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

• 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

17. 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

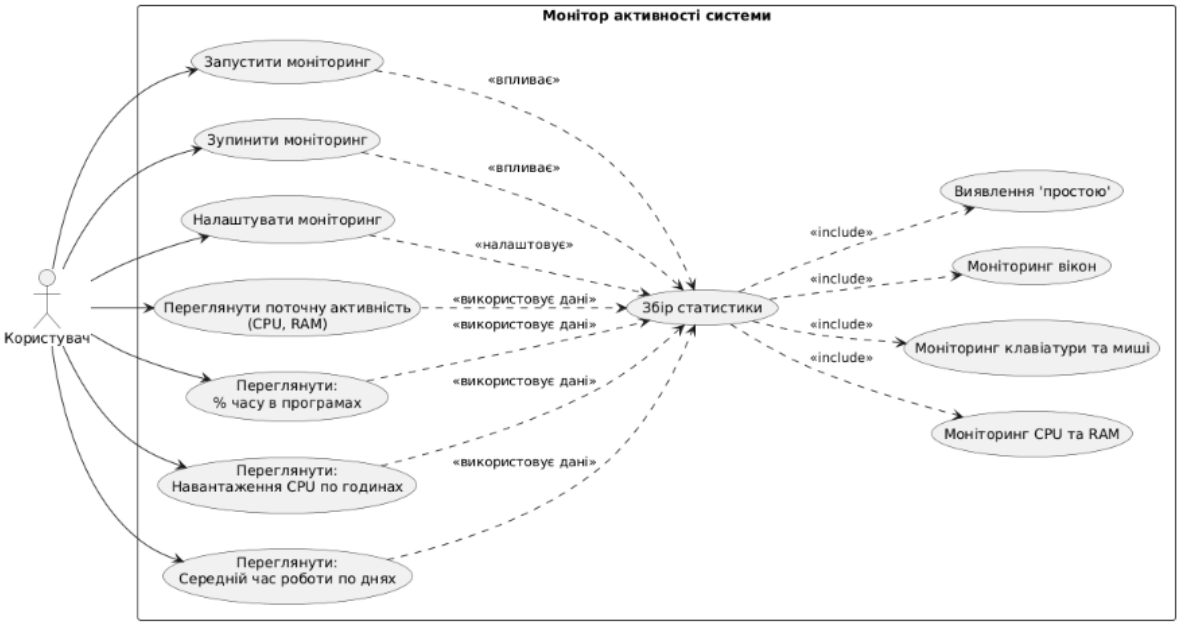

Рис. 1 – Діаграма варіантів використання

Сценарії використання

Сценарій 1. Запуск моніторингу

- Передумови:
  - Система моніторингу встановлена.
  - Фоновий процес "Збір статистики" наразі неактивний (зупинений).
- Постумови:
  - Фоновий процес "Збір статистики" активовано і починає збирати дані.
  - Користувач бачить оновлений статус ("Моніторинг активний").
- Взаємодіючі сторони:
  - Користувач, Монітор активності системи (система).
- Короткий опис: Користувач відкриває інтерфейс системи та натискає кнопку "Запустити моніторинг", щоб ініціювати фоновий процес збору даних.
- Основний перебіг подій:
  1. Користувач відкриває головний інтерфейс "Монітора активності системи".
  2. Користувач натискає кнопку "Запустити моніторинг".
  3. Система перевіряє, чи фоновий процес "Збір статистики" не є вже активним.
  4. (<<впливає>>) Система активує фоновий процес "Збір статистики".
  5. Процес "Збір статистики" починає виконувати свої <<include>> задачі (Моніторинг CPU, Моніторинг вікон тощо).

6. Система відображає користувачеві статус "Активний".

- Винятки:
  - Виняток №1: Якщо процес "Збір статистики" вже запущено. Система відображає повідомлення: "Моніторинг вже запущено".
  - Виняток №2: Якщо системі не вдалося отримати необхідні дозволи від ОС (наприклад, для "слухачів" клавіатури). Система відображає повідомлення "Помилка запуску: перевірте дозволи системи".
- Примітки:
  - Цей варіант використання є однією з дій, що керують (<<впливає>>) центральним процесом Збір статистики.

Сценарій 2. Перегляд звіту "% часу в програмах"

- Передумови:
  - Процес "Збір статистики" працював певний час.
  - У базі даних накопичено дані з Моніторингу вікон.
- Постумови:
  - Користувач отримує звіт, що показує розподіл часу між різними програмами.
- Взаємодіючі сторони:
  - Користувач, Монітор активності системи (система).
- Короткий опис: Користувач обирає в меню конкретний тип звіту. Система звертається до зібраних раніше даних і відображає результат.
- Основний перебіг подій:
  1. Користувач відкриває головний інтерфейс "Монітора активності системи".
  2. Користувач переходить до розділу "Звіти" та обирає пункт "Звіт: % часу в програмах".
  3. Користувач (опціонально) вказує часовий діапазон для аналізу.
  4. (<<використовує дані>>) Система звертається до даних, зібраних процесом "Збір статистики".
  5. Система вибирає з бази даних записи, що стосуються Моніторингу вікон.
  6. Система агрегує дані, розраховує загальний час фокусу для кожної програми.
  7. Система відображає згенерований звіт користувачеві (наприклад, у вигляді кругової діаграми).
- Винятки:
  - Виняток №1: Якщо в базі даних немає статистики для побудови

цього звіту. Система відображає повідомлення: "Дані для генерації звіту відсутні".

- Примітки:
    - На відміну від <<extend>> у попередній версії, це прямий варіант використання, який ініціює користувач.

Сценарій 3. Збір статистики (Внутрішній фоновий процес)

- Передумови:
    - Процес запущений користувачем (див. Сценарій 1 "Запуск моніторингу").
    - Система має необхідні дозволи від ОС.
- Постумови:
    - Дані про активність (ресурси, ввід, вікна, простій) збираються в реальному часі та зберігаються у внутрішній базі даних.
- Взаємодіючі сторони:
    - Монітор активності системи (система). (Користувач не взаємодіє з цим процесом напряму, а лише керує ним).
- Короткий опис: Система у фоновому режимі безперервно збирає дані про активність, доки не буде зупинена користувачем (через UC "Зупинити моніторинг").
- Основний перебіг подій:
    1. Процес перебуває в активному стані.
    2. (<<include>> Моніторинг CPU та RAM) Система з заданою періодичністю (напр., 1 раз на секунду) опитує API ОС 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 вікон) Система реєструє "слухача" в ОС для відстеження подій зміни активного (фокусного) вікна.
    5. (<<include>> Виявлення 'простою' системи) Система використовує дані з кроку 3 для керування таймером бездіяльності.
    6. Якщо таймер бездіяльності перевищує ліміт (напр., 5 хвилин), система фіксує початок статусу "Простій".
    7. Усі зібрані дані зберігаються у внутрішню базу даних системи.
    8. Процес циклічно повторюється.
- Винятки:
    - Виняток №1: Якщо ОС блокує доступ до API (наприклад, користувач

змінив дозволи під час роботи). Система реєструє помилку у своєму журналі.

    - Виняток №2: Якщо не вдається записати дані в базу (напр., закінчилося місце на диску). Система припиняє збір даних та (опціонально) інформує користувача.

- Примітки:
    - Це центральний процес системи, який надає дані для всіх звітів.

Діаграма класів

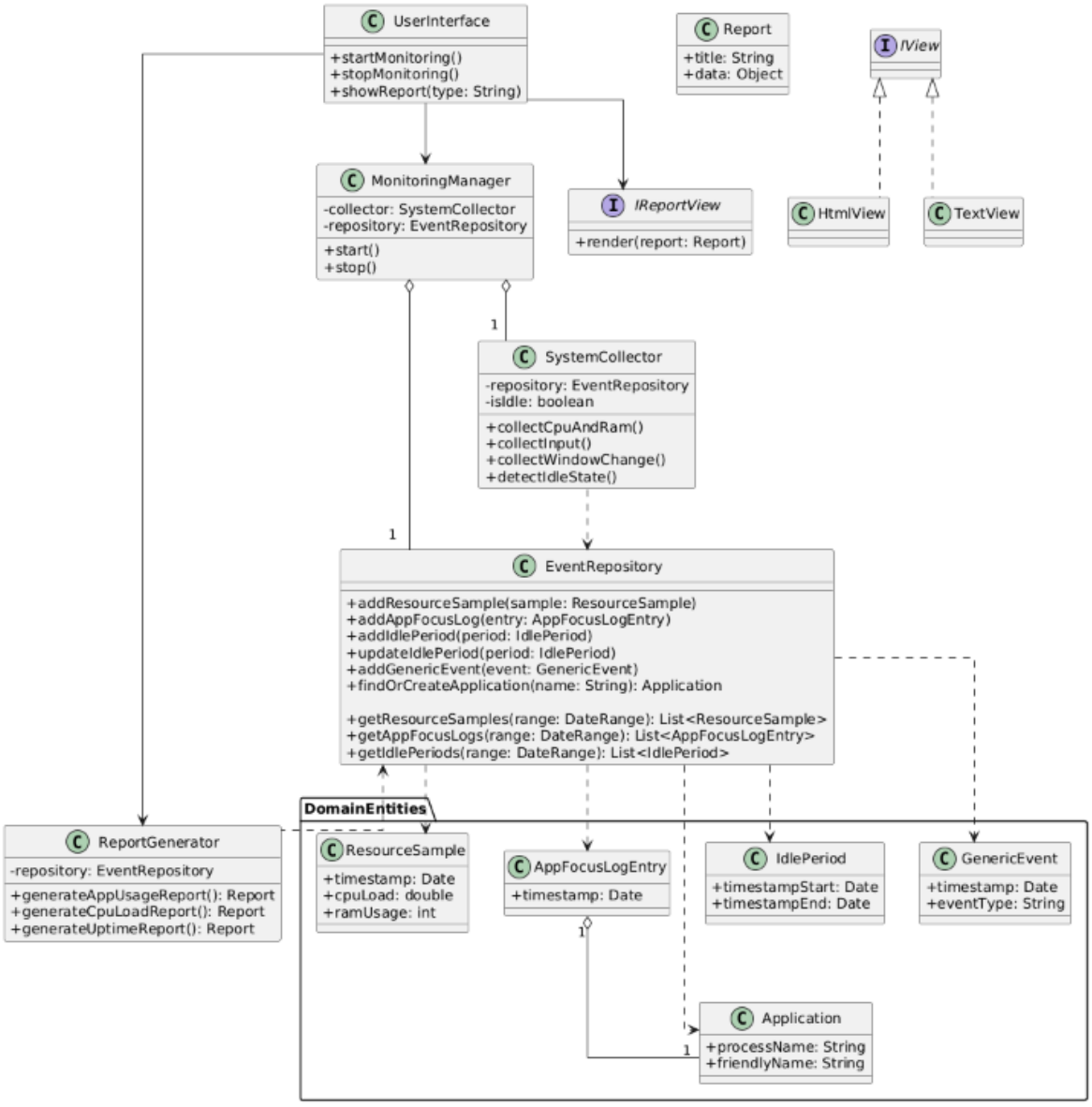

Рис. 2 – Діаграма класів

Опис зв’язків

1. UserInterface – MonitoringManager
   - Тип відносин: Асоціаціація (1 → 1).
   - Пояснення: UserInterface має посилання на MonitoringManager, щоб делегувати йому команди запуску (startMonitoring()) та зупинки (stopMonitoring()) процесу збору даних.
2. UserInterface – ReportGenerator
   - Тип відносин: Асоціаціація (1 → 1).
   - Пояснення: UserInterface звертається до ReportGenerator, щоб ініціювати створення звітів (наприклад, showReport("cpu") викличе generateCpuLoadReport()).
3. 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.

4. 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.
5. MonitoringManager – EventRepository
   - Тип відносин: Агрегація (1 → 1).
   - Пояснення: MonitoringManager володіє екземпляром EventRepository. Він передає посилання на цей репозиторій компонентам, яким він потрібен (SystemCollector для запису та ReportGenerator для читання).
6. SystemCollector – EventRepository
   - Тип відносин: Асоціація (1 → 1).
   - Пояснення: SystemCollector має постійне посилання на EventRepository (отримане від MonitoringManager). Він використовує це посилання для виклику методів "додавання", таких як addResourceSample(), addAppFocusLog() та addGenericEvent(), під час збору даних.
7. ReportGenerator – EventRepository
   - Тип відносин: Асоціація (1 → 1).
   - Пояснення: ReportGenerator має постійне посилання на EventRepository (поле repository), щоб мати можливість читати з нього дані, викликаючи методи "отримання", наприклад getResourceSamples() або getAppFocusLogs().
8. EventRepository – DomainEntities (ResourceSample, Application, ...)
   - Тип відносин: Залежність (Dependency).
   - Пояснення: EventRepository залежить від класів-сутностей. Його методи приймають ці сутності як параметри для запису в БД (наприклад, addResourceSample(sample: ResourceSample)) або повертають їх як результат читання з БД (наприклад, getResourceSamples(): List<ResourceSample>).
9. AppFocusLogEntry – Application
   - Тип відносин: Асоціація (1 → 1).
   - Пояснення: Цей зв'язок всередині DomainEntities відображає зв'язок у базі даних (Foreign Key). Кожен об'єкт AppFocusLogEntry (запис про фокус) логічно пов'язаний з одним об'єктом Application (програмою, яка була у фокусі).
10. 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()).

11. IReportView / IView – HtmlView / TextView
    - Тип відносин: Узагальнення (Реалізація).
    - Пояснення: HtmlView та TextView є конкретними реалізаціями, які наслідують інтерфейс IReportView (через IView на діаграмі). Вони обидва надають власну версію методу render(), який по-різному відображає дані Report.

Проєктування БД

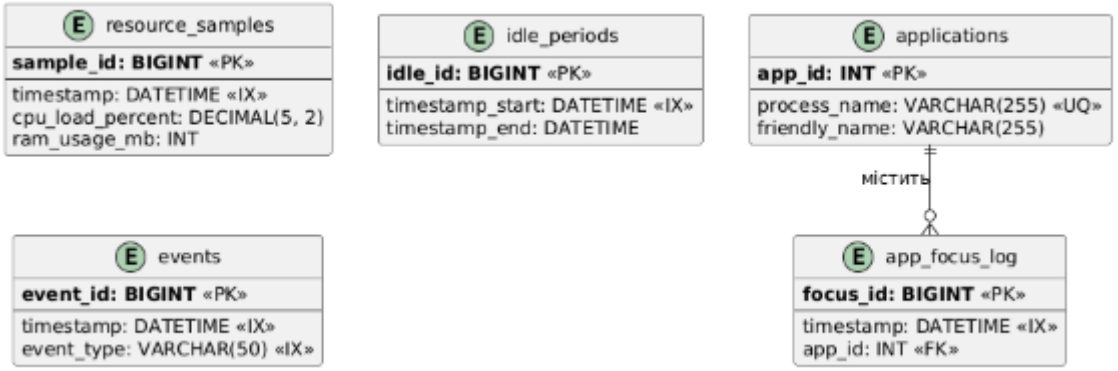

Рис. 3 – Структура БД

Опис БД

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. 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.

5. Таблиця events
- Призначення: Зберігає загальні, одноразові події (наприклад, натискання клавіш, рух миші). В основному використовується для виявлення активності користувача, щоб завершити період "простою".
- Поля:
    - event_id: BIGINT — унікальний ідентифікатор події (PK).

    - timestamp: DATETIME — точний час, коли сталася подія (IX - Indexed).
    - event_type: VARCHAR(50) — тип події (напр., 'KEY_PRESS', 'MOUSE_MOVE') (IX - Indexed).
- Зв'язок: Не має прямих зовнішніх ключів. Логічно використовується SystemCollector для оновлення записів у idle_periods (встановлення timestamp_end).

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

Ваша теза: UML — уніфікована мова моделювання, стандарт для опису та візуалізації програмних систем.

Доповнення: Це графічна мова, яка використовується не лише для *опису* (документації), але й для *проектування* (дизайну) системи до початку написання коду. Її головна мета — надати розробникам, аналітикам та замовникам спільну мову для обговорення та розуміння системи.

2. Діаграма класів UML

Ваша теза: Схема, що показує класи, їх атрибути, методи та зв’язки між ними.

Доповнення: Це основна структурна діаграма в UML. Вона моделює статичну структуру системи, тобто те, з яких "будівельних блоків" (класів) система складається, незалежно від того, що вони роблять у часі.

3. Канонічні діаграми UML

Ваша теза: Базові діаграми, без яких неможливе повноцінне моделювання: діаграма класів, варіантів використання, послідовностей, станів, діяльності,

компонентів, розгортання.

Доповнення: В UML 14+ діаграм, але ці є ключовими. Їх зазвичай поділяють на дві великі категорії:

- Структурні (Structural): Показують *статику* системи. (Діаграма класів, компонентів, розгортання).
- Поведінкові (Behavioral): Показують *динаміку* системи — як вона поводиться з часом. (Діаграма варіантів використання, послідовностей, станів, діяльності).

4. Діаграма варіантів використання (Use Case)

Ваша теза: Показує, як користувачі (актори) взаємодіють із системою через функціональні можливості (use cases).

Доповнення: Її головна мета — визначити межі (scope) системи. Вона відповідає на питання: "Що система повинна робити?" (функціональні вимоги) і "Хто і навіщо буде нею користуватися?". Вона є точкою старту для всього проектування.

5. Варіант використання (Use Case)

Ваша теза: Опис певної функціональності системи з точки зору користувача.

Доповнення: Ключовим є те, що кожен варіант використання повинен приносити конкретну, вимірювану цінність для актора. Наприклад, "Переглянути звіт" — це цінність, а "Ввести логін" — ні (це лише крок іншого, більшого Use Case).

6. Відношення на діаграмі використання

Ваша теза: Асоціація (актор–use case), include, extend, generalization.

Доповнення:

- <<include>> (включення): Використовується для винесення обов'язкової, повторюваної поведінки. (Наприклад, "Перевірка авторизації" включається в "Згенерувати звіт" і "Налаштувати моніторинг").
- <<extend>> (розширення): Використовується для опціональної поведінки, яка може відбутися, а може й ні. (Наприклад, базовий UC "Оплатити замовлення" може бути розширений UC "Використати промокод").

7. Сценарій (Scenario)

Ваша теза: Послідовність кроків взаємодії користувача з системою у межах одного варіанта використання.

Доповнення: Кожен варіант використання (Use Case) має щонайменше один основний успішний сценарій (Main Success Scenario або "happy path") та нуль або більше альтернативних сценаріїв (Alternative Paths) та сценаріїв помилок (Exception Scenarios).

8. Діаграма класів

Ваша теза: Структурна діаграма, що описує класи системи та їх зв’язки.

Доповнення: Вона є "кресленням" для розробників. На основі цієї діаграми часто пишуть "скелет" коду (класи, інтерфейси, поля) ще до реалізації методів.

9. Зв’язки між класами

Ваша теза: Асоціація, агрегація, композиція, наслідування (generalization), реалізація (implements), залежність.

Доповнення:

- Асоціація ("uses-a"): Два класи пов'язані, але існують незалежно. (Наприклад, UserInterface *використовує* ReportGenerator).
- Наслідування ("is-a"): Один клас є підтипом іншого. (Наприклад, HtmlView *є* IReportView).

10. Композиція vs Агрегація

Ваша теза: Композиція означає повну залежність частини від цілого (життєвий цикл спільний), агрегація — слабший зв’язок, частина може існувати окремо.

Доповнення:

- Приклад композиції ("has-a"): Автомобіль та Двигун. Якщо видалити Автомобіль, Двигун також буде знищено.
- Приклад агрегації ("has-a"): Університет та Професор. Професор може існувати без університету (наприклад, звільнитися і перейти в інший).

11. На діаграмі класів (позначення)

Ваша теза: Композиція позначається чорним ромбом, агрегація — білим ромбом.

Доповнення: Інші важливі позначення:

- Наслідування (Generalization): Суцільна лінія з незафарбованим трикутником (стрілкою).
- Реалізація (Realization): Пунктирна лінія з незафарбованим трикутником (стрілкою).
- Асоціація (Association): Суцільна лінія.
- Залежність (Dependency): Пунктирна лінія зі звичайною стрілкою.

12. Нормальні форми (БД)

Ваша теза: Правила структурування таблиць БД для усунення надлишковості та аномалій.

Доповнення: Головна мета — цілісність даних. Найчастіше використовуються:

- 1НФ (Перша): Атомарність значень (немає списків у комірці).
- 2НФ (Друга): Усі неключові поля залежать від *усього* складеного первинного ключа.
- 3НФ (Третя): Неключові поля не залежать від інших неключових полів.

13. Фізична vs Логічна модель БД

Ваша теза: Фізична модель БД — конкретна реалізація (таблиці, індекси, типи даних). Логічна модель — концептуальна структура даних (сутності, зв’язки, атрибути).

Доповнення: Існує три рівні:

1. Концептуальна модель (ERD): Найвищий рівень. Просто сутності та зв'язки (напр., "Додаток" пов'язаний з "Записом логу").
2. Логічна модель: Додає атрибути, первинні та зовнішні ключі, але без прив'язки до СУБД (напр., тип даних STRING).
3. Фізична модель: Найнижчий рівень. Враховує конкретну СУБД (напр., VARCHAR(255) для MySQL або NVARCHAR(255) для MS SQL), індекси, партиції.

14. Таблиці БД ↔ Програмні класи

Ваша теза: Кожна таблиця часто відповідає класу, рядок таблиці — об’єкту, стовпець — атрибуту класу.

Доповнення: Процес перетворення даних із реляційних таблиць в об'єкти коду (і навпаки) називається ORM (Object-Relational Mapping). Це фундаментальний патерн, який реалізований у фреймворках (як Hibernate для Java, Entity Framework для C#, або SQLAlchemy для Python), щоб програмісту не доводилося писати SQL-запити вручну.

Висновки

Під час виконання лабораторної роботи, ми опанували ключові етапи початкового проектування програмних систем. Ми не лише ознайомилися з теоретичними основами, але й здобули практичні навички у моделюванні, пройшовши логічний шлях від вимог до структури.

Робота почалася з вибору зручної системи для побудови UML-діаграм, що дозволило нам ефективно візуалізувати, документувати та ітеративно покращувати наші проектні рішення.

Ключовим етапом стало будування діаграми варіантів використання. Цей крок допоміг чітко визначити межі проектованої системи, її основні функціональні можливості та те, як саме користувачі будуть з нею взаємодіяти.

Для глибокого розуміння кожної функції ми розробили детальні сценарії для основних варіантів використання. Цей процес виявився критично важливим, оскільки він дозволив виявити покрокову логіку взаємодії, включаючи основний шлях, а також альтернативні потоки та потенційні виняткові ситуації, які необхідно обробити.

На основі аналізу варіантів використання та сценаріїв, ми змогли ідентифікувати ключові сутності предметної області. Результатом цього аналізу стало будування діаграми класів. Ця діаграма слугує статичним "кресленням" нашої майбутньої системи. Вона описує не лише самі класи, їхні атрибути та методи, але й важливі логічні зв’язки між ними.

Таким чином, ця лабораторна робота провела нас через повний цикл початкового аналізу та проектування: від визначення високорівневих функціональних вимог до їх та розробки статичної архітектурної моделі.

Отримані навички є фундаментальними для зменшення неоднозначності у вимогах, ефективної комунікації в команді та створення якісної, структурованої програмної системи.