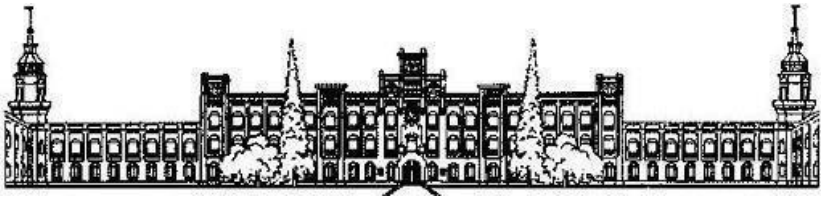

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №3

з дисципліни «Технології розроблення
програмного забезпечення»

Тема: «Основи проектування розгортання. »

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

Перевірив:
Мягкий М.Ю

Київ 2025

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.

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

1. Діаграма розгортання (Deployment Diagram)

Діаграма розгортання візуалізує фізичну архітектуру системи, показуючи, на якому обладнанні та в якому програмному оточенні будуть запускатися компоненти програмного забезпечення. Вона моделює статичне розміщення артефактів системи у фізичному світі.

Основні елементи діаграми розгортання:

Вузол (Node) - являє собою обчислювальний ресурс. Вузли бувають двох типів:

- Пристрій (Device): Фізичне обладнання, таке як комп'ютер, сервер або інший апаратний пристрій.
- Середовище виконання (Execution Environment): Програмне середовище, що знаходиться всередині пристрою і слугує для запуску іншого програмного забезпечення (наприклад, операційна система, веб-сервер, віртуальна машина Java).

Артефакт (Artifact) - фізичне представлення програмного забезпечення, що розгортається на вузлі. Зазвичай це файли, такі як виконувані файли (.exe, .jar), бібліотеки (.dll), конфігураційні файли або скрипти.

Зв'язок (Communication Path) - лінія, що з'єднує два вузли і показує, що між ними існує канал зв'язку. Зв'язок часто підписують назвою протоколу (наприклад, HTTP, TCP/IP), що використовується для взаємодії.

Діаграми розгортання допомагають зрозуміти, які апаратні та програмні ресурси необхідні для функціонування системи, та спланувати процес її встановлення та налаштування.

2. Діаграма компонентів (Component Diagram)

Діаграма компонентів представляє систему як набір модульних блоків (компонентів) та показує залежності між ними. Вона фокусується на статичній структурі системи на вищому рівні, ніж діаграма класів, і використовується для моделювання архітектури програмного забезпечення.

Основні елементи діаграми компонентів:

Компонент (Component) - це модульна, логічно відокремлена частина системи з чітко визначеним інтерфейсом. Компонент може представляти собою пакет

класів, бібліотеку, виконуваний файл або будь-яку іншу логічну одиницю системи (наприклад, "UI", "API-сервер", "Модуль доступу до даних").

Залежність (Dependency) - показується пунктирною стрілкою і означає, що один компонент (клієнт) використовує функціонал, наданий іншим компонентом (постачальником). Зміна в компоненті-постачальнику може вплинути на компонент-клієнт.

У контексті проєктованої системи, діаграма компонентів дозволяє візуалізувати розбиття програми на логічні шари (наприклад, "Клієнт", "Сервер", "Middleware"), керувати залежностями між ними та розуміти, як зміни в одній частині системи вплинуть на інші.

3. Діаграма послідовностей (Sequence Diagram)

Діаграма послідовностей моделює взаємодію об'єктів у часі для реалізації конкретного сценарію (use case). Вона показує, в якому порядку об'єкти обмінюються повідомленнями для виконання певної задачі, і є динамічним представленням системи.

Основні елементи діаграми послідовностей:

Актор (Actor) - користувач або зовнішня система, яка ініціює сценарій.

Об'єкт (Object) - учасник взаємодії (клас, компонент), представлений прямокутником. Від кожного об'єкта вниз йде вертикальна пунктирна лінія, що називається лінією життя (Lifeline).

Повідомлення (Message) - стрілка між лініями життя двох об'єктів, що позначає виклик методу або передачу інформації. Суцільна стрілка позначає синхронний виклик, а пунктирна — повернення результату.

Активація (Activation) - вузький прямокутник на лінії життя об'єкта, що показує період часу, протягом якого об'єкт активно виконує операцію.

Фрагменти взаємодії (Interaction Fragments) - спеціальні блоки, що дозволяють моделювати складну логіку:

- alt (альтернатива): Показує кілька можливих сценаріїв (аналог if-else).
- opt (опція): Блок, який виконується лише за певної умови.
- loop (цикл): Блок, який повторюється, доки виконується умова.

Діаграми послідовностей є надзвичайно корисними для деталізації сценаріїв використання, проектування логіки методів та виявлення потенційних проблем у взаємодії компонентів на ранніх етапах розробки.

Хід роботи

1. Діаграму розгортання системи

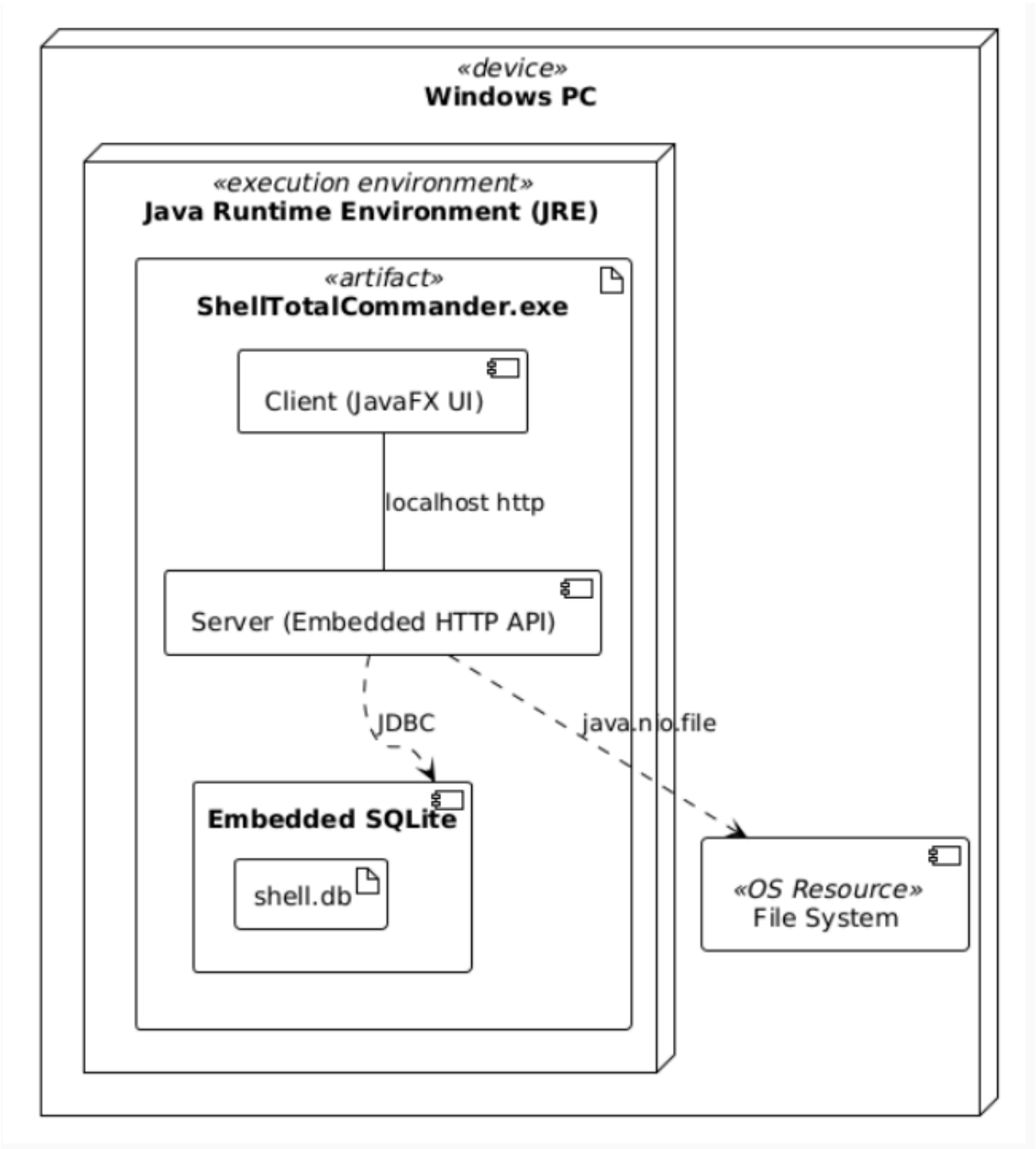

рис. 1. Діаграма розгортання системи

Діаграма розгортання візуалізує фізичну архітектуру системи. Вона демонструє, як логічно розподілена клієнт-серверна система фізично розгортається у вигляді єдиного, самодостатнього додатку на одному комп'ютері користувача.

Діаграма включає наступні ключові елементи:

1. Вузол «Windows PC» (Device):
Це основний вузол типу «Пристрій», що представляє фізичний комп'тер користувача, на якому встановлена операційна система Windows. Він є контейнером для всього програмного забезпечення.

2. Середовище виконання «Java Runtime Environment (JRE)» (Execution Environment):
Всередині фізичного пристрою знаходиться «Середовище виконання». Воно представляє собою середовище Java, необхідне для запуску програми. Завдяки використанню jpackage при збірці проєкту, JRE вбудовується безпосередньо в інсталяційний пакет, що робить додаток повністю автономним і не вимагає попередньо встановленої Java на комп'ютері користувача.

3. Артефакт «ShellTotalCommander.exe» (Artifact):
Центральним елементом є «Артефакт» — єдиний виконуваний файл, який розгортається та запускається в середовищі JRE. Цей файл інкапсулює в собі всі компоненти системи, включаючи клієнтську та серверну частини, а також вбудовану базу даних.

4. Внутрішні компоненти:

- **Client (JavaFX UI):** Компонент, що відповідає за весь графічний інтерфейс користувача.
- **Server (Embedded HTTP API):** Компонент, що реалізує всю бізнес-логіку у вигляді вбудованого локального HTTP-сервера.
- **Embedded SQLite:** Компонент, що представляє вбудовану базу даних, яка зберігає свої дані у файлі **shell.db**.

5. Ресурс ОС «File System»:
Зовнішнім по відношенню до JRE, але в межах операційної системи, є компонент «File System». Він представляє файлову систему комп'ютера як ресурс, до якого звертається серверна частина програми для виконання операцій над файлами та папками.

Опис взаємозв'язків:

- **Client - Server:** Взаємодія між клієнтським та серверним компонентами відбувається через локальні HTTP-запити (localhost http).
- **Server - Embedded SQLite:** Серверний компонент звертається до вбудованої бази даних для збереження та отримання налаштувань, історії та закладок. Ця взаємодія здійснюється за допомогою технології **JDBC**.
- **Server - File System:** Серверний компонент взаємодіє з файловою системою для виконання всіх операцій (читання, запис, видалення). Ця взаємодія реалізується через стандартну бібліотеку Java (**java.nio.file**).

2. Діаграма компонентів

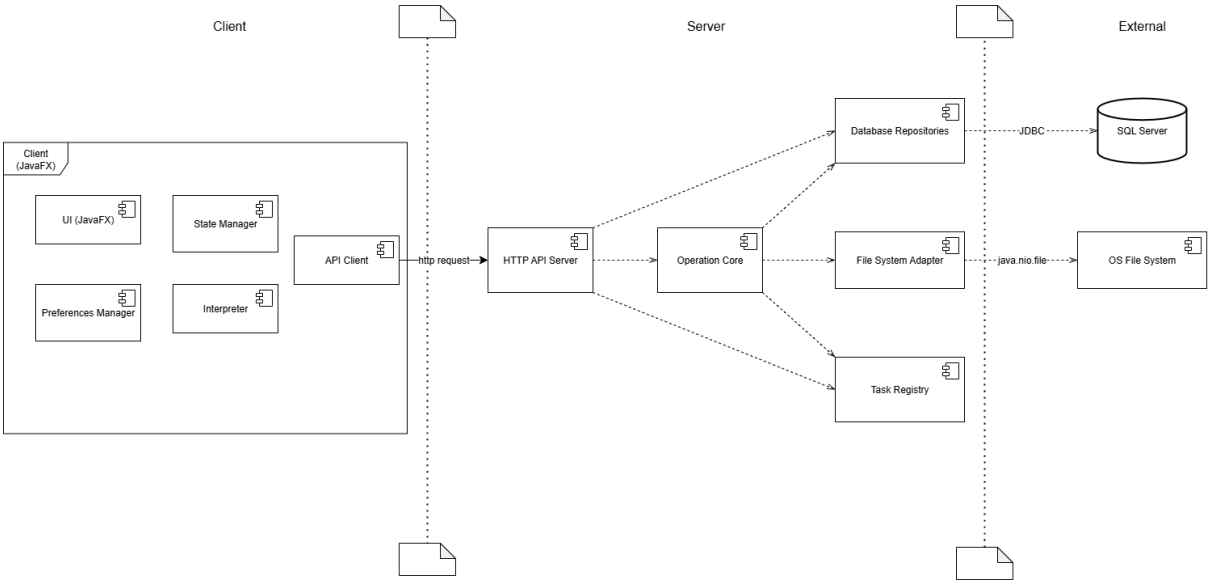

рис. 2. Діаграма компонентів

Діаграма компонентів представляє логічну архітектуру системи, структуровану у вигляді трьох основних шарів: **Client**, **Server** та **External**. Вона візуалізує модульний склад програми, залежності між ключовими компонентами та чіткий потік даних, що є основою для реалізації клієнт-серверної архітектури.

1. Шар «Client»

Цей шар об'єднує всі компоненти, відповідальні за інтерфейс та взаємодію з користувачем. Його компоненти згруповані в єдиний логічний блок, що представляє собою клієнтську частину додатку.

- **UI (JavaFX):** Основний компонент, що реалізує графічний інтерфейс користувача та виступає в ролі "диригента" на клієнтській стороні.

- **State Manager, Preferences Manager, Interpreter:** Спеціалізовані компоненти, що реалізують відповідні патерни проєктування (State, Prototype, Interpreter) та керують станами UI, налаштуваннями та обробкою команд терміналу.
- **API Client:** Інкапсулює всю логіку комунікації з серверною частиною. Він формує та відправляє HTTP-запити, забезпечуючи чіткий та ізольований інтерфейс для взаємодії з сервером.

2. Шар «Server»

Цей шар є ядром системи, що містить всю бізнес-логіку. Він не має власного UI і працює у фоновому режимі.

- **HTTP API Server:** Вхідна точка сервера. Цей компонент приймає HTTP-запити від API Client, аналізує їх та виступає в ролі шлюзу, делегуючи подальшу обробку відповідним внутрішнім сервісам.
- **Operation Core:** Центральний компонент, що оркеструє виконання файлових операцій (копіювання, переміщення тощо).
- **Task Registry:** Відповідає за реєстрацію та відстеження стану довготривалих асинхронних завдань.
- **File System Adapter:** Ізолює та інкапсулює всю низькорівневу логіку роботи з файловою системою, надаючи чистий інтерфейс для решти серверної частини.
- **Database Repositories:** Надає абстрактний шар для доступу до бази даних (патерн Repository), приховуючи деталі виконання SQL-запитів.

3. Шар «External»

Представляє зовнішні ресурси, з якими взаємодіє сервер.

- **SQL Server** (або інша СУБД): Зовнішнє сховище даних. Компонент Database Repositories звертається до нього через протокол **JDBC** для збереження та отримання налаштувань, історії та закладок.
- **OS File System**: Файлова система комп'ютера. Компонент File System Adapter взаємодіє з нею через стандартну бібліотеку **java.nio.file** для виконання всіх файлових операцій.

3. Діаграми послідовностей

рис. 3. Основна, загальна діаграма послідовності

Діаграма послідовності моделює **високорівневий асинхронний протокол взаємодії** між клієнтом та сервером під час виконання будь-якої довготривалої файлової операції (наприклад, копіювання великого обсягу даних). Її головна мета - продемонструвати, як система забезпечує відгукливість інтерфейсу, при цьому, не "заморожуючи" його на час виконання важких завдань.

Учасники взаємодії:

- **Користувач:** Ініціатор операції.
- **ShellTotalCommander (UI):** Клієнтська частина, що відповідає за інтерфейс.
- **Серверна логіка:** Серверна частина, що виконує всю бізнес-логіку.
- **Файлова система (ОС):** Зовнішній ресурс, над яким виконується дія.

Послідовність кроків:

1. **Ініціація:** Користувач виконує файлову операцію через UI.
2. **Запуск задачі:** UI надсилає синхронний запит на сервер (Запустити операцію), передаючи необхідні параметри (наприклад, джерело та призначення).
3. **Асинхронне підтвердження:** Сервер **негайно** приймає задачу в обробку, реєструє її, присвоює унікальний ідентифікатор (taskId) і відразу ж

повертає цей ID клієнту. Це ключовий крок, який **не блокує потік користувацького інтерфейсу**.

4. **Паралельне виконання:** Після отримання taskId починається паралельний процес, що змодельовано за допомогою фрагмента **par**:
    - **Фонове виконання на сервері:** Серверна логіка в окремому потоці починає фактичне виконання важкої операції (копіювання/переміщення), яка може зайняти тривалий час. Після завершення сервер оновлює фінальний статус задачі у своєму реєстрі.
    - **Опитування статусу клієнтом:** Одночасно з цим UI, отримавши taskId, відображає вікно прогресу і входить у цикл (loop). У цьому циклі він періодично (наприклад, раз на секунду) надсилає на сервер запит про статус задачі, передаючи її taskId.
5. **Оновлення прогресу:** У відповідь на запити статусу сервер повертає поточний прогрес у відсотках, який UI використовує для оновлення прогрес-бару.
6. **Завершення:** Коли сервер завершує фонову операцію, він встановлює її статус як "завершено". При наступному запиті клієнт отримує цей статус, виходить з циклу loop і повідомляє користувача про успішне завершення операції.

На діаграмі бачимо:

- **Асинхронність:** Система не змушує клієнта чекати на завершення операції, що забезпечує високу відгукливість (responsiveness) інтерфейсу.
- **Механізм опитування (Polling):** Клієнт активно запитує стан задачі, що є простою та надійною реалізацією для відстеження прогресу в клієнт-серверних системах.

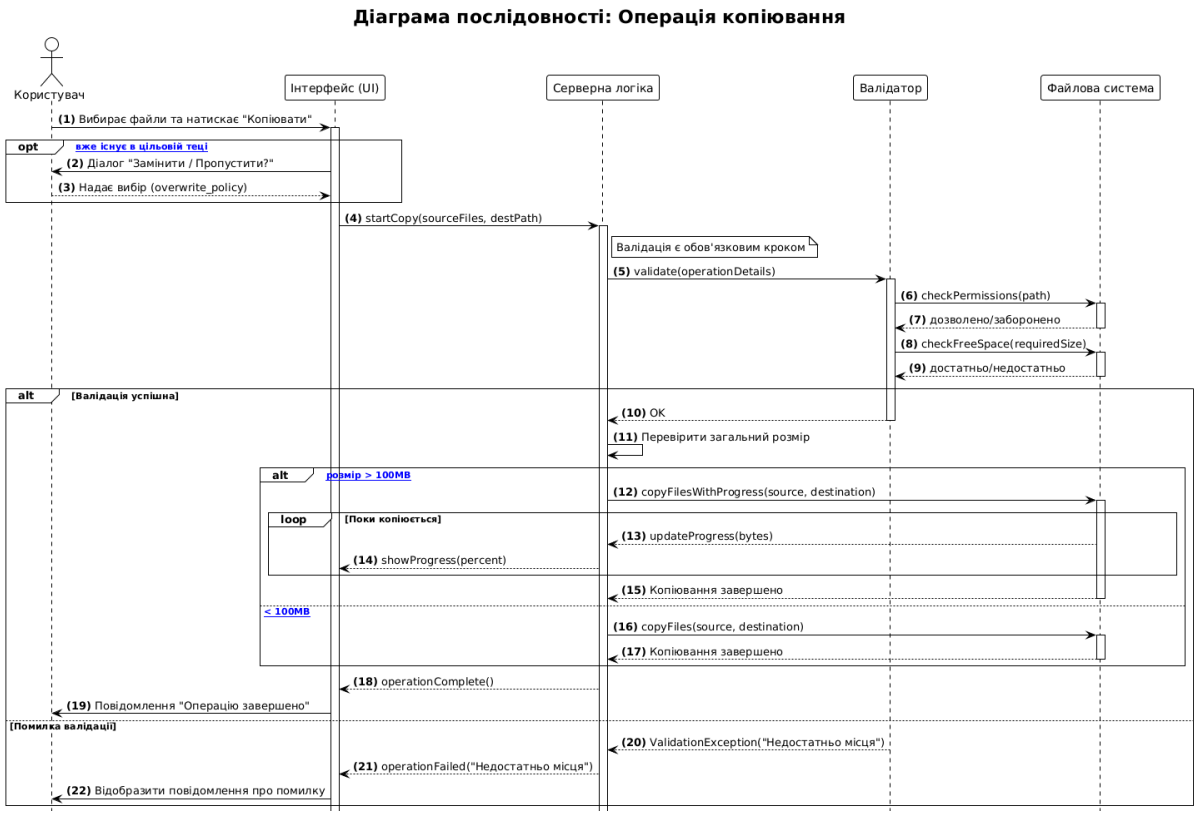

рис. 3.1. Більш конкретна діаграма послідовності

Діаграма послідовності детально моделює сценарій "Операція копіювання", розкриваючи внутрішню логіку взаємодії компонентів від моменту ініціації користувачем до завершення. На відміну від високорівневої діаграми, вона фокусується на конкретних кроках, валідації, обробці умов та помилок.

Учасники взаємодії:

- **Користувач:** Ініціює операцію.
- **Інтерфейс (UI):** Клієнтська частина, через яку користувач взаємодіє з системою.
- **Серверна логіка:** Основний компонент, що оркеструє процес копіювання.
- **Валідатор:** Спеціалізований компонент, відповідальний за перевірку можливості виконання операції. Його виокремлення демонструє принцип розділення відповідальності (Separation of Concerns).
- **Файлова система:** Зовнішній ресурс, з яким відбувається взаємодія.

Послідовність кроків:

1. **Ініціація та обробка конфліктів:** Користувач вибирає файли та натискає кнопку "Копіювати". За допомогою фрагмента **opt** (опція) моделюється опціональний крок: якщо файл з таким іменем вже існує в цільовій папці,

UI показує користувачеві діалог вибору ("Замінити / Пропустити") і отримує від нього політику перезапису (overwrite_policy).

2. **Запуск операції:** UI викликає метод startCopy на серверній логіці, передаючи список файлів для копіювання та шлях призначення.
3. **Валідація:** Це обов'язковий крок. Серверна логіка делегує перевірку умов компоненту Валідатор, викликаючи метод validateOperationDetails. Валідатор послідовно виконує кілька перевірок, звертаючись до файлової системи: перевіряє права доступу (checkPermissions), наявність вільного місця на диску (checkFreeSpace) тощо.
4. **Розгалуження логіки (Успіх / Помилка):** Результат валідації обробляється у фрагменті **alt** (альтернатива), що розділяє подальший потік на два сценарії:
    - **Сценарій "Валідація успішна":**
        - Серверна логіка отримує підтвердження (OK) від Валідатора.
        - Далі логіка розгалужується ще раз за допомогою **вкладеного alt** в залежності від розміру файлів:
            - **Якщо розмір > 100MB:** Сервер викликає метод copyFilesWithProgress. Усередині циклу **loop** серверна логіка покроково копіює дані, періодично викликаючи updateProgress. У свою чергу, UI отримує ці оновлення і відображає їх (showProgress).
            - **Якщо розмір < 100MB:** Сервер викликає простіший метод copyFiles, який виконує операцію за один крок без відстеження прогресу.
        - Після завершення копіювання сервер повідомляє UI (operationComplete), а той, у свою чергу, сповіщає користувача про успішне завершення.

    - **Сценарій "Помилка валідації":**
        - Валідатор повертає помилку (наприклад, ValidationException з повідомленням "Недостатньо місця").
        - Серверна логіка перехоплює цю помилку і викликає метод operationFailed на UI, передаючи деталі помилки.
        - UI відображає користувачеві відповідне повідомлення про помилку.

На діаграмі бачимо:

- **Детальна обробка умов:** Діаграма чітко моделює різні шляхи виконання залежно від наявності конфліктів та розміру файлів.
- **Розділення відповідальності:** Винесення логіки валідації в окремий компонент Валідатор робить код серверної логіки чистішим і більш сфокусованим.
- **Відстеження прогресу:** Детально показаний механізм покрокового виконання та оновлення інтерфейсу для довготривалих операцій.
- **Надійна обробка помилок:** Передбачено чіткий шлях для обробки помилок на етапі валідації, що робить систему більш стабільною.

Висновки

Висновки: Під час виконання даної лабораторної роботи було успішно спроєктовано ключові архітектурні діаграми для системи, що дозволило деталізувати як її статичну структуру, так і динамічну поведінку. Була розроблена діаграма розгортання, яка візуалізувала фізичну архітектуру системи, продемонструвавши розгортання клієнт-серверного застосунку у вигляді єдиного артефакту на одному комп'ютері користувача. Створена деталізована діаграма компонентів чітко визначила модульну структуру програми, її логічні шари та залежності між ними, а також явно показала компоненти, відповідальні за реалізацію ключових патернів проєктування.

Для моделювання динамічної взаємодії були розроблені дві діаграми послідовностей. Перша, високорівнева, продемонструвала асинхронний протокол виконання довготривалих файлових операцій, що забезпечує відгукливість інтерфейсу. Друга, низькорівнева, детально розкрила повний сценарій операції копіювання, включаючи кроки валідації, умовну логіку та обробку помилок. В результаті було створено комплексний та узгоджений набір UML-діаграм, який слугує надійним архітектурним фундаментом для подальшої розробки та реалізації всього запланованого функціоналу системи.

Відповіді на КЗ

1. Що собою становить діаграма розгортання?

Діаграма розгортання - це тип діаграми в UML, яка візуалізує фізичну архітектуру системи. Вона показує, як програмні компоненти (артефакти) розміщуються на фізичних пристроях (вузлах) і як ці пристрої з'єднані між собою. По суті, вона моделює статичне розміщення системи в реальному світі.

2. Які бувають види вузлів на діаграмі розгортання?

На діаграмі розгортання існують два основні види вузлів:

1. **Пристрій (Device):** Це фізичне обладнання, наприклад, комп'ютер, сервер, мобільний пристрій або інший апаратний ресурс.
2. **Середовище виконання (Execution Environment):** Це програмне забезпечення, яке знаходиться всередині пристрою і слугує для запуску інших програм, наприклад, операційна система (Windows, Linux), веб-сервер (Apache) або віртуальна машина (JRE).

3. Які бувають зв’язки на діаграмі розгортання?

На діаграмі розгортання основний тип зв'язку - це **шлях сполучення (Communication Path)**. Він зображується у вигляді суцільної лінії між двома вузлами і показує, що між ними існує канал зв'язку. Цей зв'язок зазвичай підписують назвою протоколу (наприклад, HTTP, TCP/IP, JDBC) або технології, що використовується для взаємодії.

4. Які елементи присутні на діаграмі компонентів?

Основними елементами діаграми компонентів є:

1. **Компонент (Component):** Модульна, логічно відокремлена частина системи з чітко визначеним інтерфейсом (наприклад, UI, API-сервер, модуль доступу до даних).
2. **Залежність (Dependency):** Зв'язок, що показується пунктирною стрілкою і означає, що один компонент використовує функціонал іншого.
3. **Інтерфейс (Interface):** Контракт, який визначає, які послуги надає компонент, не розкриваючи його внутрішньої реалізації.

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

Зв'язки на діаграмі компонентів переважно представляють **залежність (Dependency)**. Такий зв'язок показує, що один компонент (клієнт) для своєї роботи використовує класи або інтерфейси, надані іншим компонентом (постачальником). По суті, це означає, що зміни в компоненті-постачальнику можуть вплинути на роботу компонента-клієнта.

6. Які бувають види діаграм взаємодії?

Діаграми взаємодії - це категорія UML-діаграм, що моделюють динамічну поведінку системи. До основних видів належать:

1. **Діаграма послідовностей (Sequence Diagram):** Фокусується на часовій послідовності обміну повідомленнями.
2. **Діаграма комунікації (Communication Diagram):** Фокусується на структурі взаємозв'язків між об'єктами, а не на часі.
3. **Діаграма огляду взаємодії (Interaction Overview Diagram):** Високорівнева діаграма, що поєднує інші діаграми взаємодії для моделювання складних сценаріїв.
4. **Діаграма синхронізації (Timing Diagram):** Фокусується на зміні стану об'єктів у часі.

7. Для чого призначена діаграма послідовностей?

Діаграма послідовностей призначена для візуалізації взаємодії між об'єктами системи в межах одного сценарію. Її головна мета - показати точну послідовність обміну повідомленнями (викликів методів) між об'єктами в часі. Вона допомагає деталізувати логіку роботи, зрозуміти, як реалізується той чи інший варіант використання, та виявити потенційні проблеми у взаємодії компонентів.

8. Які ключові елементи можуть бути на діаграмі послідовностей?

Ключовими елементами діаграми послідовностей є:

1. **Актор (Actor):** Ініціатор сценарію (користувач або зовнішня система).
2. **Об'єкт (Object):** Учасник взаємодії, від якого йде лінія життя (lifeline).
3. **Повідомлення (Message):** Стрілка між лініями життя, що позначає виклик методу або передачу даних.
4. **Активація (Activation):** Прямокутник на лінії життя, що показує час виконання операції.
5. **Фрагменти взаємодії:** Спеціальні блоки для моделювання логіки, такі як alt (альтернатива), opt (опція) та loop (цикл).

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

Діаграми послідовностей є деталізацією діаграм варіантів використання. Якщо діаграма варіантів використання показує, **ЩО** система робить (наприклад, "Копіювати файл"), то діаграма послідовностей показує, **ЯК** саме система це робить. Кожен конкретний сценарій, що реалізує певний варіант використання, може бути змодельований за допомогою однієї або кількох діаграм послідовностей (наприклад, для успішного виконання та для сценаріїв з помилками).

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

Діаграми послідовностей і діаграми класів доповнюють одна одну, показуючи динамічну та статичну структуру системи відповідно. Діаграма класів визначає **структуру** (які класи існують та які методи вони мають), тоді як діаграма послідовностей показує **поведінку** (як екземпляри цих класів взаємодіють між собою, викликаючи ці методи). Об'єкти, що є учасниками на діаграмі послідовностей, є екземплярами класів, визначених на діаграмі класів, а повідомлення, якими вони обмінюються, відповідають операціям (методам) цих класів.

Код програми

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
```

```
import javafx.stage.Stage;

// --- DatabaseContext.java ---
class DatabaseContext {
    private final String url;

    public DatabaseContext(String url) {
        this.url = url;
        init();
    }

    private void init() {
        try (Connection conn = getConnection();
             Statement stmt = conn.createStatement()) {

            stmt.executeUpdate(
                "CREATE TABLE IF NOT EXISTS Settings (" +
                "profile_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                "app_name TEXT NOT NULL)");

            stmt.executeUpdate(
                "CREATE TABLE IF NOT EXISTS Preferences (" +
                "profile_id INTEGER PRIMARY KEY, " +
                "theme TEXT, " +
                "show_hidden BOOLEAN, " +
                "default_sort TEXT, " +
                "FOREIGN KEY(profile_id) REFERENCES Settings(profile_id))");

            stmt.executeUpdate(
                "CREATE TABLE IF NOT EXISTS SearchHistory (" +
                "search_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                "profile_id INTEGER, " +
                "root_path TEXT NOT NULL, " +
                "pattern TEXT, " +
                "FOREIGN KEY(profile_id) REFERENCES Settings(profile_id))");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
    public Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url);
    }
}

// --- Preferences.java ---
class Preferences {
    private int profileId;
    private String theme;
    private boolean showHidden;
    private String defaultSort;

    public int getProfileId() { return profileId; }
    public void setProfileId(int profileId) { this.profileId = profileId; }
    public String getTheme() { return theme; }
    public void setTheme(String theme) { this.theme = theme; }
    public boolean isShowHidden() { return showHidden; }
    public void setShowHidden(boolean showHidden) { this.showHidden =
showHidden; }
    public String getDefaultSort() { return defaultSort; }
    public void setDefaultSort(String defaultSort) { this.defaultSort = defaultSort; }
}

// --- PreferencesRepository.java ---
class PreferencesRepository {
    private final DatabaseContext ctx;

    public PreferencesRepository(DatabaseContext ctx) {
        this.ctx = ctx;
    }

    public void update(Preferences entity) throws SQLException {
        String sql = "UPDATE Preferences SET theme = ?, show_hidden = ?,
default_sort = ? WHERE profile_id = ?";
        try (Connection c = ctx.getConnection();
             PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setString(1, entity.getTheme());
            ps.setBoolean(2, entity.isShowHidden());
            ps.setString(3, entity.getDefaultSort());
```

```
            ps.setInt(4, entity.getProfileId());
            ps.executeUpdate();
        }
    }

    public Preferences getByProfileId(int profileId) throws SQLException {
        String sql = "SELECT * FROM Preferences WHERE profile_id = ?";
        try (Connection c = ctx.getConnection();
             PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setInt(1, profileId);
            try (ResultSet rs = ps.executeQuery()) {
                if (rs.next()) {
                    Preferences p = new Preferences();
                    p.setProfileId(rs.getInt("profile_id"));
                    p.setTheme(rs.getString("theme"));
                    p.setShowHidden(rs.getBoolean("show_hidden"));
                    p.setDefaultSort(rs.getString("default_sort"));
                    return p;
                }
            }
        }
        return null;
    }
}

// --- HttpApiServer.java---
class HttpApiServer {
    private final PreferencesRepository prefsRepo;

    public HttpApiServer(PreferencesRepository prefsRepo) {
        this.prefsRepo = prefsRepo;
    }

    public void start() { }
}

// --- SettingsForm.java ---
class SettingsForm extends Stage {
    public SettingsForm() {
        initUi();
```

```
        loadData();
    }

    private void initUi() {
        setTitle("Налаштування");
        VBox root = new VBox(10);
        TextField themeField = new TextField();
        CheckBox showHiddenCheck = new CheckBox("Показувати приховані
файли");
        Button saveButton = new Button("Зберегти");

        saveButton.setOnAction(e -> saveData(themeField.getText(),
showHiddenCheck.isSelected()));

        root.getChildren().addAll(new Label("Тема:"), themeField, showHiddenCheck,
saveButton);
        setScene(new Scene(root, 300, 200));
    }

    private void loadData() { }

    private void saveData(String theme, boolean showHidden) { }
}

// --- ViewHistoryForm.java ---
class ViewHistoryForm extends Stage {
    private TableView<Object> table;

    public ViewHistoryForm() {
        initUi();
        loadData();
    }

    private void initUi() {
        setTitle("Історія Пошуку");
        table = new TableView<>();

        Button refreshButton = new Button("Оновити");
        refreshButton.setOnAction(e -> loadData());
```

```
        BorderPane root = new BorderPane(new ScrollPane(table));
        root.setBottom(refreshButton);
        setScene(new Scene(root, 600, 400));
    }

    private void loadData() { }
}

// --- Main.java ---
public class Main extends Application {
    public static void main(String[] args) {
        DatabaseContext ctx = new DatabaseContext("jdbc:sqlite:shell_commander.db");
        PreferencesRepository prefsRepo = new PreferencesRepository(ctx);
        HttpApiServer server = new HttpApiServer(prefsRepo);
        new Thread(() -> server.start()).start();

        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        SettingsForm f1 = new SettingsForm();
        f1.setX(200);
        f1.setY(100);
        f1.show();

        ViewHistoryForm f2 = new ViewHistoryForm();
        f2.setX(550);
        f2.setY(100);
        f2.show();
    }
}
```