Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №2

з дисципліни «Теорія розробки програмного забезпечення»

Тема: «Основи проектування»

Виконав:
Студент групи ІА-34
Цап Андрій

Перевірив:
Мягкий Михайло Юрійович

Київ – 2025

Зміст

Вступ

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

У процесі розробки програмного забезпечення важливо ще на ранніх етапах визначити структуру системи, функціональні можливості, взаємодії між користувачами та компонентами. Для цього застосовується уніфікована мова моделювання UML, яка дозволяє описати архітектуру майбутньої системи у вигляді набору діаграм.

У даній лабораторній роботі виконується проектування системи “Project Management Software”.

Теоретичні відомості

Діаграма – 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.

Актором називається будь-який об'єкт, суб'єкт чи система, що взаємодіє з модельованою бізнес-системою ззовні для досягнення своїх цілей або вирішення певних завдань.

Варіанти використання **(use case)** - варіант використання служить для опису служб, які система надає актору. Інакше кажучи кожен варіант

використання визначає набір дій, здійснюваний системою під час діалогу з актором.

Відношення **(relationship)** – семантичний зв'язок між окремими елементами моделі.

Асоціація **(association)** – узагальнене, невідоме ставлення між актором та варіантом використання.Позначається суцільною лінією між актором та варіантом використання. Розрізняють ненаправлену (двонаправлену) асоціацію та однонаправлену асоціацію.

Відношення узагальнення **(generalization)** – показує, що нащадок успадковує атрибути у свого прямого батьківського елементу. Тобто, один елемент моделі є спеціальним або окремим випадком іншого елемента моделі.

Відношення залежності **(dependency)** визначається як форма взаємозв'язку між двома елементами моделі, призначена для специфікації тієї обставини, що зміна одного елемента моделі призводить до зміни деякого іншого елемента.

Відношення включення **(include)** – окремий випадок загального відношення залежності між двома варіантами використання, при якому деякий варіант використання містить поведінку, визначену в іншому варіанті використання. Графічне зображення – пунктирна стрілка з стереотипом << include>>.

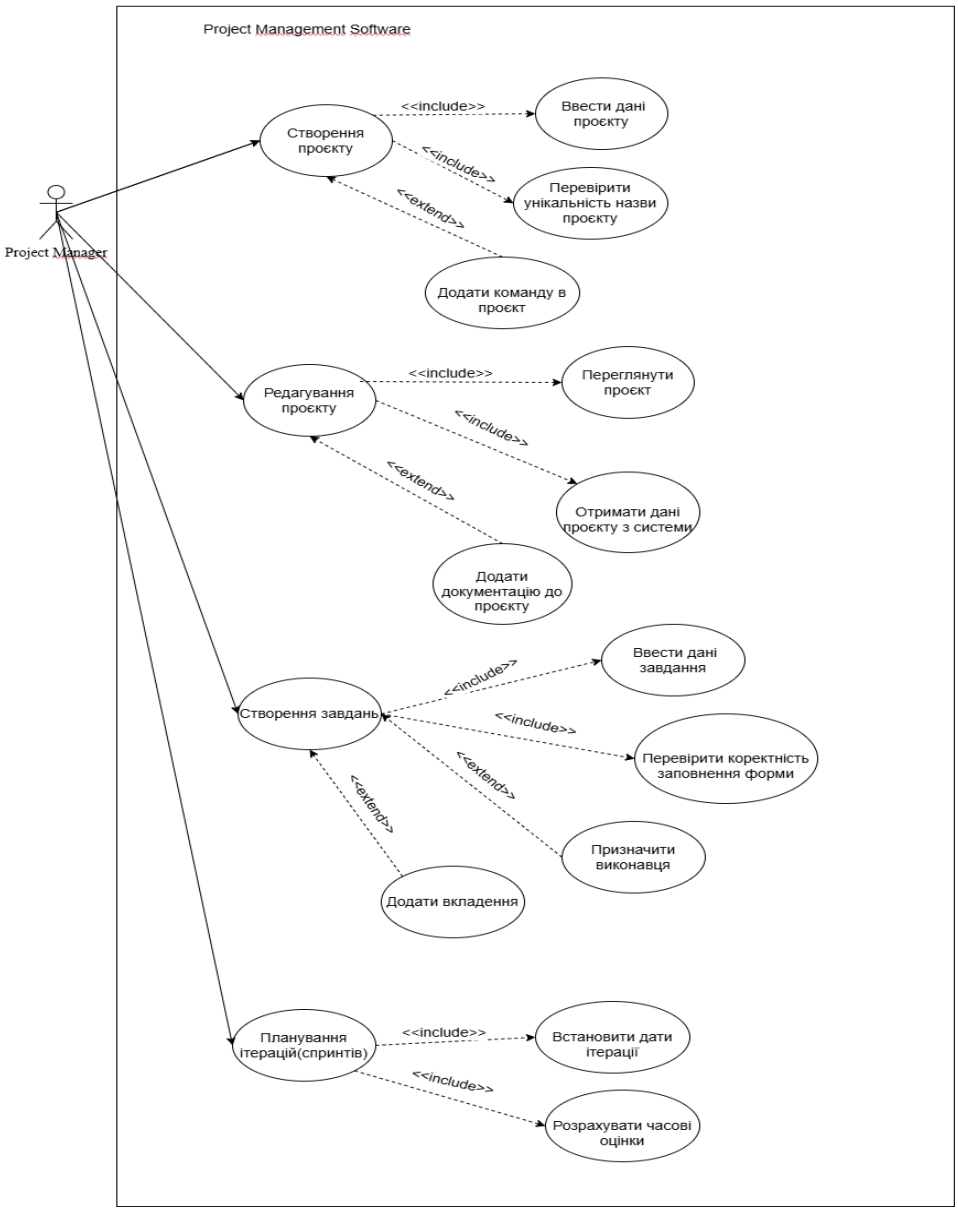
Відношення розширення **(extend)** – показує, що варіант використання розширює базову послідовність дій та вставляє власну послідовність. Графічне зображення відношення розширення – пунктирна стрілка спрямована від залежного варіанта (розширює) до незалежного варіанта (базового) з ключовим словом <extend>.

Сценарії використання – це текстові уявлення тих процесів, які відбуваються при взаємодії користувачів системи та самої системи. Вони є чітко формалізованими, покроковими інструкціями, що описують той чи інший процес у термінах кроків досягнення мети.

Розрізняють дві моделі бази даних – логічну та фізичну. Фізична модель бази даних представляє собор набір бінарних даних у вигляді файлів, структурованих та згрупованих згідно з призначенням, що використовується для швидкого та ефективного отримання інформації з жорсткого диска, а також для компактного зберігання та розміщення даних на жорсткому диску.

Хід роботи

Діаграма варіантів використання



Основним актором системи є **Project Manager**, який взаємодіє з програмним забезпеченням для управління проєктами з метою планування робіт, керування завданнями, організації спринтів. Система забезпечує створення нових проєктів, додавання завдань.

Одним з базових варіантів використання є **«Створення проєкту»**, який включає початкове введення основних даних проєкту (назва, опис,

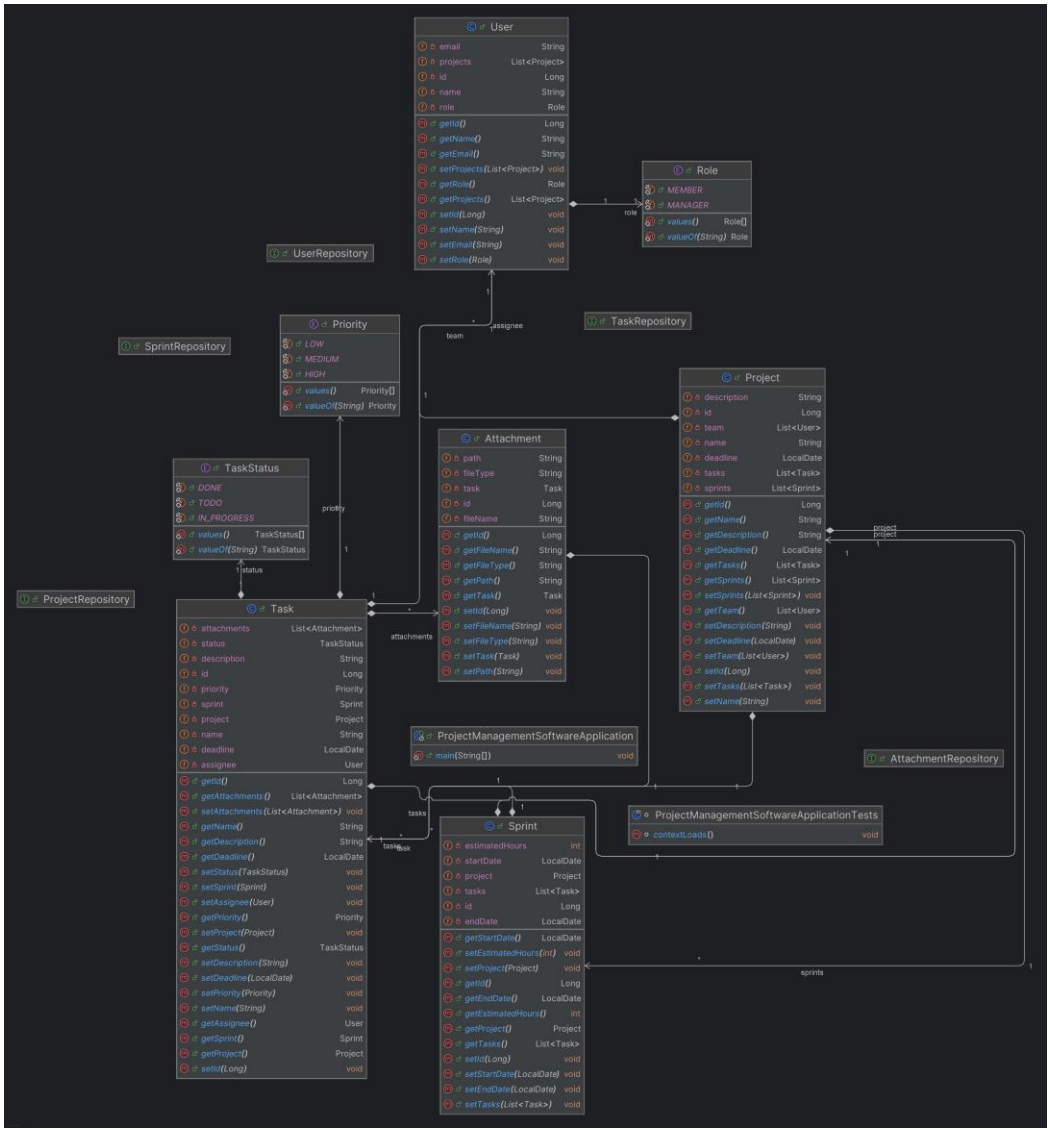
дедлайни). Цей варіант використання **включає** дію *«Ввести дані проєкту»*.

Варіант **«Редагування проєкту»** надає можливість змінювати атрибути проєкту та **включає** операцію *«Переглянути проєкт»*, що дозволяє отримати повну інформацію перед оновленнями. Розширення *«Додати завдання у проєкт»* (**extend**) дає змогу створювати завдання безпосередньо під час редагування.

Суттєвою частиною роботи менеджера є **«Створення завдання»**, яке **включає** дію *«Ввести дані завдання»* (назва, опис, пріоритет, дедлайн). Після створення завдання сценарій може бути **розширений** варіантами *«Призначити виконавця»* та *«Додати вкладення»*, що дозволяють уточнити параметри завдання та надати додаткові матеріали.

Ще одним ключовим варіантом використання є **«Планування ітерацій (спринтів)»**, яке **включає** дію *«Встановити дати ітерації»* та може **розширюватися** варіантом *«Розрахувати часові оцінки»*.

Діаграма класів предметної області



Repository Pattern

Repository<T> – це базовий підхід, що використовується для абстракції доступу до даних.
Він дозволяє відокремити логіку бізнес-процесів від логіки роботи з базою даних.

У моєму випадку Spring Data JPA автоматично генерує CRUD-операції, тому кожен репозиторій успадковує:

Основні методи:

- **save(T entity)** – збереження або оновлення об’єкта.

- **findById(Long id)** – пошук об’єкта за ідентифікатором.
- **deleteById(Long id)** – видалення об’єкта.
- **findAll()** – отримання списку всіх об’єктів.

Domain Entity

Project – модель, що описує програмний проєкт

Поля:

- **id (Long)** – унікальний ідентифікатор проєкту.
- **name (String)** – назва проєкту.
- **description (String)** – опис проєкту.
- **deadline (LocalDate)** – кінцева дата виконання.
- **tasks (List<Task>)** – список завдань, що належать цьому проєкту.
- **sprints (List<Sprint>)** – спринти, створені в рамках проєкту.
- **team (List<User>)** – користувачі, які працюють над проєктом.

Task – модель завдання

Поля:

- **id (Long)** – унікальний ідентифікатор завдання.
- **name (String)** – назва завдання.
- **description (String)** – детальний опис завдання.
- **deadline (LocalDate)** – дедлайн.
- **priority (Priority)** – пріоритет (LOW / MEDIUM / HIGH).
- **status (TaskStatus)** – статус завдання (TODO, IN_PROGRESS, DONE).
- **assignee (User)** – користувач, відповідальний за виконання.
- **project (Project)** – проєкт, до якого належить завдання.
- **attachments (List<Attachment>)** – вкладені файли, пов’язані з завданням.

Sprint – модель спринту

Поля:

- **id (Long)** – унікальний номер спринту.

- **startDate (LocalDate)** – дата початку.
- **endDate (LocalDate)** – дата завершення.
- **estimatedHours (int)** – прогнозований об’єм роботи.
- **project (Project)** – проєкт, в якому проводиться спринт.
- **tasks (List<Task>)** – завдання, включені в спринт.

Attachment – модель вкладення до завдання

Поля:

- **id (Long)** – ідентифікатор файлу.
- **fileName (String)** – назва файлу.
- **fileType (String)** – тип файлу.
- **path (String)** – шлях до збереженого файлу.
- **task (Task)** – завдання, до якого додане вкладення.

User – модель користувача

Поля:

- **id (Long)** – унікальний ідентифікатор.
- **name (String)** – ім'я користувача.
- **email (String)** – електронна пошта.
- **role (Role)** – роль користувача (MANAGER або MEMBER).

Priority – перелік можливих пріоритетів

LOW, MEDIUM, HIGH

TaskStatus – можливі стани завдання

TODO, IN_PROGRESS, DONE

Role – ролі користувачів

MANAGER, MEMBER

Сценарії варіантів використання

Сценарій 1. Створення нового проєкту

Передумови**:**

- Користувач (Project Manager) авторизований у системі.
- Користувач має роль MANAGER.

Постумови:

- Новий проєкт створено та збережено в системі.
- Проєкт з’являється у списку проєктів менеджера.
- Проєкту присвоєно порожній список завдань та спринтів.

Взаємодіючі сторони:

- Project Manager
- Project Management System

Короткий опис:

Менеджер створює новий проєкт шляхом введення необхідних даних.

Основний перебіг подій:

1. Користувач відкриває розділ “Мої проєкти”.
2. Обирає опцію «Створити проєкт».
3. Система відображає форму створення проєкту.
4. Користувач вводить:
    - назву,
    - опис,
    - дедлайн.
5. Система перевіряє коректність введених полів.

6. Система створює новий об’єкт Project та зберігає його у базі даних.
7. Система повідомляє користувача про успішне створення.
8. Створений проєкт з’являється у списку доступних проєктів.

Винятки:

Виняток 1: Поля не заповнені або містять помилки.

Система виводить повідомлення та повертає користувача до введення даних.

Сценарій 2. Створення завдання та призначення виконавця

Передумови:

- Проєкт існує.
- Project Manager або Team Member має доступ до проєкту.

Постумови:

- Створено нове завдання.
- Завдання з’являється у списку завдань проєкту.
- При призначенні виконавця поле assignee оновлюється.

Взаємодіючі сторони:

- Project Manager / Team Member
- Project Management System

Короткий опис:

Користувач створює завдання та додає до нього виконавця.

Основний перебіг подій:

1. Користувач відкриває проєкт.
2. Обирає опцію «Створити завдання».
3. Система відображає форму створення.
4. Користувач вводить:
   - назву,
   - опис,
   - пріоритет,
   - дедлайн.
5. Система перевіряє коректність даних.
6. Користувач призначає виконавця зі списку учасників проєкту.
7. Система додає завдання до проєкту та зберігає його.
8. Завдання відображається у списку задач проєкту.

Винятки:

Виняток 1: Обраний виконавець не входить до команди проєкту.

Система блокує вибір та показує повідомлення.

Сценарій 3. Додавання завдання у спринт та розрахунок часу

Передумови:

- Спринт уже створено.
- Завдання належить до того самого проєкту.
- Користувач має право редагувати спринти (MANAGER).

Постумови:

- Завдання додано до обраного спринту.
- Оновлено загальну оцінку годин (estimatedHours) у спринті.

Взаємодіючі сторони:

- Project Manager
- Project Management System

Короткий опис:

Менеджер планує спринт і розподіляє завдання, враховуючи часові оцінки.

Основний перебіг подій:

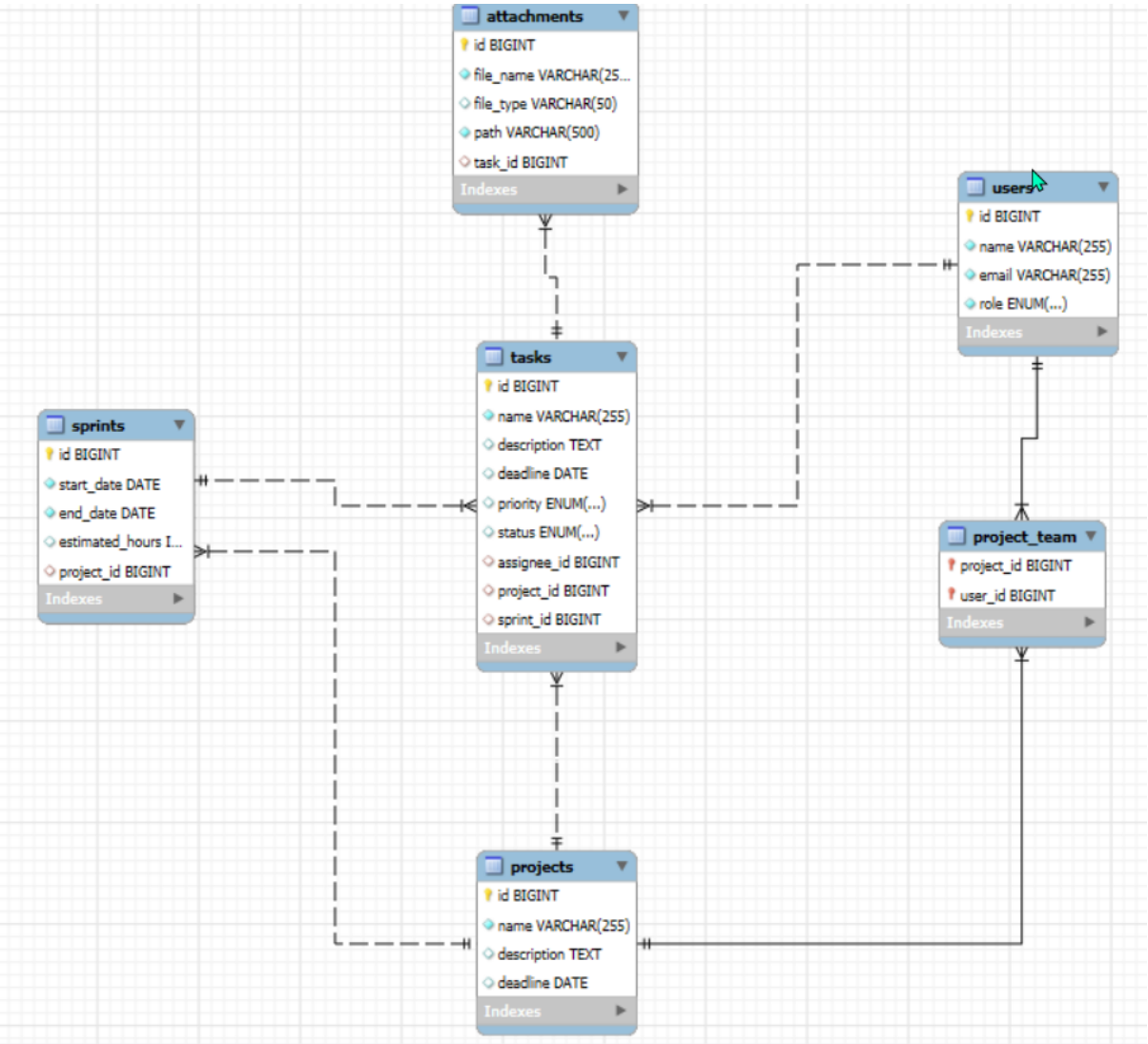
1. Менеджер відкриває сторінку спринту.
2. Обирає опцію «Додати завдання до спринту».
3. Система показує список доступних завдань проєкту зі статусом TODO.
4. Менеджер вибирає одне або кілька завдань.
5. Система пропонує ввести оцінку часу для кожного завдання (години).
6. Система автоматично оновлює поле estimatedHours спринту.
7. Завдання з’являються у списку задач спринту.

Винятки:

Виняток 1: Завдання вже належить іншому активному спринту.
Система відмовляє в додаванні.

Виняток 2: Часова оцінка не є числом або менша за 1 годину.
Система просить ввести коректне значення.

Розробка структури бази даних



Виходні коди системи

Enum Priority:

Priority.java

```
1  package org.example.projectmanagementsoftware.domain.enums;
2
3  public enum Priority {  2 usages  new *
4      LOW, MEDIUM, HIGH  no usages
5  }
6
```

Enum Role:

Role.java

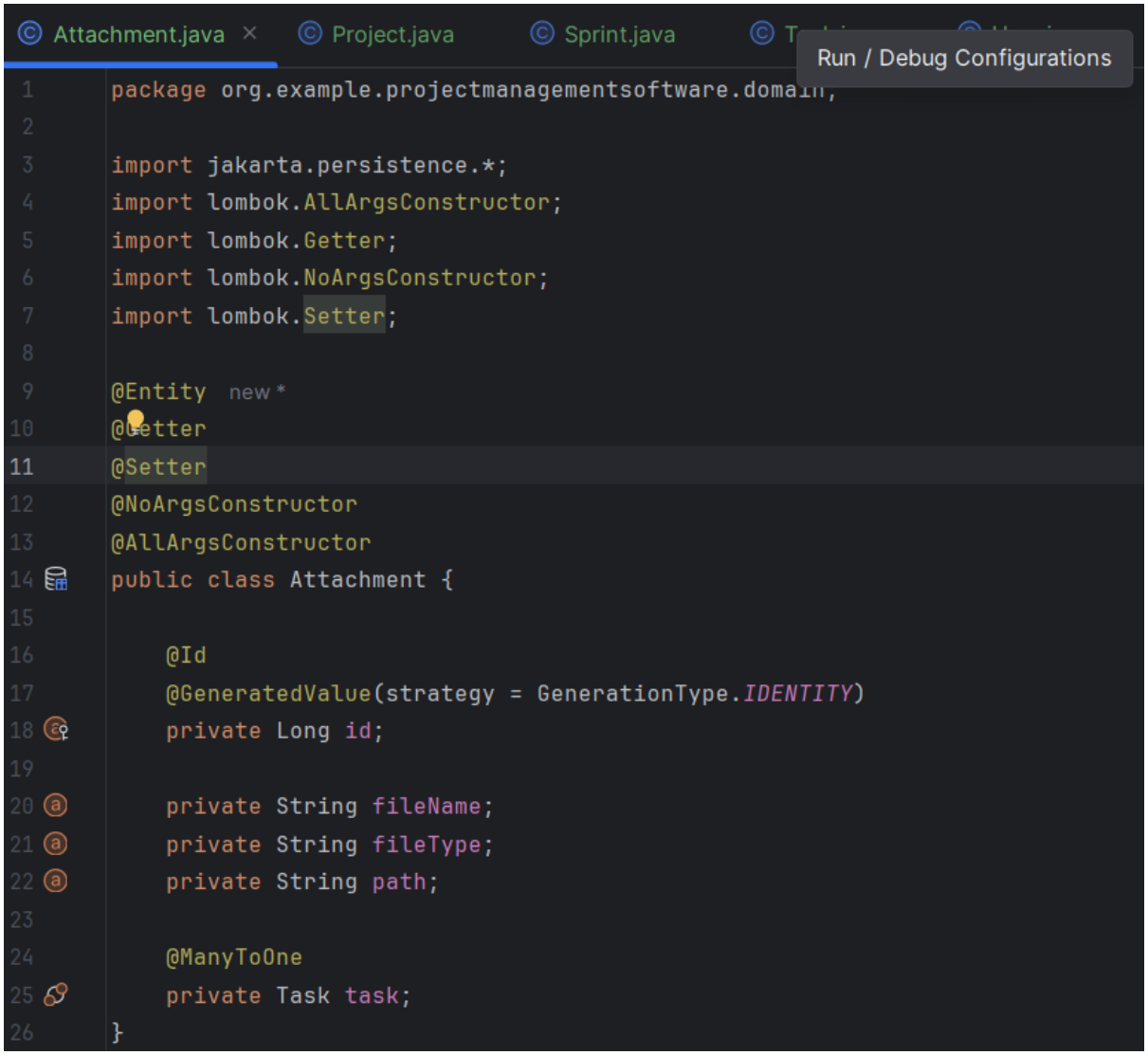
```java
package org.example.projectmanagementsoftware.domain.enums;

public enum Role {  2 usages  new *
    MANAGER, MEMBER  no usages
}

```

Enum Task Status:

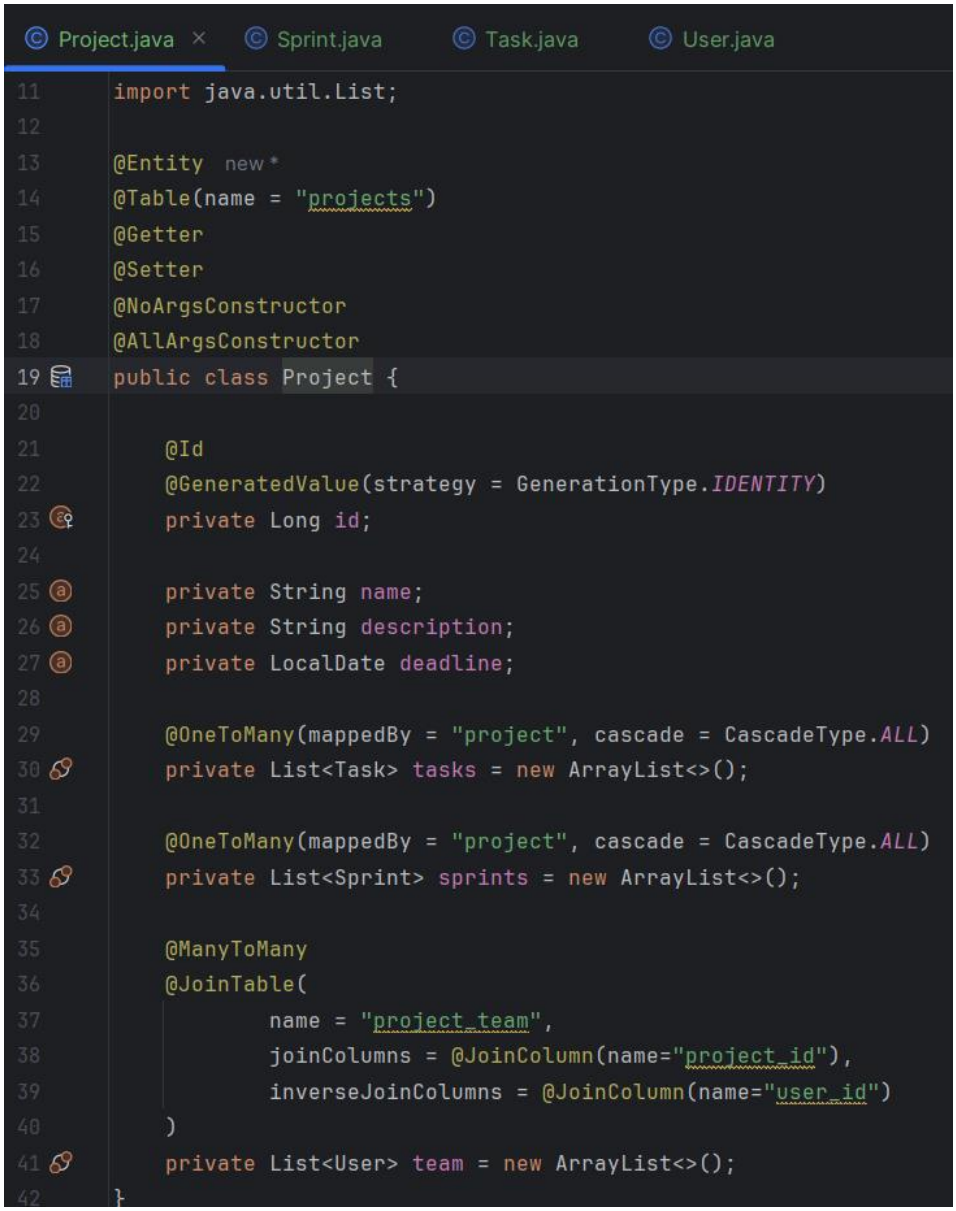
TaskStatus.java

```java
package org.example.projectmanagementsoftware.domain.enums;

public enum TaskStatus {  2 usages  new *
    TODO, IN_PROGRESS, DONE  no usages
}

```

Class Attachment:

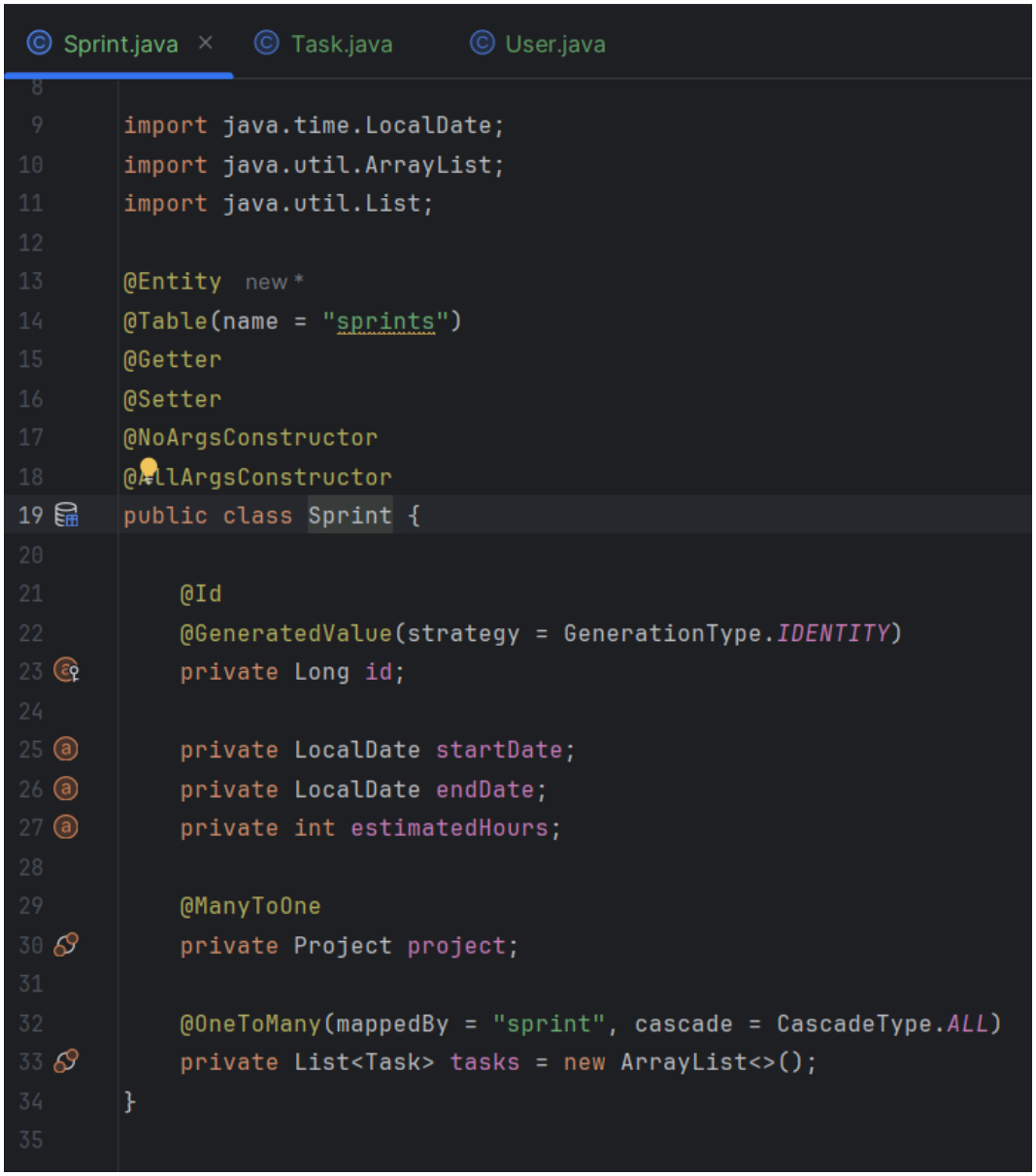


Class Project:

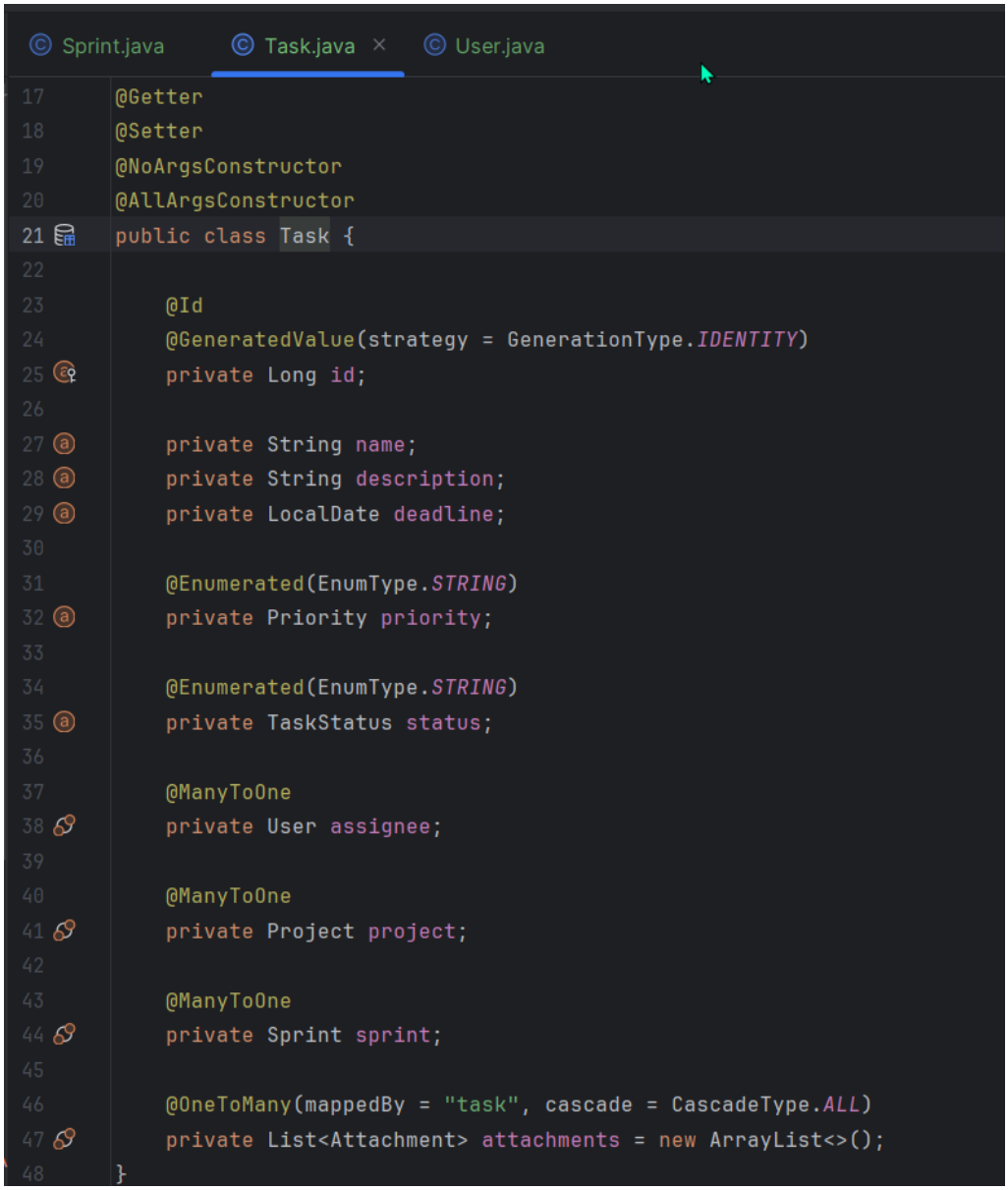


```java
import java.util.List;

@Entity  new *
@Table(name = "projects")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Project {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;
    private LocalDate deadline;

    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL)
    private List<Task> tasks = new ArrayList<>();

    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL)
    private List<Sprint> sprints = new ArrayList<>();

    @ManyToMany
    @JoinTable(
            name = "project_team",
            joinColumns = @JoinColumn(name="project_id"),
            inverseJoinColumns = @JoinColumn(name="user_id")
    )
    private List<User> team = new ArrayList<>();
}
```

Class Sprint:


Sprint.java
Task.java
User.java
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
@Entity new *
@Table(name = "sprints")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Sprint {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private LocalDate startDate;
private LocalDate endDate;
private int estimatedHours;
@ManyToOne
private Project project;
@OneToMany(mappedBy = "sprint", cascade = CascadeType.ALL)
private List<Task> tasks = new ArrayList<>();
}

Class Task:


Sprint.java
Task.java
User.java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Task {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String name;
private String description;
private LocalDate deadline;
@Enumerated(EnumType.STRING)
private Priority priority;
@Enumerated(EnumType.STRING)
private TaskStatus status;
@ManyToOne
private User assignee;
@ManyToOne
private Project project;
@ManyToOne
private Sprint sprint;
@OneToMany(mappedBy = "task", cascade = CascadeType.ALL)
private List<Attachment> attachments = new ArrayList<>();
}

Class User:

```java
import org.example.projectmanagementsoftware.domain.enums.Role

import java.util.ArrayList;
import java.util.List;

@Entity  new *
@Table(name = "users")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    @Enumerated(EnumType.STRING)
    private Role role;

    @ManyToMany(mappedBy = "team")
    private List<Project> projects = new ArrayList<> ();
}
```

Interface AttachmentRepository:

```java
package org.example.projectmanagementsoftware.repository;

import org.example.projectmanagementsoftware.domain.Attachment;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AttachmentRepository extends JpaRepository<Attachment, Long> {

}
```

Interface ProjectRepository:

ProjectRepository.java × SprintRepository.java TaskRepository.java UserRe

```java
package org.example.projectmanagementsoftware.repository;

import org.example.projectmanagementsoftware.domain.Project;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProjectRepository extends JpaRepository<Project, Long> {

}
```

Interface SprintRepository:

SprintRepository.java × TaskRepository.java UserRepository.java

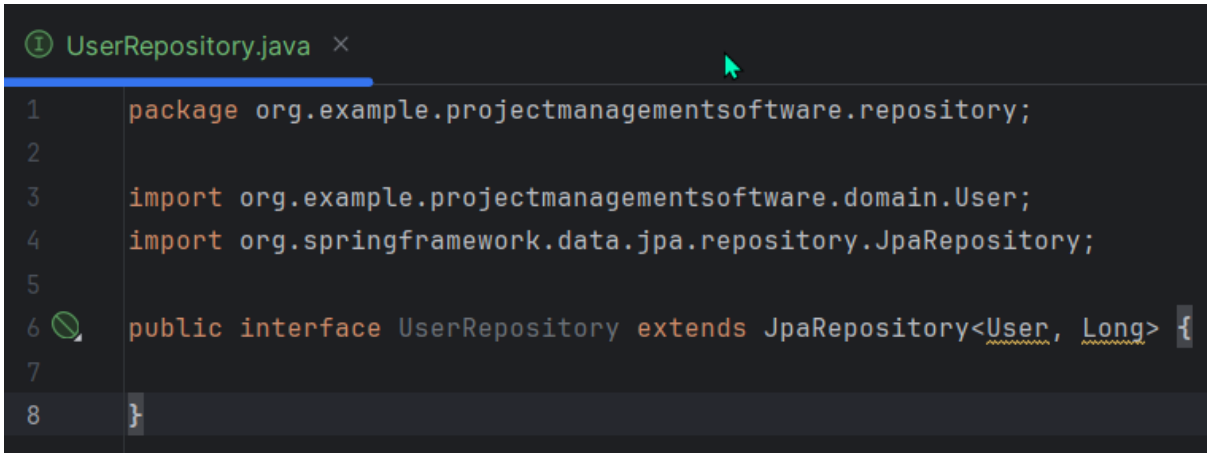
```java
package org.example.projectmanagementsoftware.repository;

import org.example.projectmanagementsoftware.domain.Sprint;
import org.springframework.data.jpa.repository.JpaRepository;

public interface SprintRepository extends JpaRepository<Sprint, Long> {

}
```

Interface TaskRepository:

TaskRepository.java × UserRepository.java

```java
package org.example.projectmanagementsoftware.repository;

import org.example.projectmanagementsoftware.domain.Task;
import org.springframework.data.jpa.repository.JpaRepository;

public interface TaskRepository extends JpaRepository<Task, Long> {

}
```

Interface UserRepository:


UserRepository.java
1 package org.example.projectmanagementsoftware.repository;
2
3 import org.example.projectmanagementsoftware.domain.User;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface UserRepository extends JpaRepository<User, Long> {
7
8 }

Висновок

У ході виконання лабораторної роботи було розглянуто основи побудови UML-діаграм та застосування їх у процесі проєктування системи управління проєктами. Було створено діаграму варіантів використання, діаграму класів предметної області та реалізованої частини системи.

На основі діаграми класів була розроблена структура бази даних та реалізовано основні класи з використанням шаблону Repository.

Виконана робота дозволила закріпити практичні навички моделювання, структурування системи та документування проєктних рішень.

Контрольні запитання

1. **Що таке UML?**

UML (Unified Modeling Language) — уніфікована мова моделювання, яка використовується для візуалізації, опису, проєктування та документування програмних систем.

2. **Що таке діаграма класів UML?**

Діаграма класів — це структурна діаграма UML, яка показує класи системи, їх атрибути, методи та зв’язки між ними.

3. **Які діаграми UML називають канонічними?**

Канонічні (основні) діаграми UML — це ті, що входять до стандартного набору UML. Вони поділяються на:

Структурні: діаграма класів, об’єктів, компонентів, розгортання, пакетів.

Поведінкові: діаграма варіантів використання, діяльності, станів, послідовності, комунікації тощо.

4. **Що таке діаграма варіантів використання?**

Це діаграма, яка показує взаємодію користувачів (акторів) із системою через різні варіанти використання (use cases) — тобто, які функції системи доступні користувачу.

5. **Що таке варіант використання?**

Варіант використання — це опис певної функціональної можливості системи, яку виконує користувач (актор), щоб досягти своєї мети.

6. **Які відношення можуть бути відображені на діаграмі використання?**

Include (включення) — один варіант завжди виконує інший.

Extend (розширення) — додатковий варіант виконується лише за певних умов.

Generalization (узагальнення) — актор або варіант використання наслідує властивості іншого.

7. **Що таке сценарій?**

Сценарій — це конкретна послідовність дій, яка описує, як саме актор взаємодіє із системою для реалізації певного варіанту використання.

8. **Що таке діаграма класів?**

Діаграма класів відображає структуру системи у вигляді класів, їх атрибутів, методів і зв’язків між ними (асоціації, наслідування, залежності тощо).

9. **Які зв’язки між класами ви знаєте?**

Асоціація — зв’язок між об’єктами двох класів.

Агрегація — “ціле–частина”, але частини можуть існувати окремо.
Композиція — “ціле–частина”, але частини не можуть існувати без цілого.

Наслідування (узагальнення) — один клас успадковує властивості іншого.

Залежність — один клас використовує інший тимчасово.

10. **Чим відрізняється композиція від агрегації?**

Композиція — сильний зв’язок: якщо знищується ціле, знищуються і частини.

Агрегація — слабкий зв’язок: частини можуть існувати незалежно.

11. **Чим відрізняється зв'язки типу агрегації від зв'язків композиції на діаграмах класів?**

Агрегація позначається порожнім ромбом біля “цілого”.

Композиція позначається заповненим ромбом біля “цілого”.

12. **Що являють собою нормальні форми баз даних?**

Нормальні форми — це правила, які допомагають структурувати таблиці бази даних для усунення надлишкових даних і забезпечення цілісності.

1НФ — кожен атрибут містить одне значення;
2НФ — кожен неключовий атрибут повністю залежить від ключа;
3НФ — відсутні транзитивні функціональні залежності неключових атрибутів від ключових

13. **Що таке фізична модель бази даних? Логічна?**

Логічна модель — опис структури даних (таблиці, зв’язки, ключі) незалежно від конкретної СУБД.

Фізична модель — конкретна реалізація логічної моделі в певній СУБД (типи даних, індекси, схеми, оптимізація).

14. **Який взаємозв'язок між таблицями БД та програмними класами?**

Кожна таблиця бази даних зазвичай відповідає класу у програмі. Рядки таблиці — це об’єкти класу. Стовпці таблиці — це поля (атрибути) класу. Зв’язки між таблицями — відповідають асоціаціям між класами.