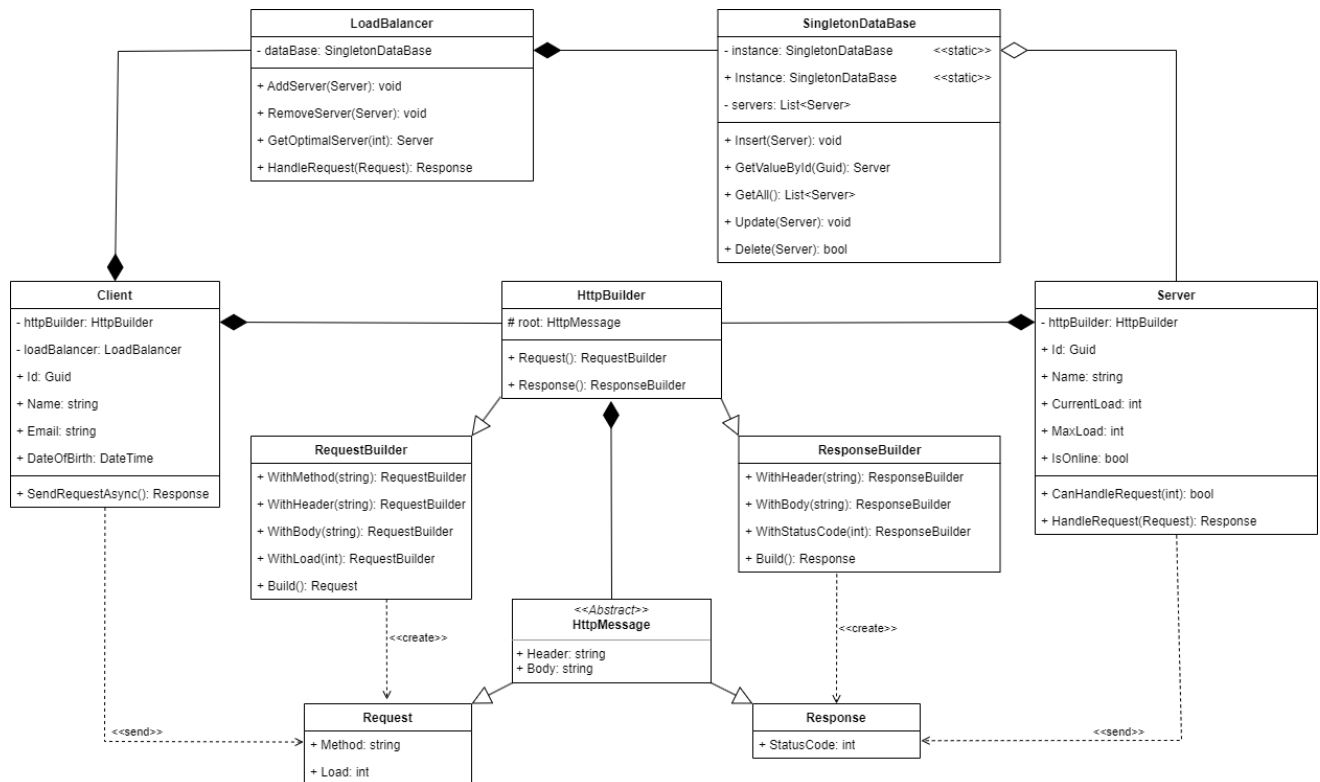
4) Навести UML-діаграму класів

5) Звіт повинен містити:

- a. обґрунтування обраного патерну (чому саме він);
- b. опис архітектури проекту (призначення методів та класів);
- c. UML-діаграму класів
- d. особливості реалізації обраного патерну
- e. текст програмного коду
- f. скріншоти результатів виконання.

2) Реалізувати задачу балансування навантаження серверів. Необхідно врахувати, що сервери можуть динамічно змінювати режим доступності (on-line або off-line), та те, що кожен запит має оброблятися тільки одним сервером, який містить інформацію про свій поточний стан.

UML-діаграма класів:



Архітектура програми

Назва класу: Server

Призначення: клас серверу для опрацювання запитів і відправлення відповідей.

Опис властивостей:

Id – унікальний ідентифікатор серверу.

Name – ім'я серверу.

CurrentLoad – поточне навантаження серверу

MaxLoad – максимально допустиме навантаження серверу.

IsOnline – поточний стан серверу.

Опис методів:

bool CanHandleRequest(int requestLoad) – метод, що перевіряє, чи здатний сервер обробити запит із введеним навантаженням.

Response HandleRequest(Request request) – метод, що обробляє введений запит та повертає відповідь.

Назва класу: SingletonDataBase

Призначення: клас бази даних, що зберігає інформацію про сервери для відправки запитів. Для цього класу реалізовано шаблон Singleton.

Опис властивостей:

Instance – статичний і єдиний екземпляр поточного класу.

Опис методів:

void Insert(IServer server) – метод для додавання нового серверу в базу даних.

IServer GetServerById(Guid guid) – метод для пошуку сервера за його унікальним ідентифікатором.

IEnumerable<IServer> GetAll() – метод, що повертає усі сервери з бази даних.

void Update(IServer server) – метод для оновлення даних введеного серверу в базі даних.

bool Delete(IServer server) – метод для видалення серверу з бази даних, повертає true в разі успіху, інакше – false.

Назва класу: LoadBalancer

Призначення: клас для балансування навантаження між серверами. Виступає посередником між клієнтом та сервером.

Опис методів:

void AddServer(IServer server) – метод для додавання нового серверу в базу даних.

void Remove(IServer server) – метод для видалення серверу з бази даних.

IServer GetOptimalServer() – метод для пошуку найменш завантаженого серверу в базі даних.

Response HandleRequest(Request request) – метод для посилання запиту найменш завантаженому серверу, повертає відповідь серверу.

Назва класу: Client

Призначення: клас клієнта, який посилає запити на сервер та отримує відповідь.

Опис властивостей:

Id – унікальний ідентифікатор клієнта.

Name – ім'я клієнта.

Email – поштова адреса клієнта.

DateOfBirth – дата народження клієнта.

Опис методів:

Response SendRequestAsync() – метод для генерації запиту, посилання його на сервер та отримання відповіді.

Назва класу: HttpResponseMessage

Призначення: абстрактний клас http-посилання.

Опис властивостей:

Header – заголовок посилання.

Body – тіло посилання.

Назва класу: Request

Призначення: клас запиту, який є нащадком класу HttpResponseMessage.

Опис властивостей:

Method – метод запиту.

Load – навантаження, яке запит створить на сервері.

Назва класу: Response

Призначення: клас відповіді, який є нащадком класу HttpResponseMessage.

Опис властивостей:

StatusCode – статус код відповіді.

Назва класу: HttpBuilder

Призначення: клас будівник http-посилань.

Опис властивостей:

root – кореневий елемент, який будується.

Опис методів:

RequestBuilder Request() – метод який починає будівництво запиту.

ResponseBuilder Response() – метод який починає будівництво відповіді.

Назва класу: RequestBuilder

Призначення: клас будівник запитів. Реалізує Fluent API.

Опис методів:

RequestBuilder WithMethod(string method) – метод для ініціалізації поля методу запиту.

RequestBuilder WithHeader(string header) – метод для ініціалізації поля заголовку запиту.

RequestBuilder WithBody(string body) – метод для ініціалізації поля тіла запиту.

RequestBuilder WithLoad(int load) – метод для ініціалізації поля навантаження запиту.

Request Build() – метод для будівництва запиту.

Назва класу: ResponseBuilder

Призначення: клас будівник відповіді. Реалізує Fluent API.

Опис методів:

ResponseBuilder WithHeader(string header) – метод для ініціалізації поля заголовку відповіді.

ResponseBuilder WithBody(string body) – метод для ініціалізації поля тіла відповіді.

ResponseBuilder WithStatusCode(int statusCode) – метод для ініціалізації поля статус коду відповіді.

Response Build() – метод для будівництва відповіді.

Шаблони

В данній лабораторній роботі було використано два породжуючі шаблони: одинак(Singleton) та будівник(Builder).

Використання одинака обумовлено тим, що база даних серверів має бути єдиним створеним екземпляром свого класу для всієї програми, щоб усі інші модулі програми мали доступ до одних і тих самих даних. Тобто об'єкт бази даних серверів має бути єдиним та глобальним для всіх модулів.

Будівник використовується для побудови http-посилання(запиту та відповіді). Сенса у створенні будівника полягає в тому, що окрім зручного інтерфейсу ініціалізації полів об'єкту в будівнику також можна прописувати правила валідації полів об'єкту(наприклад, якщо метод запиту дорівнює "GET", то запит не може мати тіла, і навпаки).

Реалізація шаблонів

Для реалізації одинака використано клас Lazy<T> для створення статичного екземпляру класу. Цей клас в конструкторі приймає лямда-вираз для ініціалізації екземпляру, якщо він ще не проініціалізований.

Для реалізації будівника було створено три класи: кореневий клас "HttpBuilder" та два похідних від нього – "RequestBuilder" та "ResponseBuilder". Ідея такої реалізації полягає в тому, що користувач має вибрати, що саме він хоче збудувати: запит чи відповідь. Залежно від цього клас "HttpBuilder" має методи Request() та Response(), які повертають об'єкти класів "RequestBuilder" та "ResponseBuilder" відповідно. Далі

дочірні класи для створення окрему запиту або відповіді мають окремі методи з Fluent API для ініціалізації властивостей запиту або відповіді.

Код програми

Код програми можна подивитись на репозиторії за посиланням:

<https://github.com/VadimkaKostin/.Net>

CustomServer.cs

```
public class CustomServer : IServer
{
    private readonly HttpBuilder _httpBuilder;

    public Guid Id { get; set; }
    public string Name { get; set; }
    public int CurrentLoad { get; set; }
    public int MaxLoad { get; set; }
    public bool IsOnline { get; set; }

    public CustomServer(string name, int maxLoad)
    {
        Id = Guid.NewGuid();
        Name = name;
        CurrentLoad = 0;
        MaxLoad = maxLoad;
        IsOnline = true;

        _httpBuilder = new HttpBuilder();
    }

    public bool CanHandleRequest(int requestLoad)
    {
        return IsOnline && (CurrentLoad + requestLoad) <= MaxLoad;
    }

    public async Task<Response> HandleRequest(Request request)
    {
        CurrentLoad += request.Load;

        int duration = new Random().Next(7, 11);

        //Опрацювання запиту
        await Task.Delay(duration * 1000);

        if (CurrentLoad == MaxLoad)
        {
            IsOnline = false;
        }
        else
            CurrentLoad -= request.Load;

        Response response =
            _httpBuilder.Response()
```



```

        .WithHeader(DateTime.Now.ToString())
        .WithBody($"Response from \'{Name}\'")
        .WithStatusCode(200)
        .Build();

    return response;
}
}

```

SingletonDataBase.cs

```

public class SingletonDataBase : IDatabase<IServer>
{
    private static Lazy<SingletonDataBase> _instance =
        new Lazy<SingletonDataBase>(() => new SingletonDataBase());
    public static SingletonDataBase Instance => _instance.Value;

    private List<IServer> _servers;

    private SingletonDataBase()
    {
        _servers = new List<IServer>();
    }

    public void Insert(IServer value)
    {
        _servers.Add(value);
    }

    public IServer GetValueById(Guid guid)
    {
        IServer serverSearched = _servers.FirstOrDefault(server => server.Id
== guid);

        return serverSearched;
    }

    public IEnumerable<IServer> GetAll()
    {
        return _servers.Select(server => server);
    }

    public void Update(IServer value)
    {
        IServer server = GetValueById(value.Id);

        if (server == null)
            return;

        foreach (var prop in server.GetType().GetProperties())
        {
            prop.SetValue(server, prop.GetValue(value));
        }
    }

    public bool Delete(Guid guid)
    {
        IServer server = GetValueById(guid);

        if (server == null)
            return false;

        return _servers.Remove(server);
    }
}

```

LaodBalancer.cs

```
public class LoadBalancer : ILoadBalancer
{
    private readonly SingletonDataBase _dataBase;

    public LoadBalancer()
    {
        _dataBase = SingletonDataBase.Instace;
    }

    public void AddServer(IServer server)
    {
        _dataBase.Insert(server);
    }

    public void RemoveServer(IServer server)
    {
        _dataBase.Delete(server.Id);
    }

    public IServer GetOptimalServer(int load)
    {
        List<IServer> servers = _dataBase.GetAll()
            .Where(server => server.CanHandleRequest(load))
            .ToList();

        if(servers.Count == 0)
            return null;

        int min = servers.Min(server => server.CurrentLoad);

        IServer optimalServer = servers.FirstOrDefault(server =>
server.CurrentLoad == min);

        return optimalServer;
    }

    public async Task<Response> HandleRequest(Request request)
    {
        IServer server = this.GetOptimalServer(request.Load);

        if (server == null)
            return null;

        Response response = await server.HandleRequest(request);

        return response;
    }
}
```

Client.cs

```
public class Client : IClient
{
    private readonly ILoadBalancer _loadBalancer;
    private readonly HttpBuilder _httpBuilder;

    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime DateOfBirth { get; set; }

    public Client(ILoadBalancer loadBalancer)
    {

```

```

        _loadBalancer = loadBalancer;

        _httpBuilder = new HttpBuilder();

        Id = Guid.NewGuid();
    }

    public async Task<Response> SendRequestAsync()
    {
        Request request = _httpBuilder.Request()
            .WithHeader($"Url: random url\n Time: {DateTime.Now}")
            .WithMethod("POST")
            .WithBody($"Id: {Id}\nName: {Name}\nEmail:{Email}\nDate of
birth:{DateOfBirth}")
            .WithLoad(5)
            .Build();

        Response response = await _loadBalancer.HandleRequest(request);

        return response;
    }
}

```

HttpMessage.cs

```

public abstract class HttpMessage
{
    public string Header { get; set; }
    public string Body { get; set; }
}

```

Request.cs

```

public class Request : HttpMessage
{
    public string Method { get; set; }
    public int Load { get; set; }
}

```

Response.cs

```

public class Response : HttpMessage
{
    public int StatusCode { get; set; }
}

```

HttpBuilder.cs

```

public class HttpBuilder
{
    protected HttpMessage _root;

    public RequestBuilder Request()
    {
        return new RequestBuilder();
    }

    public ResponseBuilder Response()
    {
        return new ResponseBuilder();
    }
}

```

```

public class RequestBuilder : HttpBuilder
{
    public RequestBuilder()
    {
        _root = new Request();
    }

    public RequestBuilder WithMethod(string method)
    {
        if(string.IsNullOrEmpty(method))
            throw new ArgumentNullException("method");

        if (method != "GET" && method != "POST")
            throw new ArgumentException("Method can supply only two values:
GET and POST.");

        if(!string.IsNullOrEmpty(_root.Body) && method == "GET")
            throw new ArgumentException("Request with method GET cannot have
request body.");

        (_root as Request).Method = method;
        return this;
    }
    public RequestBuilder WithHeader(string header)
    {
        if(string.IsNullOrEmpty(header))
            throw new ArgumentNullException("header");

        _root.Header = header;
        return this;
    }
    public RequestBuilder WithBody(string body)
    {
        if ((_root as Request).Method == "GET")
            throw new ArgumentException("Request with method GET cannot have
request body.");

        _root.Body = body;
        return this;
    }
    public RequestBuilder WithLoad(int load)
    {
        if(load < 0)
            throw new ArgumentOutOfRangeException("load");

        (_root as Request).Load = load;
        return this;
    }
    public Request Build()
    {
        return _root as Request;
    }
}

public class ResponseBuilder : HttpBuilder
{
    public ResponseBuilder()
    {
        _root = new Response();
    }

    public ResponseBuilder WithHeader(string header)
    {
        if(string.IsNullOrEmpty(header))

```

```

        throw new ArgumentNullException("header");

        _root.Header = header;
        return this;
    }
    public ResponseBuilder WithBody(string body)
    {
        if (string.IsNullOrEmpty(body))
            throw new ArgumentNullException("body");

        _root.Body = body;
        return this;
    }
    public ResponseBuilder WithStatusCode(int statusCode)
    {
        if (statusCode < 100 || statusCode > 599)
            throw new ArgumentException("statusCode");
        (_root as Response).StatusCode = statusCode;
        return this;
    }
    public Response Build()
    {
        return _root as Response;
    }
}

```

Program.cs

```

public static class Program
{
    private static readonly SingletonDataBase db = SingletonDataBase.Instance;

    public static async Task Main(string[] args)
    {
        ILoadBalancer loadBalancer = new LoadBalancer();

        loadBalancer.AddServer(new CustomServer("Server1", 50));
        loadBalancer.AddServer(new CustomServer("Server2", 45));
        loadBalancer.AddServer(new CustomServer("Server3", 55));
        loadBalancer.AddServer(new CustomServer("Server4", 50));
        loadBalancer.AddServer(new CustomServer("Server5", 50));

        IClient client = new Client(loadBalancer)
        {
            Name = "Vadim Kostin",
            Email = "vadkostinxm@gmail.com",
            DateOfBirth = Convert.ToDateTime("2003-09-26")
        };

        HttpBuilder httpBuilder = new HttpBuilder();

        while (true)
        {
            Task.Run(() => SendRequest(client));

            bool result = UpdateConsole();

            if (result)
            {
                Console.WriteLine("Error! All servers are offline.");
                break;
            }
        }
    }
}

```

```

        await Task.Delay(150);
    }

    Console.ReadLine();
}

public static async Task SendRequest(IClient client)
{
    Response response = await client.SendRequestAsync();
}

public static bool UpdateConsole()
{
    Console.Clear();

    int countOffline = 0;

    foreach(IServer server in db.GetAll())
    {
        Console.Write($"{server.Name}: ");

        if (server.IsOnline)
            Console.WriteLine($"{server.CurrentLoad}/{server.MaxLoad}");
        else
        {
            Console.WriteLine($"offline");
            countOffline++;
        }
    }

    return countOffline == db.GetAll().Count();
}
}

```

Результат роботи програми

Під час роботи програми клієнт із інтервалом 150 мс відправляє новий запит на балансувальник, той отримує запит, обирає найменш навантажений сервер та відправляє запит на нього. Для демонстрації до бази даних додано 5 серверів.

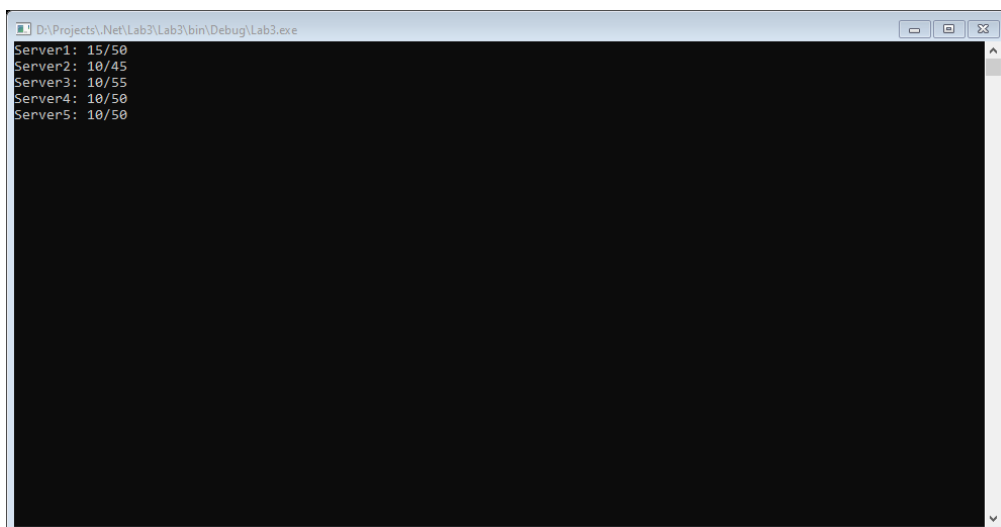


Рисунок 1.1 – спочатку поточне навантаження серверів невелике.

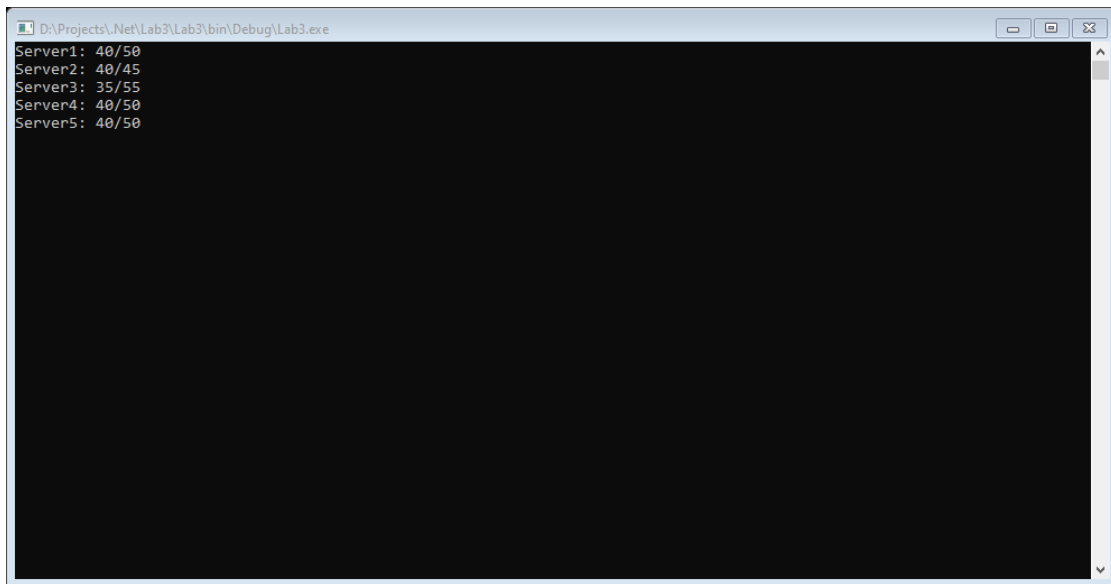


Рисунок 1.2 – після певного проміжку часу поточне навантаження серверів зростає через часте посилення запитів.

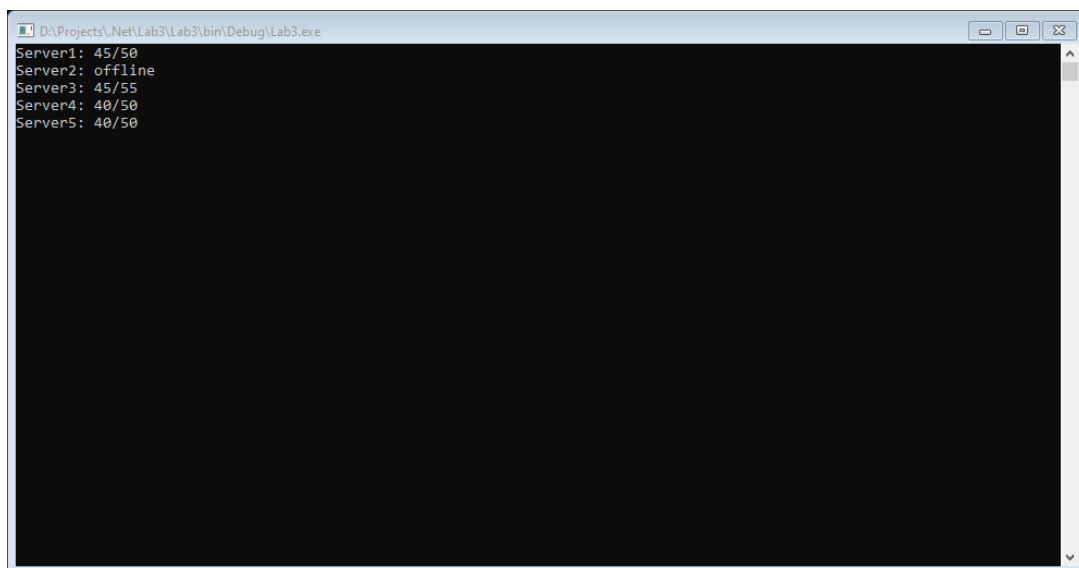
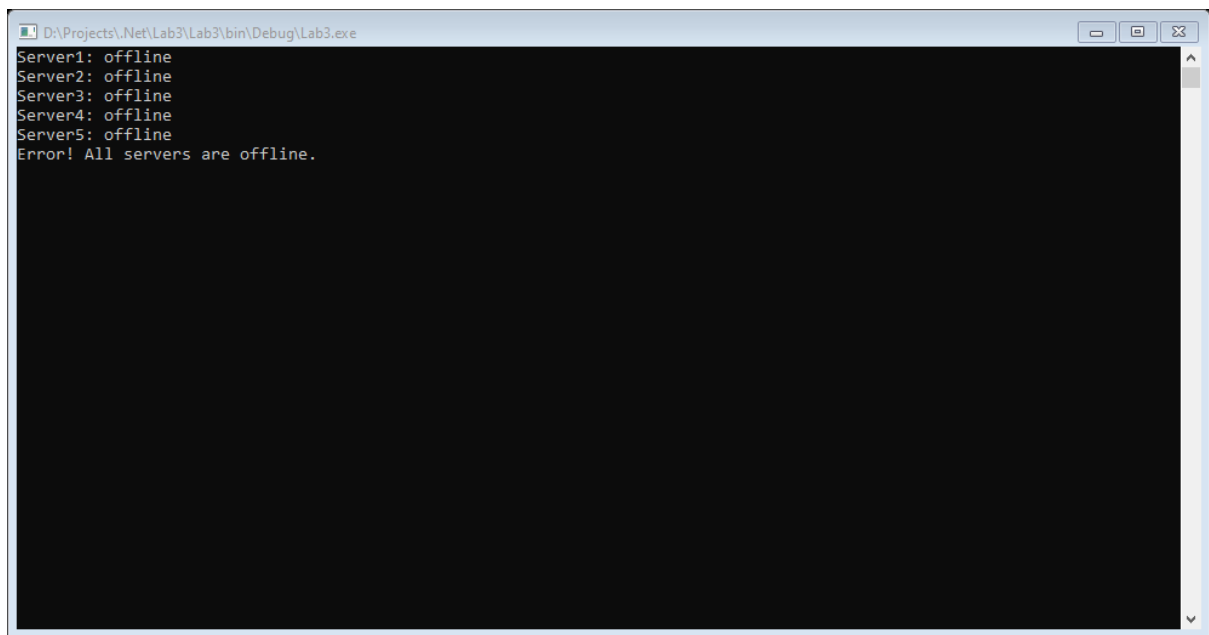


Рисунок 1.3 – коли поточне навантаження певного серверу досягає максимальної позначки, сервер змінює статус на offline.



The image shows a screenshot of a Windows command prompt window. The title bar at the top reads "D:\Projects\Net\Lab3\Lab3\bin\Debug\Lab3.exe". The window contains the following text:

```
Server1: offline  
Server2: offline  
Server3: offline  
Server4: offline  
Server5: offline  
Error! All servers are offline.
```

Рисунок 1.4 – коли всі сервери змінили статус на offline, виводиться помилка.

Висновок

Протягом третього комп'ютерного практикуму ми ознайомитися з породжуючими шаблонами. Була розроблена система балансування навантаження серверів, яка працювала за наступним принципом: клієнт посилає запит балансувальнику, який обирає сервер із найменшим поточним навантаженням та посилає цей запит йому, коли сервер обробив запит, він посилає відповідь назад балансувальнику, а той посилає її клієнту. В програмі були реалізовані шаблони одинак(Singleton) для бази даних серверів та будівник(Builder) із Fluent API для побудови http-посилання.