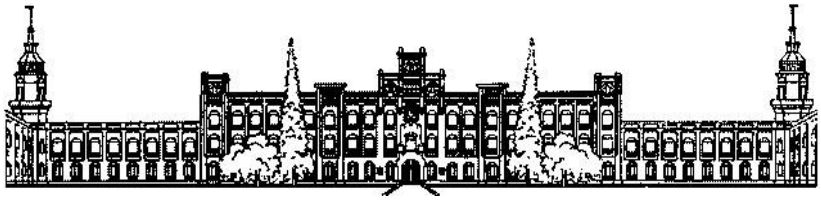

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота №2

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Основи проектування»

Виконав:

студент групи ІА-32

Нагорний Максим

Перевірив:

Мягкий М. Ю.

Київ 2025

Зміст

Вступ

Тема: Основи проектування.

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.

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.

14. **Архіватор** (strategy, adapter, factory method, facade, visitor, p2p)

Архіватор повинен являти собою візуальний додаток з можливістю створення і редагування архівів різного типу (.tar.gz, .zip, .rar, .ace) – додавання/ видалення файлів / папок, редагування метаданих (по можливості), перевірка checksum архівів, тестування архівів на наявність пошкоджень, розбиття архівів на частини.

Теоретичні відомості

UML (Unified Modeling Language) — це уніфікована мова моделювання, призначена для візуального опису, проєктування та документування програмних систем. Вона використовується як стандартний засіб для створення моделей програмного забезпечення, які допомагають розробникам, аналітикам та замовникам краще розуміти структуру, поведінку та взаємодію елементів майбутньої системи. UML не є мовою програмування, а виступає засобом графічного представлення основних аспектів системи.

Основна мета UML полягає у створенні спільної візуальної мови, яку могли б розуміти всі учасники процесу розробки. Моделі UML дають змогу зменшити складність великих систем, показати логічні взаємозв'язки між компонентами, а також використовувати ці моделі як базу для реалізації програмного коду. Вони допомагають виявити помилки ще на етапі проєктування, що робить процес створення програмного забезпечення більш ефективним.

Одним із найпоширеніших видів UML-діаграм є діаграма варіантів використання (Use Case Diagram). Вона відображає функціональні можливості системи з точки зору користувача і показує, які саме дії (варіанти використання) може виконувати кожен тип користувача (актор). Кожен актор представляє зовнішню сутність — це може бути реальна людина, інша система або пристрій, який взаємодіє з розроблюваною системою. Варіанти використання описують конкретні функції, які доступні акторові, наприклад: «увійти в систему», «створити документ», «зберегти архів» тощо. Діаграма варіантів використання дозволяє побачити систему «ззовні» — тобто так, як її бачить користувач.

Для кожного варіанту використання розробляються сценарії, що детально описують послідовність дій, які виконуються актором і системою для досягнення певної мети. Сценарій варіанта використання зазвичай містить основний потік подій, який описує звичайну ситуацію виконання дії, а також альтернативні потоки, що показують, як система реагує на помилки або виняткові випадки. Наприклад, для сценарію «Авторизація користувача» основний потік описує введення правильних даних і вхід у систему, тоді як альтернативний може описувати ситуацію, коли введено неправильний пароль.

Ще одним важливим типом UML-діаграм є діаграма класів (Class Diagram). Вона належить до структурних діаграм і використовується для відображення внутрішньої структури системи. Діаграма класів показує, з яких класів складається система, які атрибути і методи вони мають, а також які зв'язки існують між ними. Клас — це основний будівельний елемент

об’єктно-орієнтованого проєктування, який описує певний тип об’єктів, їх властивості (атрибути) і поведінку (операції). Зв’язки між класами можуть бути різного типу: асоціації, агрегації, композиції або успадкування. Наприклад, клас «Архів» може містити список об’єктів типу «Файл», що вказує на композиційний зв’язок «ціле — частина». Такі діаграми дозволяють розробникам зрозуміти логічну структуру програми і полегшують подальше програмування.

UML-діаграми створюються за допомогою спеціальних програмних засобів. Сьогодні існує багато систем для побудови UML-моделей, які відрізняються за функціональністю та зручністю використання. Серед найпопулярніших можна назвати StarUML, Visual Paradigm, Draw.io (Diagrams.net), Lucidchart, PlantUML та Enterprise Architect. Наприклад, StarUML є безкоштовним і зручним інструментом для створення базових UML-діаграм, а Visual Paradigm надає додаткові можливості, такі як генерація коду з діаграм або інтеграція з IDE. Онлайн-інструменти, такі як Draw.io або Lucidchart, дозволяють швидко створювати діаграми у браузері без встановлення додаткового програмного забезпечення.

Моделювання систем за допомогою UML має велике значення в сучасній розробці програмного забезпечення. Воно забезпечує візуальне уявлення про майбутню систему, дозволяє формалізувати вимоги, полегшує комунікацію між членами команди та дає змогу ефективно планувати архітектуру. Діаграми варіантів використання допомагають зрозуміти функціональні можливості системи, а діаграми класів — її структурну організацію. Вміння розробляти такі діаграми є невід’ємною частиною професійної підготовки розробника та аналітика.

Хід виконання

Діаграма варіантів використання

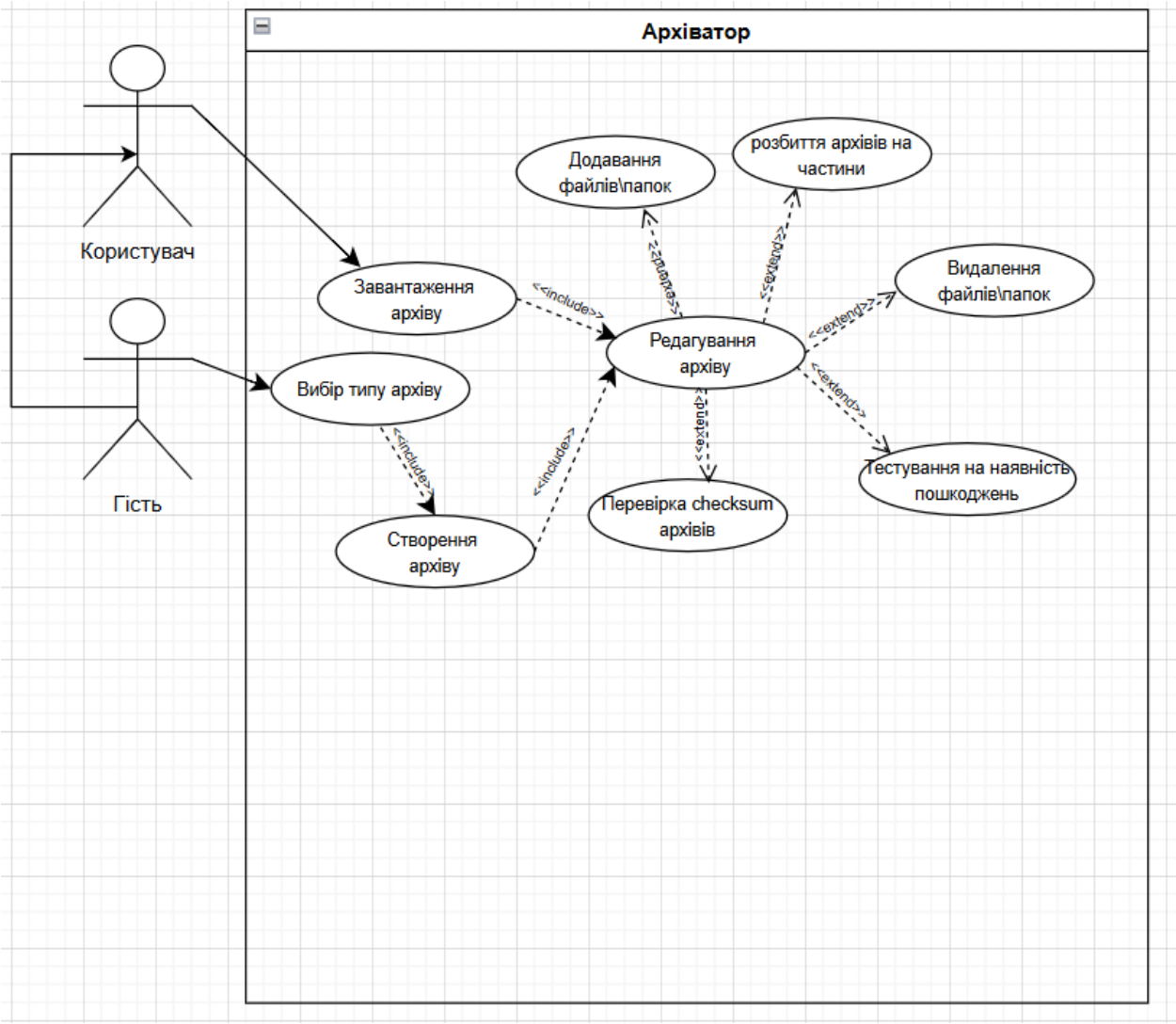

Рис. 1. Діаграма варіантів використання (Use-Cases Diagram)

На рисунку 1 представлена діаграма варіантів використання, що ілюструє функціональні можливості Архіватора.

На діаграмі визначено дві ролі:

- Гість – актор системи з обмеженими правами. Має можливість створення та редагування архіву, проте не може відкривати збережені архіви.
- Користувач – актор системи без обмежених прав. Актори цієї ролі не мають обмеження в правах, та можуть створювати, редагувати та відкривати збережені архіви

Діаграма описує наступні функціональні можливості:

- Вибір типу архіву – ця функція є обов’язковою для створення архіву, доступна всім акторам системи.
- Створення архіву – ця функція описує процес ініціалізації та створення нового архіву. Ця функція доступна всім акторам.
- Завантаження архіву – ця функція описує відкриття вже існуйочого архіву з подальшою можливістю редагування, ця функція доступна лише авторизованим акторам.
- Редагування архіву – ця функція складається з декількох підфункцій, що доступні всім акторам.
- Додавання файлів\папок – ця функція надає можливість додавання елементів до архіву, доступна всім акторам.
- Перевірка контрольної суми – ця функція дозволяє виконати перевірку цілістності архіву, доступна всім акторам.
- Видалення файлів\папок – ця функція надає можливість видаляти елементи архіву, доступна всім користувачам.
- Тестування на наявність помилок – ця функція дозволяє виконати аналіз архіву на наявність помилок.
- Розбиття архіву на частини – ця функція надає можливість розбити архів на декілька менших архівів.

Сценарії використання

1) Вхід гостя в систему архіватора
   Передумови: Немає.
   Постумови: У разі успішного виконання, гість входить до системи. У протилежному випадку система не змінюється.
   Взаємодіючі сторони: Гість , система архіватора.
   Короткий опис: Цей варіант використання визначає вхід користувача до системи архіватора.
   Основний перебіг подій: Цей варіант використання починає виконуватися, коли користувач хоче увійти до системи архіватора.
   1. Система запитує ім’я користувача та пароль.
   2. Користувач вводить ім’я та пароль.
   3. Система перевіряє ім’я та пароль, після чого відривається доступ до систем, якщо ім’я та пароль не правильні, Виняток.

   Виняток: Неправильне ім’я\пароль. Якщо під час виконання Основного потоку виявиться, що користувач ввів неправильне ім'я та пароль, система виводить повідомлення про помилку. Користувач може повернутися до початку основного потоку або відмовитись від входу в систему, при цьому виконання варіанта використання завершується.

Примітки: Відступні

2) Створення архіву
   Передумови: користувач повинен обрати тип архіву.
   Постумови: У разі успішного виконання, гість\користувач створює новий архіву. У протилежному випадку система не змінюється.
   Взаємодіючі сторони: Гість\Користувач, система архіватора.
   Короткий опис: Цей варіант використання визначає створення архіву.
   Основний перебіг подій: Цей варіант використання починає виконуватися, коли користувач хоче створити новий архів.
   1. Користувач обирає тип архіву.
   2. Користувач натискає кнопку «Створити новий архів»

   Якщо архів не створюється, Виняток

   Виняток: Не обраний тип архіву. Якщо під час натискання кнопки «Створити новий архів» виявиться, що тип архіву не обраний, система виводить повідомлення про помилку, користувач\гість може обрати тип архіву та створити його, при цьому варіанти використання завершуються.

3) Відкриття існуючого архіву
   Передумови: Виконаний вхід в систему, збережений архів.
   Постумови: У разі успішного виконання, користувач відкриє вже існуючий архів.
   Взаємодіючі сторони: Користувач, система архіватора.
   Короткий опис: Цей варіант використання визначає відкриття існуйочого архіву.
   Основний перебіг подій:
   1. Система запитує ім’я користувача та пароль.
   2. Користувач вводить ім’я та пароль.
   3. Система перевіряє ім’я та пароль, після чого відривається доступ до системи.
   4. Користувач обирає серед існуючих архівів потрібний.
   5. Користувач відкриває його, якщо архів не відкривається, Виняток.

   Виняток: Не відкривається існуючий архів. Якщо під час відкриття архіву, виявиться, що він не відкривається, виводиться повідомлення про помилку, що архів пошкоджено або він був видалений.

Діаграма класів

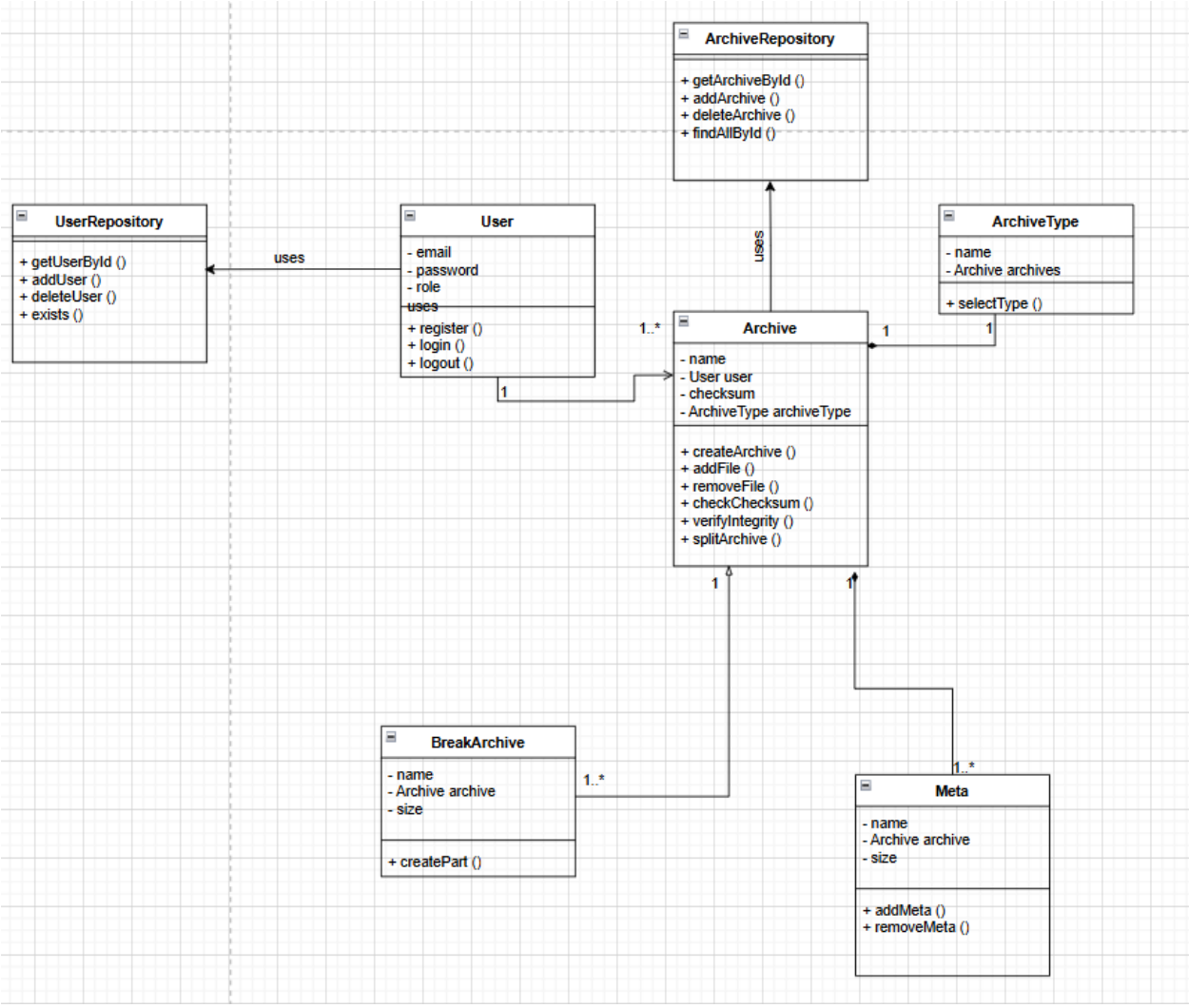

Рис. 2. Діаграма класів

Ця діаграма класів ілюструє структура системи архіватора, яка складається з 6 стуностей.

Клас User включає в себе атрибути email, password для реєстрації та авторизації і role для визначення прав. Має зв’язок типу асоціація з класом Archive (один до багатьох) один користувач може мати декілька архівів, проте архів може належати лише одному користувачеві.

Клас Archive включає в себе атрибут name та залежність від кориситувача (User user).

Клас BreakArchive відповідає за розбиття архіву на частини, та включає в себе такі атрибути, як name та size.Має зв’язок з класом Archive(багато до одного) типу наслідуваня, один архів може розбитися на декілька менших.

Клас Meta відповідає за збереження файлів, що знаходяться в архіві, містить такі атрибути, як name та size, та має зв’язок з класом Archvie (багато

до одного) типу композиції адже багато файлів можуть існувати в одному архіві, проте файл е може бути в декількох архівах.

Клас ArchiveType відповідає за тип архіву, містить такі атрибути, як name та має залежність з класом Archive (один до одного) типу композиція, адже один тип архіву до одного архіву.

Структура бази даних

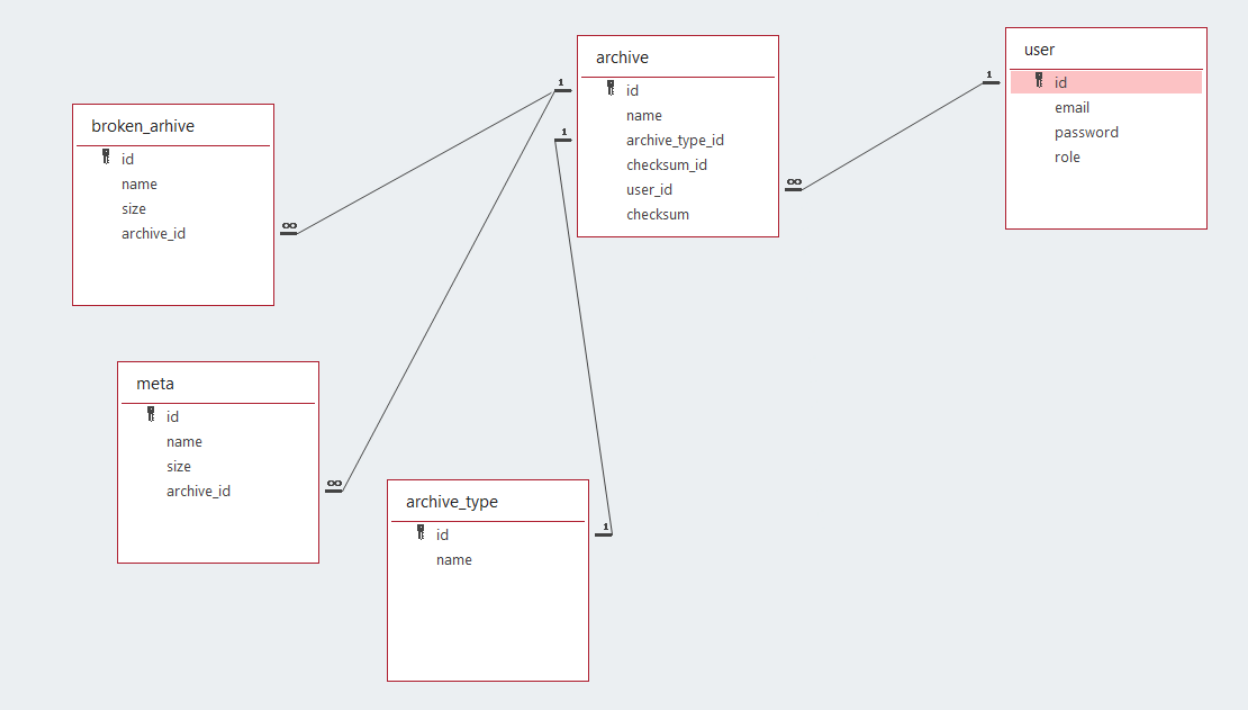

Рис. 3. Структура бази даних

Код програми

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class LabApplication {
	public static void main(String[] args) {
		SpringApplication.run(LabApplication.class, args);
		System.out.println("Hello!");
	}
}
import jakarta.persistence.*;
import java.util.Set;
```

```
@Entity
@Table(name = "archive")
public class Archive {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(nullable = false)
    private String name;
    @Column(name = "checksum_value")
    private String checksum;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "archive_type_id", nullable = false)
    private ArchiveType archiveType;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    private User user;
    @OneToMany(mappedBy = "archive", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<BreakArchive> breakArchive;
    @OneToMany(mappedBy = "archive", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<Meta> files;
    public Archive() {}
    public String getChecksum() {
        return checksum;
    }
    public void setChecksum(String checksum) {
        this.checksum = checksum;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public ArchiveType getArchiveType() {
        return archiveType;
    }
    public void setArchiveType(ArchiveType archiveType) {
        this.archiveType = archiveType;
    }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
    public Set<BreakArchive> getVolumes() {
        return breakArchive;
    }
    public void setVolumes(Set<BreakArchive> breakArchive) {
        this.breakArchive = breakArchive;
    }
    public Set<Meta> getFiles() {
        return files;
    }
    public void setFiles(Set<Meta> files) {
        this.files = files;
    }
}import jakarta.persistence.*;
import java.util.Set;
@Entity
@Table(name = "archive_type")
public class ArchiveType {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(unique = true, nullable = false)
    private String name;
    @OneToMany(mappedBy = "archiveType", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
    private Set<Archive> archive;
    public ArchiveType() {}
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
import jakarta.persistence.*;
@Entity
@Table(name = "broken_archive")
public class BreakArchive {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private long size;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "archive_id", nullable = false)
    private Archive archive;
```

```
    public BreakArchive() {}
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getSize() {
        return size;
    }
    public void setSize(long size) {
        this.size = size;
    }
    public Archive getArchive() {
        return archive;
    }
    public void setArchive(Archive archive) {
        this.archive = archive;
    }
}
import jakarta.persistence.*;
@Entity
@Table(name = "meta")
public class Meta {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(nullable = false)
```

```
    private String name;
    private long size;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "archive_id", nullable = false)
    private Archive archive;
    public Meta() {}
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getSize() {
        return size;
    }
    public void setSize(long size) {
        this.size = size;
    }
    public Archive getArchive() {
        return archive;
    }
    public void setArchive(Archive archive) {
        this.archive = archive;
    }
}
import jakarta.persistence.*;
import java.util.Set;
@Entity
```

```
@Table(name = "user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(unique = true, nullable = false)
    private String email;
    @Column(nullable = false)
    private String password;
    private String role;
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private Set<Archive> archives;
    // Конструктори, гетери та сетери
    public User() {}
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getRole() {
        return role;
```

```
    }
    public void setRole(String role) {
        this.role = role;
    }
    public Set<Archive> getArchives() {
        return archives;
    }
    public void setArchives(Set<Archive> archives) {
        this.archives = archives;
    }
}
```

Висновок

Під час виконання лабораторної роботи ми обрали систему draw.io для побудування наших uml-діаграм. Визначили функціональні блоки та акторів згідно варіанту для нашої діаграми варіантів використання, описали три сценарії використання. Визначили основні класи для нашої системи та будували діаграму класів з основними полями та методами та побудували базу даних в Microsoft Access привівши до сутності до нормальної форми та написали основні класи та реалізували шаблон Repository.

Питання до лабораторної роботи

1. Що таке UML?

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

2. Що таке діаграма класів UML?

Діаграма класів — це один із основних типів UML-діаграм, який відображає:

- класи системи
- їх атрибути (поля)
- методи (операції)
- зв’язки між ними (асоціації, агрегації, композиції, наслідування тощо).

Вона показує статичну структуру системи.

3. Які діаграми UML називають канонічними?

Канонічні діаграми UML — це основні, найуживаніші типи діаграм, що охоплюють ключові аспекти системи.До них належать:

- Діаграма варіантів використання (Use Case Diagram)
- Діаграма класів (Class Diagram)
- Діаграма послідовності (Sequence Diagram)
- Діаграма станів (State Machine Diagram)
- Діаграма діяльності (Activity Diagram)
- Діаграма компонентів (Component Diagram)
- Діаграма розгортання (Deployment Diagram)

4. Що таке діаграма варіантів використання?

Це UML-діаграма, яка показує взаємодію між користувачами (акторами) та системою через певні функціональні можливості, що називаються варіантами використання

5. Що таке варіант використання?

Варіант використання (Use Case) — це опис певного способу використання системи з точки зору користувача, який веде до досягнення конкретної мети.

6. Які відношення можуть бути відображені на діаграмі використання?

На діаграмі варіантів використання можуть бути такі зв’язки:

- Асоціація — між актором і варіантом використання.
- Include — один варіант включає інший (повторно використовує його).
- Extend — один варіант розширює інший додатковими діями.
- Узагальнення — актор або варіант використання може наслідувати властивості іншого.

7. Що таке сценарій?

Сценарій — це конкретна послідовність дій і подій, що описує, як саме виконується варіант використання.

8. Що таке діаграма класів?

Це структурна UML-діаграма, що показує:

- класи системи
- властивості ,
- методи
- типи зв’язків між ними.

9. Які зв’язки між класами ви знаєте?

Основні типи зв’язків між класами:

- Асоціація — загальний зв’язок між двома класами.
- Агрегація — “ціле–частина”, де частина може існувати окремо від цілого.
- Композиція — “ціле–частина”, але частина не може існувати без цілого.
- Наслідування — один клас є підтипом іншого.
- Залежність — один клас тимчасово використовує інший.

10. Чим відрізняється композиція від агрегації?

Агрегація —частина може існувати самостійно.

Композиція —частина знищується разом із цілим.

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

Агрегація позначається порожнім ромбом, композиція темним ромбом

12. Що являють собою нормальні форми баз даних?

Нормальні форми – це правила для нормалізації бази даних, які допомагають уникнути дублювання та колізій.

13. Що таке фізична модель бази даних? Логічна?

Логічна модель - описує структуру даних на рівні сутностей, атрибутів і зв’язків (ER-діаграма).

Фізична модель - реалізація логічної моделі у вигляді таблиць, типів даних, індексів, зовнішніх ключів, тощо, у певній СУБД.

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

Таблиці бази даних відповідають класам у програмі, рядки таблиць — об’єктам, стовпці — полям класів, а зв’язки між таблицями відображаються як зв’язки між класами.