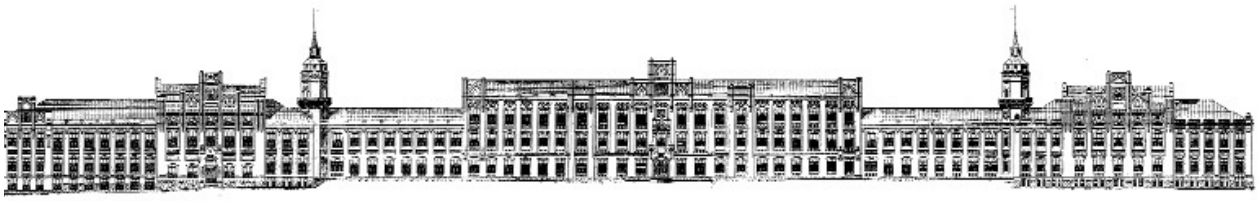

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Звіт

З дисципліни “Технології розроблення програмного забезпечення”

Варіант: 23

Виконала
студентка групи ІА-23:
Павленко Анастасія

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024

Лабораторна робота №4

Тема лабораторної роботи: Шаблони проектування «Singleton», «Iterator», «Proxy», «State», «Strategy».

Мета лабораторної роботи: Ознайомитися з основними шаблонами проектування програмного забезпечення, їх призначенням і застосуванням. Розглянути реалізацію на прикладах, вивчити їх переваги та недоліки, а також практично застосувати ці шаблони для вирішення конкретних задач програмування.

Варіант 23: Project Management software (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server)

Програмне забезпечення для управління проектами повинно мати наступні функції: супровід завдань/вимог/проектів, списків команд, поточних завдань, планування за методологіями agile/kanban/rup (включаючи дошку завдань, ітерації тощо), мати можливість прикріплювати вкладені файли до завдань та посилатися на конкретні версії програми, зберігати виконувані файли для кожної версії.

Завдання.

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

Зміст

Короткі теоретичні відомості

Шаблони проектування

Патерн проектування — це типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм.

На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм — це чіткий набір дій, то патерн — це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Патерни відрізняються за рівнем складності, деталізації та охоплення проектованої системи. Найбільш низькорівневі та прості патерни — *ідіоми*. Вони не дуже універсальні, позаяк мають сенс лише в рамках однієї мови програмування.

Найбільш універсальні — *архітектурні патерни*, які можна реалізувати практично будь-якою мовою. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Крім цього, патерни відрізняються і за призначенням. Три основні групи патернів:

- **Породжуючі патерни** піклуються про гнучке створення об’єктів без внесення в програму зайвих залежностей.
- **Структурні патерни** показують різні способи побудови зв’язків між об’єктами.
- **Поведінкові патерни** піклуються про ефективну комунікацію між об’єктами.

Singelton

Singelton — це породжувальний патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього. Singelton вирішує відразу дві проблеми (порушуючи *принцип єдиного обов'язку* класу):

1. **Гарантує наявність єдиного екземпляра класу**. Найчастіше за все це корисно для доступу до якогось спільного ресурсу, наприклад, бази даних.

Уявімо, що ми створили об'єкт, а через деякий час намагаємось створити ще один. У цьому випадку хотілося б отримати старий об'єкт замість створення нового. Таку поведінку неможливо реалізувати за допомогою звичайного конструктора, оскільки конструктор завжди повертає новий об'єкт.

2. **Надає глобальну точку доступу**. Це не просто глобальна змінна, через яку можна дістатися до певного об'єкта. Глобальні змінні не захищені від запису, тому будь-який код може підмінити їхнє значення.

Всі реалізації Singelton зводяться до того, аби приховати типовий конструктор та створити публічний статичний метод, який і контролюватиме життєвий цикл об'єкта. Якщо у нас є доступ до класу, отже, буде й доступ до цього статичного методу.

Переваги:

- Гарантує наявність єдиного екземпляра класу.
- Надає глобальну точку доступу до нього.
- Реалізує відкладену ініціалізацію об'єкта-одинака.

Недоліки:

- Порушує *принцип єдиного обов'язку класу*.
- Маскує поганий дизайн.
- Проблеми багатопоточності.
- Вимагає постійного створення Mock-об'єктів при юніт-тестуванні.

Iterator

Ітератор — це поведінковий патерн проектування, що дає змогу послідовно обходити елементи складових об’єктів, не розкриваючи їхньої внутрішньої організації.

Колекції — це набір об’єктів, зібраний в одну купу за якимись критеріями.

Незважаючи на те, яким чином структуровано колекцію, користувач повинен мати можливість послідовно обходити її елементи, щоб виконувати з ними певні дії.

Ідея патерна полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий об’єкт. Об’єкт-ітератор відстежуватиме стан обходу, поточну позицію в колекції та кількість елементів, які ще залишилося обійти. Одну і ту саму колекцію зможуть одночасно обходити різні ітератори. До того ж, якщо потрібно буде додати новий спосіб обходу, ви зможете створите окремий клас ітератора, не змінюючи існуючого коду.

Структура

1. **Ітератор** описує інтерфейс для доступу та обходу елементів колекцій.
2. **Конкретний ітератор** реалізує алгоритм обходу якоїсь конкретної колекції. Об’єкт ітератора повинен сам відстежувати поточну позицію при обході колекції, щоб окремі ітератори могли обходити одну і ту саму колекцію незалежно.
3. **Колекція** описує інтерфейс отримання ітератора з колекції. Сама колекція може створювати ітератори, оскільки вона знає, які саме ітератори здатні з нею працювати.
4. **Конкретна колекція** повертає новий екземпляр певного конкретного ітератора, зв’язавши його з поточним об’єктом колекції. Сигнатура методу повертає інтерфейс ітератора. Це дозволяє клієнтові не залежати від конкретних класів ітераторів.

5. **Клієнт** працює з усіма об’єктами через інтерфейси колекції та ітератора. Через це клієнтський код не залежить від конкретних класів, що дозволяє застосовувати різні ітератори, не змінюючи існуючого коду програми.

Застосування

- Якщо є складна структура даних, і потрібно приховати від клієнта деталі її реалізації (з питань складності або безпеки).
- Якщо потрібно мати кілька варіантів обходу однієї і тієї самої структури даних.
- Якщо необхідно мати єдиний інтерфейс обходу різних структур даних.

Переваги:

- Спрощує класи зберігання даних.
- Дозволяє реалізувати різні способи обходу структури даних.
- Дозволяє одночасно переміщуватися структурою даних у різних напрямках.

Недоліки:

- Невиправданий, якщо можна обійтися простим циклом.

Proxy

Proxy — це структурний патерн проектування, що дозволяє підставляти замість реальних об’єктів спеціальні об’єкти-замінники, які мають той самий інтерфейс. При отриманні запиту від клієнта об’єкт-замісник сам би створював примірник службового об’єкта та переадресовував би йому всю реальну роботу.

Структура

Інтерфейс сервісу визначає загальний інтерфейс для сервісу й замісника. Завдяки цьому об’єкт замісника можна використовувати там, де очікується об’єкт сервісу.

1. **Сервіс** містить корисну бізнес-логіку.

2. **Замісник** зберігає посилання на об’єкт сервісу. Після того, як замісник закінчує свою роботу (наприклад, ініціалізацію, логування, захист або інше), він передає виклики вкладеному сервісу.

Замісник може сам відповідати за створення й видалення об’єкта сервісу.

3. **Клієнт** працює з об’єктами через інтерфейс сервісу. Завдяки цьому його можна «обдурити», підмінивши об’єкт сервісу об’єктом замісника.

Застосування

- Лінива ініціалізація (віртуальний проксі). Коли є важкий об’єкт, який завантажує дані з файлової системи або бази даних.
- Захист доступу (захищаючий проксі). Коли в програмі є різні типи користувачів, і потрібно захистити об’єкт від неавторизованого доступу. Наприклад, якщо об’єкти — важлива частина операційної системи, а користувачі — сторонні програми (корисні чи шкідливі).
- Локальний запуск сервісу (віддалений проксі). Коли справжній сервісний об’єкт знаходиться на віддаленому сервері.
- Логування запитів (логуючий проксі). Коли потрібно зберігати історію звернень до сервісного об’єкта.
- Кешування об’єктів («розумне» посилання). Коли потрібно кешувати результати запитів клієнтів і керувати їхнім життєвим циклом.

Переваги:

- Дозволяє контролювати сервісний об’єкт непомітно для клієнта.
- Може працювати, навіть якщо сервісний об’єкт ще не створено.
- Може контролювати життєвий цикл службового об’єкта.

Недоліки:

- Ускладнює код програми внаслідок введення додаткових класів.
- Збільшує час отримання відклику від сервісу.

State

Стан — це поведінковий патерн проектування, що дає змогу об’єктам змінювати поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об’єкта.

Основна ідея в тому, що програма може знаходитися в одному з кількох станів, які увесь час змінюють один одного. Патерн Стан пропонує створити окремі класи для кожного стану, а потім винести туди поведінки, що відповідають цим станам. Замість того, щоб зберігати код всіх станів, початковий об’єкт, який зветься *контекстом*, міститиме посилання на один з об’єктів-станів і делегуватиме йому роботу в залежності від стану.

Завдяки тому, що об’єкти станів матимуть спільний інтерфейс, контекст зможе делегувати роботу стану, не прив’язуючись до його класу. Поведінку контексту можна буде змінити в будь-який момент, підключивши до нього інший об’єкт-стан.

Дуже важливим нюансом, який відрізняє цей патерн від Стратегії, є те, що і контекст, і конкретні стани можуть знати один про одного та ініціювати переходи від одного стану до іншого.

Структура

1. **Контекст** зберігає посилання на об’єкт стану та делегує йому частину роботи, яка залежить від станів. Контекст працює з цим об’єктом через загальний інтерфейс станів. Контекст повинен мати метод для присвоєння йому нового об’єкта-стану.
2. **Стан** описує спільний для всіх конкретних станів інтерфейс.
3. **Конкретні стани** реалізують поведінки, пов’язані з певним станом контексту. Іноді доводиться створювати цілі ієрархії класів станів, щоб узагальнити дублюючий код.
4. І контекст, і об’єкти конкретних станів можуть вирішувати, коли і який стан буде обрано наступним. Щоб перемкнути стан, потрібно подати інший об’єкт-стан до контексту.

Застосування

- Якщо є об’єкт, поведінка якого кардинально змінюється в залежності від внутрішнього стану, причому типів станів багато, а їхній код часто змінюється.
- Якщо код класу містить безліч великих, схожих один на одного умовних операторів, які вибирають поведінки в залежності від поточних значень полів класу.
- Якщо ми свідомо використовуємо табличну машину станів, побудовану на умовних операторах, але змушені миритися з дублюванням коду для схожих станів та переходів.

Переваги:

- Позбавляє від безлічі великих умовних операторів машини станів.
- Концентрує в одному місці код, пов’язаний з певним станом.
- Спрощує код контексту.

Недоліки:

- Може невиправдано ускладнити код, якщо станів мало, і вони рідко змінюються.

Strategy

Стратегія — це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна заміняти один на інший прямо під час виконання програми.

Патерн Стратегія пропонує визначити сімейство схожих алгоритмів, які часто змінюються або розширюються, й винести їх до власних класів, які називають *стратегіями*.

Замість того, щоб початковий клас сам виконував той чи інший алгоритм, він відіграватиме роль контексту, посилаючись на одну зі

стратегій та делегуючи їй виконання роботи. Щоб змінити алгоритм, вам буде достатньо підставити в контекст інший об’єкт-стратегію.

Важливо, щоб всі стратегії мали єдиний інтерфейс. Використовуючи цей інтерфейс, контекст буде незалежним від конкретних класів стратегій. З іншого боку, ви зможете змінювати та додавати нові види алгоритмів, не чіпаючи код контексту.

Структура

1. **Контекст** зберігає посилання на об’єкт конкретної стратегії, працюючи з ним через загальний інтерфейс стратегій.
2. **Стратегія** визначає інтерфейс, спільний для всіх варіацій алгоритму. Контекст використовує цей інтерфейс для виклику алгоритму.
3. **Конкретні стратегії** реалізують різні варіації алгоритму.
4. Під час виконання програми контекст отримує виклики від клієнта й делегує їх об’єкту конкретної стратегії.
5. Клієнт повинен створити об’єкт конкретної стратегії та передати його до конструктора контексту. Крім того, клієнт повинен мати можливість замінити стратегію на льоту, використовуючи сетер поля стратегії. Завдяки цьому, контекст не знатиме про те, яку саме стратегію зараз обрано.

Застосування

- Якщо потрібно використовувати різні варіації якого-небудь алгоритму всередині одного об’єкта.
- Якщо у вас є безліч схожих класів, які відрізняються лише деякою поведінкою.
- Якщо ви не хочете оголювати деталі реалізації алгоритмів для інших класів.
- Якщо різні варіації алгоритмів реалізовано у вигляді розлогого умовного оператора. Кожна гілка такого оператора є варіацією алгоритму.

Переваги:

- Гаряча заміна алгоритмів на льоту.
- Ізолює код і дані алгоритмів від інших класів.
- Заміна спадкування делегуванням.
- Реалізує *принцип відкритості/закритості*.

Недоліки:

- Ускладнює програму внаслідок додаткових класів.
- Клієнт повинен знати, в чому полягає різниця між стратегіями, щоб вибрати потрібну.

Хід виконання роботи

Використаємо паттерн Проксі (Замісник) для реалізації роботи з Проєктами. У нашому прикладі Замісник допоможе додати до програми механізм ледачої ініціалізації та кешування результатів роботи отримання всіх Проєктів.

Структура

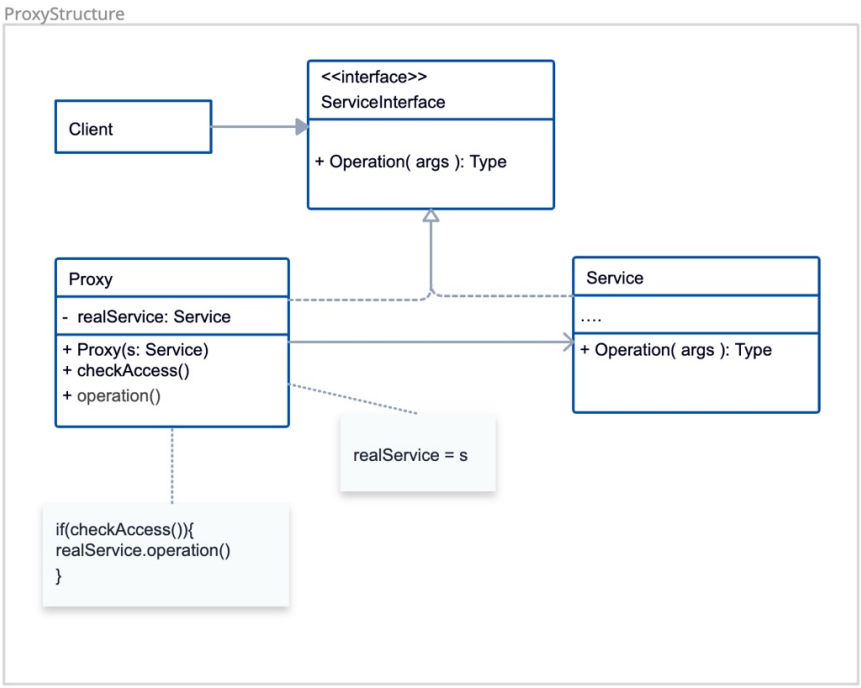

Рисунок 1. Загальна структура патерна

Кроки реалізації паттерна Замісник

1. Визначимо інтерфейс, який би зробив замісника та оригінальний об’єкт взаємозамінними.

У нашому випадку це інтерфейс IProjectRepository, який визначає необхідні методи для роботи з проектами. Цей інтерфейс використовуватиметься як для реального репозиторію (ProjectRepository), так і для замісника (ProjectProxy).

```
public interface IProjectRepository : IDisposable
{
    1 usage  2 implementations
    Task<List<Project?>> GetAllAsync();
    1 usage  2 implementations
    Task<Project?> GetByIdAsync(Guid id);
    1 usage  2 implementations
    Task<Project?> CreateProjectAsync(CreateProjectDTO project);
    1 usage  2 implementations
    Task<Project?> UpdateProjectAsync(Guid id, UpdateProjectDTO project);
    1 usage  2 implementations
    Task<bool> DeleteProjectAsync(Guid id);
}
```

Рисунок 2. Інтерфейс IProjectRepository

2. Створимо клас замісника. Він повинен містити посилання на сервісний об’єкт. Частіше за все сервісний об’єкт створюється самим замісником.

Клас замісника (ProjectProxy) буде містити посилання на реальний об'єкт сервісу — ProjectRepository. Замісник виконує додаткову обробку перед тим, як передати запит реальному репозиторію, або обробляє відповідь перед поверненням клієнту.

```
public class ProjectProxy : IProjectRepository
{
    private readonly ProjectRepository _projectRepository;
    private List<Project?>? _cachedProjects;

    public ProjectProxy(ProjectRepository projectRepository)
    {
        _projectRepository = projectRepository;
    }
```

Рисунок 3. Клас замісника (ProjectProxy)

3. Реалізуємо методи замісника в залежності від його призначення. У більшості випадків, виконавши якусь корисну роботу, методи замісника повинні передати запит сервісному об’єкту.

```
public Task<Project?> CreateProjectAsync(CreateProjectDTO project)
{
    return _projectRepository.CreateProjectAsync(project);
}

0+1 usages
public Task<Project?> UpdateProjectAsync(Guid id, UpdateProjectDTO project)
{
    return _projectRepository.UpdateProjectAsync(id, project);
}

0+1 usages
public Task<bool> DeleteProjectAsync(Guid id)
{
    return _projectRepository.DeleteProjectAsync(id);

}
```

Рисунок 4. Замісник передає запит реальному об'єкту

```
public async Task<List<Project?>> GetAllAsync()
{
    if (_cachedProjects == null)
    {
        Console.WriteLine("Loading all projects from the database...");
        _cachedProjects = await _projectRepository.GetAllAsync();
    }
    else
    {
        Console.WriteLine("Retrieving all projects from cache...");
    }

    return _cachedProjects;
}

0+1 usages
public Task<Project?> GetByIdAsync(Guid id)
{
    Console.WriteLine($"Getting project with ID: {{id}}");
    return _projectRepository.GetByIdAsync(id);
}
```

Рисунок 5. Методи замісника виконують додаткові операції

Методи замісника виконують додаткові операції, наприклад кешування для GetAllAsync, і потім передають запит до реального репозиторію.

1. **GetAllAsync** — Якщо кеш порожній, замісник викликає реальний репозиторій для завантаження проектів і зберігає їх у кеші. Якщо кеш уже заповнений, замісник повертає дані з кешу.
2. **GetByIdAsync** — Проксі без змін передає запит на отримання конкретного проекту до реального репозиторію.
3. **CreateProjectAsync** — Оскільки створення проектів не потребує кешування або інших додаткових операцій, метод просто делегує виклик реальному репозиторію.

Реалізація патерна "Замісник" дозволяє нам додавати додаткові функціональності (як кешування, лінива ініціалізація) до класу, що працює з базою даних, без зміни самого класу реального репозиторію. Клієнт може використовувати проксі, і з часом буде не помічати, коли він працює з проксі, а коли — з реальним об'єктом.

Реалізація додаткових класів

Крім того, у цій лабораторній роботі створимо клас **AppDbContext**.

```
public class AppDbContext: IdentityDbContext<AppUser, UserRole, Guid>
{

    public AppDbContext(DbContextOptions dbContextOptions):base(dbContextOptions)
    {

    }


    6 usages
    public DbSet<Project?> Projects { get; set; }
    public DbSet<Task> Tasks { get; set; }
    public DbSet<File> Files { get; set; }
    1 usage
    public DbSet<Team> Teams { get; set; }
}
```

Рисунок 6. Клас AppDbContext

Клас AppDbContext є підкласом IdentityDbContext, який надає підтримку для роботи з авторизацією та автентифікацією за допомогою ASP.NET Identity. В даному випадку клас AppDbContext використовується для роботи з базою даних, що зберігає проекти, завдання, файли та команди.

Також створимо **Data Transfer Objects** - об'єкти, які використовуються для передачі даних між компонентами додатку. Вони зазвичай мають поля, які відповідають вимогам бізнес-логіки або інтерфейсу користувача.

```
public class CreateProjectDTO
{
    [Required]
    [StringLength(255)]
    1 usage
    public string ProjectName { get; set; }

    1 usage
    public string Description { get; set; }

    [Required]
    [DataType(DataType.Date)]
    1 usage
    public DateTime StartDate { get; set; }

    [DataType(DataType.Date)]
    1 usage
    public DateTime? EndDate { get; set; }

    1 usage
    public ICollection<Guid> TeamIds { get; set; } = new List<Guid>();
}
```

Рисунок 7. Клас CreateProjectDTO

CreateProjectDTO: Це клас, який містить лише ті поля, які потрібні для створення нового проекту.

Контролер приймає цей DTO в тілі запиту ([FromBody]) і передає його до репозиторію для створення проекту. Це дозволяє чітко контролювати, які саме дані можна відправити на сервер і як ці дані будуть оброблені.

```
public class UpdateProjectDTO
{
    [StringLength(255)]
    1 usage
    public string ProjectName { get; set; }

    1 usage
    public string Description { get; set; }
}
```

Рисунок 8. Клас UpdateProjectDTO

UpdateProjectDTO: Це клас, який містить лише ті поля, які потрібні для оновлення існуючого проекту.

Також створимо **ProjectController** для роботи з проектами в API за допомогою ASP.NET Core. Контролер забезпечує CRUD-операції (створення, читання, оновлення, видалення) для проектів.

```
public class ProjectController : ControllerBase
{
    private readonly IProjectRepository _projectRepository;

    public ProjectController(IProjectRepository projectRepository)
    {
        _projectRepository = projectRepository;
    }

    [HttpGet]
    public async Task<IActionResult> GetAllProjects()
    {
        var projects :List<Project?> = await _projectRepository.GetAllAsync();
        return Ok(projects);
    }

    [HttpGet( template: "{id}")]
    2 usages
    public async Task<IActionResult> GetProjectById(Guid id)
    {
        var project = await _projectRepository.GetByIdAsync(id);
        if (project == null)
            return NotFound($"project not found");

        return Ok(project);
    }
```

Рисунок 9. Клас ProjectController

1. **Операція "Отримати всі проекти" (HTTP GET)**

Метод обробляє HTTP GET запит. У даному випадку він відповідає на запит до "api/project".

GetAllProjects(): Цей метод отримує всі проекти за допомогою асинхронного виклику _projectRepository.GetAllAsync(). Коли дані отримано, повертається статус 200 (OK) разом з переліком проектів.

2. **Операція "Отримати проект за ID" (HTTP GET)**

Метод обробляє GET запит з параметром в URL ({id}). URL буде виглядати як "api/project/{id}".

GetProjectById(Guid id): Цей метод отримує проект за його id. Якщо проект не знайдений, повертається статус 404 (Not Found) з відповідним повідомленням. Якщо проект знайдений, повертається статус 200 (OK) з проектом.

3. **Операція "Створити новий проект" (HTTP POST)**

Метод обробляє POST запит. Він очікує на дані в тілі запиту, які потрібно використати для створення нового проекту.

CreateProject([FromBody] CreateProjectDTO project): Метод отримує дані про проект (DTO) з тіла запиту і передає їх до репозиторію для створення нового проекту. Після створення проекту, повертається статус 201 (Created) з новим проектом.

4. **Операція "Оновити проект" (HTTP PUT)**

Метод обробляє PUT запит для оновлення проекту за конкретним ID.

UpdateProject(Guid id, [FromBody] UpdateProjectDTO project): Метод отримує ID проекту та нові дані для оновлення (DTO) через тіло запиту. Якщо проект знайдений і оновлений, повертається статус 200 (OK) з оновленим проектом. Якщо проект не знайдений, повертається статус 404 (Not Found).

5. **Операція "Видалити проект" (HTTP DELETE)**

Метод обробляє DELETE запит для видалення проекту за конкретним ID.

DeleteProject(Guid id): Метод видаляє проект за допомогою _projectRepository.DeleteProjectAsync(id). Якщо проект успішно видалено, повертається статус 204 (No Content). Якщо проект не знайдений, повертається статус 404 (Not Found).

Отже, цей клас дозволяє створювати RESTful API для управління проектами з підтримкою всіх основних операцій CRUD.

Висновок:

У ході виконання лабораторної роботи було здійснено ознайомлення з основними патернами проектування, їх застосуванням у реальних проектах та практичними аспектами використання цих патернів. Основною метою роботи було розібратися в основних принципах проектування, зокрема, в таких патернах як Проксі, Фабрика, Декоратор, Стратегії тощо.

У рамках лабораторної роботи я реалізував патерн Проксі для вирішення задачі з використанням репозиторіїв. Це дозволило не лише підвищити ефективність доступу до даних, а й зберегти абстракцію над реальними операціями з базою даних.

Також важливим аспектом було ознайомлення з використанням DTO (Data Transfer Object) для обміну даними між різними компонентами системи. Це дозволило забезпечити чіткий розподіл логіки обробки запитів та мінімізувати взаємодію з базою даних в контролерах.

Протягом роботи я також отримала досвід у використанні Entity Framework Core для взаємодії з базою даних, створення контексту бази та реалізації CRUD операцій для різних сутностей проекту.

В цілому, лабораторна робота допомогла мені поглибити розуміння патернів проектування, їх важливості для організації коду, а також принципів архітектури програмного забезпечення.

Вихідний код можна переглянути в GitHub у папці лабораторної роботи №4