



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

Лабораторна робота №2  
**Технології розроблення програмного забезпечення**  
«Основи проектування»

Тема: Особиста бухгалтерія

Виконала:  
студентка групи ІА-32  
Коляда М. С.

Перевірів:  
Мягкий М.Ю.

## **Зміст**

<b>Завдання.....</b>	<b>3</b>
<b>Теоретичні відомості.....</b>	<b>3</b>
<b>Діаграма класів.....</b>	<b>8</b>
<b>Схема бази даних.....</b>	<b>10</b>
<b>Вихідні коди класів системи.....</b>	<b>10</b>
<b>Питання до лабораторної роботи.....</b>	<b>19</b>

**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 спроектувати діаграму варіантів використання відповідно до обраної теми лабораторного циклу.
- Спроектувати діаграму класів предметної області.
- Вибрати 3 варіанти використання та написати за ними сценарії використання.
- На основі спроектованої 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.

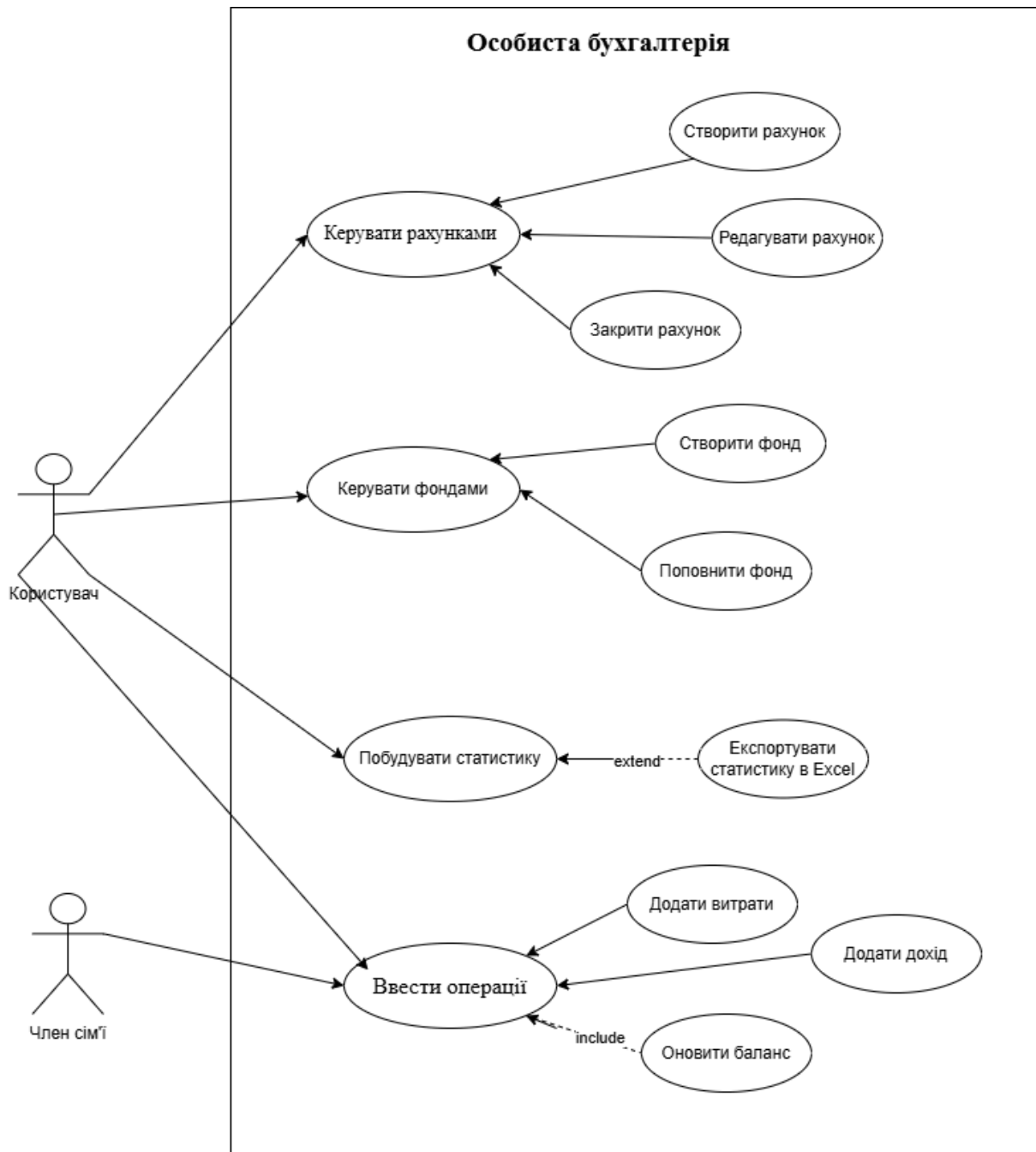
### **Теоретичні відомості**

UML (Unified Modeling Language) – це стандартна мова візуального моделювання, яку застосовують для опису, проєктування та документування програмних систем і бізнес-процесів. Вона дає змогу будувати модель системи на різних рівнях абстракції: від загального уявлення про призначення системи до логічної структури та фізичної реалізації. У межах об'єктно-орієнтованого аналізу й проєктування

система розглядається як набір уявлень (представлень), де кожне відображає певний аспект її поведінки або будови. Одним із базових засобів UML є діаграми. Вони фіксують модель у вигляді формальних графічних конструкцій. Серед основних типів діаграм виділяють діаграми варіантів використання, класів, послідовностей, станів, діяльності, компонентів, розгортання тощо. Кожен вид діаграми відповідає за свій зріз системи: статичну структуру, динаміку взаємодій, життєвий цикл об'єктів, фізичне розміщення компонентів. Діаграма варіантів використання (Use Case Diagram) відображає систему «очима користувача». На ній показують зовнішніх учасників – акторів – та варіанти використання, тобто послуги або функції, які система їм надає. Зв'язки між акторами й варіантами використання задаються відношеннями: асоціації описують факт взаємодії; узагальнення показує наслідування властивостей між акторами або варіантами; відношення «extend» дає змогу додати до базового варіанта додаткову поведінку за певних умов. Для кожного варіанта зазвичай складають текстовий сценарій, у якому фіксують передумови, основний потік подій, альтернативні гілки та результат. Сукупність діаграм і сценаріїв забезпечує чітке та однозначне формулювання функціональних вимог до системи.

### **Хід роботи**

#### **Діаграма використання (Use Case Diagram)**



## Прецедент 1. Ввести операції

**Передумови:** Користувач запустив додаток і знаходиться на головному екрані.

**Постумови:** У разі успішного виконання в системі збережено запис про фінансову операцію, а поточний баланс рахунку перераховано.

**Взаємодіючі сторони:** Користувач, Член сім'ї.

**Короткий опис:** Дозволяє користувачеві додати інформацію про витрату або дохід у систему.

**Основний потік подій:**

1. Користувач обирає функцію додавання нової операції.
2. Система відображає форму для введення даних (сума, категорія, дата, вибір рахунку).
3. Користувач вводить дані (наприклад, сума «500», категорія «Продукти») і натискає «Зберегти».
4. Система перевіряє коректність даних (чи заповнені поля, чи є сума числом).
5. Система зберігає запис про операцію в базу даних.
6. Система автоматично виконує включений варіант використання «Оновити баланс» (перераховує залишок коштів на обраному рахунку).
7. Система виводить повідомлення «Операцію успішно додано».

**Винятки:**

Невірний формат даних. Якщо користувач ввів літери замість цифр у полі «Сума», система підсвічує поле червоним і просить виправити помилку. Сценарій повертається до кроку 3.

**Прецедент 2. Побудувати статистику**

**Передумови:** У системі наявна хоча б одна збережена операція.

**Постумови:** Користувач переглянув графічний звіт про свої фінанси.

**Взаємодіючі сторони:** Користувач.

**Короткий опис:** Відображає розподіл витрат і доходів за обраний період у вигляді діаграми.

**Основний потік подій:**

1. Користувач переходить у розділ «Статистика».

2. Система запитує період звітності (поточний місяць, рік або довільний діапазон).
3. Користувач обирає період.
4. Система вибирає дані з бази та формує візуальну діаграму.
5. Система відображає статистику на екрані.
6. (Точка розширення) Якщо користувач бажає зберегти звіт у файл, виконується розширений варіант використання «Експортувати статистику в Excel».
7. Варіант використання завершується, коли користувач покидає екран статистики.

**Винятки:**

Дані відсутні. Якщо за обраний період не було транзакцій, система виводить повідомлення: «За цей період операцій не знайдено».

**Прецедент 3. Створити рахунок**

**Передумови:** Користувач знаходиться в меню «Керувати рахунками».

**Постумови:** У системі створено новий гаманець (рахунок) з початковим балансом.

**Взаємодіючі сторони:** Користувач.

**Короткий опис:** Дозволяє додати нове місце зберігання коштів (наприклад, «Готівка», «Картка Приват», «Скарбничка»).

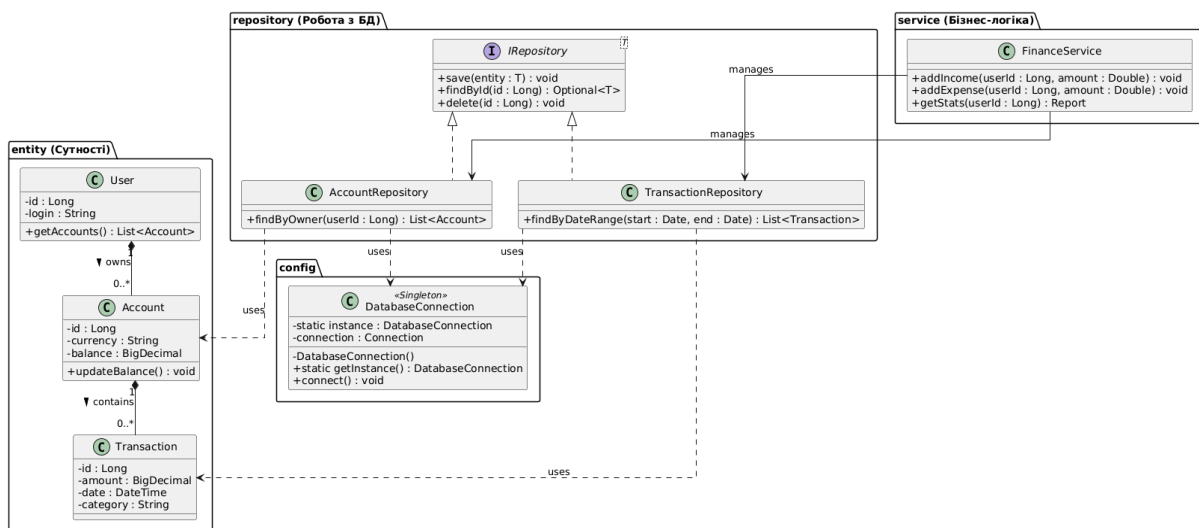
**Основний потік подій:**

1. Користувач натискає кнопку «Додати новий рахунок».
2. Система запитує назву рахунку, валюту та початковий баланс.
3. Користувач вводить дані (наприклад: Назва- «Спортбанк», Валюта- «UAH», Баланс - «0»).
4. Система перевіряє, чи не існує вже рахунку з такою самою назвою.
5. Система створює новий об'єкт рахунку в базі даних.
6. Система оновлює список рахунків на екрані, додаючи новий пункт.

## Винятки:

Дублювання назви. Якщо рахунок з такою назвою вже існує, система повідомляє: «Рахунок з такою назвою вже створено. Будь ласка, введіть іншу назву».

## Діаграма класів



## Repository Pattern

`IRepository<T>`: базовий інтерфейс, що визначає методи `save(T entity)`, `findById(Long id)`, `delete(Long id)`.

## Моделі (предметна область особистої бухгалтерії)

- User { id, login, accounts } – сутність користувача системи (власника фінансів).
- Account { id, name, currency, balance, transactions } – клас рахунку (гаманця). Зберігає поточний баланс та список прив'язаних до нього транзакцій.
- Transaction { id, amount, date, category, accountId } – клас фінансової операції. Відображає конкретний дохід або витрату. Може реалізовувати патерн Prototype (для створення копій регулярних платежів).



## Репозиторії

- AccountRepository (findByOwner) — для пошуку всіх рахунків конкретного користувача.
- TransactionRepository (findByDateRange, findByCategory) — для вибірки операцій за певний період або категорією (для статистики).

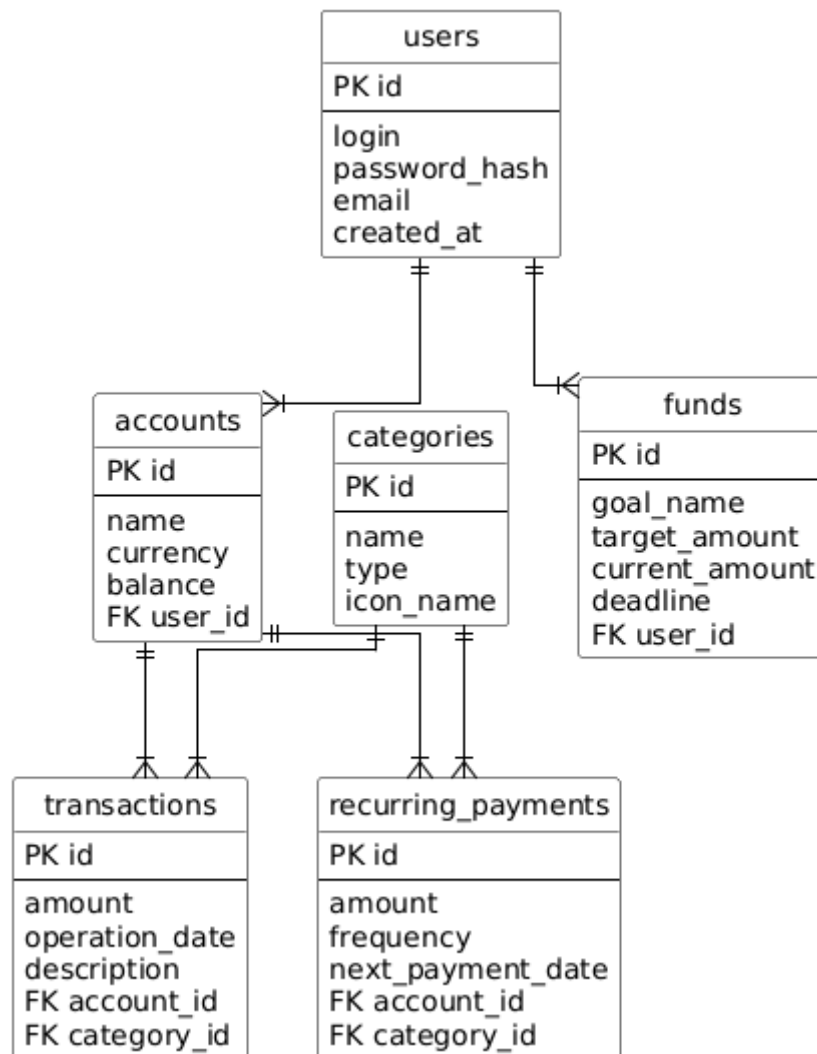
## Зв'язки між класами

- User 1 — 0..\* Account: (Композиція) Один користувач є власником багатьох рахунків. При видаленні користувача видаляються і його рахунки.
- Account 1 — 0..\* Transaction: (Композиція) Рахунок обов'язково містить історію транзакцій. Транзакція не може існувати без прив'язки до рахунку.
- FinanceService --> AccountRepository: (Асоціація) Сервіс використовує репозиторій для керування даними рахунків.
- AccountRepository ..> DatabaseConnection: (Залежність) Репозиторій залежить від класу підключення для виконання SQL-запитів.

## Сервіси та Utility-класи

- FinanceService: addIncome(amount), addExpense(amount), getStats().  
Відповідає за бізнес-логіку фінансів: при додаванні транзакції викликає оновлення балансу рахунку.
- DatabaseConnection: Клас, реалізований за патерном Singleton, що відповідає за єдине підключення до бази даних та ізолює технічні деталі з'єднання.

## Схема бази даних



## Вихідні коди класів системи

### DatabaseConnection.java

```
package ia32.koliada.finance.config;
```

```
import java.sql.Connection;
```

```
public class DatabaseConnection {  
    private static DatabaseConnection instance;
```

```

private Connection connection;

private DatabaseConnection() {
    System.out.println(">>> [DB] Підключення до бази даних
встановлено");
}

public static synchronized DatabaseConnection getInstance() {
    if (instance == null) {
        instance = new DatabaseConnection();
    }
    return instance;
}

public Connection getConnection() {
    return connection;
}

public void disconnect() {
    System.out.println(">>> [DB] Відключення від бази даних");
}
}

```

### **Account.java**

```

package ia32.koliada.finance.entity;

import java.math.BigDecimal;

public class Account {

```

```
private Long id;
private Long userId;
private String name;
private BigDecimal balance;
```

```
public Account(Long id, Long userId, String name, BigDecimal balance) {
    this.id = id;
    this.userId = userId;
    this.name = name;
    this.balance = balance;
}
```

```
public Long getId() { return id; }
public BigDecimal getBalance() { return balance; }
public void setBalance(BigDecimal balance) { this.balance = balance; }
public String getName() { return name; }
```

```
@Override
```

```
public String toString() {
    return "Account{id=" + id + ", name=" + name + ", balance=" + balance +
    "}" + "\n";
}
}
```

### **Transaction.java**

```
package ia32.koliada.finance.entity;
```

```
import java.math.BigDecimal;
import java.time.LocalDateTime;
```

```

public class Transaction {
    private Long id;
    private Long accountId;
    private Long categoryId;
    private BigDecimal amount;
    private LocalDateTime date;
    private String description;

    public Transaction(Long id, Long accountId, Long categoryId, BigDecimal
amount, String description) {
        this.id = id;
        this.accountId = accountId;
        this.categoryId = categoryId;
        this.amount = amount;
        this.date = LocalDateTime.now();
        this.description = description;
    }

    public BigDecimal getAmount() { return amount; }

    @Override
    public String toString() {
        return String.format("Transaction{amount=%s, date=%s, desc='%s'",
amount, date, description);
    }
}

```

**RecurringPayment.java**

```

package ia32.koliada.finance.entity;

import java.math.BigDecimal;

public class RecurringPayment implements Cloneable {
    private Long accountId;
    private Long categoryId;
    private BigDecimal amount;
    private String frequency;

    public RecurringPayment(Long accountId, Long categoryId, BigDecimal
amount, String frequency) {
        this.accountId = accountId;
        this.categoryId = categoryId;
        this.amount = amount;
        this.frequency = frequency;
    }

    public Transaction createTransaction() {
        return new Transaction(
            null,
            this.accountId,
            this.categoryId,
            this.amount,
            "Auto-payment: " + this.frequency
        );
    }
}

@Override

```

```

public RecurringPayment clone() {
    try {
        return (RecurringPayment) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
}

```

### **IRepository.java**

```

package ia32.koliada.finance.repository;

```

```

import java.util.Optional;

```

```

public interface IRepository<T> {
    void save(T entity);
    Optional<T> findById(Long id);
    void delete(Long id);
}

```

### **AccountRepository.java**

```

package ia32.koliada.finance.repository;

```

```

import ia32.koliada.finance.config.DatabaseConnection;
import ia32.koliada.finance.entity.Account;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

```

```

public class AccountRepository implements IRepository<Account> {
    private Map<Long, Account> fakeDb = new HashMap<>();

    public AccountRepository() {
        DatabaseConnection.getInstance();
    }

    @Override
    public void save(Account account) {
        fakeDb.put(account.getId(), account);
        System.out.println("SQL >>> Збережено рахунок: " + account);
    }

    @Override
    public Optional<Account> findById(Long id) {
        return Optional.ofNullable(fakeDb.get(id));
    }

    @Override
    public void delete(Long id) {
        fakeDb.remove(id);
    }
}

```

### **TransactionRepository.java**

```

package ia32.koliada.finance.repository;

import ia32.koliada.finance.entity.Transaction;
import ia32.koliada.finance.config.DatabaseConnection;

```



```

import java.util.Optional;

public class TransactionRepository implements IRepository<Transaction> {

    public TransactionRepository() {
        DatabaseConnection.getInstance();
    }

    @Override
    public void save(Transaction transaction) {
        System.out.println("SQL >>> Вставка транзакції в БД: " + transaction);
    }

    @Override
    public Optional<Transaction> findById(Long id) {
        return Optional.empty();
    }

    @Override
    public void delete(Long id) {}
}

```

### **FinanceService.java**

```

package ia32.koliada.finance.service;

import ia32.koliada.finance.entity.Account;

import ia32.koliada.finance.entity.Transaction;

import ia32.koliada.finance.repository.AccountRepository;

```

```
import ia32.koliada.finance.repository.TransactionRepository;

import java.math.BigDecimal;

public class FinanceService {

    private AccountRepository accountRepo;

    private TransactionRepository transactionRepo;

    public FinanceService() {

        this.accountRepo = new AccountRepository();

        this.transactionRepo = new TransactionRepository();

    }

    public void addTransaction(Long accountId, Long categoryId, BigDecimal
amount, String desc) {

        Account account = accountRepo.findById(accountId)

            .orElseThrow(() -> new RuntimeException("Рахунок не
знайдено!"));

        Transaction tx = new Transaction(null, accountId, categoryId, amount,
desc);
```

```
transactionRepo.save(tx);

BigDecimal newBalance = account.getBalance().add(amount);

account.setBalance(newBalance);

accountRepo.save(account);

System.out.println(">>> Операцію виконано. Новий баланс: " +
newBalance);

}

}
```

## **Питання до лабораторної роботи**

### **1. Що таке UML?**

UML (Unified Modeling Language) — це уніфікована графічна мова моделювання для опису, проектування та документування програмних систем і бізнес-процесів. Вона задає набір стандартних діаграм і позначень.

### **2. Що таке діаграма класів UML?**

Це діаграма, що описує статичну структуру системи. Вона показує класи системи, їхні атрибути (поля), методи (функції) та взаємозв'язки між цими класами. Це "скелет" програми.

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

Канонічними діаграмами називають основний набір із дев'яти типів діаграм UML (прецедентів, класів, об'єктів, послідовності, кооперації, станів, діяльності, компонентів, розгортання), що складають ядро мови.

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

Це діаграма поведінки, яка показує, що робить система з точки зору користувача, але не показує, як це реалізовано всередині. Вона складається з Акторів (дійових осіб) та Прецедентів (варіантів використання).

#### **5. Що таке варіант використання (Use Case)?**

Варіант використання (прецедент) — це специфікація послідовності дій, які система може виконувати у відповідь на дії актора для досягнення певної мети

#### **7. Що таке сценарій?**

Сценарій — це текстовий опис варіанту використання, у якому покроково подано основний хід подій і можливі альтернативні реакції системи на дії користувача.

#### **8. Що таке діаграма класів?**

Діаграма класів — це графічна схема, що відображає незмінну (статичну) структуру системи: які в ній є класи (типи об'єктів) та які зв'язки існують між цими класами.

#### **9. Які зв'язки між класами ви знаєте?**

Між класами виділяють такі основні види зв'язків: асоціація, агрегація, композиція, узагальнення (спадкування), реалізація та залежність.

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

Композиція має більш сильний зв'язок, ніж агрегація: у композиції

об'єкт-частина не існує окремо від об'єкта-цілого і зникає разом із ним, тоді як при агрегації частини можуть існувати самостійно.

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

На діаграмі класів: агрегація позначається лінією з порожнім ромбом біля цілого; композиція — лінією із заштрихованим (чорним) ромбом біля цілого.

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

Нормальні форми – це система правил проєктування таблиць, яка допомагає правильно організувати дані, зменшити їх дублювання й уникнути помилок (аномалій) під час додавання, зміни або видалення записів.

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

Логічна модель описує дані у вигляді сутностей, їхніх атрибутів та зв'язків між ними, без прив'язки до конкретної СУБД. Фізична модель показує, як ця логічна структура реалізується в конкретній СУБД: у вигляді таблиць, полів із певними типами даних, індексів тощо.

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

Взаємозв'язок полягає в тому, що клас у програмі зазвичай відповідає таблиці в базі даних: об'єкти (екземпляри) цього класу зберігаються як рядки таблиці, а його поля (атрибути) – як стовпці цієї таблиці.

**Висновок:** під час виконання лабораторної роботи, було обрано зручну систему побудови UML-діаграм та навчилася будувати діаграми варіантів використання для системи що проєктується, розробляла сценарії варіантів використання та будувала діаграми класів предметної області.