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 розроблення програмного забезпечення
«Основи проектування розгортання»

Виконав
студент групи ІА–34:
Бородай А.С.

Київ 2025

Зміст

Теоретичні відомості

Діаграми розгортання (**Deployment Diagram**) – представлення фізичного розташування системи, яке показує, 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).

Між вузлами можуть стояти **зв'язки**, які зазвичай зображують у вигляді прямої лінії. Як і на інших діаграмах, у зв'язків можуть бути атрибути множинності (для показання, наприклад, підключення 2х і більше клієнтів до одного сервера) і назва. У назві, як правило, міститься спосіб зв'язку між двома вузлами – це може бути назва протоколу (HTTP, IPC) або технологія, що використовується для забезпечення взаємодії вузлів (.NET Remoting, WCF).

Вузли можуть містити артефакти (**artifacts**), які є фізичним уособленням програмного забезпечення; зазвичай це файли. Такими файлами можуть бути виконувані файли (такі як файли .exe, двійкові файли, файли DLL, файли JAR, збірки або сценарії) або файли даних, конфігураційні файли, HTML-документи тощо. Перелік артефактів усередині вузла вказує на те, що на даному вузлі артефакт розгортається в систему, що запускається.

Артефакти можна зображати у вигляді прямокутників класів або перераховувати їхні імена всередині вузла. Якщо ви показуєте ці елементи у вигляді прямокутників класів, то можете додати значок документа або ключове слово «artifact». Можна супроводжувати вузли або артефакти значеннями у вигляді міток, щоб вказати різну

цікаву інформацію про вузол, наприклад постачальника, операційну систему, місце розташування – загалом, усе, що спаде вам на думку.

Часто у вас буде безліч фізичних вузлів для розв'язання однієї й тієї самої логічної задачі. Можна відобразити цей факт, намалювавши безліч прямокутників вузлів або поставивши число у вигляді значення-мітки.

Діаграми розгортань розрізняють двох видів: **описові та екземплярні**.

На **діаграмах описової форми** вказуються вузли, артефакти і зв'язки між вузлами без вказівки конкретного обладнання або програмного забезпечення, необхідного для розгортання. Такий вид діаграм корисний на ранніх етапах розроблення для розуміння, які взагалі фізичні пристрої необхідні для функціонування системи або для опису процесу розгортання в загальному ключі.

Діаграми екземплярної форми несуть у собі екземпляри обладнання, артефактів і зв'язків між ними. Під екземплярами розуміють конкретні елементи – ПК із відповідним набором характеристик і встановленим ПЗ; цілком може бути, у межах однієї організації це може бути якийсь конкретний вузол. Діаграми екземплярної форми розробляють на завершальних стадіях розроблення ПЗ – коли вже відомі та сформульовані вимоги до програмного комплексу, обладнання закуплено і все готово до розгортання. Діаграми такої форми являють собою скоріше план розгортання в графічному вигляді, ніж модель розгортання.

Діаграма компонентів UML є представленням проєктованої 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.

Коли на діаграмі представляють **фізичне розбиття**, то в такому разі на діаграмі компонентів показують компоненти та залежності між ними. Залежності показують, що класи в з одного компонента використовують класи з іншого компонента. Фізична

модель використовується для розуміння які компоненти повинні бути зібрані в інсталяційний пакет.

Компоненти можуть поділятися за фізичними одиницями – окремі вузли розподіленої системи – набір комп'ютерів і серверів; на кожному з вузлів можуть бути встановлені різні виконувані компоненти. Такий вид діаграм компонентів застарів і зазвичай замість нього використовують діаграму розгортань.

На діаграмах компонентів з виконуваним поділом компонентів кожен компонент являє собою деякий файл – виконувані файли (.exe), файли вихідних кодів, сторінки html, бази даних і таблиці тощо. У цьому разі діаграма схожа на діаграму класів, але на більш верхньому рівні – рівні виконуваних файлів або процесів.

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.

Actors – зазвичай позначаються піктограмами або назвами. Це користувачі чи інші системи, які взаємодіють із системою. Актори можуть бути зовнішніми стосовно моделювання системи.

Об'єкти або класи – 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).

Тема: Основи проектування розгортання.

Мета: Навчитися проєктувати 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.

Хід роботи

20. **Mind-mapping software**

Візуальний додаток для складання "карт пам'яті" з можливістю роботи з декількома картами (у вкладках), автоматичного промальовування ліній, додавання вкладених файлів, картинок, відеофайлів (попередній перегляд); можливість додавання значків категорій / терміновості, обведення областей карти (поділ пунктирною лінією).

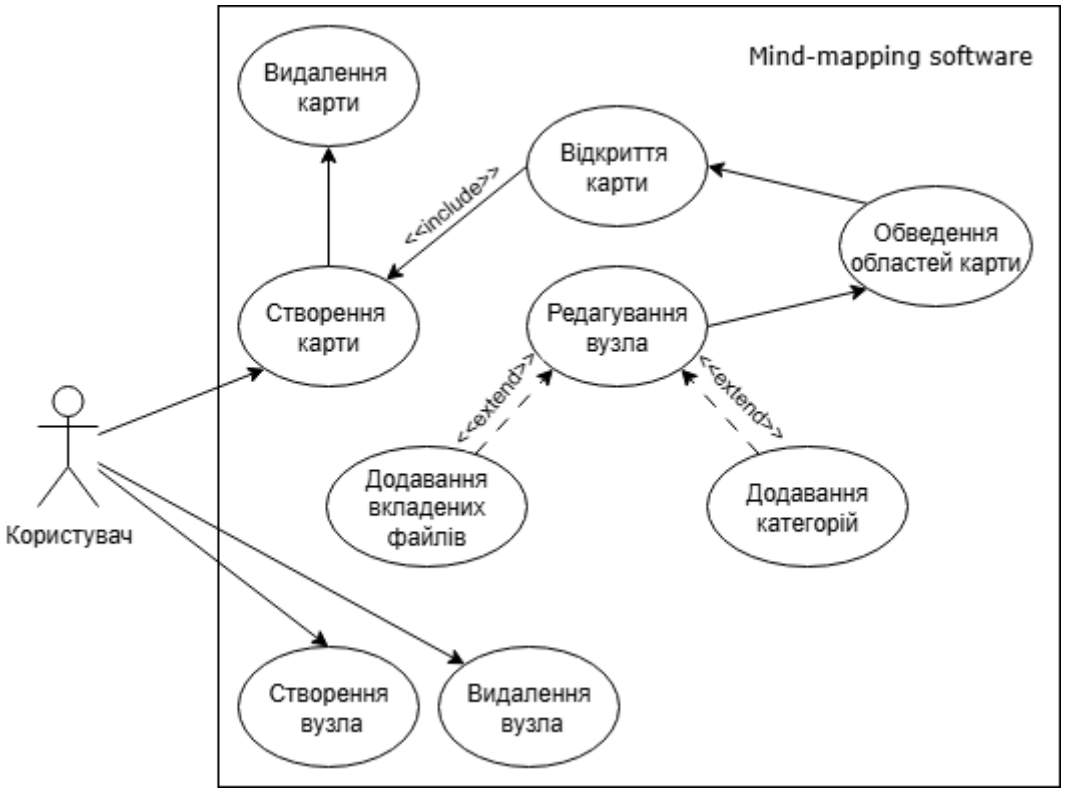

Рисунок 1 – Діаграма варіантів використання із попередньої роботи

Deployment Diagram

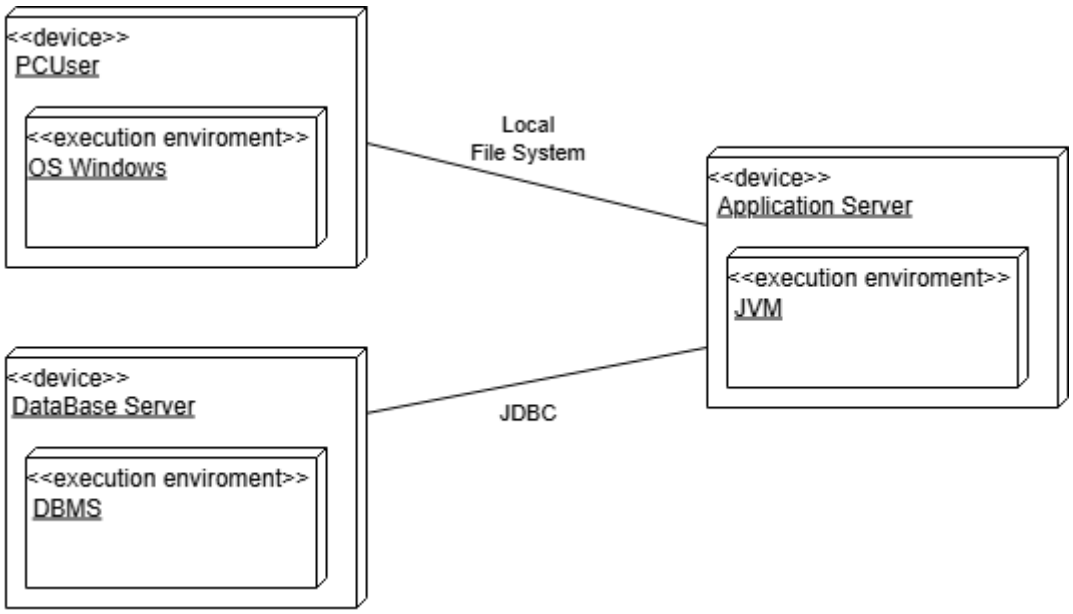

Рисунок 2 – Діаграма розгортання

<<device>> PCUser – фізичний пристрій користувача.

<<execution enviroment>> OS Windows – операційна система, на якій працює застосунок.

<<device>> Application Server – локальна серверна машина.

<< execution enviroment >> JVM – середовище виконання для сервісів обробки.

<<device>> DataBase Server – сервер для зберігання даних.

<< execution enviroment >> DBMS – СУБД, для зберігання карт у json файлах.

Component diagram

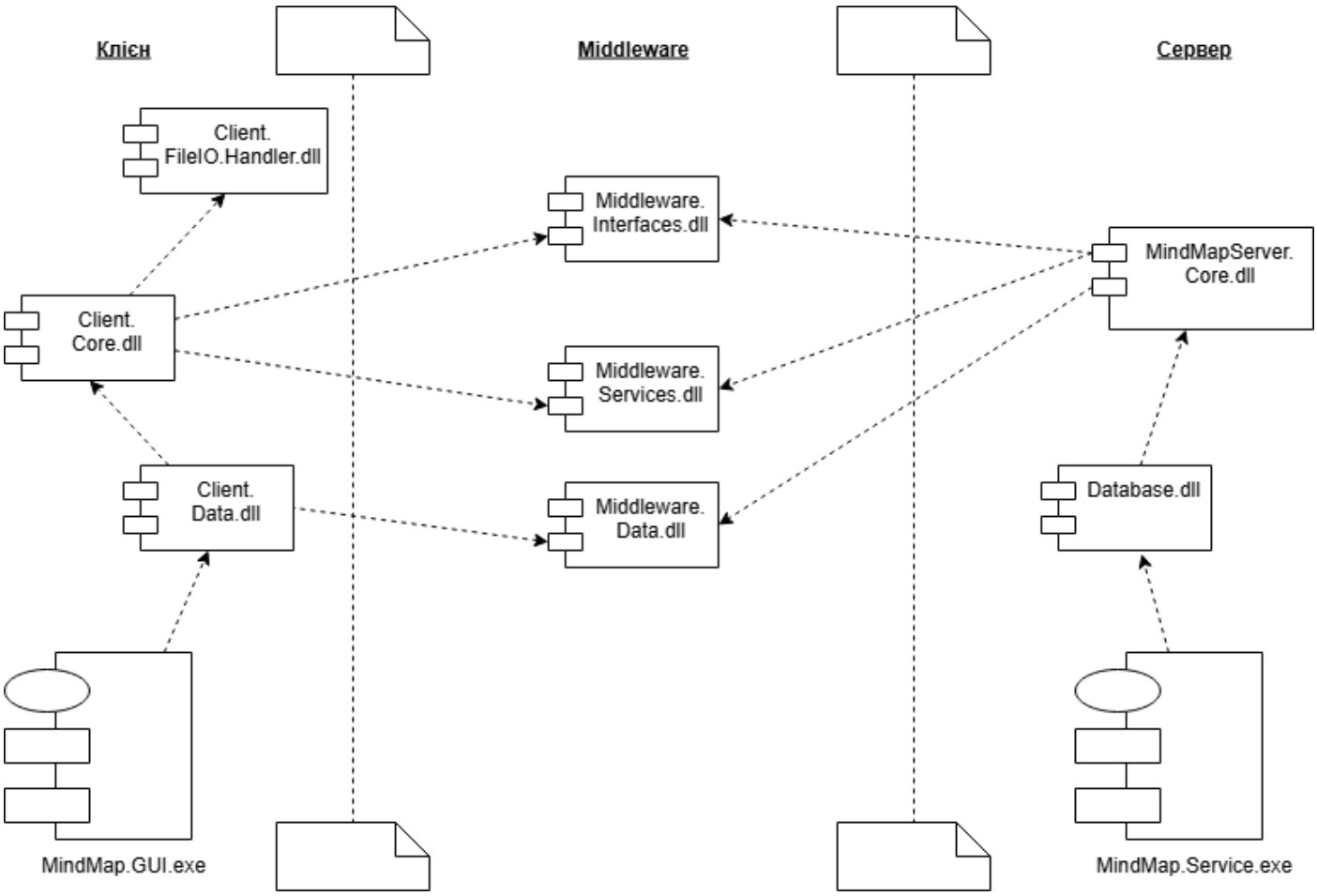

Рисунок 3 – Діаграма компонентів

Client:

- MindMap.GUI.exe – файл інтерфейсу користувача.
- Client.Data.dll – файл для роботи із даними на клієнті.
- Client.Core.dll – головний виконуваний файл (головна логіка застосунку).
- Client.FileIO.Handler.dll – відповідає за роботу із файлами.

Middleware:

- Middleware.Interfaces.dll – спільні інтерфейси сервісів, для узгодженість між клієнтом та сервером.
- Middleware.Services.dll – спільна бізнес-логіка між клієнтом та сервером.
- Middleware.Data.dll – DTO, для узгодженості між клієнтом та сервером.

Server:

- MindMap.Service.exe – серверний застосунок, що оброблює запити.

- Database.dll – модуль доступу до бази даних.
- Middleware.Server.Core.dll – серверна бізнес-логіка.

Sequence Diagram

Діаграми послідовностей, для сценаріїв використання прописаних в попередній лабораторній роботі:

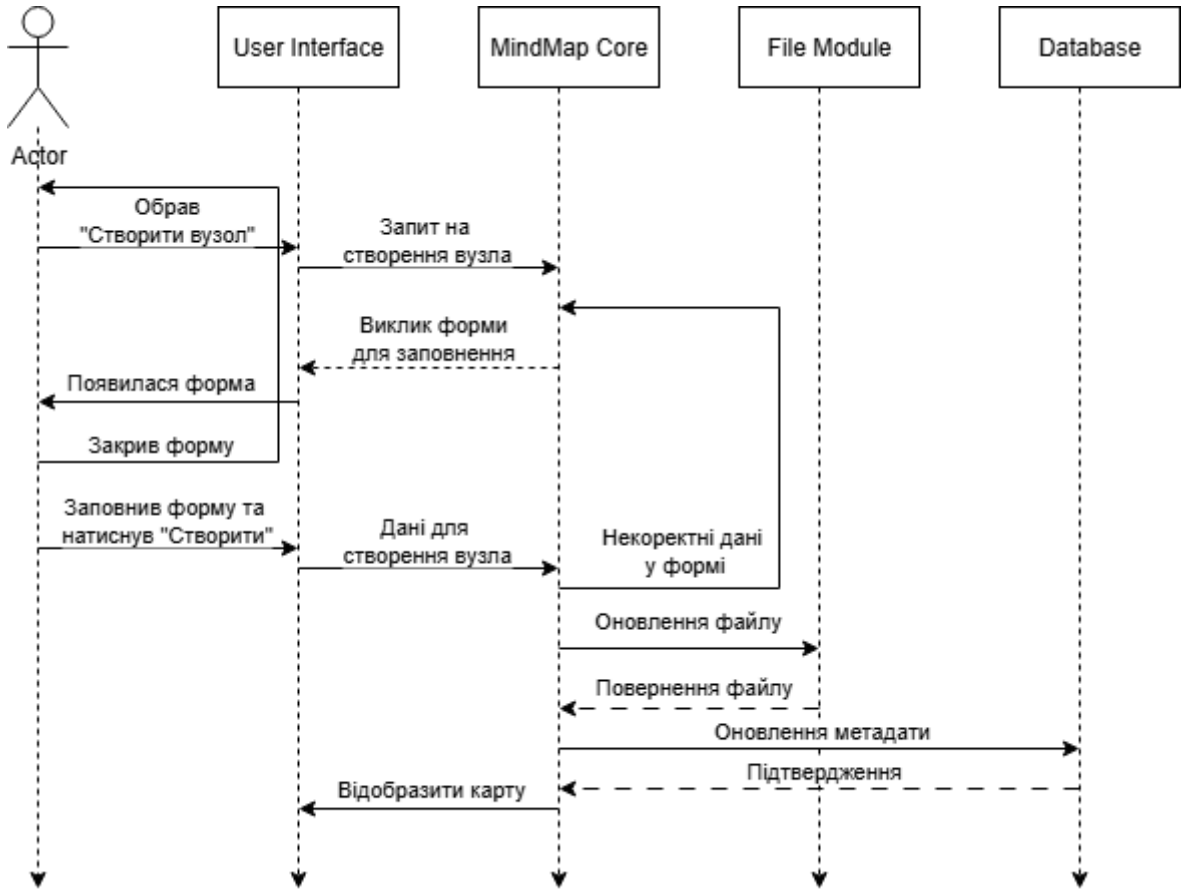

Рисунок 4 – Діаграма послідовності для варіанту використання: «Створення вузла»

Опис:

1. Користувач натискає “Створити вузол”.
2. Інтерфейс викликає програму для оброблення дій користувача.
3. Ядро викликає форму для створення вузла.
4. Користувач закриває форму або заповнює форму.
5. Інтерфейс передає дані для створення вузла програмі, якщо форма заповнена.
6. Програма створює новий вузол.
7. File Module оновлює файл карти та повертає оновлений файл.

8. Застосунок відправляє файл до бази даних.
9. База даних дає підтвердження, що файл оброблений.
10. Інтерфейс відображає оновлену карту із доданим вузлом.

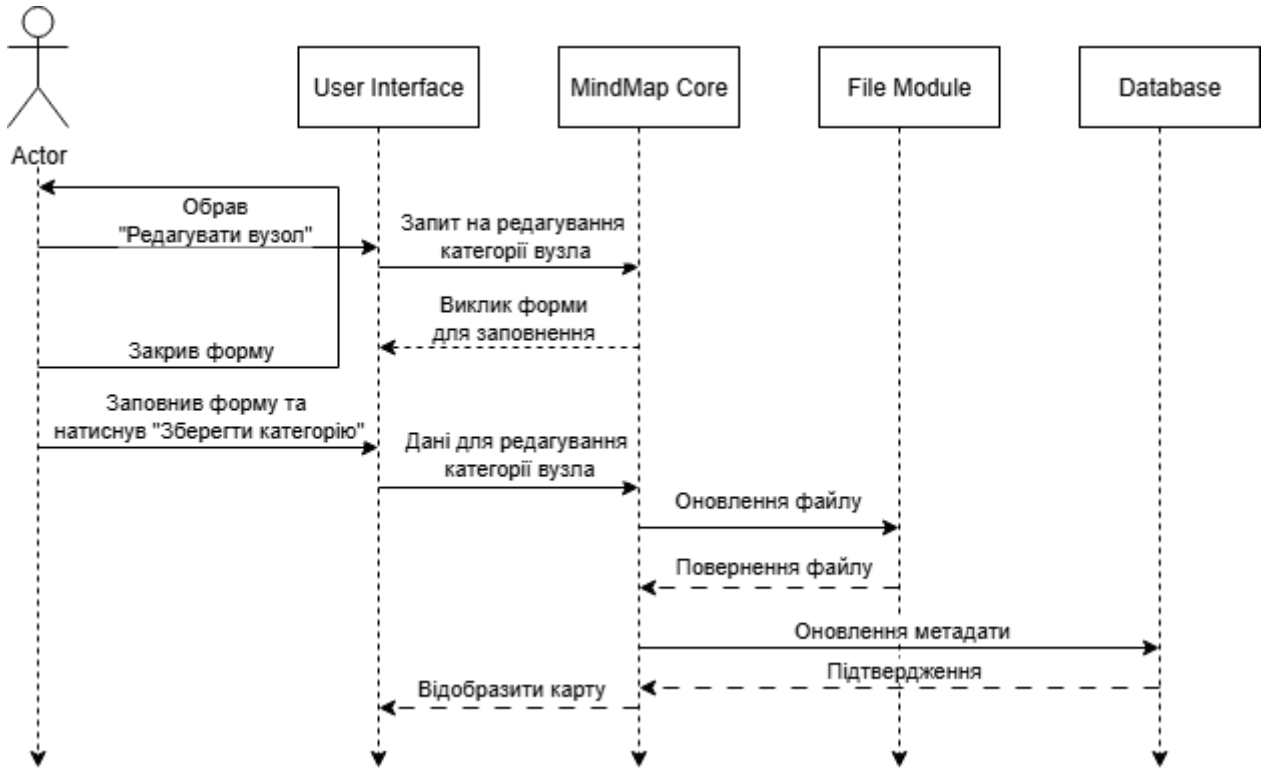

Рисунок 5 – Діаграма послідовності для варіанту використання: «Додавання категорії»

Опис:

1. Користувач натискає “Редагувати вузол”.
2. Інтерфейс викликає програму для оброблення дій користувача.
3. Ядро викликає форму для заповнення.
4. Користувач закриває форму або заповнює форму.
5. Інтерфейс передає дані для програмі, якщо форма заповнена.
6. Програма додає категорію.
7. File Module оновлює файл карти та повертає оновлений файл.
8. Застосунок відправляє файл до бази даних.
9. База даних дає підтвердження, що файл оброблений.
10. Інтерфейс відображає оновлену карту із доданим вузлом.

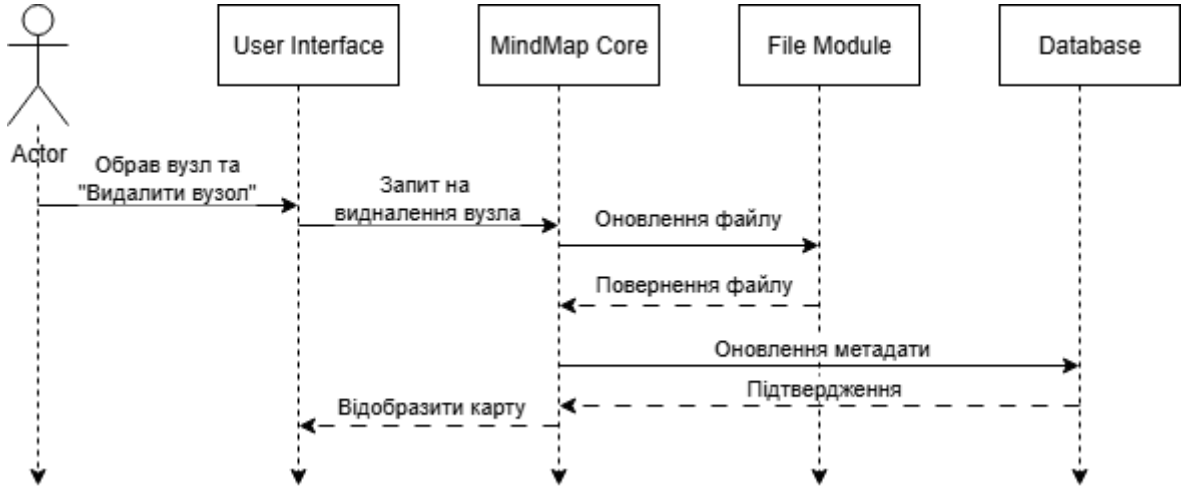

Рисунок 5 – Діаграма послідовності для варіанту використання: «Видалення вузла»

Опис:

1. Користувач натискає “Видалити вузол”.
2. Інтерфейс викликає програму для оброблення дій користувача.
3. Програма додає категорію.
4. File Module оновлює файл карти та повертає оновлений файл.
5. Застосунок відправляє файл до бази даних.
6. База даних дає підтвердження, що файл оброблений.
7. Інтерфейс відображає оновлену карту із доданим вузлом.

Висновок

Під час виконання лабораторної роботи були розроблені основні діаграми UML, які показують структуру системи (діаграма компонентів) та її поведінку (діаграма використання та послідовностей).

Також було визначено апаратне та програмне середовище для виконання застосунку, показане на діаграмі розгортання.

Нові діаграми будуть використані для подальшого розуміння системи, розробки нашого застосунку та подальших дій у відпрацювання UML діаграм.

Відповіді на питання для самоперевірки

1. Що собою становить діаграма розгортання?

Це представлення фізичного розташування системи, яке показує, на якому фізичному обладнанні запускається та чи інша складова програмного забезпечення.

Node – це те, що може містити програмне забезпечення.

2. Які бувають види вузлів на діаграмі розгортання?

Device – це фізичне обладнання: комп'ютер або пристрій, пов'язаний із системою.

Execution environment – це програмне забезпечення, яке саме може включати інше програмне забезпечення, наприклад операційну систему або процес-контейнер (наприклад, вебсервер).

3. Які бувають зв’язки на діаграмі розгортання?

Зазвичай зображують у вигляді прямої лінії. У зв'язків можуть бути атрибути множинності (для показання, наприклад, підключення 2х і більше клієнтів до одного сервера) і назва. У назві, як правило, міститься спосіб зв'язку між двома вузлами – це може бути назва протоколу (HTTP, IPC) або технологія, що використовується для забезпечення взаємодії вузлів (.NET Remoting, WCF).

4. Які елементи присутні на діаграмі компонентів?

Окремі вузли розподіленої системи – набір комп'ютерів і серверів; на кожному з вузлів можуть бути встановлені різні виконувані компоненти. Такий вид діаграм компонентів застарів і зазвичай замість нього використовують діаграму розгортань.

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

Зв’язки показують, що класи в з одного компонента використовують класи з іншого компонента. Також така діаграма показує зміни в якому компоненті будуть впливати на інші компоненти.

6. Які бувають види діаграм взаємодії?

Основними видами цих діаграм є діаграма послідовностей, яка акцентує увагу на часовому порядку повідомлень, та діаграма комунікації, яка зосереджена на структурі зв'язків між об'єктами.

7. Для чого призначена діаграма послідовностей?

Sequence Diagram – це один із типів діаграм у моделюванні UML, 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.

8. Які ключові елементи можуть бути на діаграмі послідовностей?

Actors – зазвичай позначаються піктограмами або назвами. Це користувачі чи інші системи, які взаємодіють із системою. Актори можуть бути зовнішніми стосовно моделювання системи.

Об'єкти або класи – 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.

9. Як діаграми послідовностей пов’язані з діаграмами варіантів використання?

Діаграми послідовностей є корисними для моделювання бізнес-процесів, проєктування архітектури систем і тестування. Вони дають змогу візуалізувати логіку взаємодії компонентів та виявити потенційні проблеми ще на етапі проєктування.

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

Вони показують різні аспекти однієї системи: діаграми класів описують її статичну структуру, а діаграми послідовностей - динамічну поведінку в рамках конкретного сценарію. Безпосередній зв'язок між ними полягає в тому, що кожне повідомлення на діаграмі послідовностей відповідає виклику методу в класі-отримувачі на діаграмі класів.

Код системи, який було зроблено в цій лабораторній

Node.java

```
package com.mindmap.model;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class Node implements Serializable {
        private String id;
        private String text;
        private double x;
        private double y;
        private String attachedFilePath;
        private List<Node> children;

   public Node(String text, double x, double y, children) {
                this.id = UUID.randomUUID().toString();
                this.text = text;
                this.x = x;
                this.y = y;
                this.children = new ArrayList<>();}

    public String getId() { return id; }
```

```
    public String getText() { return text; }
    public void setText(String text) { this.text = text; }
    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public List<Node> getChildren() { return children; }
    public String getAttachedFilePath() { return attachedFilePath; }
    public void setAttachedFilePath(String attachedFilePath) { this.attachedFilePath = attachedFilePath; }
}
```

MindMap.java

```
package com.mindmap.model;
import java.io.Serializable;
import java.util.UUID;

public class MindMap implements Serializable {
        private String id;
        private String name;
        private Node rootNode;

    public MindMap(String name, Node rootNode) {
        this.id = UUID.randomUUID().toString();
        this.name = name;
        this.rootNode = rootNode;
        this.rootNode.setText(this.name);
    }

    public String getId() { return id; }
    public String getName() { return name; }
```

```
    public void setName(String name) { this.name = name; }
    public Node getRootNode() { return rootNode; }
    public void setRootNode(Node rootNode) { this.rootNode = rootNode; }
}
```

MindMapRepository.java

```
package com.mindmap.repository;
import com.mindmap.model.MindMap;
import java.io.IOException;
import java.util.Optional;

public interface MindMapRepository {
        MindMap save(MindMap map) throws IOException;
        Optional<MindMap> findById(String id) throws IOException;
        void update(MindMap map) throws IOException;
        void delete(String id) throws IOException;
        List<MindMap> findAll() throws IOException;
}
```

FileMindMapRepository.java

```
package com.mindmap.repository;
import com.mindmap.model.MindMap;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.File;
import java.io.IOException;
import java.util.Optional;

public class FileMindMapRepository implements MindMapRepository {
        private static final String STORAGE_DIR = "mindmaps/";
        private final ObjectMapper objectMapper = new ObjectMapper();

    public FileMindMapRepository() {
```

```
        new File(STORAGE_DIR).mkdirs();}

    @Override
    public MindMap save(MindMap map) throws IOException {
        String filePath = STORAGE_DIR + map.getId() + ".json";
        objectMapper.writerWithDefaultPrettyPrinter().writeValue(new File(filePath), map);
       return map;}

    @Override
    public Optional<MindMap> findById(String id) throws IOException {
        String filePath = STORAGE_DIR + id + ".json";
        File file = new File(filePath);
        if (file.exists()) {
                return Optional.of(objectMapper.readValue(file, MindMap.class));
        }
        return Optional.empty();}

    @Override
    public void update(MindMap map) throws IOException {
       save(map);}

    @Override
    public void delete(String id) throws IOException {
       String filePath = STORAGE_DIR + id + ".json";
       new File(filePath).delete();}
}
```

MindMapService.java

```
package com.mindmap.service;

import com.mindmap.model.MindMap;
```

```
import com.mindmap.model.Node;
import java.io.IOException;

public interface MindMapService {
        MindMap createNewMap();
        void deleteMap(MindMap map) throws IOException;
        Node createNode(Node parent, double x, double y);
        void deleteNode(MindMap map, Node node);
        void editNode(Node node, String newText, double newX, double newY, String newAttachedFilePath);
        void saveMap(MindMap map) throws IOException;
}
```

MindMapServiceImpl.java

```
package com.mindmap.service;
import com.mindmap.model.MindMap;
import com.mindmap.model.Node;
import com.mindmap.repository.MindMapRepository;
import java.io.IOException;
import java.util.Objects;

public class MindMapServiceImpl implements MindMapService {
        private final MindMapRepository repository;
    public MindMapServiceImpl(MindMapRepository repository) {
        this.repository = repository;}
```

```
@Override
public MindMap createNewMap() {
     MindMap newMap = new MindMap();
     try {
            return repository.save(newMap);
     } catch (IOException e) {
       System.err.println("Помилка при збереженні нової карти: " + e.getMessage());
            return newMap; // Повертаємо, але не збережену
   }
}

@Override
public void deleteMap(MindMap map) throws IOException {
   repository.delete(map.getId());
}

@Override
public Node createNode(Node parent, double x, double y) {
   Node newNode = new Node();
   newNode.setX(x);
   newNode.setY(y);
   if (parent != null) {
      parent.getChildren().add(newNode);}
   return newNode;}

@Override
public void deleteNode(MindMap map, Node node) {
     if (Objects.equals(map.getRootNode().getId(), node.getId())) {
            System.out.println("Неможливо видалити кореневий вузол.");}
}

@Override
```

```
        public void editNode(Node node, String newText, double newX, double newY, String
newAttachedFilePath) {
        node.setText(newText);
        node.setX(newX);
        node.setY(newY);
        node.setAttachedFilePath(newAttachedFilePath);
          }

    @Override
    public void saveMap(MindMap map) throws IOException {
       repository.update(map);
    }
}
```

MindMapApp.java

```
package com.mindmap.ui;

import com.mindmap.repository.FileMindMapRepository;
import com.mindmap.repository.MindMapRepository;
import com.mindmap.service.MindMapService;
import com.mindmap.service.MindMapServiceImpl;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;

public class MindMapApp extends Application {
    private MindMapService mindMapService;
```

```
    @Override
    public void start(Stage stage) throws IOException {
        MindMapRepository repository = new FileMindMapRepository();
        this.mindMapService = new MindMapServiceImpl(repository);
        FXMLLoader loader = new FXMLLoader(getClass().getResource("MindMap.fxml"));
        Parent root = loader.load();
        MindMapController controller = loader.getController();
        controller.setMindMapService(this.mindMapService)
        Scene scene = new Scene(root, 1024, 768);
        stage.setTitle("MindMap Desktop App");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch();
    }}
```

MindMapController

```
package com.mindmap.ui;
import com.mindmap.model.MindMap;
import com.mindmap.service.MindMapService;
import javafx.fxml.FXML;
import javafx.scene.control.TabPane;
import java.io.IOException;

public class MindMapController {
        @FXML
        private TabPane tabPane;
```

```
    private MindMapService mindMapService;
  public void setMindMapService(MindMapService mindMapService) {
      this.mindMapService = mindMapService;
      createNewMapAction();}

  @FXML
  private void createNewMapAction() {
      MindMap newMap = mindMapService.createNewMap();
      MindMapTab newTab = new MindMapTab(newMap, mindMapService);
      tabPane.getTabs().add(newTab);
      tabPane.getSelectionModel().select(newTab);}

  @FXML
  private void deleteCurrentMapAction() {
      MindMapTab selectedTab = (MindMapTab)
      tabPane.getSelectionModel().getSelectedItem();
      if (selectedTab != null) {
            try {
            mindMapService.deleteMap(selectedTab.getMap());
            tabPane.getTabs().remove(selectedTab);
            } catch (IOException e) {
            System.err.println("Помилка видалення карти: " + e.getMessage());
       }}}
}
```