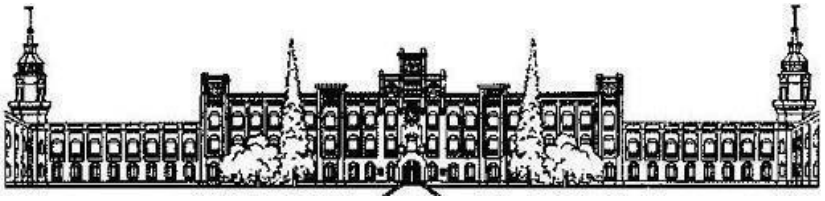

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №2

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Основи проектування»

Виконав:
студент групи - ІА-32
Діденко Я.О

Перевірив:
Мягкий М.Ю

Київ 2025

Мета роботи: 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.

18. **Shell (total commander)** (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі – перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Теоретичні відомості

UML (Unified Modeling Language) - універсальна мова візуального моделювання для специфікації, візуалізації, проєктування та документування програмних систем і бізнес-процесів. Модель складної системи розглядають у кількох представленнях (views) і на різних рівнях абстракції: концептуальний (початкова, найбільш загальна модель), логічний (структура та поведінка) і фізичний (реалізація та розгортання). Діаграма - графічне подання елементів моделі у вигляді пов’язаного графа з чіткою семантикою вузлів і зв’язків. Сукупність узгоджених діаграм фіксує вимоги та архітектуру системи. UML підтримує широкий набір типів діаграм, що відображають як статичні, так і динамічні аспекти системи. Це дозволяє розробникам, аналітикам і замовникам працювати з єдиною нотацією незалежно від конкретної предметної області чи мови програмування. Основною метою використання UML є зменшення неоднозначностей у процесі розробки та забезпечення єдиного бачення системи всіма учасниками проєкту. Важливо, що UML не визначає процес розроблення, а лише пропонує стандартні графічні засоби для опису моделей. Саме тому UML є базовим інструментом об’єктно-орієнтованого аналізу та проєктування, а створені моделі можуть бути використані для подальшої генерації коду або як документація.

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ (USE-CASE)
Призначення: визначити межі системи та функціональні вимоги у вигляді взаємодій користувачів із системою. Це концептуальна модель, яка не описує внутрішній устрій.

Основні елементи:

Актор (Actor) — зовнішній користувач або система, що взаємодіє із моделлю.

Варіант використання (Use case) - послуга/поведінка, яку система надає актору; позначається еліпсом із назвою.

Межа системи - прямокутник, що окреслює, які функції належать системі.
Відношення:

Асоціація - зв’язок між актором і варіантом (суцільна лінія; за потреби з напрямом).

Узагальнення (Generalization) - спадкування між акторами або між варіантами (стрілка з порожнім трикутником до предка).

<<include>> - базовий варіант завжди включає поведінку іншого (виділення спільної частини).

<<extend>> - додаткові кроки виконуються за умови у точці розширення (опційна поведінка).
Специфікація сценарію: передумови; післяумови; учасники; короткий опис; основний потік подій; виключення/альтернативи; примітки. Такий опис усуває неоднозначності та готує матеріал для подальшого аналізу й тестування.

ДІАГРАМА КЛАСІВ (CLASS)

Призначення: логічна статична структура: класи, атрибути, операції та відношення

Основні елементи:

Асоціація - «користується/посилається на», підтримує множинність (1, 0..1, 0.., 1..).

Агрегація - відношення «has-a» (слабкий склад), життєві цикли не пов’язані жорстко.

Композиція - відношення «owns-a» (сильний склад), життєвий цикл частини залежить від цілого.

Узагальнення - спадкування атрибутів та операцій (забезпечує повторне використання й поліморфізм).

Клас зображається прямокутником із трьома секціями: назва; атрибути (із видимістю); операції (із параметрами й типами). Видимість: public, protected, private.

БАЗИ ДАНИХ: ЛОГІЧНА МОДЕЛЬ І НОРМАЛІЗАЦІЯ

Логічна модель БД описує таблиці, ключі, зв’язки, представлення й індекси; фізична модель - зберігання на носіях та внутрішні структури СУБД.

Ключові поняття:

Первинний ключ (PK) - унікальна ідентифікація рядка.

Зовнішній ключ (FK) - забезпечення цілісності посилань між таблицями.

Індекси — прискорення пошуку/сортування за вибраними стовпцями.

Обмеження цілісності - правила ON DELETE/UPDATE (CASCADE/RESTRICT/ SET NULL).

Нормалізація зменшує логічну надмірність і запобігає аномаліям оновлення:
1НФ - атомарні значення, відсутність повторюваних груп.
2НФ - 1НФ + повна функціональна залежність неключових атрибутів від усього ключа.

Хід роботи

1. Побудуємо Use-case діаграму

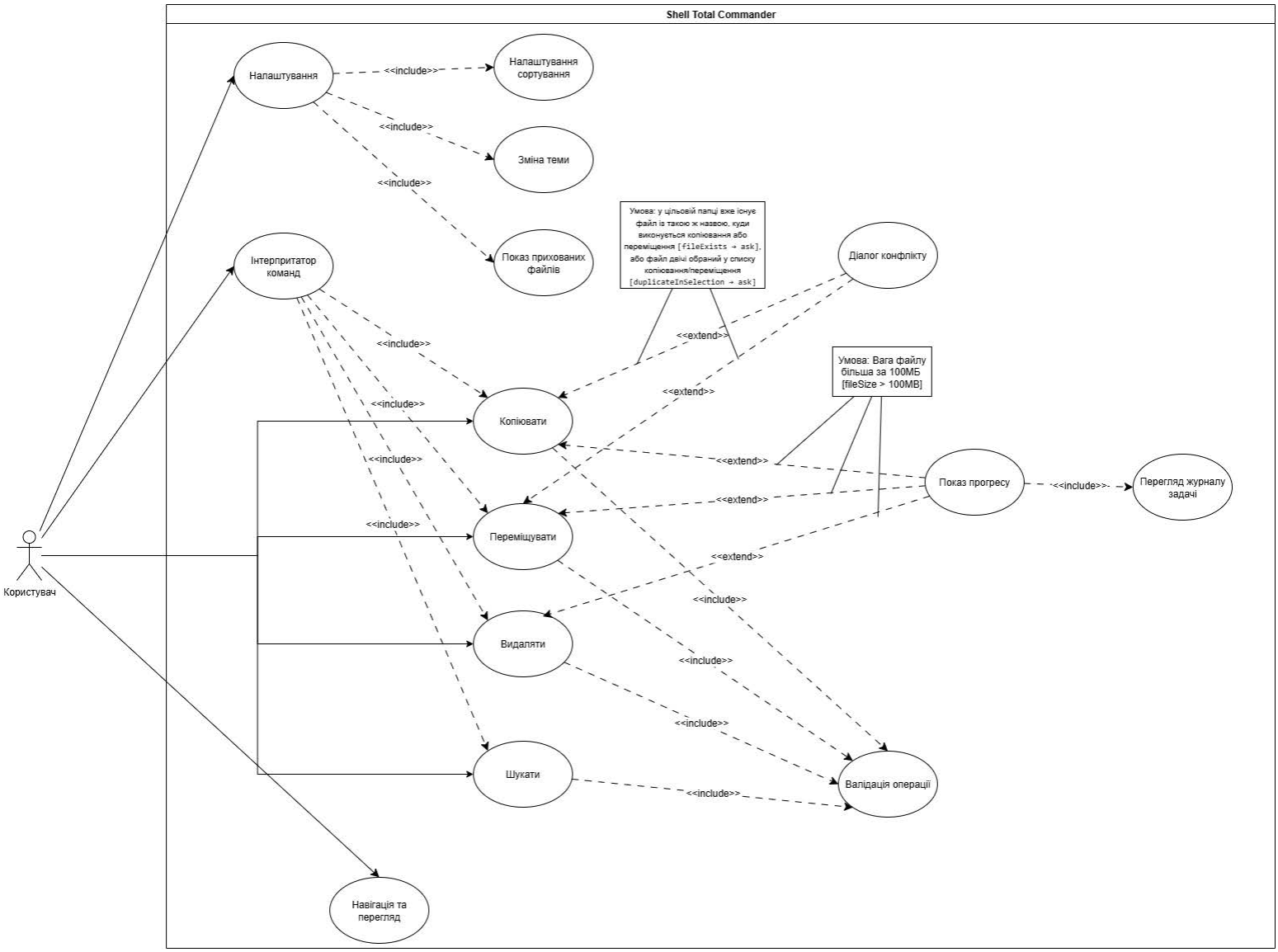

Рис.1 Use-case діаграма

Діаграма варіантів використання для Shell Total Commander описує взаємодію користувачів із файловим менеджером та його основні функції.

Актори:

Користувач (операції з файлами через термінал) - взаємодіє з системою за допомогою інтерпретатора команд.

Користувач (операції з файлами через інтерфейс) - працює з графічним UI.

Файлова система - зовнішній актор, що надає доступ до файлів і папок.

Основні варіанти використання:

Навігація та перегляд файлової системи.

Операції над файлами: Копіювати, Переміщувати, Видаляти, Шукати.

Налаштування (сортування, зміна теми, показ прихованих файлів).

Інтерпретатор команд як альтернативний спосіб виконання дій.

Додаткові варіанти:

Валідація операцій - завжди включається у виконання основних дій.

Діалог конфлікту - виконується у випадку дублювання імені або при наявності файлу в цільовій папці.

Показ прогресу - розширення для операцій із загальним обсягом даних понад 100 МБ.

Перегляд журналу задач - включається до сценарію відображення прогресу.

2. Оберемо 3 прецеденти і напишемо на їх основі прецеденти.

UC-01: Копіювати вибрані елементи
Актор: Користувач (через UI або термінал)
Мета: Скопіювати файли/папки у вибрану директорію.
Передумови: Обидві панелі відкриті; у лівій/правій панелі є вибір елементів; цільова директорія доступна для запису.
Тригер: Натискання “Копіювати” або команда у термінал сору ....

Основний сценарій:
1. Система зчитує вибрані елементи та цільовий шлях.
2. Валідація операції (існування джерел, доступ, вільне місце, коректність шляху).
3. Якщо в цілі вже існують однойменні елементи → показати діалог конфлікту (ask) з варіантами Замінити / Пропустити / Перейменувати (для кожного конфлікту або “для всіх”).
4. Якщо загальна вага > 100 MB - створити фонову задачу з вікном прогресу та журналом.
5. Виконати копіювання (поштучно/чанками).
6. Після завершення показати підсумок (успішно/помилки) і записати подію в журнал задач.

Альтернативи / винятки:

2a. Немає прав читання/запису - повідомлення Access denied, задача помічена як Error.
3a. Користувач обирає Пропустити - відповідні елементи не копіюються.
4a. Вага ≤ 100 MB - виконання без вікна прогресу (швидкий режим).
5a. Нестача місця - зупинка, Error, показ причини.

Постумови: Цільова директорія містить (за обраною політикою) нові копії; створено запис у Tasks, TaskItems, TaskLogs.

UC-02: Перемістити (вирізати/вставити) елементи
Актор: Користувач
Мета: Перемістити файли/папки до іншої директорії.
Передумови: Є вибрані елементи та доступ до джерела/цілі.
Тригер: “Перемістити” або команда у терміналі move ....

Основний сценарій:
1. Система зчитує вибір і цільовий шлях.
2. Валідація (існування джерела, права доступу, вільне місце в цілі).
3. Для конфліктних імен у цілі показується діалог конфлікту (ask).
4. Якщо загальна вага > 100 MB - задача з прогресом і журналом.
5. Виконати переміщення: копіювання у ціль - верифікація - видалення джерела.
6. Показати підсумок і записати результат у журнал задач.

Альтернативи / винятки:

2а. Заборонено видаляти зі джерела - зупинка, Error.
3а. Користувач обирає Перейменувати - система пропонує нове ім’я (додає суфікс).
4а. Вага ≤ 100 MB - без вікна прогресу.
5а. Копіювання пройшло, але видалення не вдалося - частковий успіх, попередження в журналі.

Постумови: Елементи знаходяться в цілі; журнал містить повну історію; у разі помилок джерела можуть залишитися (залежно від кроку зупинки).

UC-03: Пошук файлів
Актор: Користувач
Мета: Знайти файли/папки за маскою/патерном у кореневому каталозі.
Передумови: Вказано root (панель або поле), задано pattern (напр. *.jpg).
Тригер: Кнопка “Шукати” або команда у термінал search root=... pattern=... recursive=true.

Основний сценарій:
1. Користувач задає параметри: root, pattern, recursive (так/ні).
2. Система валідує параметри (існує корінь, патерн коректний).
3. Створюється задача Search (фоново, якщо результатів очікується багато).
4. Сканування директорій (за recursive), відбір за маскою/умовами.
5. Результати відображаються у списку; доступні дії:
Відкрити розташування, Скопіювати/Перемістити вибране.
6. Параметри пошуку заносяться до SearchHistory.

Альтернативи / винятки:

2а. Кореневий шлях недоступний - помилка Path not found / Access denied.
3а. Відміна користувачем - стан Canceled, часткові результати можуть бути показані.
5а. Подвійний клік по результату - фокус панелі переходить у відповідну теку з виділенням файла.

Постумови: Користувач бачить результати; записано задачу та історію пошуку.

3. Зобразимо діаграму класів

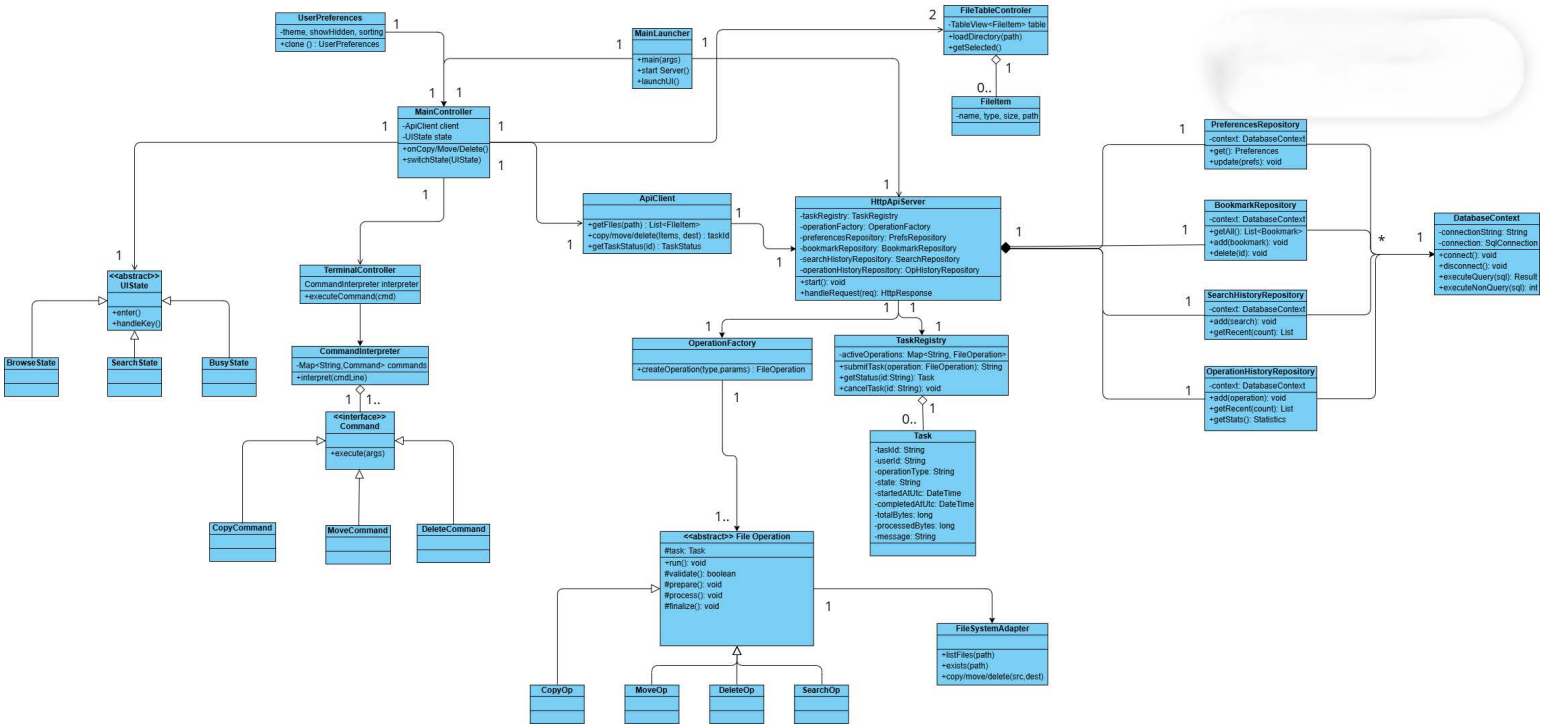

Рис.2 Діаграма класів

1. Серверна частина (HTTP API):

MainLauncher - Головний клас, точка входу в програму. Відповідає за запуск HTTP-сервера та клієнтського UI.

HttpApiServer - Основний клас сервера, що приймає та обробляє HTTP-запити від клієнта. Керує життєвим циклом сервера та делегує обробку запитів відповідним компонентам, таким як OperationFactory та репозиторії.

OperationFactory - Реалізує патерн Factory Method. Створює екземпляри конкретних файлових операцій (CopyOp, MoveOp) на основі запиту від клієнта.

FileOperation - Абстрактний клас, що реалізує патерн Template Method. Визначає загальний шаблон (алгоритм) для виконання файлових операцій: validate(), prepare(), process(), finalize(). Його підкласи (CopyOp, MoveOp, DeleteOp, SearchOp) реалізують конкретну логіку для кожної операції.

TaskRegistry - Реєстр фонових завдань. Зберігає активні операції, дозволяє клієнту запитувати їхній статус за ідентифікатором (taskId).

Task - Модель даних, що описує стан фонового завдання: його ID, статус, прогрес, повідомлення.

FileSystemAdapter - Адаптер для роботи з файловою системою. Інкапсулює всю низькорівневу логіку взаємодії з файлами та папками (отримання списків, копіювання, переміщення), надаючи чистий інтерфейс для решти серверної частини.

2. Клієнтська частина (JavaFX UI):

`MainController` - головний контролер, керує станом інтерфейсу та взаємодією з сервером.
`FileTableController` - відповідає за відображення вмісту папки у вигляді таблиці.
`TerminalController` - реалізує взаємодію з користувачем через термінал команд.
`UIState` (абстрактний клас) та його підкласи `BrowseState`, `SearchState`, `BusyState` реалізують патерн State для роботи з різними режимами інтерфейсу.
`UserPreferences` - зберігає налаштування користувача (тема, відображення прихованих файлів, сортування) та реалізує Prototype через метод `clone()`.

3. Інтерпретатор команд:

CommandInterpreter - Відповідає за розбір (парсинг) командного рядка, введеного у терміналі, та виклик відповідних об'єктів-команд.

Command - Інтерфейс та його конкретні реалізації (CopyCommand, MoveCommand, DeleteCommand), що втілюють патерн Interpreter. Кожен клас інкапсулює логіку виконання однієї команди терміналу.

4. База даних та репозиторій:

DatabaseContext - Клас, що керує з'єднанням з базою даних (наприклад, SQLite) та виконанням SQL-запитів.

PreferencesRepository, BookmarkRepository, SearchHistoryRepository, OperationHistoryRepository - Класи, що реалізують патерн Repository. Кожен репозиторій надає абстрактний інтерфейс для роботи з даними (CRUD-операції) для конкретної сутності, приховуючи деталі взаємодії з базою даних.

4. Спроєктуємо БД

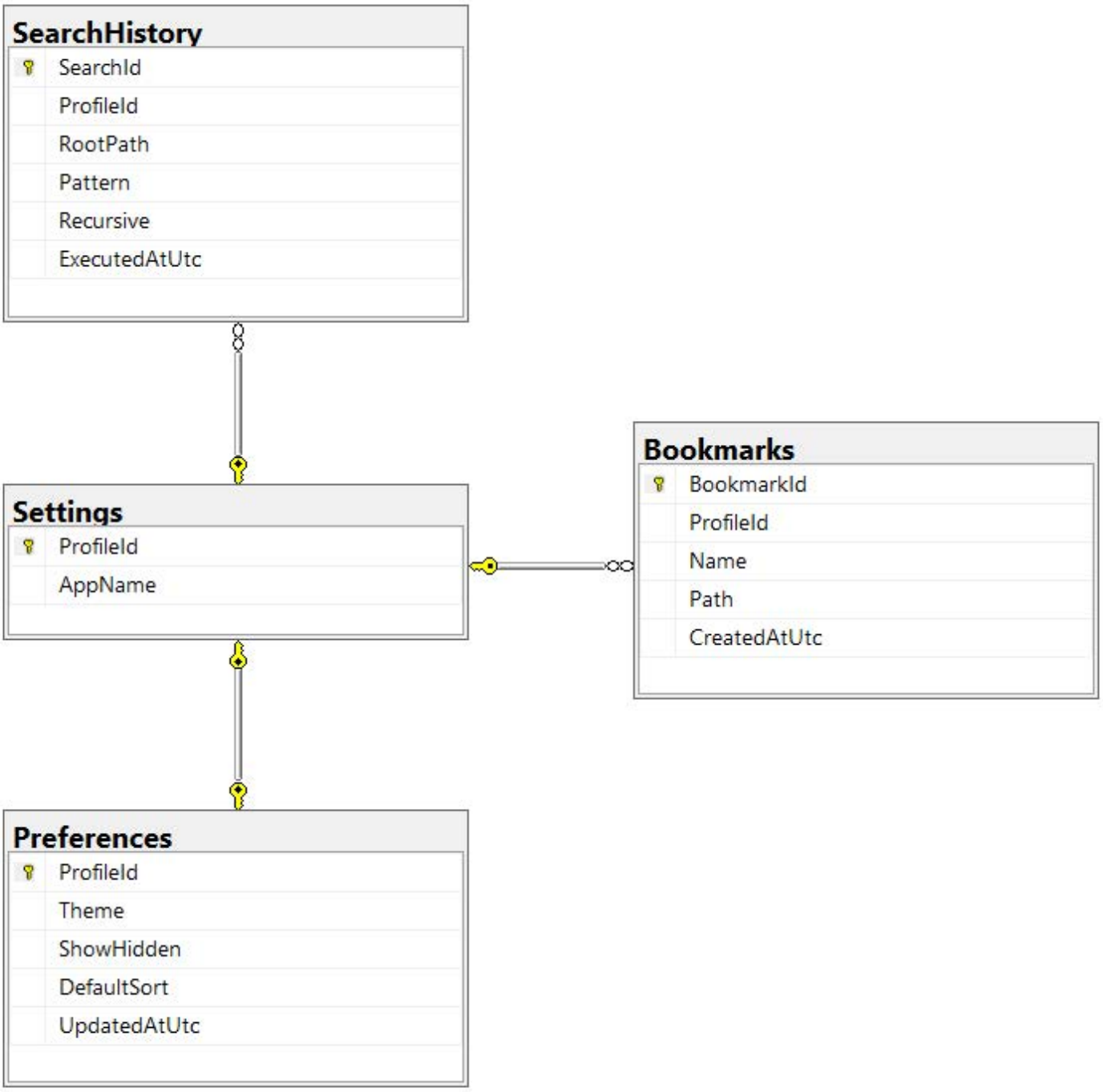

Рис.3 Спроєктована БД

Модель бази даних побудована з урахуванням потреб збереження історії виконаних операцій, журналів, налаштувань користувачів та результатів пошуку. Вона складається з кількох взаємопов’язаних таблиць:

1. **Settings** - Центральна таблиця, що ідентифікує унікальний профіль налаштувань. Кожен запис у цій таблиці представляє один профіль, до якого прив'язані всі інші налаштування.

2. **Preferences** - Зберігає налаштування зовнішнього вигляду та поведінки програми для конкретного профілю. Містить такі поля, як тема (Theme), налаштування показу прихованих файлів (ShowHidden) та сортування за замовчуванням (DefaultSort).

3. **SearchHistory** - Зберігає історію пошукових запитів, виконаних користувачем у рамках профілю. Кожен запис містить параметри пошуку: кореневий шлях (RootPath), шаблон (Pattern) та час виконання.

4. **Bookmarks** - Зберігає закладки користувача на певні папки або файли. Кожен запис містить назву закладки (Name), шлях (Path) та дату створення.

Ключові зв’язки:

'Settings' - 'Preferences' (1:1). Кожен профіль (Settings) має рівно один набір візуальних налаштувань (Preferences).

'Settings' - 'SearchHistory' (1:N). Один профіль (Settings) може мати багато записів в історії пошуку (SearchHistory).

'Settings' - 'Bookmarks' (1:N). Один профіль (Settings) може мати багато закладок (Bookmarks).

База даних спроєктована відповідно до принципів нормалізації (3НФ), що дозволяє уникнути надмірності та забезпечити логічну узгодженість інформації.

Контрольні запитання

1. Що таке UML?
UML (Unified Modeling Language) - це уніфікована мова візуального моделювання, яка використовується для специфікації, проєктування, документування та візуалізації програмних систем і бізнес-процесів.

2. **Що собою становить діаграма класів UML?**
Діаграма класів - це статичне подання системи, яке показує класи, їхні атрибути та операції, а також відношення між класами (асоціації, узагальнення, агрегації, композиції).

3. Які діаграми UML називають канонічними?
До канонічних UML-діаграм належать: діаграма варіантів використання, класів, послідовності, кооперації, станів, діяльності, компонентів і розгортання.

4. Що собою становить діаграма варіантів використання?
Діаграма варіантів використання (Use Case) — це концептуальна модель, що показує функціональні вимоги до системи через взаємодію акторів (користувачів чи зовнішніх систем) із варіантами використання (сценаріями роботи).

5. Чим відрізняються зв'язок типу агрегації від зв'язків композиції на діаграмі класів?
Агрегація («has-a») - слабкий зв'язок «ціле–частина»: частина може існувати окремо від цілого.
Композиція («owns-a») - сильний зв'язок: життєвий цикл частини залежить від цілого, при знищенні цілого знищуються всі його частини.

6. Що собою становлять нормальні форми баз даних?
Нормальні форми — це правила організації таблиць БД, що усувають надмірність і аномалії при оновленні даних. Основні:

1НФ - атрибути атомарні;
2НФ - усі неключові атрибути залежать від усього ключа;
3НФ - відсутні транзитивні залежності від ключа;
BCNF - посилена 3НФ.

7. Що таке фізична модель баз даних? Логічна?
Логічна модель бази даних - це абстрактний опис структури даних, який включає таблиці, поля, ключі, зв’язки, індекси та обмеження цілісності. Вона показує, які сутності існують у системі, як вони пов’язані між собою, але не залежить від конкретної СУБД.

Фізична модель бази даних - це конкретна реалізація логічної моделі на рівні обраної СУБД. Вона описує зберігання даних на носіях, використання файлів і сторінок, структуру індексів, механізми доступу та оптимізацію продуктивності.

8. Який взаємозв’язок між таблицями БД та програмними класами? Таблиці бази даних і програмні класи відображають одні й ті самі сутності предметної області, але на різних рівнях:

Класи описують логіку та поведінку об’єктів у програмі (атрибути, методи). Таблиці зберігають ці ж об’єкти у вигляді рядків і стовпців.

Між ними встановлюється відображення (mapping):
«Один клас — одна таблиця» (найпоширеніший варіант).
«Один клас — кілька таблиць» (якщо дані рознесені).
«Кілька класів — одна таблиця» (якщо реалізується спадкування або спільне зберігання).

Висновки:

У роботі було розглянуто основи мови UML та побудовано діаграми для системи Shell Total Commander. Діаграми варіантів використання, класів, послідовностей, компонентів і розгортання відобразили функціональні вимоги, архітектуру та динаміку роботи системи. Також розроблено логічну модель бази даних для збереження налаштувань, історії пошуку та журналу виконаних завдань. При проєктуванні застосовано шаблони проєктування (Client–Server, State, Factory Method, Template Method, Prototype, Interpreter).
Побудовані моделі створюють основу для подальшої реалізації й тестування програмного продукту.

Додаток

Вихідні коди класів системи

```
class MainLauncher {
    public static void main(String[] args) {
        HttpApiServer server = new HttpApiServer();
        server.start();
        MainController controller = new MainController();
        controller.switchState(new BrowseState(controller));
    }
}

class UserPreferences implements Cloneable {
    private String theme = "Light";
    private boolean showHidden = false;
    private String sorting = "name";
    public UserPreferences clone() {
        try { return (UserPreferences) super.clone(); } catch
(CloneNotSupportedException e) { throw new
RuntimeException(e); }
    }
    public String getTheme() { return theme; }
    public boolean isShowHidden() { return showHidden; }
    public String getSorting() { return sorting; }
    public void setTheme(String v) { theme = v; }
    public void setShowHidden(boolean v) { showHidden =
v; }
    public void setSorting(String v) { sorting = v; }
}

class MainController {
    private final ApiClient client = new ApiClient();
    private UIState state;
    public ApiClient getClient() { return client; }
    public void copy(List<String> items, String dest)
{ client.postOperation("copy", items, dest); }
    public void move(List<String> items, String dest)
{ client.postOperation("move", items, dest); }
    public void delete(List<String> items)
{ client.postOperation("delete", items, null); }
    public void switchState(UIState s) { state = s; if (state !
= null) state.enter(); }
}
```

```
abstract class UIState {
   protected final MainController ui;
   protected UIState(MainController ui) { this.ui = ui; }
   public abstract void enter();
   public abstract void handleKey(String key);
}

class BrowseState extends UIState {
   public BrowseState(MainController ui) { super(ui); }
   public void enter() {}
   public void handleKey(String key) {}
}

class SearchState extends UIState {
   public SearchState(MainController ui) { super(ui); }
   public void enter() {}
   public void handleKey(String key) {}
}

class BusyState extends UIState {
   public BusyState(MainController ui) { super(ui); }
   public void enter() {}
   public void handleKey(String key) {}
}

class TerminalController {
   private final CommandInterpreter interpreter = new CommandInterpreter();
   public void executeCommand(String cmd) { interpreter.interpret(cmd); }
}

interface Command { void execute(String[] args); }

class CopyCommand implements Command { public void execute(String[] args) {} }
class MoveCommand implements Command { public void execute(String[] args) {} }
class DeleteCommand implements Command { public void execute(String[] args) {} }

class CommandInterpreter {
   private final Map<String, Command> commands = new HashMap<>();
   public CommandInterpreter() {
      commands.put("copy", new CopyCommand());
      commands.put("move", new MoveCommand());
      commands.put("delete", new DeleteCommand());
   }
   public void interpret(String cmdLine) {}
}
```

```
class ApiClient {
    public List<FileItem> getFiles(String path) { return List.of(); }
    public String postOperation(String type, List<String> items, String dest) { return
UUID.randomUUID().toString(); }
    public TaskStatus getTaskStatus(String id) { return new TaskStatus(); }
}

class OperationFactory {
    public FileOperation createOperation(String type, List<String> params) {
        if ("copy".equals(type)) return new CopyOp(params, "");
        if ("move".equals(type)) return new MoveOp(params, "");
        if ("delete".equals(type)) return new DeleteOp(params);
        if ("search".equals(type)) return new SearchOp(params, "");
        throw new IllegalArgumentException(type);
    }
}

abstract class FileOperation {
    protected final TaskStatus status = new TaskStatus();
    public final TaskStatus run() { validate(); prepare(); process(); finalizeOk(); return status; }
    protected abstract void validate();
    protected abstract void prepare();
    protected abstract void process();
    protected void finalizeOk() {}
    public TaskStatus getStatus() { return status; }
}

class CopyOp extends FileOperation {
    private final List<String> items;
    private final String dest;
    public CopyOp(List<String> items, String dest) { this.items = items; this.dest = dest; }
    protected void validate() {}
    protected void prepare() {}
    protected void process() {}
}
```

```
class MoveOp extends FileOperation {
   private final List<String> items;
   private final String dest;
   public MoveOp(List<String> items, String dest) { this.items = items; this.dest = dest; }
   protected void validate() {}
   protected void prepare() {}
   protected void process() {}
}

class DeleteOp extends FileOperation {
   private final List<String> items;
   public DeleteOp(List<String> items) { this.items = items; }
   protected void validate() {}
   protected void prepare() {}
   protected void process() {}
}

class SearchOp extends FileOperation {
   private final List<String> items;
   private final String query;
   public SearchOp(List<String> items, String query) { this.items = items; this.query = query; }
   protected void validate() {}
   protected void prepare() {}
   protected void process() {}
}

class TaskRegistry {
   private final Map<String, TaskStatus> tasks = new HashMap<>();
   public String submit(TaskStatus status) { String id = UUID.randomUUID().toString();
tasks.put(id, status); return id; }
   public TaskStatus getStatus(String id) { return tasks.get(id); }
}

class TaskStatus {
   private String state = "queued";
   private int progress;
   private String message;
   public String getState() { return state; }
   public void setState(String v) { state = v; }
   public int getProgress() { return progress; }
   public void setProgress(int v) { progress = v; }
   public String getMessage() { return message; }
   public void setMessage(String v) { message = v; }
}
```

```
class FileSystemAdapter {
    public List<FileItem> list(String path) { return List.of(); }
    public boolean exists(String path) { return false; }
    public void copy(String src, String dest) {}
    public void move(String src, String dest) {}
    public void delete(String src) {}
}

class FileItem {
    private String name;
    private String type;
    private long size;
    private String path;
    public FileItem() {}
    public FileItem(String name, String type, long size, String path) { this.name = name; this.type
= type; this.size = size; this.path = path; }
    public String getName() { return name; }
    public String getType() { return type; }
    public long getSize() { return size; }
    public String getPath() { return path; }
    public void setName(String v) { name = v; }
    public void setType(String v) { type = v; }
    public void setSize(long v) { size = v; }
    public void setPath(String v) { path = v; }
}

class HttpRequest {
    public int id;
    public String method;
    public String url;
    public String headers;
    public String body;
    public int responseCode;
    public long createdAt;
}
```

```
class TaskRecord {
    public int id;
    public String taskId;
    public String state;
    public int progress;
    public String message;
}

class DatabaseContext {
    public String connectionString;
    public List<HttpRequest> requests = new ArrayList<>();
    public List<TaskRecord> tasks = new ArrayList<>();
    public void connect() {}
    public void disconnect() {}
    public void saveChanges() {}
}

class RequestRepository implements IRepository<HttpRequest> {
    private final DatabaseContext context;
    public RequestRepository(DatabaseContext context) { this.context = context; }
    public void add(HttpRequest entity) { context.requests.add(entity); context.saveChanges(); }
    public HttpRequest getById(int id) { return context.requests.stream().filter(r -> r.id ==
id).findFirst().orElse(null); }
    public List<HttpRequest> getAll() { return new ArrayList<>(context.requests); }
    public void update(int id, HttpRequest entity) { remove(id); add(entity); }
    public void remove(int id) { context.requests.removeIf(r -> r.id == id); context.saveChanges(); }
}

class SearchHistoryRepository {
    DatabaseContext context;

    void add(SearchHistory search) { }

    List<SearchHistory> getAllForProfile(int profileId) { }
}

class PreferencesRepository {
    DatabaseContext context;

    Preferences getForProfile(int profileId) { }

    void update(Preferences prefs) { }
}
```

```
class BookmarkRepository {
    DatabaseContext context;

    void add(Bookmark bookmark) { }

    List<Bookmark> getAllForProfile(int profileId)
{ }
}

class DatabaseContext {
    Connection connection;

    void connect() { }

    void disconnect() { }

    ResultSet executeQuery(String sql, ...) { }

    int executeUpdate(String sql, ...) { }
}

class HttpApiServer {
    int port;
    HttpServer server;

    SearchHistoryRepository searchHistoryRepo;
    PreferencesRepository preferencesRepo;
    OperationFactory operationFactory;

    void start() {
        server = HttpServer.create(new
InetSocketAddress("localhost", port));

        server.createContext("/api/fs/list", (request,
response) -> {
            String path = request.getParameter("path");
            FileOperation op =
operationFactory.create("list", path);
            List<FileItem> files = op.run();
            response.send(200, toJson(files));
        });
```

```
        server.createContext("/api/preferences/
update", (request, response) -> {
            Preferences prefs =
fromJson(request.getBody(), Preferences.class);
            preferencesRepo.update(prefs);
            response.send(200, "{ \"status\": \"ok\" }");
        });

        server.createContext("/api/search/history/add",
(request, response) -> {
            SearchHistory history =
fromJson(request.getBody(), SearchHistory.class);
            searchHistoryRepo.add(history);
            response.send(201, "{ \"status\": \"created
\" }");
        });

        server.start();
    }

    void stop() {
        server.stop();
    }
}
```