



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 №3
Технології розробки
програмного
забезпечення

*«Основи проектування
розгортання»*

«System activity monitor»

Виконав:

студент групи IA-33

Ничик Олександр

Перевірив:

Мягкий Михайло

Юрійович

Тема: Основи проектування розгортання

Мета: Навчитися проектувати діаграми розгортання та компонентів для системи що проєктується, а також розробляти діаграми взаємодії, а саме діаграми послідовностей, на основі сценаріїв зроблених в попередній лабораторній роботі.

Зміст

Завдання.....	2
Теоретичні відомості	3
Тема проєкту	4
Діаграма розгортання	5
Опис діаграми розгортання	5
Діаграма компонентів.....	6
Опис діаграми компонентів	7
Діаграми послідовностей.....	8
Код програми.....	11
Контрольні запитання	21
Висновки	22

Завдання

- Ознайомитись з короткими теоретичними відомостями.
- Проаналізувати діаграми створені в попередній лабораторній роботі а також тему системи та спроєктувати діаграму розгортання використання відповідно до обраної теми лабораторного циклу.
- Розробити діаграму компонентів для проєктованої системи.
- Розробити діаграму розгортання для проєктованої системи.
- Розробити як мінімум дві діаграми послідовностей для сценаріїв прописаних в попередній лабораторній роботі.
- На основі спроєктованих діаграм розгортання та компонентів доопрацювати програмну частину системи. Реалізація системи, додатково до попередньої реалізації, повинна містити як мінімум дві візуальні форми. В системі вже повинен бути повністю реалізована архітектура (повний цикл роботи з даними від вводу на формі до збереження їх в БД і подальшій виборці з БД та відображенням на UI).
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт

повинен містити: діаграму розгортання з описом, діаграму компонентів системи з описом, діаграми послідовностей, а також вихідний код системи, який було додано в цій лабораторній роботі.

Теоретичні відомості

Діаграми компонентів — показують модулі (компоненти), їхні залежності і артефакти (.jar, таблиці, html). Використовуються для логічного/фізичного поділу системи.

Діаграма компонентів UML є представленням проектованої системи, розбитої на окремі модулі. Залежно від способу поділу на модулі розрізняють три види діаграм компонентів:

- логічні;
- фізичні;
- виконувані.

Коли використовують логічне розбиття на компоненти, то у такому разі проєктовану систему віртуально уявляють як набір самостійних, автономних модулів (компонентів), що взаємодіють між собою.

Діаграми розгортання — показують фізичні вузли (devices) і середовища виконання (execution environments), на яких розгортаються артефакти; зв'язки зазначають протоколи (HTTP, SQL/ODBC).

Діаграми розгортання представляють фізичне розташування системи, показуючи, на якому фізичному обладнанні запускається та чи інша складова програмного забезпечення.

Головними елементами діаграми є вузли, пов'язані інформаційними шляхами. Вузол (node) – це те, що може містити програмне забезпечення. Вузли бувають двох типів. Пристрій (device) – це фізичне обладнання: комп'ютер або пристрій, пов'язаний із системою. Середовище виконання (execution environment) – це програмне забезпечення, яке саме може включати інше програмне забезпечення, наприклад операційну систему або процес-контейнер (наприклад, вебсервер).

Діаграми послідовностей — моделюють часову послідовність повідомлень між акторами/об'єктами (HTTP POST/GET: Browser – Server DB – Browser).

Діаграма послідовностей (Sequence Diagram) – це один із типів діаграм у моделюванні UML (Unified Modeling Language), який використовується для моделювання взаємодії між об'єктами системи у певній послідовності часу. Вона відображає, як об'єкти обмінюються повідомленнями, показуючи порядок і логіку виконання операцій. Діаграма складається з таких основних елементів:

Актори (Actors): Зазвичай позначаються піктограмами або назвами. Це користувачі чи інші системи, які взаємодіють із системою. Актори можуть бути

Об'єкти або класи: Розміщуються горизонтально на діаграмі. Вони позначаються прямокутниками з іменем об'єкта або класу під прямокутником. Кожен об'єкт має «життєвий цикл», який представлений вертикальною пунктирною лінією (лінія життя).

Повідомлення: Це лінії зі стрілками, які з'єднують об'єкти. Вони показують передачу повідомлень чи виклик методів. Стрілка може бути синхронною (звичайна стрілка) або асинхронною (лінія з відкритим трикутником) та з пунктирною лінією, що показує повернення результату.

Активності: Вказують періоди, протягом яких об'єкт виконує певну дію. На діаграмі це позначається прямокутником, накладеним на лінію життя. **Контрольні структури:** Використовуються для відображення умов, циклів або альтернативних сценаріїв. Наприклад, блоки "alt" (альтернатива) або "loop" (цикл).

Тема проекту

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.

Діаграма розгортання

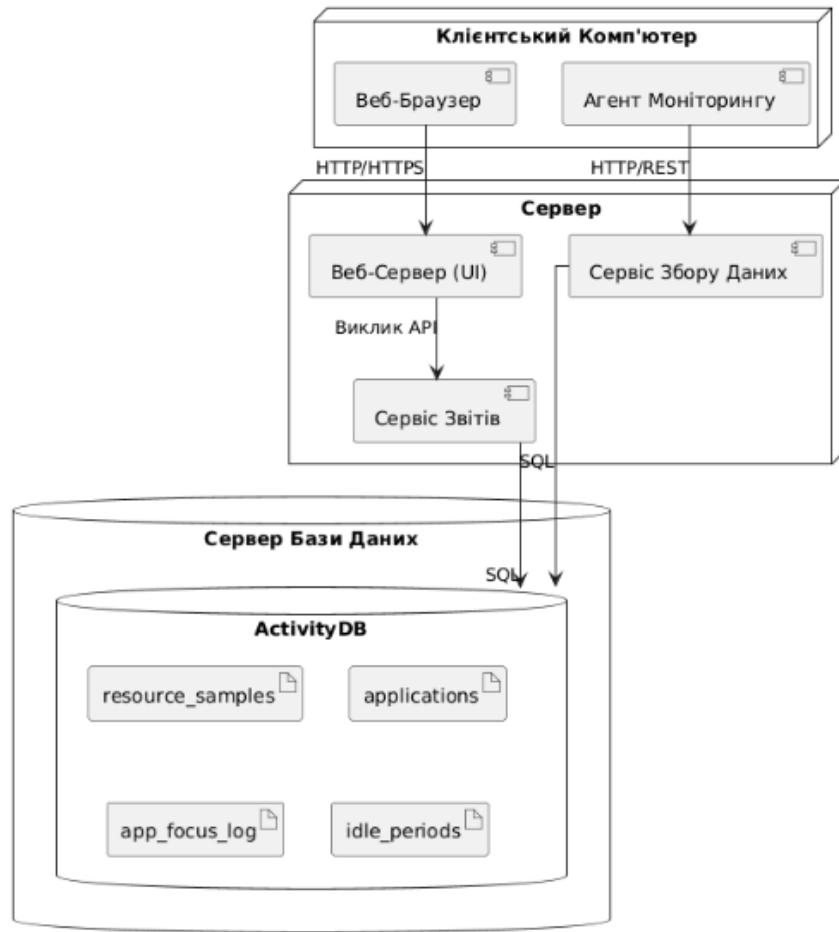


Рис.1 – Діаграма розгортання

Опис діаграми розгортання

1. Вузол «Клієнтський Комп'ютер» (Клієнтська сторона)

Цей вузол представляє кінцевий пристрій користувача. Він виконує дві функції: 1) на ньому відбувається безпосередній збір статистики, 2) він слугує точкою доступу для перегляду звітів.

- Пристрій (<<device>>): Client PC під управлінням операційної системи (наприклад, Windows, macOS або Linux OS).
- Середовище виконання (<<execution environment>>): Подвійне:
 1. Operating System (Service/Daemon): Середовище, в якому у фоновому режимі виконується артефакт Агент Моніторингу для збору даних.
 2. Web Browser: Стандартний браузер, який слугує середовищем для запуску клієнтського веб-інтерфейсу для перегляду звітів.
- Артефакт (<<artifact>>):
 3. monitor-agent.exe (або monitor-agent.daemon): Це Агент Моніторингу. Скомпільований нативний виконуваний файл, який працює у фоновому режимі, збирає дані (CPU, RAM, вікна, простій) і відправляє їх на сервер.
 4. (Standard Browser): Веб-Браузер є стандартним компонентом системи, який отримує UI від Веб-Сервера.

2. Вузол «Сервер» (Серверна сторона / SOA)

Цей вузол відповідає за обробку бізнес-логіки згідно з архітектурою SOA (Сервіс-орієнтована архітектура). Він приймає дані, обробляє запити на звіти та надає веб-інтерфейс.

- Пристрій (<<device>>): Application Server під управлінням серверної операційної системи (наприклад, Linux OS).
- Середовище виконання (<<execution environment>>): Java Application Server (наприклад, Apache Tomcat або JBoss), який керує життєвим циклом Java-додатків та обробляє HTTP-запити до сервісів.
- Артефакт (<<artifact>>): activity-monitor-services.war. Це розгортуваний веб-архів, який містить скомпільовані серверні класи для всіх сервісів SOA:
 - Сервіс Збору Даних: Реалізує API (REST endpoint) для прийому даних від Агента.
 - Сервіс Звітів: Реалізує API для Веб-Сервера (UI), виконуючи складну логіку генерації звітів (з патерном Visitor).
 - Веб-Сервер (UI): Надає статичний HTML/JS/CSS (або рендерить) для Веб-Браузера користувача.

3. Вузол «Сервер Бази Даних» (Рівень даних)

Цей вузол призначений для персистентного (довготривалого) зберігання всієї зібраної статистики.

- Пристрій (<<device>>): Database Server. Спеціалізований сервер, оптимізований для роботи систем управління базами даних (СУБД).
- Середовище виконання (<<execution environment>>): ActivityDB (PostgreSQL). Система управління реляційними базами даних (PostgreSQL), що відповідає за цілісність, зберігання та доступ до даних.
- Артефакт (<<artifact>>): ActivityDB_Schema. Даний артефакт представляє собою розгорнуту схему бази даних, що включає таблиці (resource_samples, applications, app_focus_log, idle_periods), їхні зв'язки, індекси та обмеження цілісності.

Комуунікаційні Протоколи

- Зв'язок Клієнтський Комп'ютер - Сервер (HTTP/HTTPS та HTTP/REST):
 - Агент Моніторингу → Сервіс Збору Даних (HTTP/REST): Агент надсилає зібрану статистику (наприклад, у форматі JSON) на серверний API методом POST.
 - Веб-Браузер → Веб-Сервер (UI) (HTTP/HTTPS): Браузер запитує веб-сторінку (інтерфейс звітів) і отримує HTML/JS/CSS. Далі цей інтерфейс спілкується з Сервісом Звітів для отримання даних.
- Зв'язок Сервер - Сервер Бази Даних (SQL/JDBC):
 - Сервер додатку взаємодіє з базою даних за допомогою технології JDBC (Java Database Connectivity) для виконання SQL-запитів:
 - Сервіс Збору Даних → ActivityDB: Виконує операції INSERT для збереження нової статистики.
 - Сервіс Звітів → ActivityDB: Виконує складні операції SELECT (з GROUP BY, JOIN, AVG) для читання та агрегації даних при побудові звітів.

Діаграма компонентів

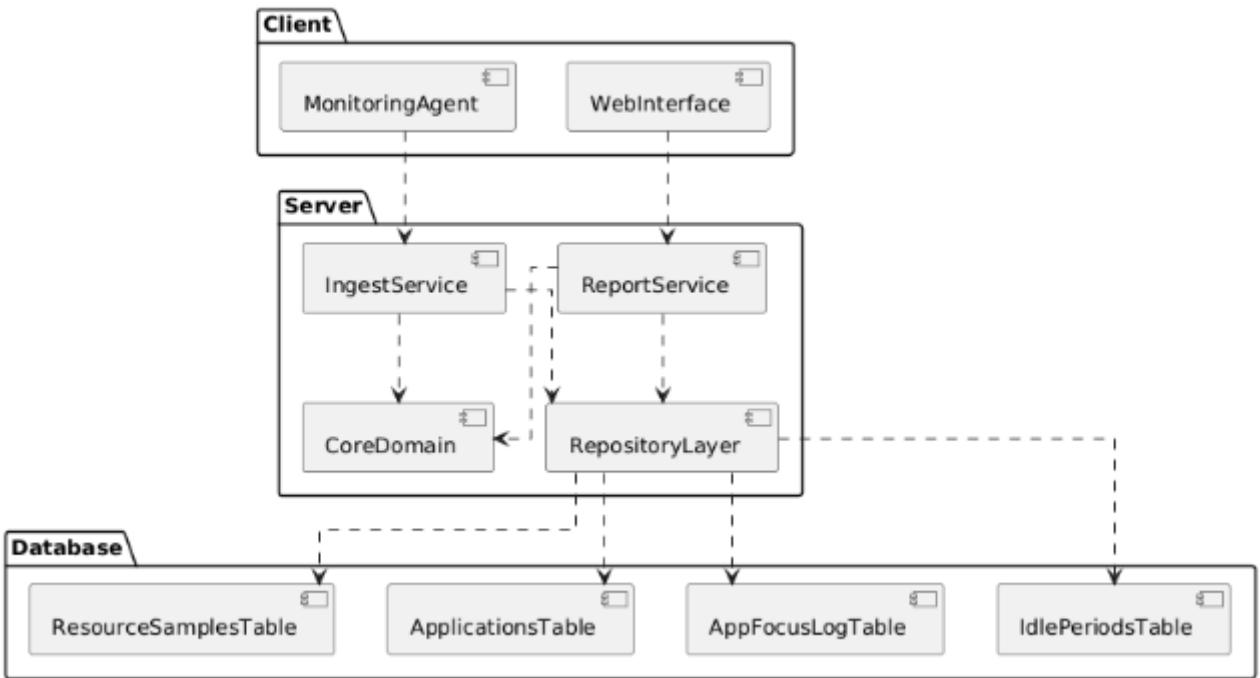


Рис.2 – Діаграма компонентів

Опис діаграми компонентів

Рівень Клієнта (Client)

Цей рівень представляє кінцевий пристрій користувача. На ньому виконуються два окремі компоненти: один для фонового збору даних, інший для взаємодії з користувачем.

- MonitoringAgent: Компонент, що являє собою фоновий додаток (агент). Він відповідає за збір всієї системної статистики (CPU, RAM, активні вікна, простій) та ініціалізує запити до IngestService для відправки цих даних на сервер.
- WebInterface: Компонент, що являє собою клієнтський веб-додаток (UI). Він відповідає за користувацький інтерфейс, через який користувач може запитувати та переглядати звіти. Ініціює запити до ReportService.

Рівень Сервера (Server)

Серверний рівень реалізує всю основну бізнес-логіку та логіку доступу до даних, слідуючи архітектурі SOA (Сервіс-орієнтована архітектура) з двома основними точками входу.

- IngestService: Виконує роль точки входу (API) для *прийому* даних. Його єдина відповідальність — отримати дані від MonitoringAgent, валідувати їх, використати CoreDomain для структурування та передати на RepositoryLayer для збереження.
- ReportService: Виконує роль точки входу (API) для *генерації* звітів. Він отримує запити від WebInterface, виконує складну бізнес-логіку (агрегацію,

аналіз, реалізацію патерну Visitor) та звертається до RepositoryLayer для отримання даних.

- CoreDomain: Компонент, що містить визначення всіх бізнес-сущностей системи (класи-моделі для ResourceSample, Application, IdlePeriod тощо). Він забезпечує уніфіковану структуру даних, яка використовується на сервісному рівні та рівні доступу до даних.
- RepositoryLayer: Шар абстракції доступу до даних, що реалізує шаблон "Repository". Він інкапсулює всю логіку взаємодії з базою даних (SQL-запити), надаючи бізнес-сервісам (IngestService, ReportService) простий та зрозумілий інтерфейс для управління персистентністю даних.

Рівень Бази Даних (Database)

- ResourceSamplesTable, ApplicationsTable, AppFocusLogTable, IdlePeriodsTable: Компоненти, що символізують таблиці в реляційній базі даних. Вони є кінцевою точкою для зберігання даних, і з ними безпосередньо взаємодіє лише RepositoryLayer.

Загальний потік взаємодії

Система має два основні, незалежні потоки взаємодії:

Потік 1: Збір даних

1. MonitoringAgent на клієнті збирає дані про активність (напр., навантаження CPU).
2. Агент надсилає запит до єдиної точки входу для запису — IngestService.
3. IngestService використовує сущності з CoreDomain та звертається до RepositoryLayer для збереження даних.
4. RepositoryLayer трансформує запит у SQL (INSERT) та взаємодіє з відповідними таблицями (напр., ResourceSamplesTable).

Потік 2: Отримання звіту

1. Користувач через WebInterface запитує звіт (напр., "Середнє навантаження CPU по годинах").
2. WebInterface надсилає запит до точки входу для читання — ReportService.
3. ReportService виконує бізнес-логіку (агрегацію) та звертається до RepositoryLayer для отримання необхідних даних.
4. RepositoryLayer формує та виконує SQL-запити (SELECT, GROUP BY) до ResourceSamplesTable і повертає дані сервісу для подальшої обробки та відправки на WebInterface.

Діаграми послідовностей

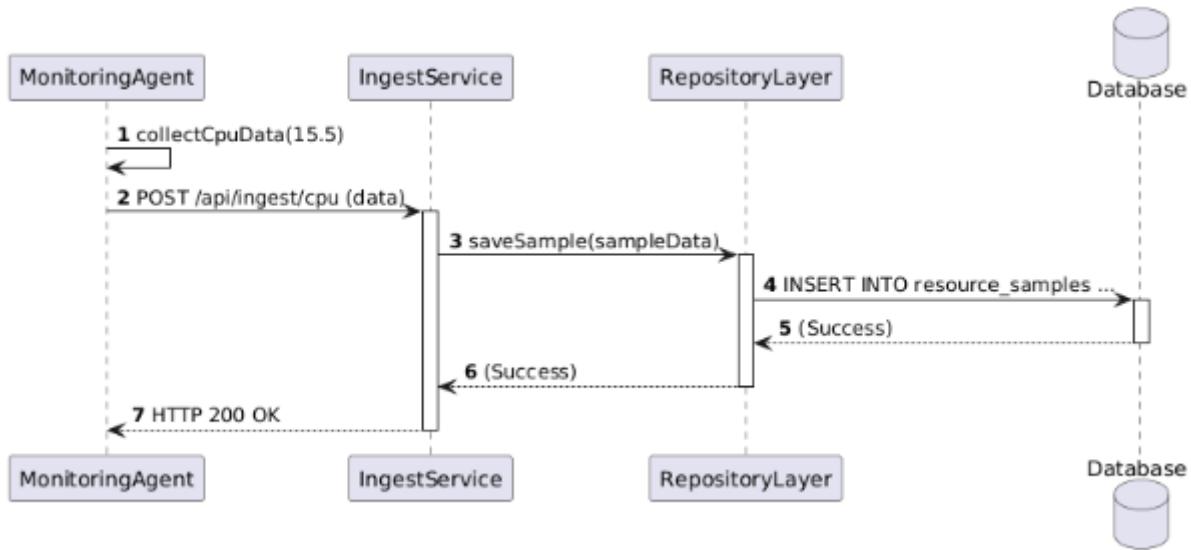


Рис.3 – Діаграма послідовності для збору статистики

Основний перебіг подій (успішний сценарій)

- MonitoringAgent, що працює у фоновому режимі, виконує внутрішню операцію збору даних (наприклад, collectCpuData(15.5)).
- Агент формує пакет даних і відправляє HTTP POST-запит на кінцеву точку IngestService (/api/ingest/cpu).
- IngestService отримує запит, валідує дані та викликає метод saveSample() у RepositoryLayer, передаючи йому оброблені дані.
- RepositoryLayer трансформує дані в SQL-запит і виконує операцію INSERT у Database.
- Database успішно зберігає дані та повертає RepositoryLayer підтвердження (Success).
- RepositoryLayer повертає підтвердження (Success) на IngestService.
- IngestService завершує HTTP-транзакцію, повертаючи MonitoringAgent відповідь HTTP 200 OK.

Альтернативний перебіг подій (сценарій помилки)

- У разі, якщо IngestService не може зберегти дані в Database (наприклад, через збій з'єднання з БД або порушення цілісності даних).
- Database повертає помилку на RepositoryLayer.
- RepositoryLayer переоплює цю помилку і передає її вище, на IngestService.
- IngestService реєструє помилку (логує її) і не повертає HTTP 200 OK.
- IngestService повертає MonitoringAgent відповідь про помилку сервера (наприклад, HTTP 500 Internal Server Error).
- (Опціонально) MonitoringAgent отримує помилку і не видаляє зібраний дані, а ставить їх у чергу на повторну відправку.

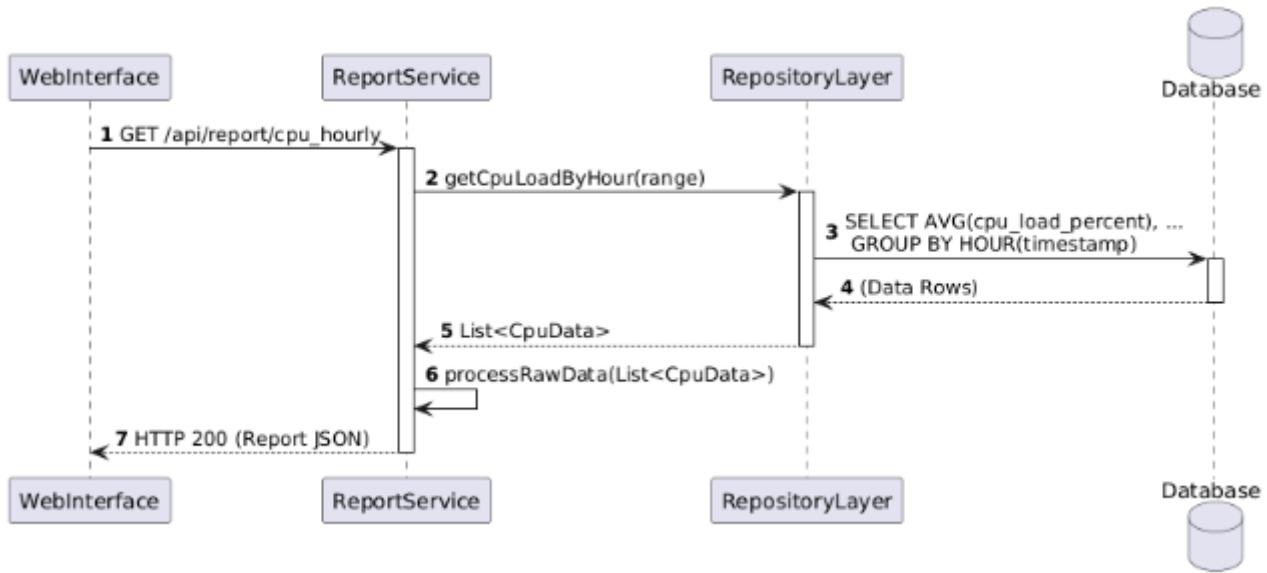


Рис.4 – Діаграма послідовності генерації звіту

Основний перебіг подій (успішний сценарій)

- Процес починається, коли користувач через WebInterface ініціює запит на генерацію звіту (наприклад, обирає "Звіт по CPU").
- WebInterface надсилає HTTP GET-запит до ReportService на відповідну кінцеву точку (/api/report/cpu_hourly).
- ReportService отримує запит і делегує отримання даних до RepositoryLayer, викликаючи метод getCpuLoadByHour().
- RepositoryLayer виконує агрегуючий SQL-запит SELECT ... GROUP BY до Database.
- Database обробляє запит і повертає набір рядків (Data Rows) до RepositoryLayer.
- RepositoryLayer перетворює "сирі" дані у список об'єктів (List<CpuData>) і повертає його до ReportService.
- ReportService виконує фінальну обробку та форматування даних (внутрішній виклик processRawData).
- ReportService завершує операцію, надсилаючи WebInterface відповідь HTTP 200 OK з готовим звітом у форматі JSON.

Альтернативний перебіг подій (сценарій помилки)

- Цей сценарій активується, якщо під час взаємодії RepositoryLayer з Database виникає помилка (наприклад, невірний SQL-запит або недоступність БД).
- Database (або RepositoryLayer) генерує помилку.
- Ця помилка передається вгору по ланцюжку викликів: від RepositoryLayer до ReportService.
- ReportService, обробивши виняткову ситуацію, формує HTTP-відповідь зі статусом помилки сервера (наприклад, 500 Internal Server Error).

- WebInterface отримує відповідь про невдачу і повідомляє про це користувача (наприклад, "Не вдалося завантажити звіт").

Код програми

```
// // IReportVisitor.java
```

```
public interface IReportVisitor {
    void visit(ResourceEvent event);
    void visit(WindowEvent event);
    void visit(IdleEvent event);
    Report getReport();
}
```

```
// // ActivityEvent.java
```

```
public abstract class ActivityEvent {
    public Date timestamp;
    public ActivityEvent() {
        this.timestamp = new Date();
    }
    public abstract void accept(IReportVisitor visitor);
}
```

```
// // ResourceEvent.java
```

```
public class ResourceEvent extends ActivityEvent {
    public double cpuLoad;
    public double ramUsage;

    public ResourceEvent(double cpu, double ram) {
        super();
        this.cpuLoad = cpu;
        this.ramUsage = ram;
    }
}
```

```
@Override
```

```
public void accept(IReportVisitor visitor) {
    visitor.visit(this);
}
```

```
}
```

```
// // WindowEvent.java
public class WindowEvent extends ActivityEvent {
    public String processName;
    public String windowTitle;

    public WindowEvent(String process, String title) {
        super();
        this.processName = process;
        this.windowTitle = title;
    }
}
```

```
@Override
public void accept(IReportVisitor visitor) {
    visitor.visit(this);
}
}
```

```
// // IdleEvent.java
public class IdleEvent extends ActivityEvent {
    public enum IdleState { START, END }
    public IdleState state;

    public IdleEvent(IdleState state) {
        super();
        this.state = state;
    }
}
```

```
@Override
public void accept(IReportVisitor visitor) {
    visitor.visit(this);
}
}
```

```
// // Report.java
public class Report {
```

```
public String title;
public Map<String, Object> data;

public Report(String title) {
    this.title = title;
    this.data = new HashMap<>();
}

}

// // IReportView.java
public interface IReportView {
    void render(Report report);
}

// // TextView.java
public class TextView implements IReportView {
    @Override
    public void render(Report report) {
        System.out.println("--- " + report.title + " ---");
        for (Map.Entry<String, Object> entry : report.data.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue().toString());
        }
        System.out.println("-----");
    }
}

// // HtmlView.java
public class HtmlView implements IReportView {
    @Override
    public void render(Report report) {
        System.out.println("<html><body>");
        System.out.println(" <h1>" + report.title + "</h1>");
        System.out.println(" <ul>");
        for (Map.Entry<String, Object> entry : report.data.entrySet()) {
            System.out.println("   <li><b>" + entry.getKey() + ":</b> " +
entry.getValue().toString() + "</li>");
        }
    }
}
```

```
        System.out.println(" </ul>");
        System.out.println("</body></html>");
    }
}

// // EventRepository.java
public class EventRepository {
    private List<ActivityEvent> eventLog = new ArrayList<>();

    public void addEvent(ActivityEvent event) {
        System.out.println("[Repo] Збережено: " + event.getClass().getSimpleName());
        eventLog.add(event);
    }

    public List<ActivityEvent> getEvents(Date start, Date end) {
        return new ArrayList<>(eventLog);
    }
}

// // CpuLoadVisitor.java
public class CpuLoadVisitor implements IReportVisitor {
    private double totalLoad = 0;
    private int samples = 0;

    @Override
    public void visit(ResourceEvent event) {
        totalLoad += event.cpuLoad;
        samples++;
    }

    @Override public void visit(WindowEvent event) { }
    @Override public void visit(IdleEvent event) { }

    @Override
    public Report getReport() {
        Report report = new Report("Середнє Навантаження CPU");
        double avg = (samples == 0) ? 0 : totalLoad / samples;
    }
}
```

```
report.data.put("Середнє навантаження", String.format("%,.2f%%", avg));
report.data.put("Кількість вимірювань", samples);
return report;
}

}

// // AppUsageVisitor.java
public class AppUsageVisitor implements IReportVisitor {
    private Map<String, Integer> appCounts = new HashMap<>();

    @Override
    public void visit(WindowEvent event) {
        String app = event.processName;
        appCounts.put(app, appCounts.getOrDefault(app, 0) + 1);
    }

    @Override public void visit(ResourceEvent event) { }
    @Override public void visit(IdleEvent event) { }

    @Override
    public Report getReport() {
        Report report = new Report("Звіт по % часу в програмах");
        report.data.put("Дані (по подіях)", appCounts.toString());
        return report;
    }
}

// // ReportService.java
public class ReportService {
    private EventRepository repository;

    public ReportService(EventRepository repository) {
        this.repository = repository;
    }

    public Report generateReport(IReportVisitor visitor) {
        Date now = new Date();
        visitor.visit(new WindowEvent(now));
        visitor.visit(new ResourceEvent(now));
        visitor.visit(new IdleEvent(now));
    }
}
```

```
Date past = new Date(System.currentTimeMillis() - 86400000);

List<ActivityEvent> events = repository.getEvents(past, now);

System.out.println("[ReportService] Отримано " + events.size() + " подій для
аналізу.");
Iterator<ActivityEvent> iterator = events.iterator();
while (iterator.hasNext()) {
    ActivityEvent event = iterator.next();
    event.accept(visitor);
}

return visitor.getReport();
}

}

// // ISystemCollector.java
public interface ISystemCollector {
    void collect();
}

// // WindowsCollector.java
public class WindowsCollector implements ISystemCollector {
    private EventRepository repository;
    public WindowsCollector(EventRepository repo) { this.repository = repo; }

    @Override
    public void collect() {
        repository.addEvent(new ResourceEvent(15.5, 2048));
        repository.addEvent(new WindowEvent("chrome.exe", "Google"));
    }
}

// // LinuxCollector.java
public class LinuxCollector implements ISystemCollector {
    private EventRepository repository;
    public LinuxCollector(EventRepository repo) { this.repository = repo; }
```

```
@Override
public void collect() {
    repository.addEvent(new ResourceEvent(10.1, 1024));
    repository.addEvent(new WindowEvent("/usr/bin/firefox", "Mozilla"));
}
}

// // IMonitorFactory.java
public interface IMonitorFactory {
    ISystemCollector createCollector(EventRepository repository);
}

// // WindowsMonitorFactory.java
public class WindowsMonitorFactory implements IMonitorFactory {
    @Override
    public ISystemCollector createCollector(EventRepository repository) {
        System.out.println("[Factory] Створено WindowsCollector");
        return new WindowsCollector(repository);
    }
}

// // MonitoringManager.java
public class MonitoringManager {
    private ISystemCollector collector;
    private Timer timer;

    public MonitoringManager(IMonitorFactory factory, EventRepository repository) {
        this.collector = factory.createCollector(repository);
    }

    public void start() {
        System.out.println("[Manager] Запуск моніторингу...");
        timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
```

```
        collector.collect();
    }
}, 1000, 5000);
}

public void stop() {
    System.out.println("[Manager] Зупинка моніторингу.");
    if (timer != null) {
        timer.cancel();
    }
}

// // ICommand.java
public interface ICommand {
    void execute();
}

// // StartCommand.java
public class StartCommand implements ICommand {
    private MonitoringManager manager;

    public StartCommand(MonitoringManager manager) {
        this.manager = manager;
    }

    @Override
    public void execute() {
        manager.start();
    }
}

// // StopCommand.java
public class StopCommand implements ICommand {
    private MonitoringManager manager;

    public StopCommand(MonitoringManager manager) {
```

```
        this.manager = manager;
    }

    @Override
    public void execute() {
        manager.stop();
    }
}

// // UserInterface.java
public class UserInterface {
    private ICommand startCmd;
    private ICommand stopCmd;

    private ReportService reportService;

    private IReportView view;

    public UserInterface(MonitoringManager manager, ReportService reportSvc) {
        this.startCmd = new StartCommand(manager);
        this.stopCmd = new StopCommand(manager);

        this.reportService = reportSvc;

        this.view = new TextView();
    }

    public void setView(IReportView view) {
        System.out.println("\n[UI] Змінено вигляд на: " +
view.getClass().getSimpleName());
        this.view = view;
    }

    public void pressStartButton() {
        System.out.println("\n[UI] Натиснуто 'Start'");
        startCmd.execute();
    }
}
```

```
public void pressStopButton() {
    System.out.println("\n[UI] Натиснуто 'Stop'");
    stopCmd.execute();
}

public void pressShowCpuReport() {
    System.out.println("\n[UI] Запит звіту по CPU...");
    Report report = reportService.generateReport(new CpuLoadVisitor());
    view.render(report);
}

public void pressShowAppReport() {
    System.out.println("\n[UI] Запит звіту по Програмах...");
    Report report = reportService.generateReport(new AppUsageVisitor());
    view.render(report);
}

// // Main.java
public class Main {
    public static void main(String[] args) throws InterruptedException {

        IMonitorFactory factory = new WindowsMonitorFactory();

        EventRepository repository = new EventRepository();
        MonitoringManager manager = new MonitoringManager(factory, repository);
        ReportService reportService = new ReportService(repository);

        UserInterface ui = new UserInterface(manager, reportService);

        ui.pressStartButton();

        Thread.sleep(6000);

        ui.pressStopButton();
```

```
Thread.sleep(6000);

ui.pressStartButton();
Thread.sleep(6000);
ui.pressStopButton();

ui.pressShowCpuReport();
ui.pressShowAppReport();

ui.setView(new HtmlView());
ui.pressShowCpuReport();

}

}
```

Контрольні запитання

1. Що собою становить діаграма розгортання?

Діаграма розгортання (Deployment Diagram) у UML показує фізичне розміщення апаратних вузлів (серверів, клієнтів) і компонентів системи на цих вузлах. Вона відображає, як програмне забезпечення реалізується на апаратних елементах.

2. Які бувають види вузлів на діаграмі розгортання?

Фізичні вузли (Node): реальні апаратні пристрої (сервер, комп'ютер, мобільний пристрій).

Вузли виконання (Execution Environment): середовище, у якому запускаються компоненти (операційна система, віртуальна машина, контейнер).

3. Які бувають зв'язки на діаграмі розгортання?

Асоціація між вузлами (Communication Path): показує можливість обміну даними між вузлами.

Залежність (Dependency): вказує, що один вузол або компонент залежить від іншого.

4. Які елементи присутні на діаграмі компонентів?

- Компоненти (Component): самостійні частини ПЗ.
- Інтерфейси (Interface): порти, через які компоненти взаємодіють.

- Пакети (Package): групування компонентів.
- Зв'язки (Dependency, Association): взаємозв'язки між компонентами.

5. Що становлять собою зв'язки на діаграмі компонентів?

Зв'язки відображають залежності між компонентами: хто на кого покладається, хто надає чи використовує інтерфейс іншого компонента.

6. Які бувають види діаграм взаємодії?

Діаграма послідовностей (Sequence Diagram) – показує порядок повідомлень між об'єктами.

Діаграма комунікацій (Communication Diagram) – показує зв'язки між об'єктами та обмін повідомленнями.

Діаграма часу (Timing Diagram) – показує зміни станів об'єктів у часі.

Діаграма взаємодії (Interaction Overview Diagram) – поєднує елементи кількох діаграм взаємодії.

7. Для чого призначена діаграма послідовностей?

Вона описує порядок і часову послідовність взаємодії між об'єктами системи для реалізації певного сценарію чи варіанту використання.

8. Які ключові елементи можуть бути на діаграмі послідовностей?

- Об'єкти/Актори (Lifelines)
- Повідомлення (Messages) – виклики методів між об'єктами
- Активності (Activation bars) – час виконання дії об'єкта
- Події створення/знищення об'єктів

9. Як діаграми послідовностей пов'язані з діаграмами варіантів

використання?

Кожна діаграма послідовностей реалізує сценарій певного варіанту використання, деталізуючи, як актори взаємодіють із системою крок за кроком.

10. Як діаграми послідовностей пов'язані з діаграмами класів?

Повідомлення на діаграмі послідовностей зазвичай відповідають методам класів, а об'єкти на діаграмі є екземплярами класів із діаграми класів.

Висновки

Під час виконання даної лабораторної роботи я навчився проєктувати та створювати діаграми розгортання, діаграми компонентів та діаграми послідовностей, також я реалізував мінімальну робочу модель монітору активності зі збереженням історії.