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 розробки програмного забезпечення

«Основи проектування»

Виконав:
студент групи ІА-33
Ничик Олександр

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

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.

Зміст

Завдання 2
Теоретичні відомості 2
Тема 4
Діаграма варіантів використання 4
Сценарії використання 5
Діаграма класів 7
Опис зв'язків 8
Проєктування БД 10
Опис БД 10
Вихідний код без реалізації на мові Java 11
Контрольні запитання 15
Висновки 16

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.

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 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 (diagram) – 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.

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.

Діаграми варіантів використання є відправною точкою при збиранні вимог до програмного продукту та його реалізації. Дана модель будується на аналітичному етапі побудови програмного продукту (збір та аналіз вимог) і дозволяє бізнес-аналітикам отримати більш повне уявлення про необхідне програмне забезпечення та документувати його.

Діаграма варіантів використання складається з низки елементів. Основними елементами є: варіанти використання або прецеденти (use case), актор або дійова особа (actor) та відносини між акторами та варіантами використання (relationship).

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. Сценарії використання однозначно визначають кінцевий результат.

Діаграми класів використовуються при моделюванні програмних систем

найчастіше. Вони є однією із форм статичного опису системи з погляду її проєктування, показуючи її структуру [3]. Діаграма класів не відображає динамічної поведінки об'єктів зображених на ній класів. На діаграмах класів показуються класи, інтерфейси та відносини між ними.

Клас – це основний будівельний блок програмної системи. Це поняття є і в мовах програмування, тобто між класами UML та програмними класами є відповідність, що є основою для автоматичної генерації програмних кодів або для виконання реінжинірингу. Кожен клас має назву, атрибути та операції. Клас на діаграмі показується як прямокутник, розділений на 3 області. У верхній міститься назва класу, у середній – опис атрибутів (властивостей), у нижній – назви операцій – послуг, що надаються об'єктами цього класу.

Логічна модель бази даних є структурою таблиць, уявлень, індексів та інших логічних елементів бази даних, що дозволяють власне програмування та використання бази даних. Процес створення логічної моделі бази даних зветься проєктування бази даних (database design). Проєктування відбувається у зв'язку з опрацюванням архітектури програмної системи, оскільки база даних створюється зберігання даних, одержуваних з програмних класів.

Тема

System activity monitor (iterator, command, abstract factory, bridge, visitor, SOA) Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

Хід роботи

Діаграма варіантів використання

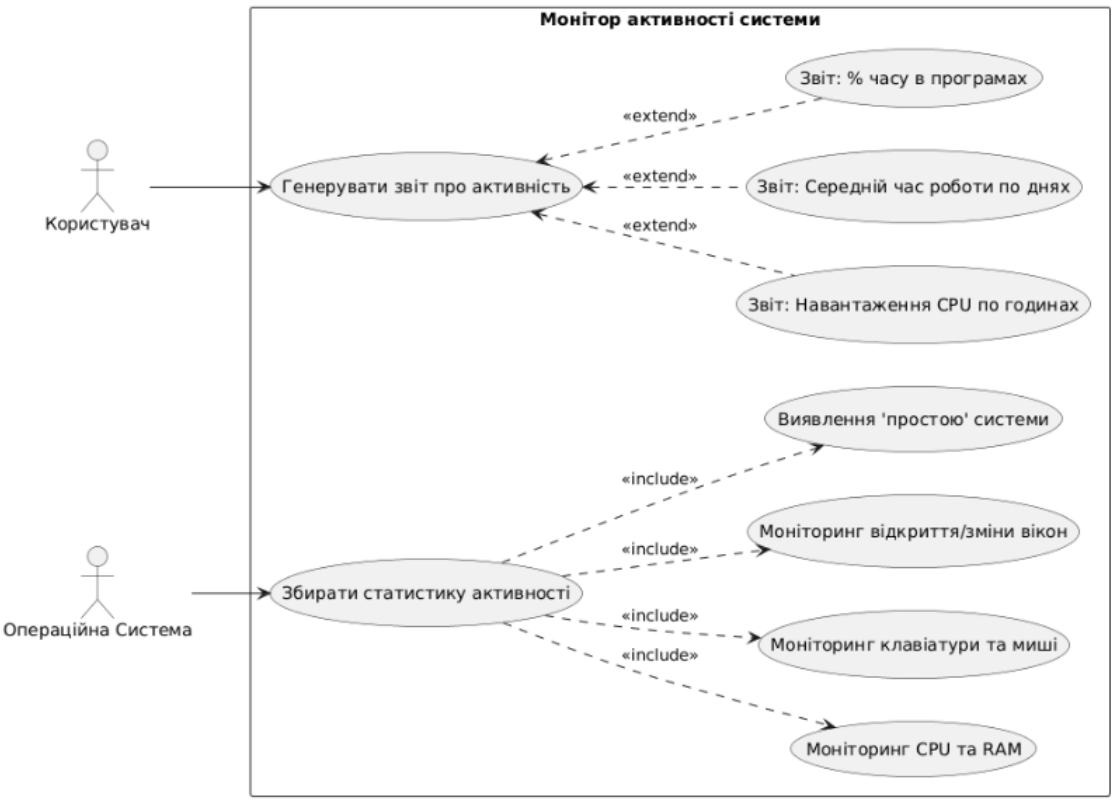

Рис. 1 – Діаграма варіантів використання

Сценарії використання

Сценарій 1. Генерація звіту про активність

- Передумови:
  - Система моніторингу встановлена та активна.
  - У базі даних системи накопичено статистику за певний період.
- Постумови:
  - Користувач отримує згенерований звіт за обраними критеріями.
- Взаємодіючі сторони:
  - Користувач, Монітор активності системи (система).
- Короткий опис: Користувач відкриває інтерфейс монітора та обирає один із доступних критеріїв для звіту. Система обробляє збережені дані та відображає звіт у зрозумілому вигляді.
- Основний перебіг подій:
  1. Користувач відкриває головний інтерфейс "Монітора активності системи".
  2. Користувач переходить до розділу "Звіти".
  3. Система відображає список доступних типів звітів:
     - Звіт: % часу в програмах
     - Звіт: Середній час роботи по днях

        - Звіт: Навантаження CPU по годинах
    4. Користувач обирає один із типів звіту (наприклад, "Звіт: % часу в програмах").
    5. Користувач (опціонально) вказує часовий діапазон для аналізу.
    6. Система звертається до своєї бази даних, витягує збережену статистику активності вікон.
    7. Система агрегує дані, розраховує відсоткове співвідношення часу для кожної програми.
    8. Система відображає згенерований звіт користувачеві (наприклад, у вигляді кругової діаграми або таблиці).
- Винятки:
    - Виняток №1: Якщо в базі даних немає статистики за обраний період. Система відображає повідомлення: "Дані для генерації звіту відсутні".
- Примітки:
    - Кроки 4-8 є реалізацією випадків використання <<extend>>. Основний випадок — "Генерувати звіт", а вибір конкретного типу звіту (по програмах, CPU, часу роботи) — це його розширення.

Сценарій 2. Збір статистики активності (Фоновий процес)

- Передумови:
    - Система моніторингу запущена (наприклад, при старті операційної системи).
    - Система має необхідні дозволи від ОС для збору даних.
- Постумови:
    - Дані про активність (ресурси, ввід, вікна, простій) збираються в реальному часі та зберігаються у внутрішній базі даних.
- Взаємодіючі сторони:
    - Операційна Система, Монітор активності системи (система).
- Короткий опис: Система у фоновому режимі безперервно збирає дані від Операційної Системи, включаючи використання ресурсів, дії користувача та активні вікна, а також фіксує періоди бездіяльності.
- Основний перебіг подій:
    1. Система запускає фоновий сервіс моніторингу.
    2. (<<include>> Моніторинг CPU та RAM) Система з заданою періодичністю (напр., 1 раз на секунду) опитує API Операційної Системи для отримання поточного % навантаження на процесор та обсягу зайнятої оперативної пам'яті.
    3. (<<include>> Моніторинг клавіатури та миші) Система реєструє

глобальні "слухачі" (hooks) в ОС для фіксації подій натискання клавіш та рухів/кліків миші.

4. (<<include>> Моніторинг відкриття/зміни вікон) Система реєструє "слухача" в ОС для відстеження подій зміни активного (фокусного) вікна, фіксуючи назву процесу та заголовок вікна.
5. (<<include>> Виявлення 'простою' системи) Система запускає внутрішній таймер бездіяльності.
6. Якщо події від клавіатури або миші (крок 3) не надходять протягом визначеного часу (напр., 5 хвилин), система фіксує початок статусу "Простій".
7. При надходженні першої події (крок 3) після початку "Простою", система фіксує завершення періоду бездіяльності.
8. Усі зібрані дані (кроки 2, 3, 4, 6, 7) зберігаються у внутрішню базу даних системи з відповідними часовими мітками.
9. Процес циклічно повторюється, поки "Монітор активності системи" увімкнений.

- Винятки:
  - Виняток №1: Якщо ОС блокує доступ до API (наприклад, через відсутність прав адміністратора). Система реєструє помилку у своєму журналі (log-файлі) і (опціонально) сповіщає користувача про неможливість збору даних.
  - Виняток №2: Якщо не вдається записати дані в базу (напр., закінчилося місце на диску). Система припиняє збір даних та інформує користувача про проблему.
- Примітки:
  - Цей сценарій є ключовим, оскільки він забезпечує даними "Сценарій 1 (Генерація звіту)". Без нього генерація звітів неможлива.

Діаграма класів

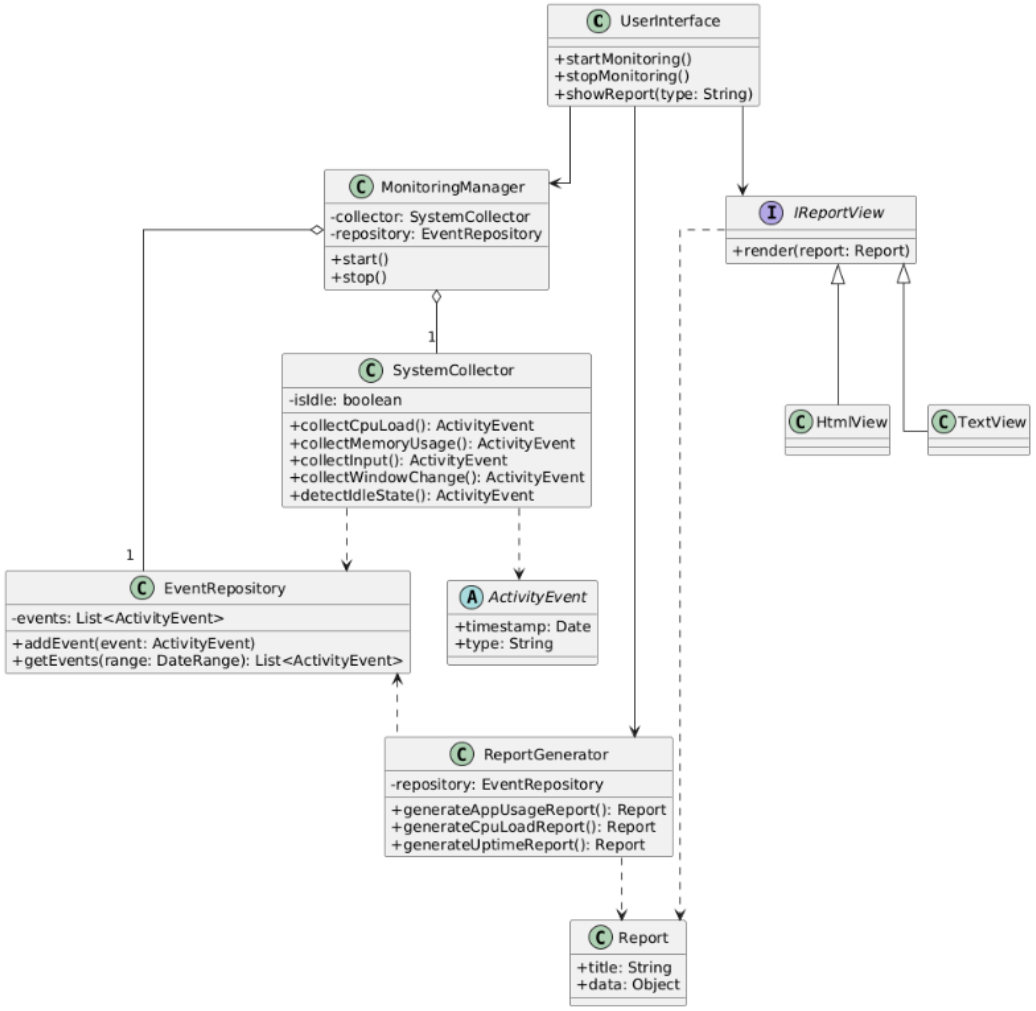

Рис. 2 – Діаграма класів

Опис зв’язків

UserInterface – MonitoringManager

- Тип відносин: Асоціація (1 → 1).
- Пояснення: UserInterface має посилання на MonitoringManager, щоб делегувати йому команди запуску (startMonitoring()) та зупинки (stopMonitoring()) процесу збору даних.

UserInterface – ReportGenerator

- Тип відносин: Асоціація (1 → 1).
- Пояснення: UserInterface звертається до ReportGenerator (який він, ймовірно, отримує від MonitoringManager або створює сам), щоб ініціювати створення звітів (наприклад, showReport("cpu") викличе generateCpuLoadReport()).

MonitoringManager – SystemCollector

- Тип відносин: Агрегація (1 → 1).
- Пояснення: MonitoringManager володіє та керує життєвим циклом SystemCollector. Менеджер запускає та зупиняє колектор, але колектор є окремим класом, що відповідає за логіку збору даних.

MonitoringManager – EventRepository

- Тип відносин: Агрегація (1 → 1).
- Пояснення: MonitoringManager володіє екземпляром EventRepository. Він передає посилання на цей репозиторій компонентам, яким він потрібен (наприклад, SystemCollector для запису та ReportGenerator для читання).

SystemCollector – EventRepository

- Тип відносин: Залежність (Dependency).
- Пояснення: SystemCollector не володіє репозиторієм, але використовує його. Коли колектор збирає нову подію (наприклад, collectCpuLoad()), він викликає метод addEvent() у EventRepository, щоб зберегти цю подію.

EventRepository – ActivityEvent

- Тип відносин: Композиція (1 → *).
- Пояснення: EventRepository містить список (List<ActivityEvent>) подій. ActivityEvent є невід'ємною частиною стану репозиторію. Якщо EventRepository буде знищено, всі події, що в ньому зберігаються, також будуть втрачені.

ReportGenerator – EventRepository

- Тип відносин: Асоціація (1 → 1).
- Пояснення: ReportGenerator має постійне посилання на EventRepository (поле repository), щоб мати можливість читати з нього дані (getEvents()) для подальшого аналізу та побудови звітів.

ReportGenerator – Report

- Тип відносин: Залежність (Dependency).
- Пояснення: ReportGenerator створює та повертає об'єкти Report. Він не зберігає їх; об'єкт Report є результатом роботи його методів (наприклад, generateAppUsageReport()).

IReportView – HtmlView / TextView

- Тип відносин: Узагальнення (Реалізація).
- Пояснення: HtmlView та TextView є конкретними реалізаціями інтерфейсу IReportView. Вони обидва надають власну версію методу render(), який по-різному відображає дані.

UserInterface – IReportView

- Тип відносин: Залежність (Dependency).
- Пояснення: UserInterface використовує інтерфейс IReportView для відображення звіту. Він створює конкретний HtmlView (або інший) і викликає його метод render(), передаючи йому об'єкт Report, отриманий від ReportGenerator.

Проєктування БД

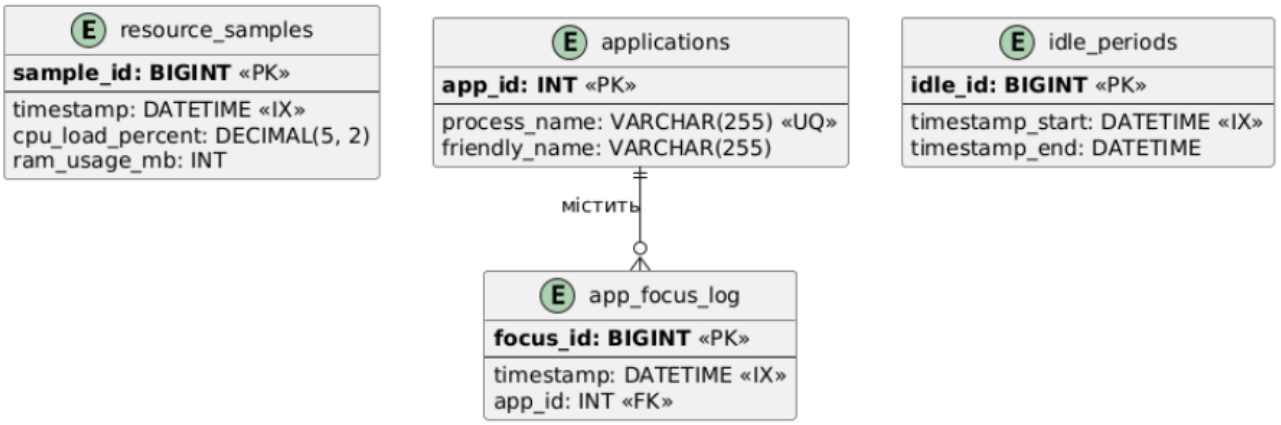

Рис. 3 – Структура БД

Опис БД

1. Таблиця applications
   - Призначення: Зберігає довідник унікальних програм (процесів), які відстежує монітор.
   - Поля:
     - app_id: INT — унікальний ідентифікатор програми (PK).
     - process_name: VARCHAR(255) — назва виконуваного файлу процесу, наприклад 'chrome.exe' (UQ - Unique).
     - friendly_name: VARCHAR(255) — "людська" назва програми, наприклад 'Google Chrome'.
   - Зв'язок: Одна програма може мати багато записів у журналі фокусу (app_focus_log). Це зв'язок "один-до-багатьох".
2. Таблиця app_focus_log
   - Призначення: Зберігає журнал подій, яка програма і коли ставала активною (отримувала фокус).
   - Поля:
     - focus_id: BIGINT — унікальний ідентифікатор події фокусу (PK).
     - timestamp: DATETIME — точний час, коли програма отримала фокус (IX - Indexed).
     - app_id: INT — зовнішній ключ, що посилається на applications(app_id) (FK).
   - Зв'язок: "Багато-до-одного" з applications. Багато подій фокусу можуть посилатися на один запис у applications.
3. Таблиця resource_samples
   - Призначення: Зберігає періодичні виміри системних ресурсів (навантаження CPU та використання RAM).
   - Поля:

    - sample_id: BIGINT — унікальний ідентифікатор виміру (PK).
    - timestamp: DATETIME — точний час, коли було зроблено вимір (IX - Indexed).
    - cpu_load_percent: DECIMAL(5, 2) — відсоток навантаження на процесор на момент виміру.
    - ram_usage_mb: INT — кількість використаної оперативної пам'яті (в МБ) на момент виміру.
- Зв'язок: Не має прямих зовнішніх ключів до інших таблиць. Вона логічно пов'язана з app_focus_log та idle_periods через поле timestamp, що дозволяє ReportGenerator аналізувати ресурси в певні проміжки часу.

4. Таблиця idle_periods
- Призначення: Зберігає часові інтервали, коли користувач був неактивний ("простій").
- Поля:
    - idle_id: BIGINT — унікальний ідентифікатор періоду простою (PK).
    - timestamp_start: DATETIME — час, коли система зафіксувала початок "простою" (IX - Indexed).
    - timestamp_end: DATETIME — час, коли "простій" закінчився (користувач проявив активність).
- Зв'язок: Не має прямих зовнішніх ключів. Логічно пов'язана з іншими таблицями через часові проміжки для виключення цих періодів зі звітів про активність.

Вихідний код без реалізації на мові Java

```
import java.util.Date;
import java.util.List;
import java.util.ArrayList;

abstract class ActivityEvent {
   public Date timestamp;
   public String type;
}

class Report {
   public String title;
   public Object data;
```

```
}

class EventRepository {
    private Object databaseConnection;

    public void addEvent(ActivityEvent newEvent) {
        // ...
    }

    public List<ActivityEvent> getEvents(Date startTime, Date endTime) {
        // ...
        return new ArrayList<ActivityEvent>();
    }
}

class SystemCollector {
    private boolean isIdle;
    private EventRepository repository;

    public SystemCollector(EventRepository repository) {
        this.repository = repository;
        this.isIdle = false;
    }

    public void collectCpuLoad() {
        // ActivityEvent evt = ...
        // repository.addEvent(evt);
    }

    public void collectMemoryUsage() {
        // ...
    }

    public void collectInput() {
        // ...
    }
```

```
    public void collectWindowChange() {
        // ...
    }

    public void detectIdleState() {
        // ...
    }
}

class MonitoringManager {
    private SystemCollector collector;
    private EventRepository repository;
    private Object timer;

    public MonitoringManager() {
        this.repository = new EventRepository();
        this.collector = new SystemCollector(this.repository);
    }

    public void start() {
        // ...
    }

    public void stop() {
        // ...
    }

    public EventRepository getRepository() {
        return repository;
    }
}

class ReportGenerator {
    private EventRepository repository;

    public ReportGenerator(EventRepository repository) {
        this.repository = repository;
```

```
    }

    public Report generateAppUsageReport() {
        // 1. List<ActivityEvent> events = repository.getEvents(...)
        // 2. Process events
        // 3. return new Report();
        return new Report();
    }

    public Report generateCpuLoadReport() {
        // ...
        return new Report();
    }

    public Report generateUptimeReport() {
        // ...
        return new Report();
    }
}

interface IReportView {
    void render(Report report);
}

class HtmlView implements IReportView {
    @Override
    public void render(Report report) {
        // System.out.println("<html><h1>" + report.title + "</h1>...</html>");
    }
}

class TextView implements IReportView {
    @Override
    public void render(Report report) {
        // System.out.println("--- " + report.title + " ---");
    }
}
```

```
public class UserInterface {
    private MonitoringManager manager;
    private ReportGenerator reportGenerator;
    private IReportView currentView;

    public UserInterface() {
        this.manager = new MonitoringManager();
        this.reportGenerator = new ReportGenerator(manager.getRepository());
        this.currentView = new TextView();
    }

    public void onStartClick() {
        manager.start();
    }

    public void onStopClick() {
        manager.stop();
    }

    public void onShowReportClick(String reportType) {
        Report report = null;

        if ("cpu".equals(reportType)) {
            report = reportGenerator.generateCpuLoadReport();
        } else if ("apps".equals(reportType)) {
            report = reportGenerator.generateAppUsageReport();
        }
        // ...

        if (report != null) {
            currentView.render(report);
        }
    }
}
```

Контрольні запитання

1. UML — уніфікована мова моделювання, стандарт для опису та візуалізації програмних систем.
2. Діаграма класів UML — схема, що показує класи, їх атрибути, методи та зв’язки між ними.
3. Канонічні діаграми UML — це базові діаграми, без яких неможливе повноцінне моделювання: діаграма класів, варіантів використання, послідовностей, станів, діяльності, компонентів, розгортання.
4. Діаграма варіантів використання — показує, як користувачі (актори) взаємодіють із системою через функціональні можливості (use cases).
5. Варіант використання — це опис певної функціональності системи з точки зору користувача.
6. Відношення на діаграмі використання: асоціація (актор–use case), include, extend, generalization.
7. Сценарій — послідовність кроків взаємодії користувача з системою у межах одного варіанта використання.
8. Діаграма класів — структурна діаграма, що описує класи системи та їх зв’язки.
9. Зв’язки між класами: асоціація, агрегація, композиція, наслідування (generalization), реалізація (implements), залежність.
10. Композиція vs агрегація: композиція означає повну залежність частини від цілого (життєвий цикл спільний), агрегація — слабший зв’язок, частина може існувати окремо.
11. На діаграмі класів: композиція позначається чорним ромбом, агрегація — білим ромбом.
12. Нормальні форми — правила структурування таблиць БД для усунення надлишковості та аномалій.
13. Фізична модель БД — конкретна реалізація (таблиці, індекси, типи даних). Логічна модель — концептуальна структура даних (сутності, зв’язки, атрибути).
14. Таблиці БД ↔ програмні класи: кожна таблиця часто відповідає класу, рядок таблиці — об’єкту, стовпець — атрибуту класу.

Висновки

Під час виконання лабораторної роботи, ми навчилися основам проектування, обрали зручну систему побудови UML-діаграм та навчилися будувати діаграми варіантів використання для системи що проєктується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.