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

Лабораторна робота № 2

із дисципліни «Технології розроблення програмного забезпечення»

Тема: «Основи проектування»

Варіант - 12

Виконав
Студент групи ІА-31: Перевірив:
Губар Б. О. Мягкий М. Ю.

Київ 2025

Зміст

1. **Мета:**

Обрати зручну систему побудови UML-діаграм та навчитися будувати діаграми варіантів використання для системи що проєктується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.

2. **Теоретичні відомості:**

Мова UML є загальноцільовою мовою візуального моделювання, яка розроблена для специфікації, візуалізації, проєктування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем. Мова UML є досить строгим та потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних та графічних 18 моделей складних систем різного цільового призначення. Ця мова увібрала в себе найкращі якості та досвід методів програмної інженерії, які з успіхом використовувалися протягом останніх років при моделюванні великих та складних систем.

Діаграма – це графічне уявлення сукупності елементів моделі у формі зв'язкового графа, вершинам і ребрам (дугам) якого приписується певна семантика. Нотація канонічних діаграм є основним засобом розробки моделей мовою UML.

У нотації мови UML визначено такі види діаграм:

- варіантів використання (use case diagram);
- класів (class diagram);
- кооперації (collaboration diagram);
- послідовності (sequence diagram);
- станів (statechart diagram);
- діяльності (activity diagram);
- компонентів (component diagram);
- розгортання (deployment diagram).

Діаграма варіантів використання (Use-Cases Diagram) – це UML діаграма за допомогою якої у графічному вигляді можна зобразити вимоги до системи, що розробляється. Діаграма варіантів використання – це вихідна концептуальна модель проєктованої системи, вона не описує внутрішню побудову системи. Діаграма використання складається з:

- Акторів - будь-які об'єкти, суб'єкти чи системи, що взаємодіють з модельованою бізнес-системою ззовні для досягнення своїх цілей або вирішення певних завдань.
- Варіантів використання - служать для опису служб, які система надає актору.
- Відношень: асоціація, узагальнення, залежність, включення, розширення.

Діаграми класів використовуються при моделюванні програмних систем найчастіше. Вони є однією із форм статичного опису системи з погляду її проєктування, показуючи її структуру. Діаграма класів не відображає динамічної поведінки об'єктів зображених на ній класів. На діаграмах класів показуються класи, інтерфейси та відносини між ними.

Діаграма класів містить у собі класи, їхні методи та атрибути, зв’язки. Методи та атрибути мають 4 модифікатори доступу: public, package, protected, private. Зв’язки налічують у собі асоціацію, агрегацію, композицію, успадкування тощо.

3. Хід роботи:

Завдання:

- Ознайомитись з короткими теоретичними відомостями.
- Проаналізувати тему та спроєктувати діаграму варіантів використання відповідно до обраної теми лабораторного циклу.
- Спроєктувати діаграму класів предметної області.
- Вибрати 3 варіанти використання та написати за ними сценарії використання.
- На основі спроєктованої діаграми класів предметної області розробити основні класи та структуру бази даних системи. Класи даних повинні реалізувати шаблон Repository для взаємодії з базою даних.
- Нарисувати діаграму класів для реалізованої частини системи.
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму варіантів використання відповідно, діаграму
- класів системи, вихідні коди класів системи, а також зображення структури бази даних

Тема : CI server (state, command, decorator, mediator, visitor, soa)

Діаграма варіантів використання

Актори:

- Розробник (Developer): Користувач, який пише код, налаштовує проєкти та переглядає результати.
- Система контролю версій (VCS - GitHub/GitLab): Зовнішня система, яка зберігає код та надсилає вебхуки про зміни.
- Агент збірки (Build Agent): Компонент, що безпосередньо виконує команди (компіляцію, тести).

Варіанти використання:

- Налаштування проєкту: Створення нового проєкту в CI, вказання репозиторію.
- Запуск збірки (Trigger Build): Ініціалізація процесу (автоматично через вебхук або вручну).
- Виконання пайплайну: Процес проходження кроків (клонування, білд, тест).
- Перегляд історії збірок: Отримання списку минулих запусків та їх статусів.

- Перегляд логів: Читання консольного виводу конкретної збірки.
- Збереження артефактів: Збереження бінарних файлів після успішної збірки.

Зв’язки:

- Розробник — Налаштування проєкту
- Розробник — Запуск збірки (ручний)
- Розробник — Перегляд історії збірок
- Розробник — Перегляд логів
- VCS — Запуск збірки (через Webhook)
- Агент збірки — Виконання пайплайну
- Виконання пайплайну — (include) — Збереження артефактів

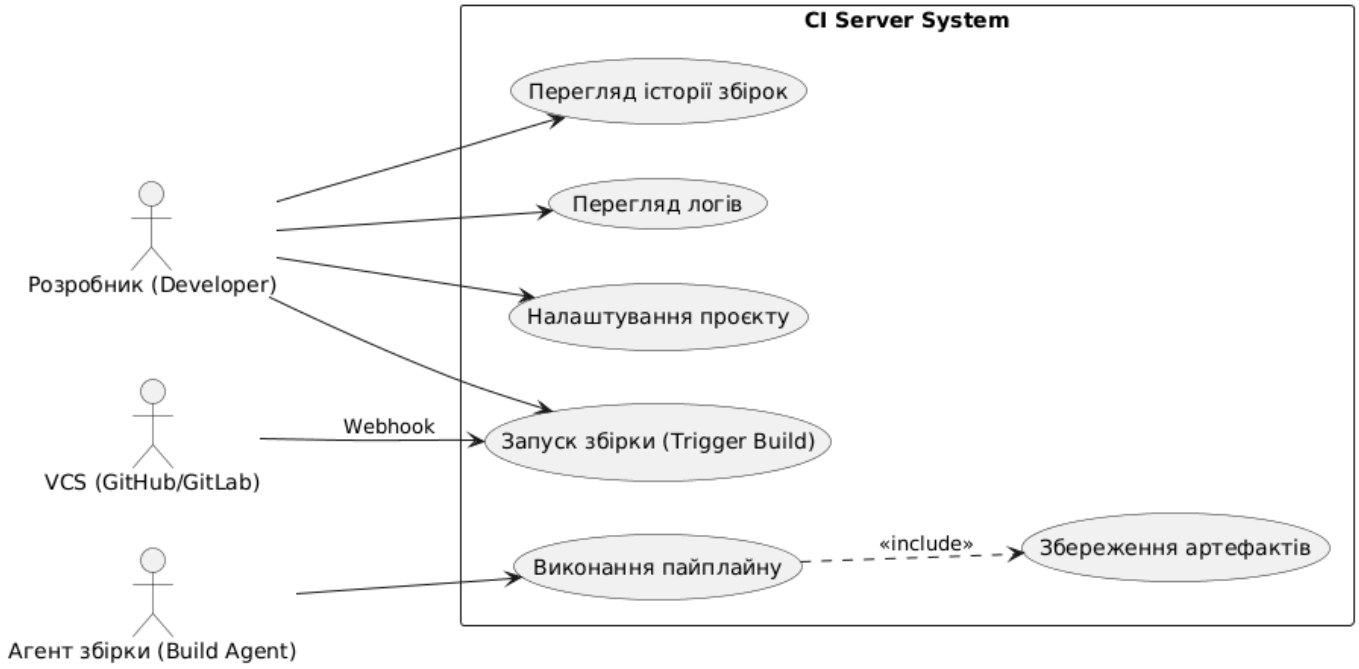

Рис. 1 - Діаграма використання

Сценарії використання:

Автоматичний запуск збірки	
Передумови	Проєкт налаштований у CI сервері, у VCS налаштований Webhook.
Постумови	Створено нову задачу (Build Job) у черзі, статус збірки "Pending".
Взаємодіючі сторони	VCS (GitHub/GitLab), CI Server.
Короткий опис	Розробник робить push у репозиторій, VCS повідомляє сервер, сервер створює задачу.

Основний потік подій	Розробник відправляє код у репозиторій.
	VCS відправляє HTTP POST запит (Webhook) на CI Server.
	CI Server валідує запит.
	CI Server створює запис Build у базі даних зі статусом Pending.
	CI Server додає задачу в чергу на виконання.
Винятки	Невалідний токен Webhook. Сервер відхиляє запит (403 Forbidden).

Виконання пайплайну	
Передумови	У черзі є задача зі статусом Pending, є вільний Агент.
Постумови	Збірка завершена зі статусом Success або Failed, логи збережені.
Взаємодіючі сторони	CI Server, Build Agent.
Короткий опис	Агент отримує задачу та виконує послідовність команд.
Основний потік подій	1. Агент запитує задачу у Сервера.
	2. Сервер змінює статус збірки на Running.
	3. Агент клонує репозиторій.
	4. Агент виконує кроки (Build, Test).
	5. Агент відправляє логи на Сервер у реальному часі.
	6. Агент завершує роботу і оновлює фінальний статус у базі даних.
Винятки	Помилка компіляції. Агент перериває виконання, ставить статус Failed і записує це в БД.

Перегляд логів	
Передумови	Збірка вже запущена або завершена, у базі є записи логів.
Постумови	Розробник бачить текстовий лог виконання.
Взаємодіючі сторони	Розробник, CI Server.
Короткий опис	Розробник відкриває сторінку збірки, щоб дізнатися причину помилки або хід виконання.
Основний потік подій	1. Розробник обирає конкретну збірку зі списку історії.
	2. Розробник натискає "View Logs".
	3. Система робить запит до БД в таблицю BuildLogs.
	4. Система відображає список рядків логу, відсортованих за часом.
Винятки	Збірка не знайдена. Система видає помилку 404.

Структура БД:

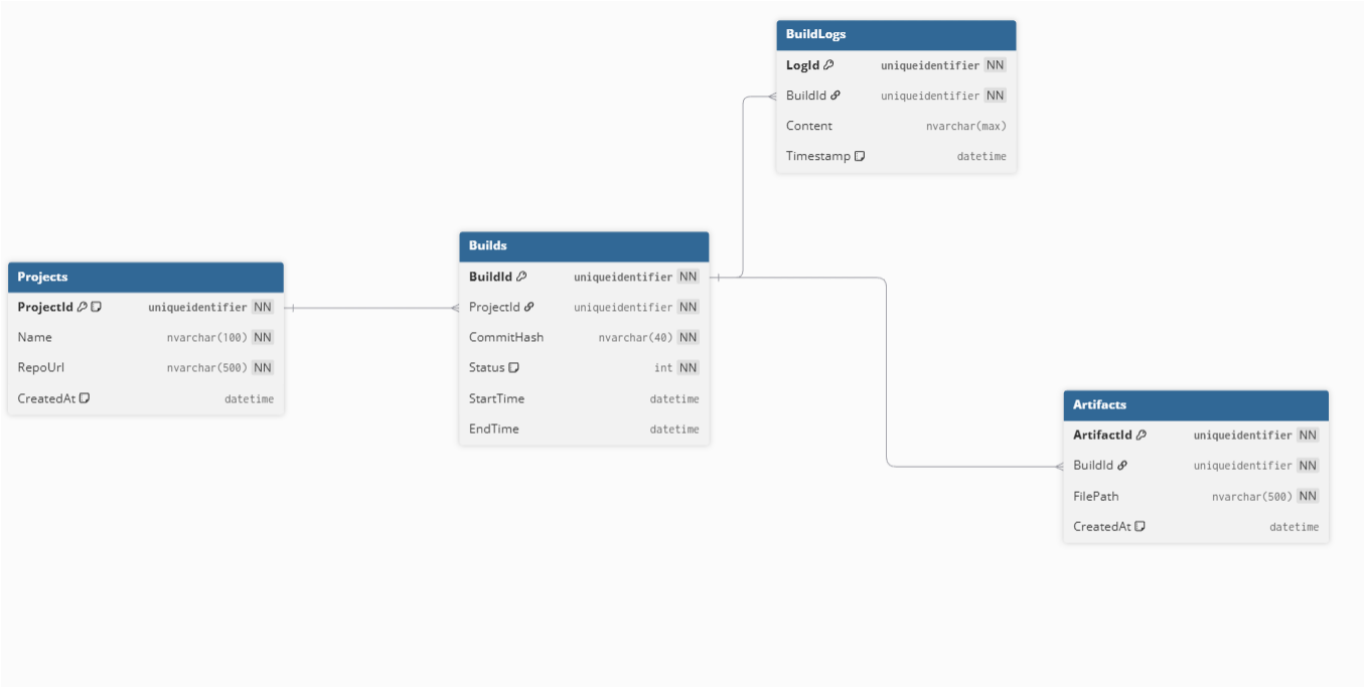

Рис. 2 - Структура БД

Projects: Головна сутність. Зберігає налаштування репозиторію.

Builds: Пов'язана з проєктом відношенням "один-до-багатьох" (один проєкт має багато збірок). Поле Status зберігається як int, що відповідатиме enum у коді C#.

BuildLogs: Логи виконання. Пов'язані з конкретним білдом. Використовується каскадне видалення (delete: cascade): якщо видалити білд, логи теж зникнуть.

Artifacts: Файли (наприклад, .exe або .zip), які були створені в результаті успішної збірки.

Діаграми класів:

Репозиторії:

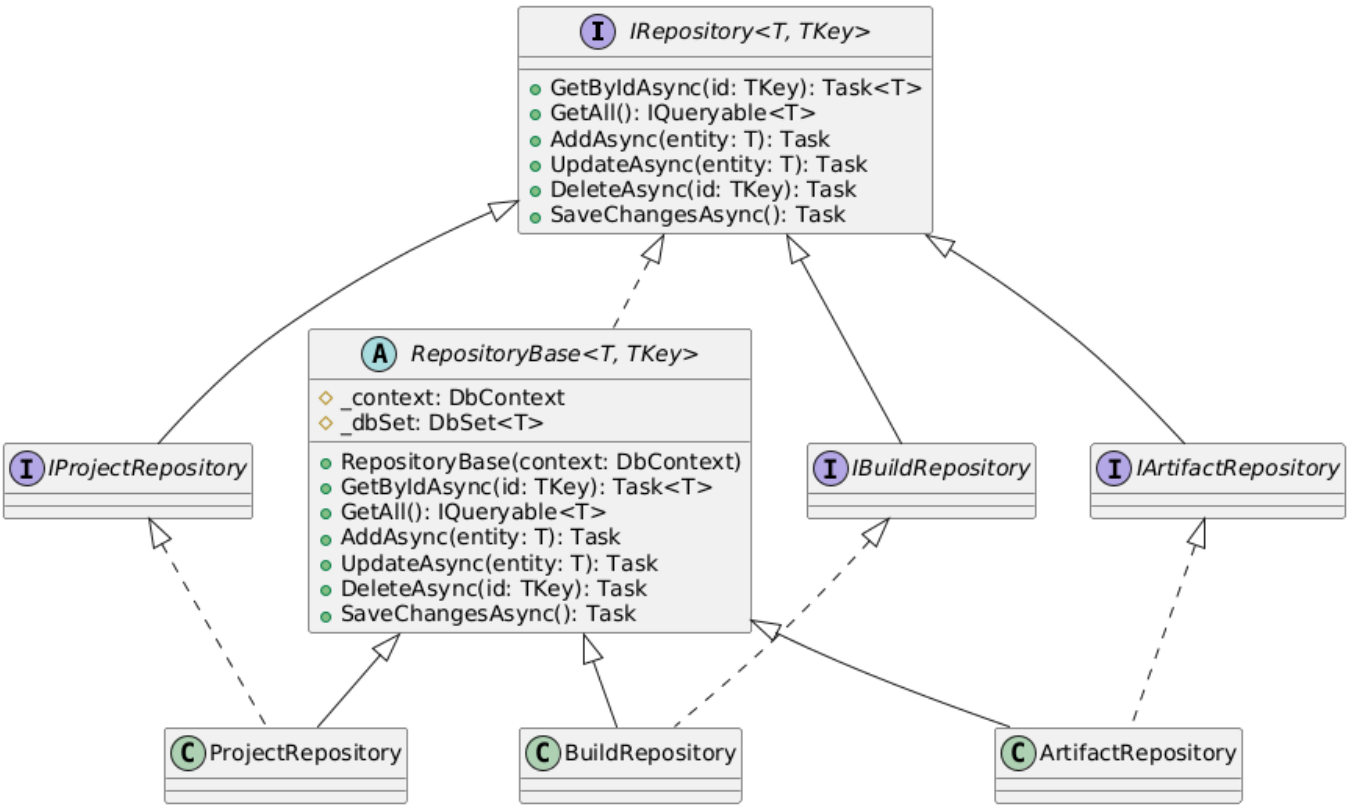

Рис. 3 - Діаграма класів репозиторіїв

IRepository<T, TKey>: Універсальний інтерфейс, що визначає контракт CRUD-операцій.

RepositoryBase<T, TKey>: Абстрактний клас, що реалізує цей інтерфейс, використовуючи DbContext (Entity Framework). Це дозволяє уникнути дублювання коду.

Concrete Repositories (ProjectRepository, etc.): Конкретні класи для кожної сутності. Вони успадковують базову логіку, але можуть містити специфічні методи (наприклад, GetBuildsByStatus).

Класи даних:

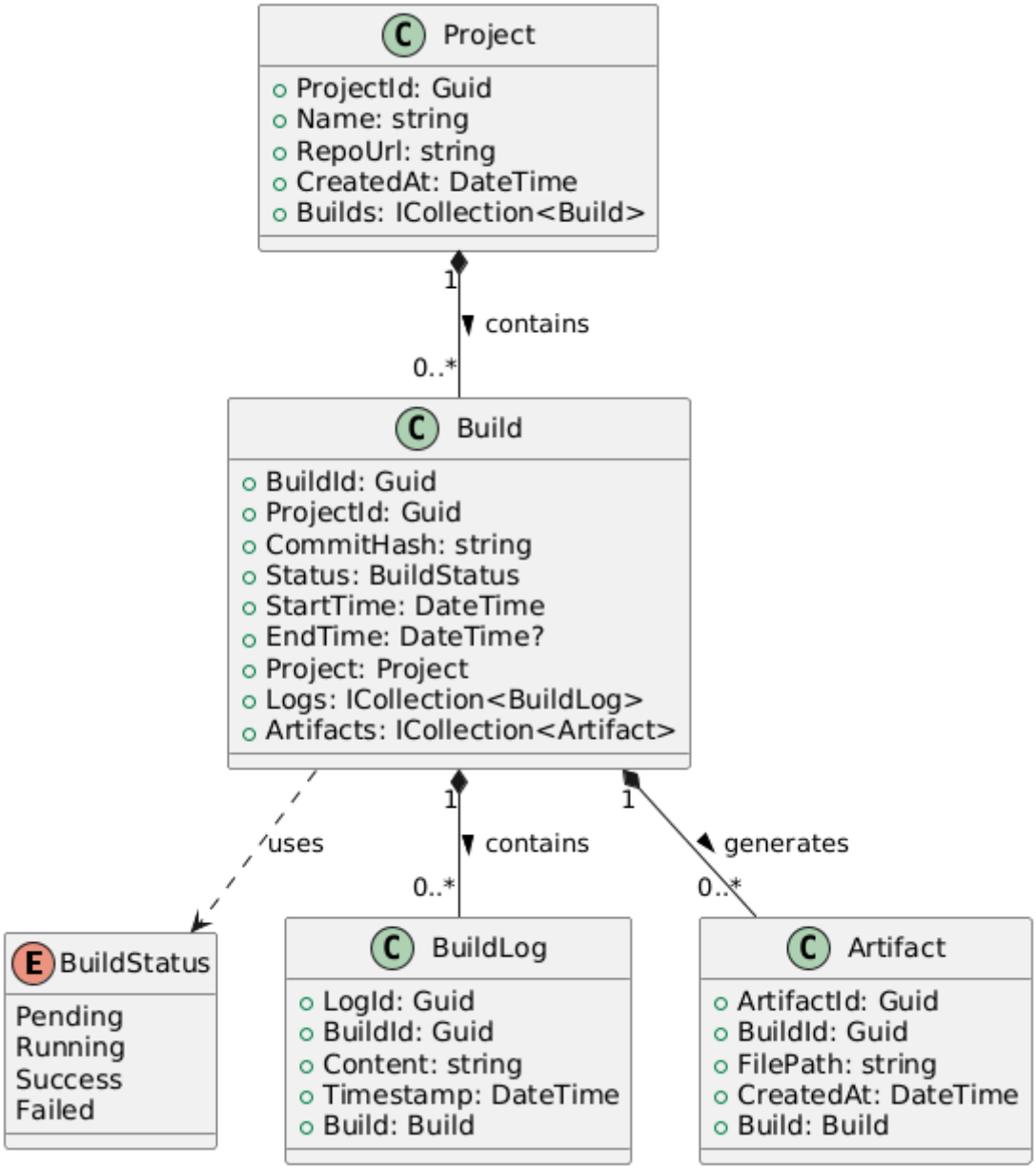

Рис. 4 - Діаграма класів даних

Асоціації (Composition *--): Ми використовуємо "зафарбований ромб" (Composition), щоб показати сильну залежність. Build не може існувати без Project, а BuildLog не має сенсу без Build.

Навігаційні властивості: Класи містять посилання один на одного (наприклад, Project має список Builds), що дозволяє легко отримувати пов'язані дані в коді (наприклад, project.Builds.Last()).

Enum: Використання перерахування для статусів робить код чистішим і безпечнішим, ніж використання простих чисел чи рядків.

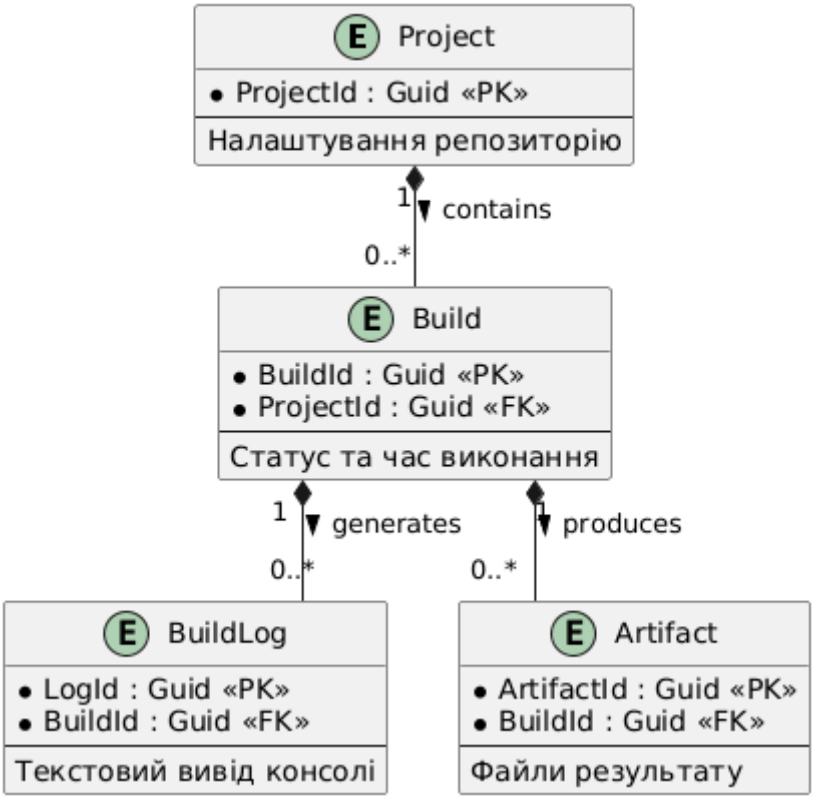

Рис. 4 - Діаграма зв’язків

Зв’язки між класами (Multiplicity)

У системі реалізовано реляційну модель даних, де сутності пов'язані між собою відношеннями "Один-до-Багатьох" (One-to-Many). Це забезпечує цілісність даних та каскадне видалення.

1. Project ↔ Build (1 : N)

- Тип: Один до багатьох (One-to-Many).
- Опис: Один Project може мати багато запусків Build. Кожен Build обов'язково належить одному конкретному проєкту.
- Реалізація:
  - У класі Project: колекція ICollection<Build> Builds.
  - У класі Build: зовнішній ключ Guid ProjectId та навігаційна властивість Project Project.
- Поведінка: При видаленні Проєкту автоматично видаляються всі його Білди (Cascade Delete).

2. Build ↔ BuildLog (1 : N)

- Тип: Композиція (Composition) / Один до багатьох.

- Опис: Один Build генерує множину записів логів BuildLog. Лог не може існувати без прив'язки до конкретного білда.
- Реалізація:
    - У класі Build: колекція ICollection<BuildLog> Logs.
    - У класі BuildLog: зовнішній ключ Guid BuildId та навігаційна властивість Build Build.

3. Build ↔ Artifact (1 : N)

- Тип: Один до багатьох.
- Опис: В результаті успішного виконання один Build може створити кілька артефактів (файлів) Artifact (наприклад: .exe, .xml звіт, .zip архів).
- Реалізація:
    - У класі Build: колекція ICollection<Artifact> Artifacts.
    - У класі Artifact: зовнішній ключ Guid BuildId та посилання на об'єкт Build Build.

Вихідний код:

```
namespace CiServer.Core.Entities;

0 references
public class Project
{
    [Key]
    0 references
    public Guid ProjectId { get; set; }
    0 references
    public string Name { get; set; } = string.Empty;
    0 references
    public string RepoUrl { get; set; } = string.Empty;
    0 references
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

    0 references
    public ICollection<Build> Builds { get; set; } = new List<Build>();
}
```

Рис. 6 - Project.cs (Сутність Проєкту)

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace CiServer.Core.Entities;
0 references
public class Build
{
    [Key]
    0 references
    public Guid BuildId { get; set; }
    [ForeignKey("Project")]
    0 references
    public Guid ProjectId { get; set; }
    0 references
    public string CommitHash { get; set; } = string.Empty;
    0 references
    public BuildStatus Status { get; set; }
    0 references
    public DateTime StartTime { get; set; }
    0 references
    public DateTime? EndTime { get; set; }
    0 references
    public Project? Project { get; set; }
    0 references
    public ICollection<BuildLog> Logs { get; set; } = new List<BuildLog>();
    0 references
    public ICollection<Artifact> Artifacts { get; set; } = new List<Artifact>();
}
```

Рис. 7 - Build.cs (Сутність Збірки)

```
namespace CiServer.Core.Interfaces;

0 references
public interface IRepository<T, TKey> where T : class
{
    0 references
    Task<T?> GetByIdAsync(TKey id);
    0 references
    IQueryable<T> GetAll();
    0 references
    Task AddAsync(T entity);
    0 references
    Task UpdateAsync(T entity);
    0 references
    Task DeleteAsync(TKey id);
    0 references
    Task SaveChangesAsync();
}
```

Рис. 8 - IRepository.cs (Інтерфейс Репозиторію)

```
namespace CiServer.Data;
2 references
public class ApplicationDbContext : DbContext
{
    0 references
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
    0 references
    public DbSet<Project> Projects { get; set; }
    0 references
    public DbSet<Build> Builds { get; set; }
    0 references
    public DbSet<BuildLog> BuildLogs { get; set; }
    0 references
    public DbSet<Artifact> Artifacts { get; set; }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Project>()
            .HasMany(p => p.Builds)
            .WithOne(b => b.Project)
            .HasForeignKey(b => b.ProjectId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

Рис. 9 - ApplicationDbContext.cs (Контекст Бази Даних)

4. Висновок

У процесі виконання лабораторної роботи з теми "CI Server" (Сервер безперервної інтеграції) було здійснено комплексний аналіз предметної області та спроєктовано архітектуру програмної системи. За допомогою мови UML розроблено моделі системи, включаючи діаграми варіантів використання для визначення функціональних вимог, діаграми класів для відображення статичної структури та схему бази даних для MS SQL Server. Практична частина роботи полягала у створенні базової архітектури коду на платформі .NET. Було реалізовано багаторівневу структуру проєкту (Core, Data, Web) з використанням патерну Repository та ORM Entity Framework Core, що забезпечує слабку зв'язність компонентів, гнучкість та можливість масштабування.

5. Питання до лабораторної роботи:

1. Що таке UML? – універсальна мова для опису та візуалізації систем.
2. Що таке діаграма класів UML? – схема з класами, їхніми полями, методами та зв’язками.
3. Які діаграми UML називають канонічними? – стандартні: класів, об’єктів, варіантів використання, послідовностей тощо.
4. Що таке діаграма варіантів використання? – показує акторів і функції, які вони виконують у системі.
5. Що таке варіант використання? – дія або функція, яку виконує користувач.
6. Які відношення можуть бути відображені на діаграмі використання? – асоціація, include, extend, узагальнення.
7. Що таке сценарій? – опис кроків взаємодії користувача з системою.
8. Що таке діаграма класів? – схема структури системи через класи та їх зв’язки.
9. Які зв’язки між класами ви знаєте? – асоціація, агрегація, композиція, наслідування, залежність.
10. Чим відрізняється композиція від агрегації? – у композиції частини не існують без цілого, в агрегації можуть.
11. Чим відрізняється зв'язки типу агрегації від зв'язків композиції на діаграмах класів? – агрегація має порожній ромб, композиція – зафарбований.
12. Що являють собою нормальні форми баз даних? – правила організації таблиць для уникнення дублювання.
13. Що таке фізична модель бази даних? Логічна? – фізична: як реально зберігається; логічна: концептуальна схема.
14. Який взаємозв'язок між таблицями БД та програмними класами? – таблиця ≈ клас, рядок ≈ об’єкт, стовпчик ≈ поле.