НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра інформаційних систем та технологій

Тема Powershell terminal

Курсова робота

З дисципліни «ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

Керівник
доц. Амонс О.А.

«Допущений до захисту»

_____

(Особистий підпис керівника)
« » _____2025р.

Захищений з оцінкою

_____

(оцінка)

Виконавець
ст. Панащук Р. А.

залікова книжка № ІА – 3315

_____

(особистий підпис виконавця)
«___»_____2025р.

Члени комісії:

_____ _____
(особистий підпис) (розшифровка підпису)

_____ _____
(особистий підпис) (розшифровка підпису)

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

(назва навчального закладу)

Дисципліна «Технології розроблення програмного забезпечення»

Курс 3 Група_ІА-33__ Семестр__5_____

ЗАВДАННЯ

на курсову роботу студента

Панащука Романа Андрійовича

(прізвище, ім’я, по батькові)

1.Тема роботи: Powershell terminal

2. Строк здачі студентом закінченої роботи: 25.11.2025

3. Вихідні дані до роботи:

Powershell terminal; шаблони проєктування, які сліди використати: strategy, command, abstract factory, bridge, interpreter, client-server. Термінал для powershell повинен нагадувати типовий термінал з можливістю налаштування кольорів синтаксичних конструкцій, розміру вікна, фону вікна, а також виконання команд powershell і виконуваних файлів, а також працювати в декількох вікнах терміналу (у вкладках або одночасно шляхом розділення вікна).

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці):

Проектування: огляд існуючих рішень, загальний опис проєкту, вимоги до застосунку системи (функціональні вимоги та нефункціональні вимоги), опис сценаріїв використання, концептуальна модель системи, вибір БД, вибір мови програмування та середовища розробки, проєктування розгортання; Реалізація: структура БД, архітектура системи (специфікація системи), вибір та обгрунтування патернів реалізації, інструкція користувача.

5. Дата видачі завдання:_17.09.2025р.

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Підбір та вивчення літератури	30.09.2025	
2.	Проектування та написання розділу 1	31.10.2025	
3.	Розробка та написання розділу 2	20.11.2025	
4.	Подання курсової роботи на перевірку	25.11.2025	
5.	Захист курсової роботи	08.12.2025	
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			
18.			

Студент _____ _Роман Панащук_____
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник_____ __Олександр АМОНС__
(підпис) (Ім’я ПРІЗВИЩЕ)

«____»_____ 20___ р.

ВСТУП

Сучасні інформаційні технології стрімко розвиваються, і разом із цим зростає потреба у створенні ефективних інструментів для управління системами та автоматизації процесів. Одним із таких інструментів є PowerShell, який надає розширені можливості для автоматизації адміністративних завдань та взаємодії з операційною системою. Проте, для зручнішого використання PowerShell у великих проєктах необхідно мати зручний інтерфейс, який би дозволяв користувачу налаштовувати вигляд та функціональність терміналу відповідно до його потреб.

Актуальність даної теми полягає у створенні терміналу для PowerShell, який, на відміну від стандартного, надає додаткові можливості, такі як налаштування кольорів синтаксичних конструкцій, зміна розміру та фону вікна, а також робота з декількома вікнами одночасно. Це підвищує ефективність роботи адміністраторів систем, розробників і DevOps-інженерів, які використовують PowerShell для виконання своїх завдань.

Мета роботи полягає в розробці інтерактивного терміналу для PowerShell з підтримкою основних функцій налаштування зовнішнього вигляду та багатовіконного режиму, а також можливості виконання команд і скриптів. Для досягнення цієї мети будуть використані шаблони проєктування, такі як strategy, command, abstract factory, interpreter, client-server, що дозволить створити гнучку та розширювану архітектуру.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1 Огляд існуючих рішень

На сьогоднішній день існує декілька популярних терміналів, які підтримують виконання команд PowerShell та забезпечують базові можливості для користувача. Одним із найвідоміших рішень є Windows Terminal, який підтримує кілька типів командних інтерпретаторів, включаючи PowerShell, Command Prompt, WSL та інші. Він надає користувачам можливість роботи з вкладками, налаштуванням кольорів і шрифтів, але його можливості налаштування вигляду та поведінки терміналу обмежені стандартним інтерфейсом.

Іншим популярним рішенням є ConEmu, який пропонує більш розширені можливості налаштування інтерфейсу, такі як зміна кольорових схем, багатовіконний режим, і підтримка кількох інтерпретаторів команд. Він також забезпечує високий

рівень гнучкості, але складність його налаштування може стати перешкодою для новачків.

Terminus є ще одним інструментом, який підтримує роботу з PowerShell. Його основними перевагами є кросплатформеність та підтримка розширених функцій налаштування терміналу. Цей термінал надає можливість налаштування вигляду, розміру та роботи з декількома вікнами, однак для складніших користувацьких потреб може не вистачати вбудованих функцій.

Окремо варто згадати Visual Studio Code, яке є одним із найпопулярніших редакторів коду з вбудованим терміналом, що підтримує PowerShell через відповідне розширення. Visual Studio Code надає зручне інтегроване середовище для розробки та виконання скриптів PowerShell, а також можливості налаштування підсвітки синтаксису, роботи з багатьма вкладками та виконання команд безпосередньо в терміналі. Проте, основний акцент Visual Studio Code робить на редагуванні коду, і його термінал не завжди задовольняє потреби користувачів, яким потрібен гнучкіший інструмент саме для командної роботи в PowerShell.

Незважаючи на широкий вибір інструментів, більшість із них або обмежені у функціональності, або вимагають складних налаштувань для інтеграції з іншими системами. Вони також не завжди дозволяють легко змінювати синтаксичне підсвічування або виконувати специфічні налаштування для окремих команд чи скриптів. Крім того, існуючі рішення не завжди підтримують роботу в багатовіконному режимі із зручною інтеграцією з вкладками.

Таким чином, актуальним є створення терміналу для PowerShell, який поєднує в собі всі ці можливості, забезпечуючи зручний інтерфейс для налаштування зовнішнього вигляду, багатовіконний режим роботи та виконання команд, а також реалізує гнучку архітектуру завдяки використанню сучасних шаблонів проєктування.

1.2 Загальний опис проєкту

Метою проєкту є розробка інтерактивного терміналу для PowerShell, який забезпечить користувачам покращені можливості роботи з командним інтерпретатором. Проєкт спрямований на створення зручного інтерфейсу, який дозволяє налаштовувати кольори синтаксичних конструкцій, розміри та фон вікна, а також підтримує роботу в багатовіконному режимі (через вкладки). Окрім цього, термінал буде підтримувати виконання команд PowerShell і скриптів, що значно підвищить ефективність взаємодії користувачів з системою.

Для реалізації цього проєкту буде використано кілька шаблонів проєктування, таких як strategy, command, abstract factory, interpreter, і client-server, що дозволить

забезпечити гнучкість, модульність і легкість розширення системи. Важливою особливістю є можливість масштабування терміналу для додаткових функцій у майбутньому, що дозволить інтегрувати нові можливості без значної зміни основної архітектури.

Основними компонентами терміналу є:

1. Налаштування інтерфейсу — можливість зміни кольорових схем синтаксичного підсвічування, розмірів вікна та фону.
2. Виконання команд PowerShell — інтерпретатор команд PowerShell для виконання скриптів і команд безпосередньо в терміналі.
3. Багатовіконний режим — можливість одночасної роботи з декількома терміналами через вкладки.
4. Модульність та масштабованість — використання шаблонів проєктування для забезпечення гнучкості архітектури та можливості додавання нових функцій у майбутньому.

Проєкт спрямований на користувачів, які працюють із PowerShell для автоматизації адміністративних завдань, розробки або тестування програмного забезпечення. Він надасть їм зручний інструмент для виконання щоденних завдань із можливістю індивідуальних налаштувань інтерфейсу та оптимізації робочих процесів.

1.3 Вимоги до застосунків системи

1.3.1 Функціональні вимоги до системи

Система Powershell terminal повинна відповідати наступним функціональним вимогам:

- Сервер має обробляти HTTP-запити, які надходять від клієнта (JavaFX), через REST API, використовуючи Spring Boot для організації сервісів.
- Сервер має надавати детальні відповіді з помилками у разі невдалого виконання команд.
- Сервер має використовувати PostgreSQL для зберігання даних про вкладки. - Користувач повинен мати можливість налаштовувати колір фону вкладки.
- Користувач повинен мати можливість вводити PowerShell-команди через інтерфейс терміналу в JavaFX, після чого команда відправляється на сервер для виконання.

- Клієнт має підтримувати багато вікон терміналу, що дозволяє одночасно виконувати кілька команд у різних вкладках.
- Результат виконання команд або помилки мають відображатися в терміналі JavaFX в реальному часі.
- Користувач повинен мати можливість налаштовувати колір синтаксичних конструкцій та фону.

1.3.2 Нефункціональні вимоги до системи

Система Powershell terminal повинна відповідати наступним нефункціональним вимогам:

- Налаштування кольорів синтаксису та фону мають бути доступними через інтерфейс терміналу.
- У разі збоїв у виконанні команд термінал має виводити інформативні повідомлення про помилки без завершення роботи всього застосунку.
- Інтерфейс терміналу повинен бути інтуїтивно зрозумілим і відповідати стандартам типового терміналу.
- Забезпечення простоти розширення функціоналу шляхом використання стандартних шаблонів проєктування.

1.4 Сценарії використання системи

Діаграма сценаріїв використання представлена на рисунку 1.4.1. Сценарії використання - це текстові уявлення тих процесів, які відбуваються під час взаємодії користувачів з терміналом та самого терміналу. Вони є чітко формалізованими, покроковими інструкціями, що описують той чи інший процес.

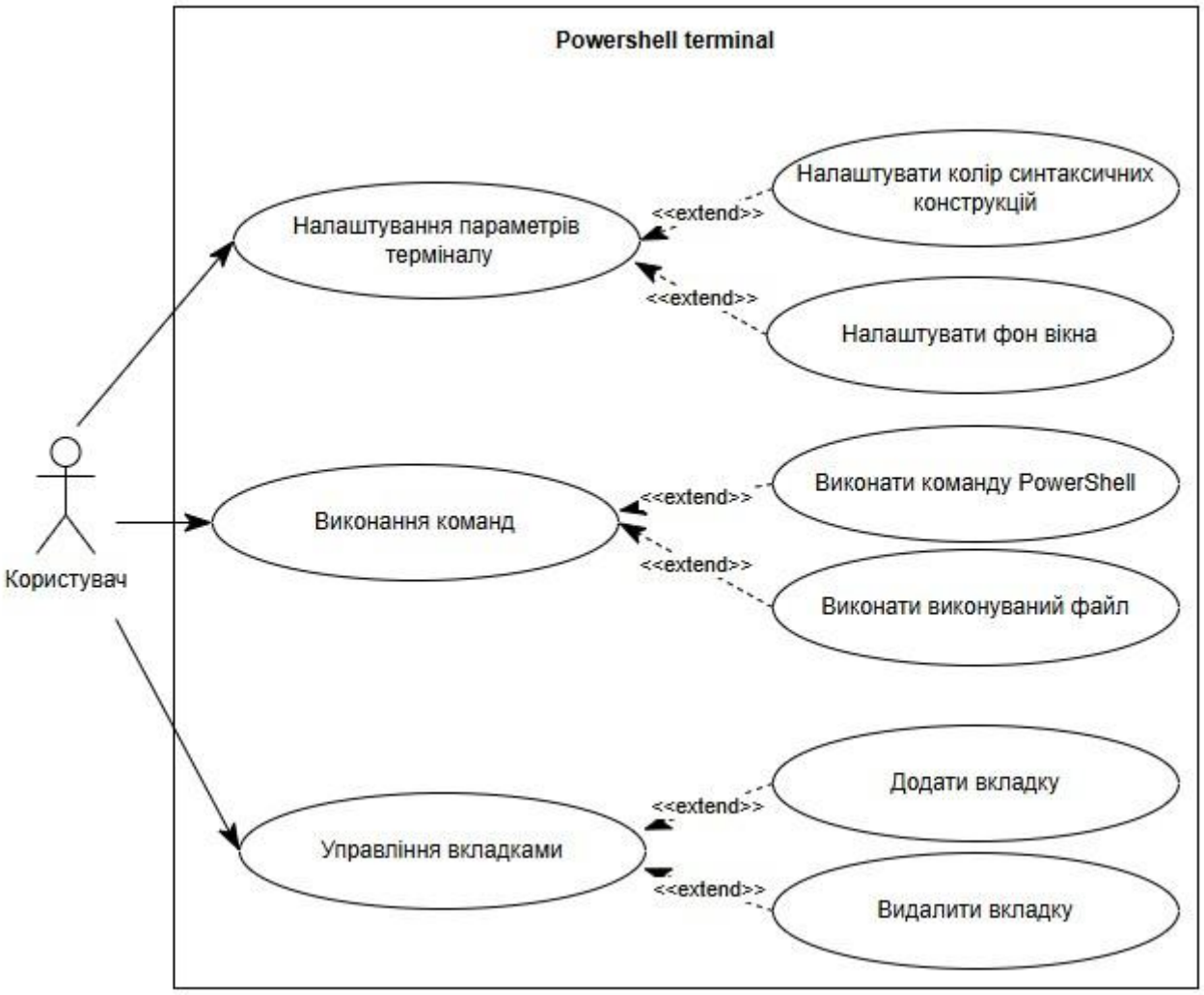

Рисунок 1.4.1 – Діаграма варіантів використання системи Детальний опис сценаріїв використання наведений у таблицях 1.4.1- 1.4.3.

Таблиця 1.4.1. - Прецедент 1: Налаштування параметрів терміналу.

Передумови	Користувач відкрив термінал
Постумови	Параметри терміналу (колір, фон, розмір) налаштовані відповідно до вибору користувача
Короткий опис	Користувач змінює налаштування кольору синтаксису, фону вікна або розміру вікна

Основний хід подій	1. Користувач натискає на кнопку для вибору кольору для фону та тексту або змінює розмір вікна. 2. Система пропонує обрати колір синтаксису, фон вікна. 3. Користувач вказує потрібні значення для одного або декількох параметрів. 4. Система надсилає дані на сервер та записує їх до бази даних, де вони зберігаються для поточної вкладки та застосовує нові налаштування.
Винятки	- Якщо користувач не обрав колір, система не застосовує налаштування. - Якщо введені дані некоректні, система повідомляє про помилку і повертається до попередніх налаштувань.
Примітки	Параметри можна змінити в будь-який момент без необхідності перезапуску терміналу.

Таблиця 1.4.2 - Прецедент 2: Виконання команди PowerShell.

Передумови	Користувач має доступ до терміналу і ввів команду PowerShell
Постумови	Вказана команда PowerShell успішно виконана або система показала повідомлення про помилку.
Короткий опис	Користувач вводить команду PowerShell, або додає виконуваний файл для виконання в терміналі.
Основний хід подій	1. Користувач вводить команду PowerShell або виконуваний файл у терміналі. 2. Команда або файл надсилаються на сервер. 3. Система перевіряє коректність команди, визначає чи це Powershell команда, чи виконуваний файл. 3. Команда виконується, і результат надсилається до клієнта. 4. Система показує результат або повідомлення про помилку у випадку невдалої спроби.

Винятки	- Якщо команда містить помилки, система виводить відповідне повідомлення із зазначенням помилки.

Таблиця 1.4.3 - Прецедент 3: Управління вкладками.

Передумови	Користувач працює з терміналом і має відкриту вкладку.
Постумови	Додана або видалена вкладка у терміналі.
Короткий опис	Користувач може додавати або видаляти вкладки для одночасної роботи з декількома сесіями терміналу.
Основний хід подій	1. Користувач натискає кнопку "Нова вкладка". 2. Клієнт надсилає запит до сервера, сервер створює нову вкладку з стандартними кольорами фону та тексту і додає цю інформацію до бази даних. 3. Після отримання відповіді від сервера, клієнт відкриває нову вкладку в терміналі. 4. Користувач може переключатися між вкладками для роботи з різними сесіями. 5. Користувач може видалити вкладку, якщо вона більше не потрібна для цього також надсилається запит до сервера та вкладка з відповідним Id видаляється з бази даних.
Винятки	Відсутні
Примітки	- Коли користувач закриває термінал, надсилається запит до сервера та видаляються всі збережені у базі даних вкладки, що були відкриті. - Кількість вкладок може бути обмежено системними ресурсами.

Також наведемо діаграму послідовностей для основних сценаріїв використання терміналу, вона зображена на рисунку 1.4.2. Діаграма використовується для моделювання процесу виконання операцій у мові UML. Кожен стан на діаграмі

послідовностей відповідає виконанню деякої елементарної операції, а перехід у наступний стан виконується тільки при завершенні цієї операції.

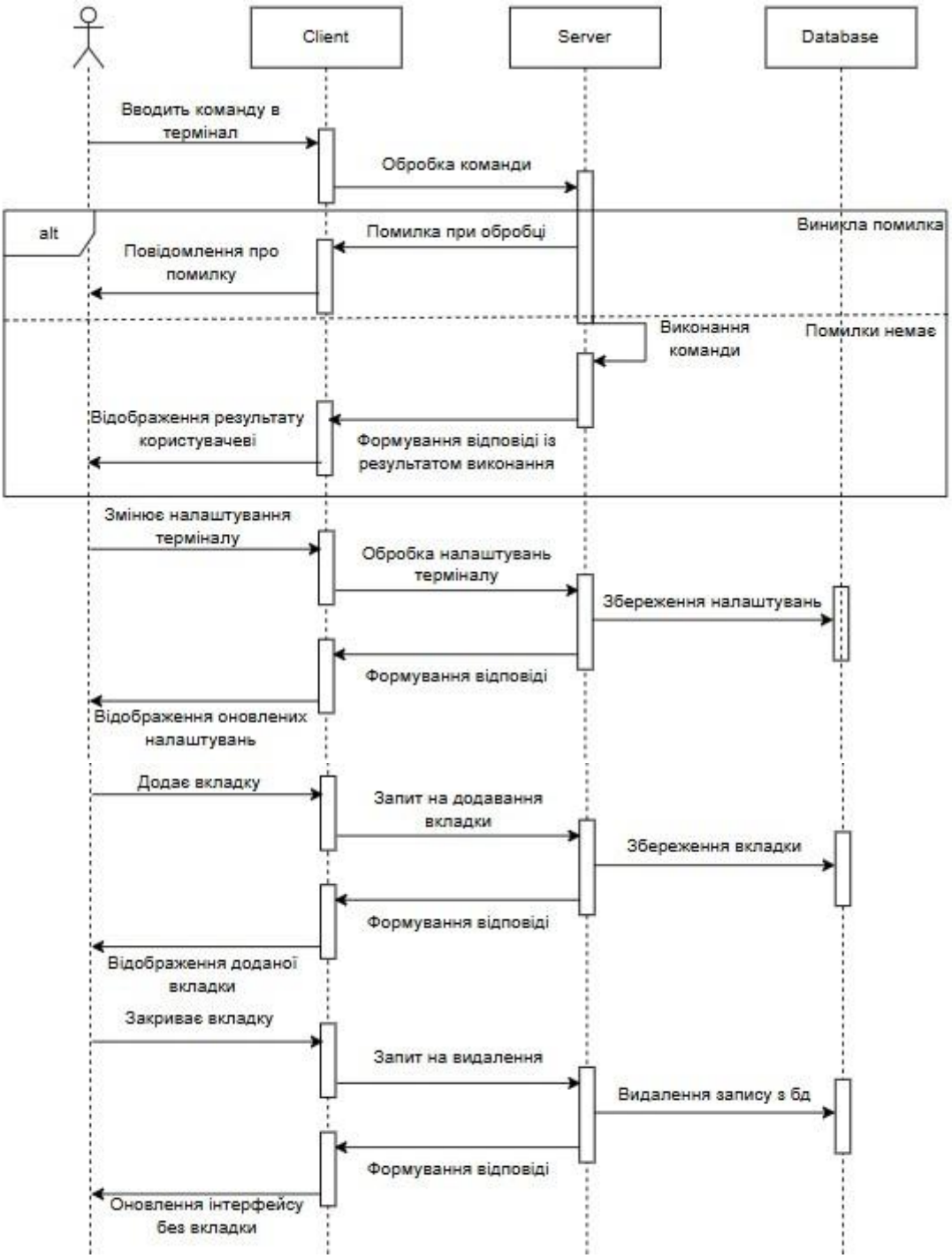

Рисунок 1.4.2 – Діаграма послідовностей основних сценаріїв застосунку

1.5 Концептуальна модель системи

Концептуальна модель — це абстрактне представлення системи або її частини, що використовується для опису основних понять, їх властивостей, взаємозв’язків і обмежень без деталізації технічних або реалізаційних аспектів. Вона створюється на ранніх етапах розробки для розуміння сутності проблемної області та формування спільного бачення між учасниками проєкту.

Основні особливості концептуальної моделі: абстракція (концентрується на високорівневих поняттях, відкидаючи деталі реалізації), фокус на предметній області (Визначає ключові об’єкти, а саме суб’єкти, процеси, події та їхні взаємозв’язки), міждисциплінарність (використовується для спілкування між різними сторонами, наприклад замовниками, розробниками, бізнес-аналітиками, без складних технічних термінів), гнучкість і незалежність від технологій (вона не залежить від мови програмування, бази даних чи архітектурного рішення).

На рисунку 1.5.1 зображена концептуальна модель системи у форматі діаграми компонентів.

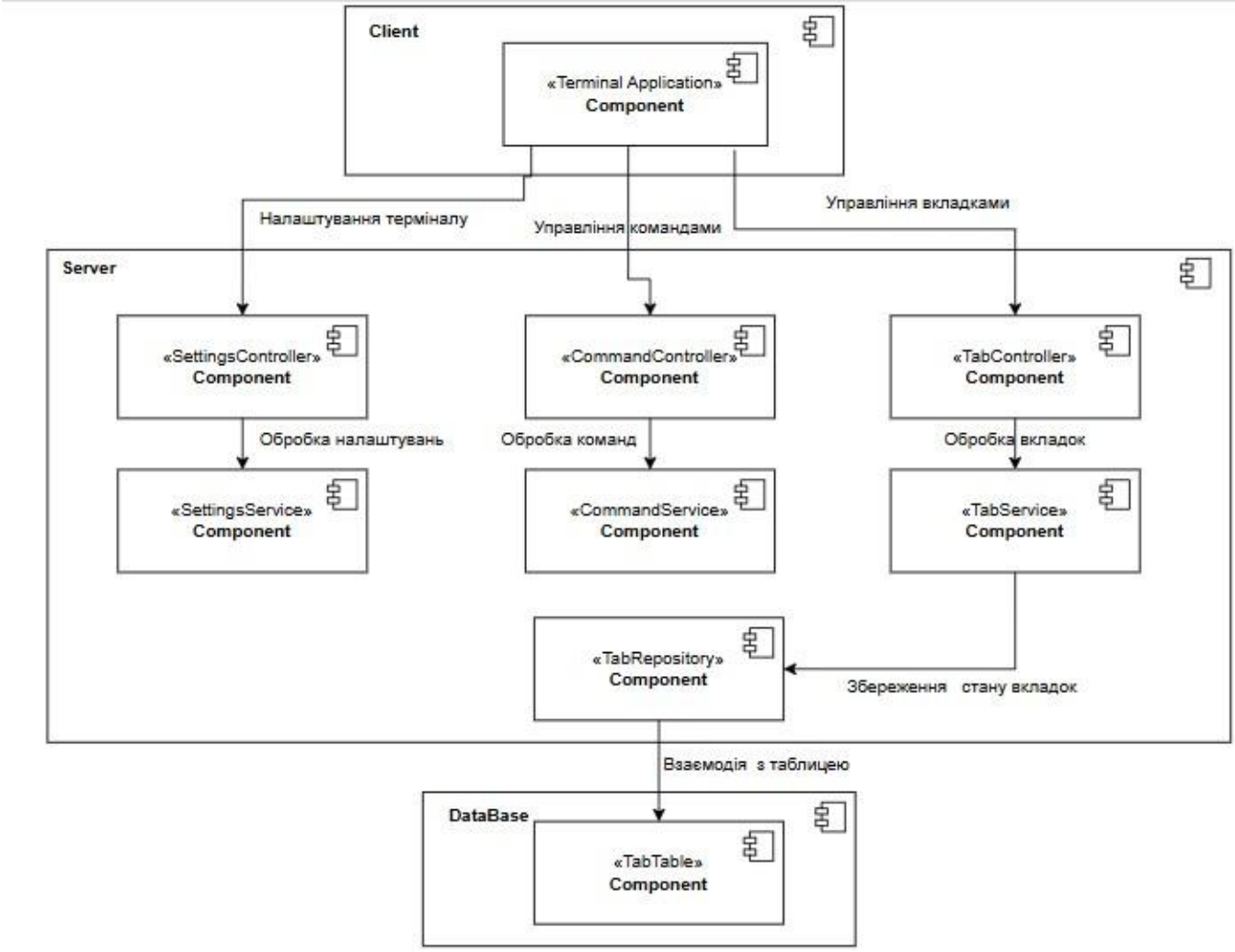

Рисунок 1.5.1 – Діаграма компонентів

Опис діаграми компонентів:

- Client(JavaFX):
  - User Interface (UI): відповідає за графічний інтерфейс терміналу. Включає компоненти для відображення вікна, вкладок, налаштувань кольору, фону та розміру вікна, керує налаштуваннями терміналу, обробляє відкриття, перемикання та видалення вкладок у терміналі, підтримує можливість одночасної роботи з декількома сесіями, відповідає за взаємодію користувача з терміналом, пов'язану з введенням та отриманням результатів команд. - Server (Spring Boot):
  - CommandController Component – обробляє запити на виконання команд від клієнта, перевіряє їх на коректність і передає для виконання.

- SettingsController Component – обробляє запити щодо налаштувань терміналу, отримує та передає налаштування в сервіс налаштувань.
- TabController Component – обробляє запити, пов'язані з управлінням вкладками, забезпечує можливість зберігання вкладок та їх інформації.
- CommandService Component – реалізує бізнес-логіку для виконання команд, перевіряє команди на виконує команди і повертає результати.
- TabService Component – керує вкладками, обробляє додавання, видалення та збереження стану вкладок.
- SettingsService Component - реалізує бізнес-логіку налаштування параметрів терміналу, працює з TabRepository для збереження і оновлення даних налаштувань у таблиці Tab.
- TabRepository Component - зберігає інформацію про вкладки, їхню поточну директорію, кольори для фону та тексту.

- Таблиці:
  - TabTable Component – таблиця, яка містить дані про вкладки, зокрема унікальний ідентифікатор, назву вкладки, колір фону, колір синтаксичних конструкцій та поточну директорію.

1.6 Вибір бази даних

Для реалізації проєкту PowerShell Terminal, який передбачає створення клієнтсерверного застосунку з підтримкою налаштувань користувача, виконання команд та роботи з кількома вікнами, я обрав PostgreSQL як систему управління базами даних. Вибір цієї СУБД зумовлений наступними причинами:

- PostgreSQL є реляційною СУБД, яка забезпечує підтримку складних структур даних, таких як JSON, масиви та таблиці з вкладеними типами. Це важливо для зберігання налаштувань кожної вкладки терміналу (колір синтаксису, розмір шрифту, фон тощо). PostgreSQL дозволяє використовувати розширення та функції, що спрощують адаптацію бази даних до специфічних потреб проєкту. Завдяки цьому можна ефективно масштабувати систему для підтримки великої кількості користувачів або вкладок.

- Виконання команд PowerShell і змін у налаштуваннях терміналу може вимагати збереження проміжних станів і забезпечення цілісності даних. PostgreSQL гарантує транзакційну обробку, що запобігає втраті або пошкодженню даних у разі помилок. PostgreSQL має потужну інтеграцію з Java, що робить її ідеальною для використання у клієнт-серверному застосунку на основі Spring Boot. Це спрощує розробку REST API для взаємодії з клієнтською частиною (JavaFX).

- PostgreSQL забезпечує високу швидкість обробки запитів навіть для великих обсягів даних, що важливо для виконання команд PowerShell і маніпуляцій із вкладками в реальному часі. Система має вбудовані механізми аутентифікації, шифрування та розмежування доступу до даних, що дозволяє надійно зберігати інформацію.

- PostgreSQL є безкоштовним рішенням з відкритим кодом, що робить його економічно вигідним для використання в навчальних проєктах. Активна спільнота розробників забезпечує регулярне оновлення, підтримку та наявність якісної документації.

Таким чином, PostgreSQL є оптимальним вибором для реалізації бази даних у проєкті PowerShell Terminal завдяки своїй надійності, функціональності та сумісності з обраними технологіями.

1.7 Вибір мови програмування та середовища розробки

Для реалізації проєкту PowerShell Terminal було обрано мову програмування Java та середовище розробки IntelliJ IDEA. Цей вибір зумовлений особливостями задачі та необхідністю використання сучасних інструментів для створення надійного клієнт-серверного застосунку з розширеним функціоналом.

Java є однією з найпоширеніших і найпотужніших мов програмування, яка забезпечує кросплатформенність. Це дозволяє запускати застосунок на різних операційних системах, включаючи Windows, Linux та macOS. Завдяки своїй об'єктно-орієнтованій природі, Java підтримує використання патернів проєктування, таких як strategy, command, abstract factory, та interpreter, що є важливим для структурування та організації коду проєкту. Однією з ключових переваг Java є розвинена екосистема бібліотек. Java також забезпечує високу безпеку завдяки сильній типізації та управлінню пам’яттю. Це дозволяє уникнути поширених помилок, пов’язаних із роботою з пам’яттю, і гарантує стабільність роботи застосунку. У контексті проєкту PowerShell Terminal це особливо важливо, оскільки виконання команд і маніпуляції з даними мають бути надійними та передбачуваними. Для серверної частини застосунку використовується Spring Boot,

що спрощує створення REST API і забезпечує зручну інтеграцію з базою даних PostgreSQL. Для клієнтської частини обрано JavaFX, який дозволяє створювати сучасний інтерфейс користувача з можливістю налаштувань кольору і розмірів вікна. Завдяки наявності бібліотек для роботи з потоками, Java полегшує реалізацію багатозадачності, що необхідна для підтримки роботи в кількох вкладках терміналу. Ще однією важливою перевагою вибору Java та IntelliJ IDEA є велика кількість документації, навчальних матеріалів та активна спільнота розробників, що дозволяє швидко вирішувати будь-які питання, які виникають під час розробки. Як для новачків, так і для досвідчених розробників доступні численні ресурси для навчання та вдосконалення знань.

Середовище розробки IntelliJ IDEA було обрано через його потужний функціонал, який робить розробку на Java максимально продуктивною. Це середовище забезпечує вбудовану підтримку Java, Spring Boot і JavaFX, що значно спрощує реалізацію проєкту. Інструменти автодоповнення, рефакторингу та автоматичного форматування дозволяють зосередитися на розв'язанні ключових задач проєкту, мінімізуючи час на вирішення технічних проблем. IntelliJ IDEA також інтегрується з системами контролю версій, такими як Git, що дозволяє ефективно працювати над проєктом. Завдяки вбудованим інструментам для роботи з базами даних можна безпосередньо виконувати запити, переглядати структуру бази PostgreSQL і налагоджувати взаємодію сервера з базою.

Окрім цього, IntelliJ IDEA забезпечує зручний інструментарій для тестування. Завдяки підтримці JUnit, тести можна запускати безпосередньо з середовища, що спрощує перевірку коректності виконання команд і змін налаштувань терміналу. Інструменти для створення шаблонів коду та генерації типових конструкцій сприяють швидкому впровадженню патернів проєктування.

Таким чином, вибір Java як основної мови програмування обґрунтований її універсальністю, стабільністю та широкою екосистемою. Середовище розробки IntelliJ IDEA, у свою чергу, забезпечує комфортні умови для ефективної розробки, налагодження та тестування застосунку, роблячи його оптимальним рішенням для реалізації проєкту PowerShell Terminal.

1.8 Проєктування розгортання системи

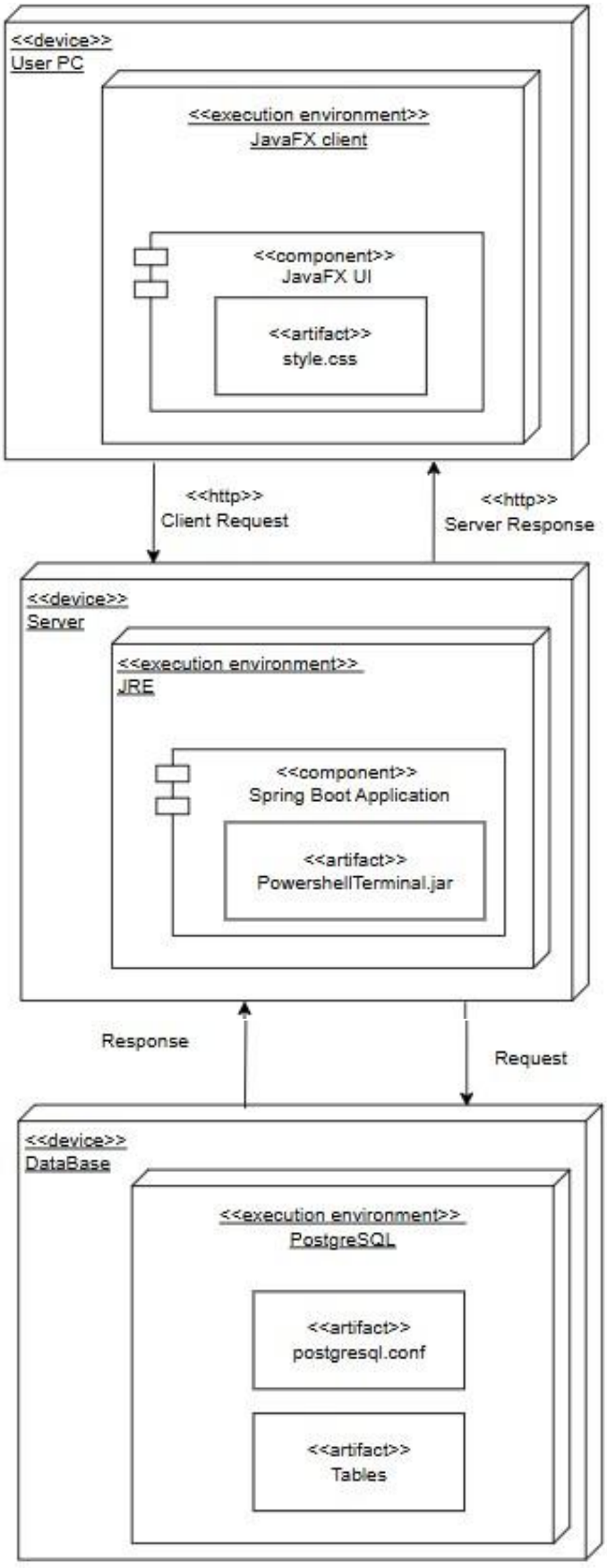

Рисунок 1.8.1 – Діаграма розгортання системи

Діаграма розгортання — це тип діаграми в UML, яка використовується для візуалізації фізичного розташування компонентів системи, їх розгортання на апаратних і програмних платформах, а також зв’язків між ними. Вона показує, як програмні модулі (компоненти, артефакти) взаємодіють з апаратними пристроями або середовищами виконання.

Опис діаграми розгортання:

1. User PC (<<device>>):

- Це пристрій користувача, з якого здійснюється доступ до PowerShell терміналу.
- JavaFX Client (<<execution environment>>): середовище виконання JavaFX на клієнтському пристрої, яке відповідає за запуск інтерфейсу користувача.
- JavaFX UI (<<component>>): це компонент інтерфейсу користувача, який представляє графічний інтерфейс терміналу. Він надає користувачеві можливість взаємодіяти з терміналом — вводити команди, змінювати налаштування (кольори, розмір, фон) і переглядати результати виконання команд.
- style.css (<<artifact>>): артефакт, що представляє файл стилів CSS, використовується для налаштування зовнішнього вигляду інтерфейсу терміналу, зокрема стилів для синтаксичних конструкцій, фону, кольору тексту тощо.

2. Server (<<device>>):

- Це основний сервер, де виконуються всі серверні обчислення, обробляються команди терміналу і надсилаються налаштування.
- JRE (<<execution environment>>): Java Runtime Environment (JRE), необхідне для запуску застосунків на Java. Це середовище виконання, яке підтримує роботу Spring Boot на сервері, що забезпечує обробку запитів і виконання команд.
- Spring Boot Application (<<component>>): основний компонент, який обробляє запити клієнта. У рамках цього компонента відбувається обробка команд, надісланих із клієнта, виконується серверна логіка та комунікація з базою даних.

- PowershellTerminal.jar (<<artifact>>): зібраний артефакт застосунку у вигляді JAR-файлу, що містить весь код і залежності для запуску серверної частини терміналу. Цей файл розгорнутий на сервері й виконується в середовищі JRE.

3. DataBase (<<device>>):

- Це сервер бази даних, який зберігає всі дані, необхідні для роботи PowerShell терміналу.

- PostgreSQL (<<execution environment>>): середовище виконання для бази даних PostgreSQL. Відповідає за обробку запитів на читання та запис даних.

- postgresql.conf (<<artifact>>): конфігураційний файл PostgreSQL, який містить параметри налаштування для бази даних, такі як параметри підключення, обсяг пам'яті, що використовується, і налаштування для оптимізації роботи бази даних. • Tables (<<artifact>>): таблиці бази даних, у яких зберігаються дані про налаштування інтерфейсу тощо. Цей артефакт відображає сховище, де зберігаються всі необхідні дані.

4. Зв'язки:

- Client Request (<<http>>) та Server Response (<<http>>): ці зв’язки представляють клієнт-серверну комунікацію між User PC і сервером. Client Request: користувач із JavaFX UI відправляє запити на сервер (наприклад, для виконання команд або змін налаштувань). Server Response: сервер обробляє запит, виконує потрібні команди або оновлення налаштувань і повертає результат або підтвердження виконання клієнту. Використовується протокол HTTP або HTTPS, залежно від потреби у безпеці зв'язку.

- Request та Response між сервером і базою даних: Request: сервер надсилає запити до бази даних для отримання, збереження або оновлення даних (наприклад, зберігання налаштувань інтерфейсу). Response: база даних відповідає на запити сервера, повертаючи результати запитів або підтвердження успішного запису даних.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1 Структура бази даних

Структура бази даних показана на рисунку 2.1.1.

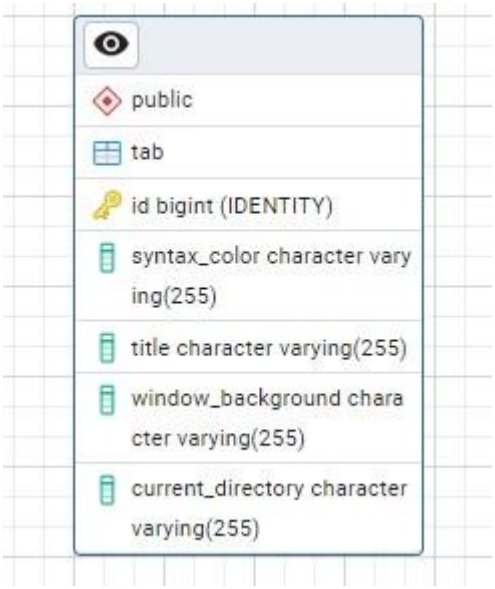

Рисунок 2.1.1 – Структура бази даних

Система використовує одну таблицю Tab, для того, щоб зберігати інформацію про стиль вкладки, щоб зберігати поточну директорію для виконання команд змінюючи їх та зберігати назву вкладки, це покращить інтерфейс користувача, адже кожна вкладка матиме свою назву. На рисунку 2.1.2 зображено її структуру.

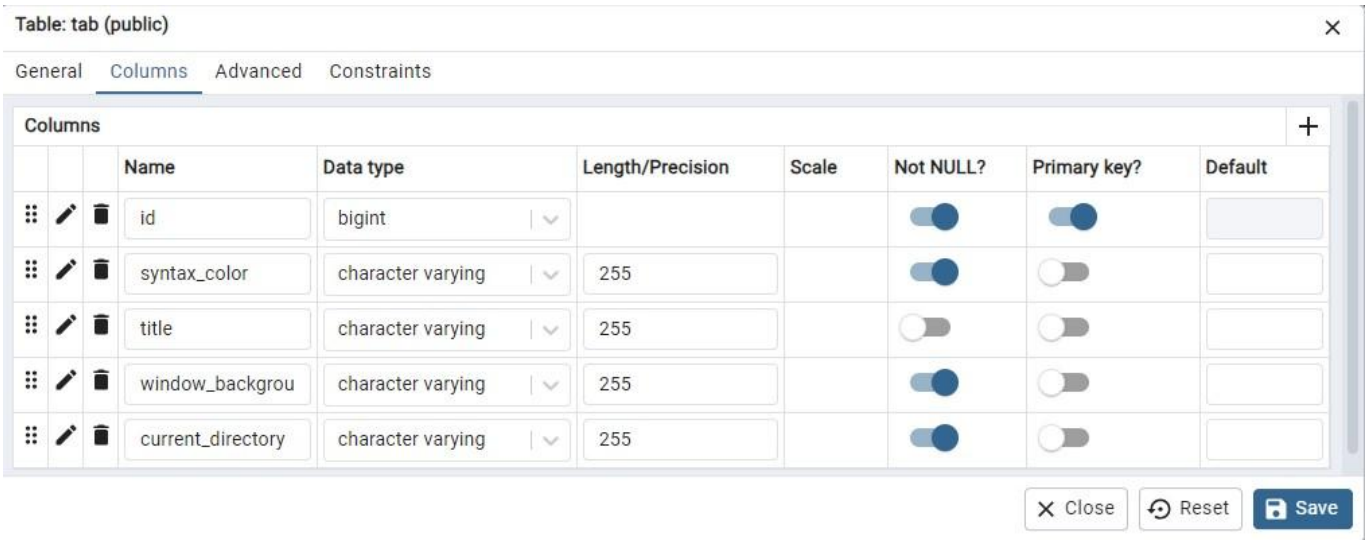

Рисунок 2.1.2 – Структура таблиці Tab

2.2 Архітектура системи

Архітектура системи зображена на рисунку 2.2.1.

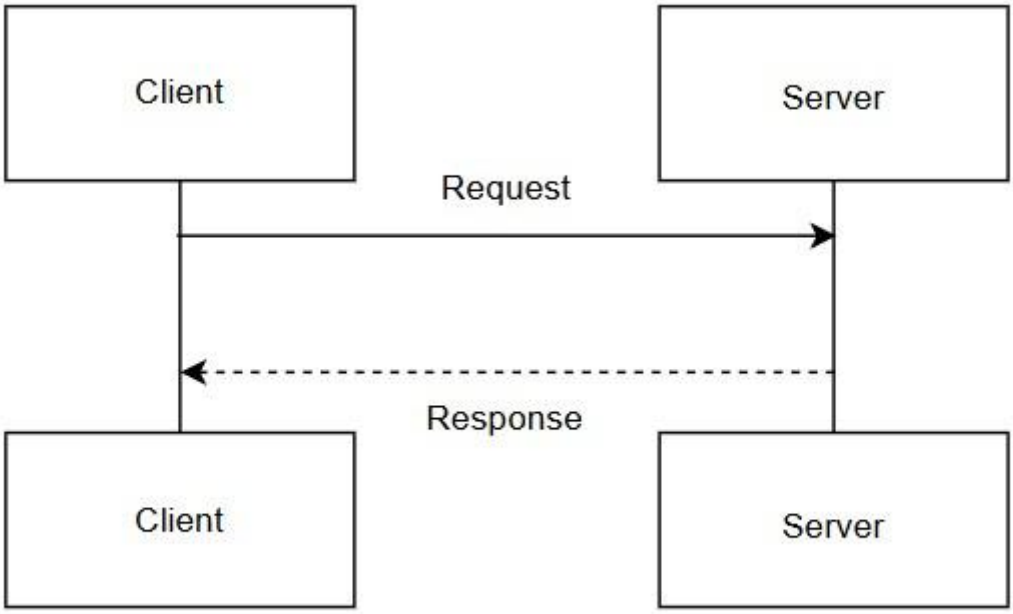

Рисунок 2.2.1 – Архітектура системи client-server

2.2.1 Специфікація системи

Загальна структура проєкту та діаграма класів зображені на рисунку 2.2.1.1 – 2.2.1.2.

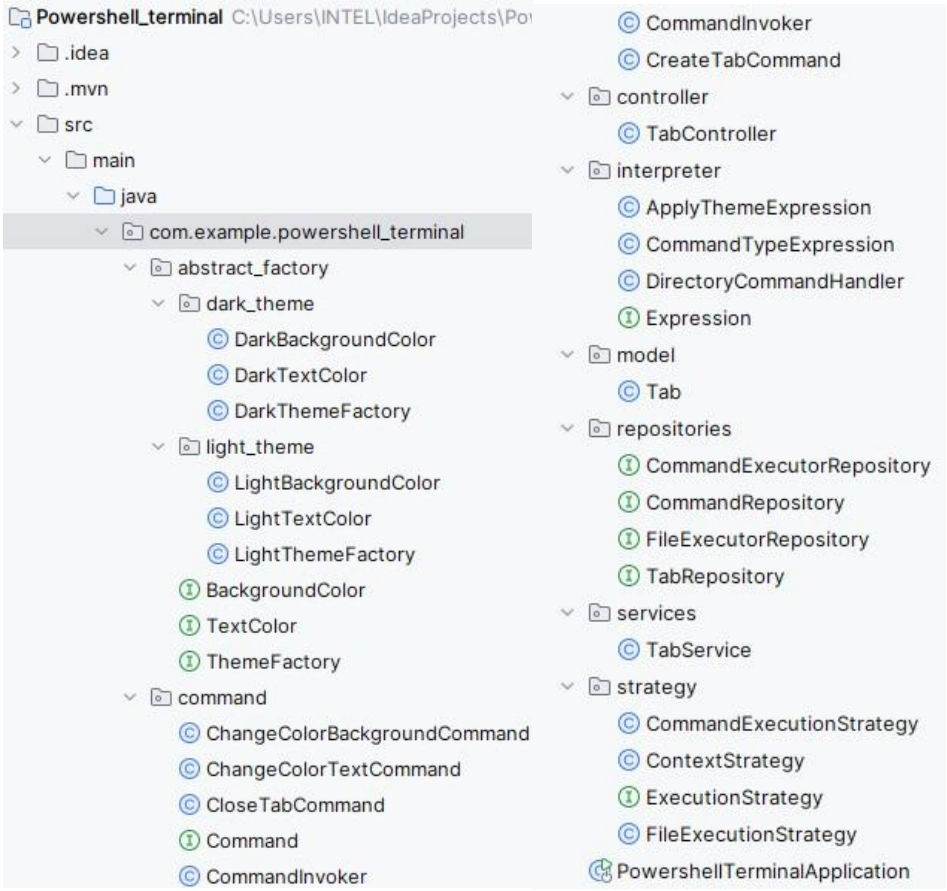

Рисунок 2.2.1.1 – Структура проєкту

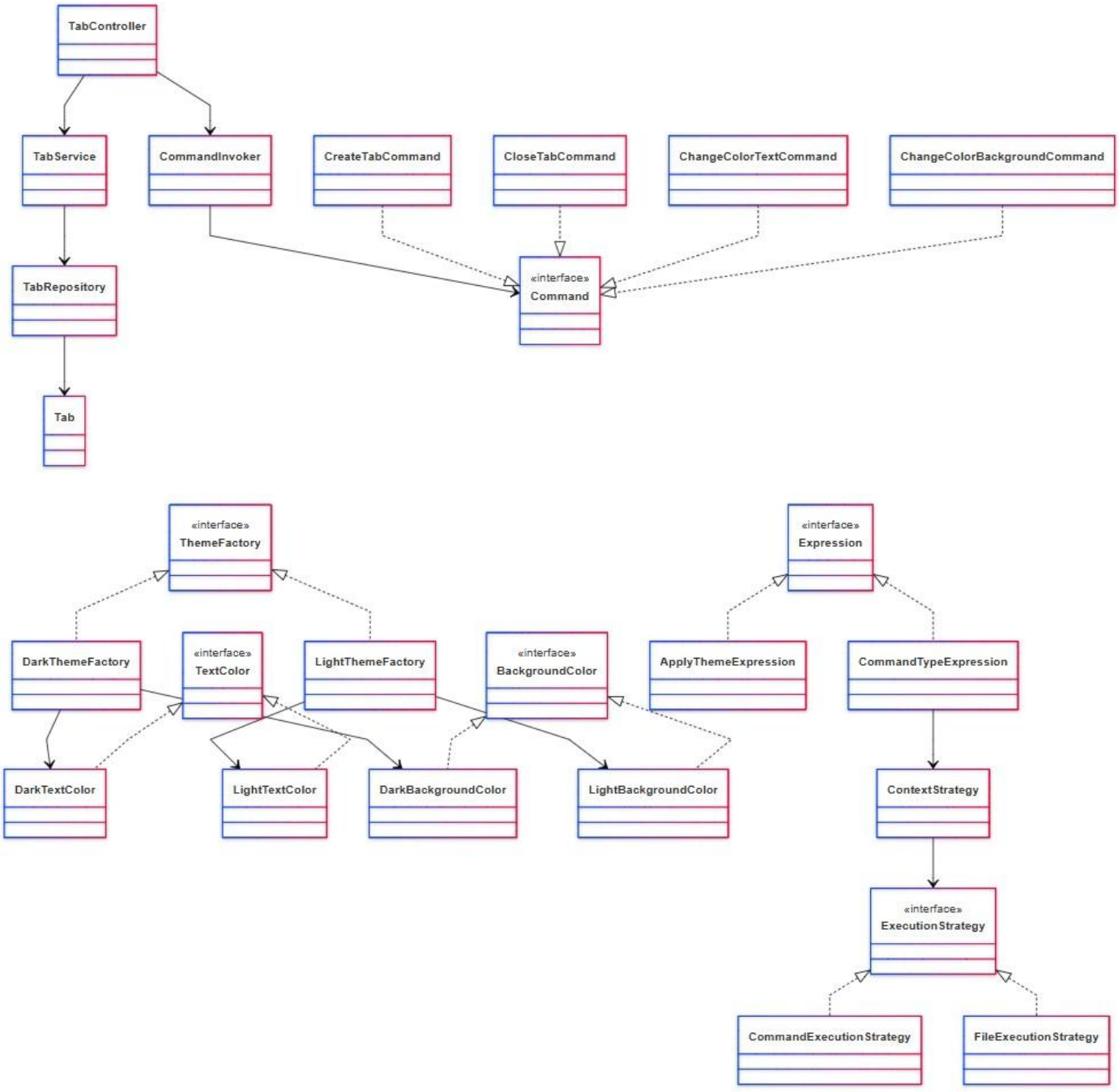

Рисунок 2.2.1.2 – Діаграма класів

2.2.2 Вибір та обґрунтування патернів реалізації

Виконання Powershell-команд та виконуваних файлів залежно від стратегії (шаблон “Strategy”).

Стратегія — це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна заміняти один на інший прямо під час виконання програми.

Я обрав для реалізації шаблон Strategy. Реалізація цього шаблону полягатиме у створенні різних стратегій виконання команд у терміналі залежно від їх типу. Наприклад, якщо команда є PowerShell скриптом, вона вимагає спеціальної обробки для правильного виконання в середовищі PowerShell. Інший випадок — виконання зовнішніх виконуваних файлів. Цей сценарій потребує іншого підходу. Таким чином, моє завдання полягає у розробці двох стратегій: CommandExecutionStrategy для виконання PowerShell-команд і FileExecutionStrategy для запуску виконуваних файлів. Це дозволить забезпечити розширюваність і гнучкість системи, а також спростить додавання нових типів виконання в майбутньому. На рисунку 2.2.2.1 наведено основні класи для реалізації шаблону Strategy.

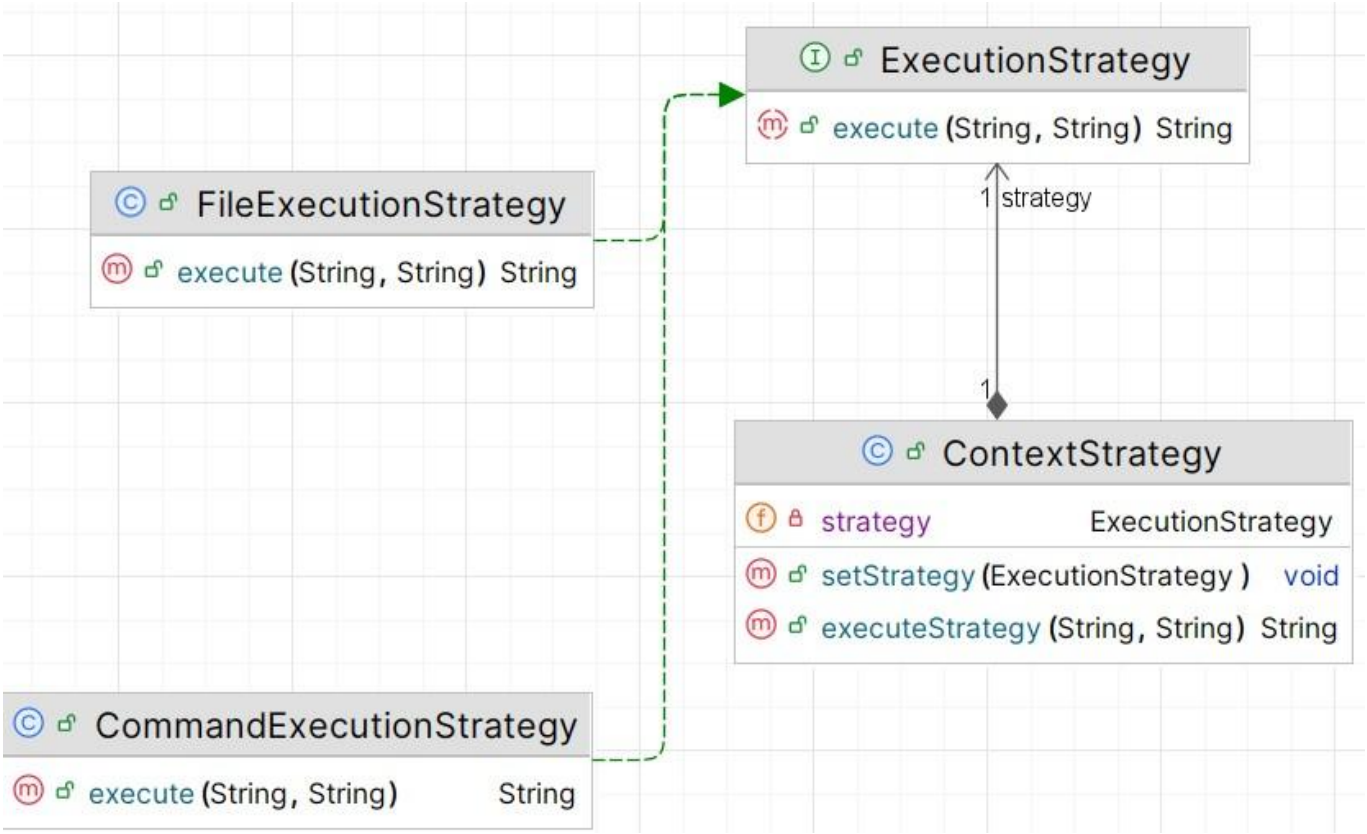

Рисунок 2.2.2.1 – Діаграма класів реалізації шаблону Strategy

Інтерфейс ExecutionStrategy визначає контракт для виконання команд або файлів через метод execute(String, String): String, де перший параметр — команда, а другий — директорія, в якій він виконується. Клас FileExecutionStrategy реалізує інтерфейс для виконання файлів, а CommandExecutionStrategy — для виконання PowerShell команд. Контекстний клас ContextStrategy зберігає поточну стратегію в полі strategy і дозволяє змінювати її через метод setStrategy(ExecutionStrategy strategy): void. Метод executeStrategy(String, String): String делегує виконання поточній стратегії. Ця структура дозволяє динамічно змінювати стратегії залежно від типу виконання та легко розширювати функціонал, додаючи нові стратегії.

Інкапсулюція різних дій, у вигляді окремих об’єктів команд (шаблон “Command”).

Команда — це поведінковий патерн проектування, який перетворює запити на об’єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

Я обрав реалізувати шаблон Command. Реалізація цього шаблону дозволить інкапсулювати різні дії, у вигляді окремих об'єктів-команд. Це спрощує додавання нових операцій, знижує залежність між компонентами та полегшує впровадження додаткових функціональностей. У межах лабораторної роботи я реалізував наступні команди: створення вкладки, закриття вкладки, зміна розміру тексту, зміна кольору тексту та зміна фону. Нижче на рисунку 2.2.2.2 наведено ключові класи, що використовуються у реалізації шаблону Command.

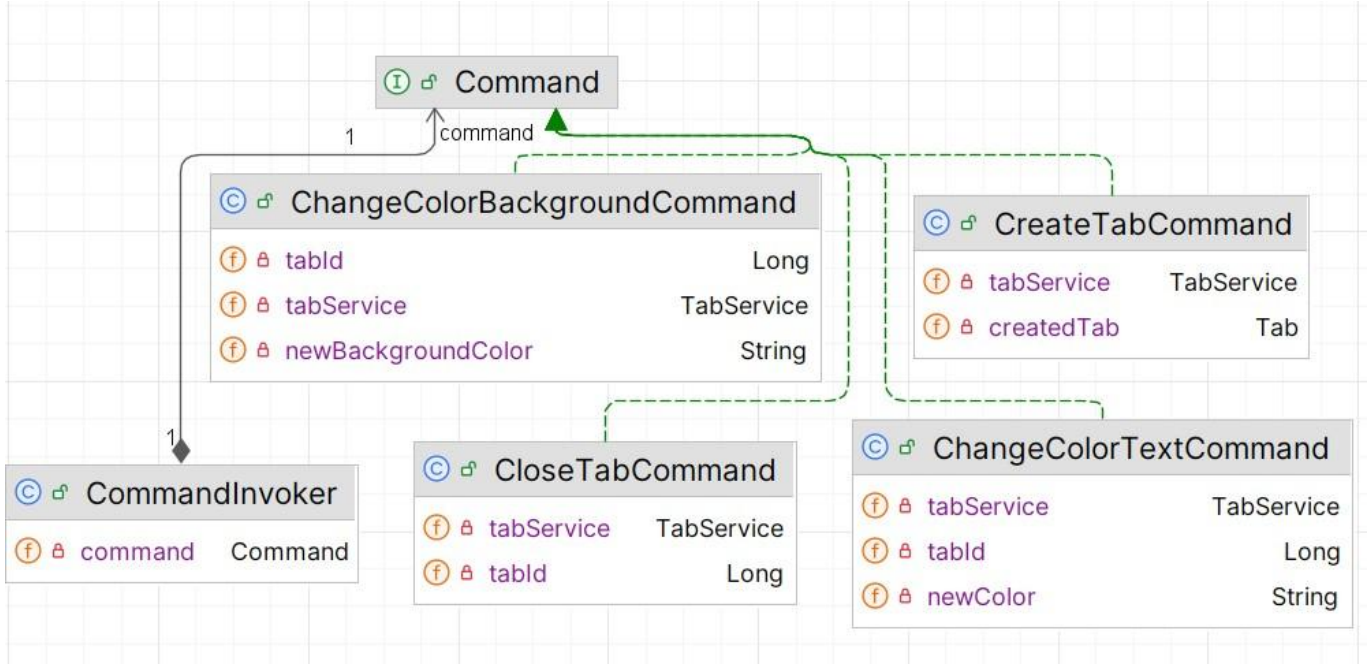

Рисунок 2.2.2.2 – Діаграма класів реалізації шаблону Command

Інтерфейс Command є основою шаблону, що визначає метод execute(). Цей метод реалізується всіма командами для виконання операцій. Команда CreateTabCommand відповідає за створення нової вкладки у терміналі. Вона містить всю необхідну логіку для додавання вкладки з вказаним іменем. Використання такої команди дозволяє легко інтегрувати функціонал створення вкладок у загальну архітектуру програми, а також робить систему більш масштабованою та зручною для тестування. CloseTabCommand - команда для закриття вкладки у терміналі. Її реалізація включає видалення вкладки за унікальним ідентифікатором, забезпечуючи контроль над операціями закриття.

ChangeColorTextCommand команда, яка реалізує зміну кольору тексту у вибраній вкладці. Вона дозволяє задати новий колір, що застосовується до текстового контенту. Використання цієї команди робить систему більш гнучкою і дозволяє легко адаптувати функціонал. ChangeColorBackgroundCommand команда дозволяє змінювати фон вкладки, задаючи новий колір.

CommandInvoker клас виконує роль ініціатора виконання команд. Він відповідає за збереження посилання на команду та її виконання через метод executeCommand(). Цей механізм дозволяє інкапсулювати виклик команд у одному місці, забезпечуючи гнучкість і простоту інтеграції команд у систему.

Реалізація різних тем терміналу (шаблон “Abstract Factory”).

Абстрактна фабрика — шаблон проєктування, що забезпечує інкапсуляцію окремих фабрик під єдиною схемою, упускаючи їхню деталізацію. Належить до класу твірних шаблонів. В типових випадках застосування, клієнтський код створює конкретну реалізацію абстрактної фабрики, а потім використовує загальний універсальний інтерфейс фабрики, для створення екземплярів об'єктів, які є частиною схеми. Клієнтський код не знає (або не бере до уваги), які саме конкретно об'єкти він отримує від цих фабрик, оскільки він використовує універсальний інтерфейс для їхнього створення. Шаблон розмежовує деталі реалізації множини об'єктів від їхнього загального використання в коді, оскільки створення об'єкта здійснюється за допомогою методів, що забезпечуються інтерфейсом фабрики. Я обрав реалізувати шаблон Abstract Factory. Реалізація цього шаблону дозволяє створювати сімейства взаємопов’язаних об’єктів без вказівки їхніх конкретних класів. Це забезпечує узгодженість між об'єктами, спрощує модифікацію та заміну компонентів і робить архітектуру програми більш гнучкою. У межах лабораторної роботи я реалізував фабрику для створення елементів теми, таких як колір тексту та колір фону. На рисунку 2.2.2.3 наведено ключові класи, що використовуються у реалізації шаблону Abstract factory.

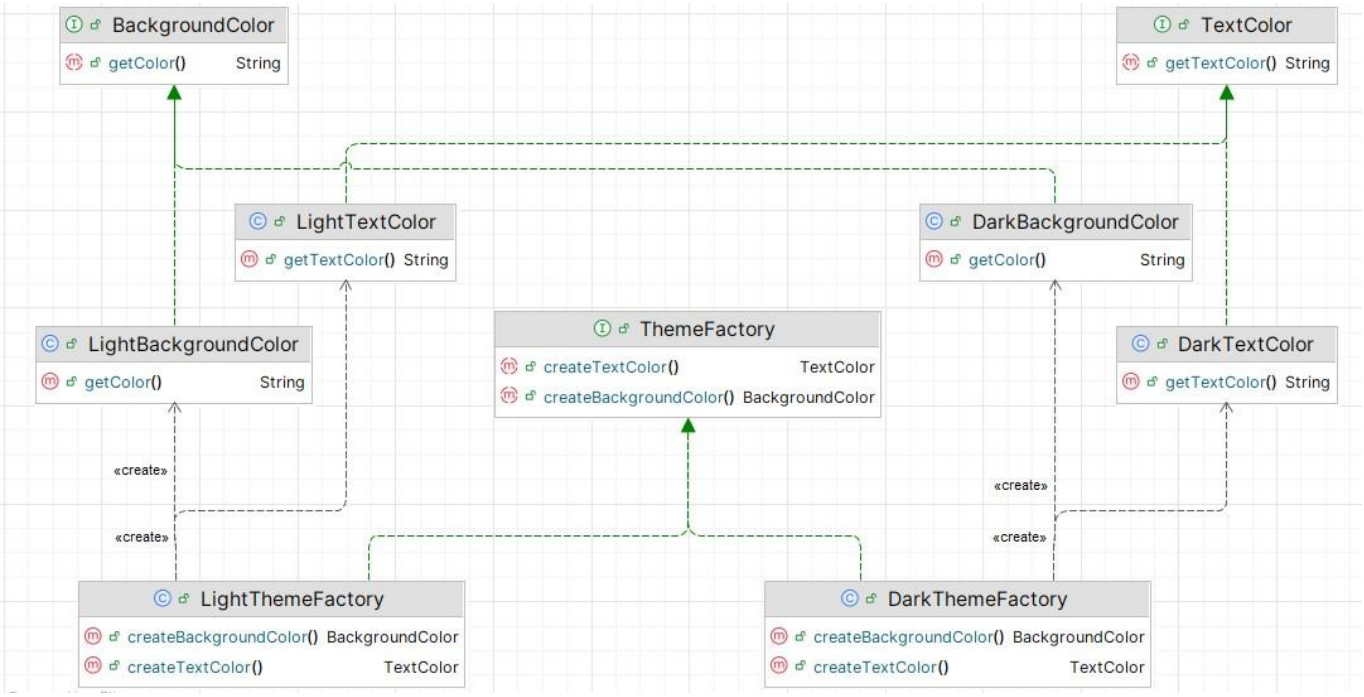

Рисунок 2.2.2.3 – Діаграма класів реалізації шаблону Abstract factory

Інтерфейс фабрики визначає набір методів для створення об’єктів, пов’язаних із темою. У цьому випадку він включає методи createTextColor(), createBackgroundColor() для створення відповідних компонентів теми.

Фабрика, яка реалізує методи інтерфейсу ThemeFactory для створення компонентів світлої теми. Вона генерує об’єкти, такі як чорний текст, білий фон. Ця реалізація забезпечує узгодженість компонентів світлої теми. Фабрика, яка створює компоненти темної теми. Вона включає об'єкти білого тексту, чорного фону.

Кожен інтерфейс визначає поведінку компонентів. TextColor – отримує колір тексту через метод getTextColor(). BackgroundColor – отримує колір фону через метод getColor(). Компоненти LightTextColor, LightBackgroundColor реалізують інтерфейси продуктів для світлої теми. Наприклад, LightTextColor задає чорний текст, а LightBackgroundColor – білий фон. Компоненти DarkTextColor, DarkBackgroundColor відповідають за темну тему. Вони реалізують білий текст, чорний фон, відповідно.

Реалізація визначення типу команди(кастомна, Powershell-команда, виконуваний файл чи команда зміни директорії). (Шаблон “Interpreter”). Інтерпретатор — шаблон проєктування, належить до класу шаблонів поведінки. Для заданої мови визначає представлення її граматики, а також інтерпретатор речень цієї мови. Шаблон Інтерпретатор слід використовувати, коли є мова для інтерпретації, речення котрої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще шаблон працює коли граматика проста та ефективність не є головним критерієм.

Я обрав патерн Interpreter для реалізації обробки команд, оскільки він дозволяє створити систему, що ефективно обробляє різноманітні команди PowerShell та інші спеціалізовані команди в терміналі. Патерн дозволяє інкапсулювати складні логічні вирази в об'єкти, що зменшує складність коду і дозволяє зручно розширювати систему новими типами команд. Завдяки цьому патерну можна ефективно реалізувати команду для виконання операцій з каталогами, для виконання файлів, а також для зміни тем у терміналі.

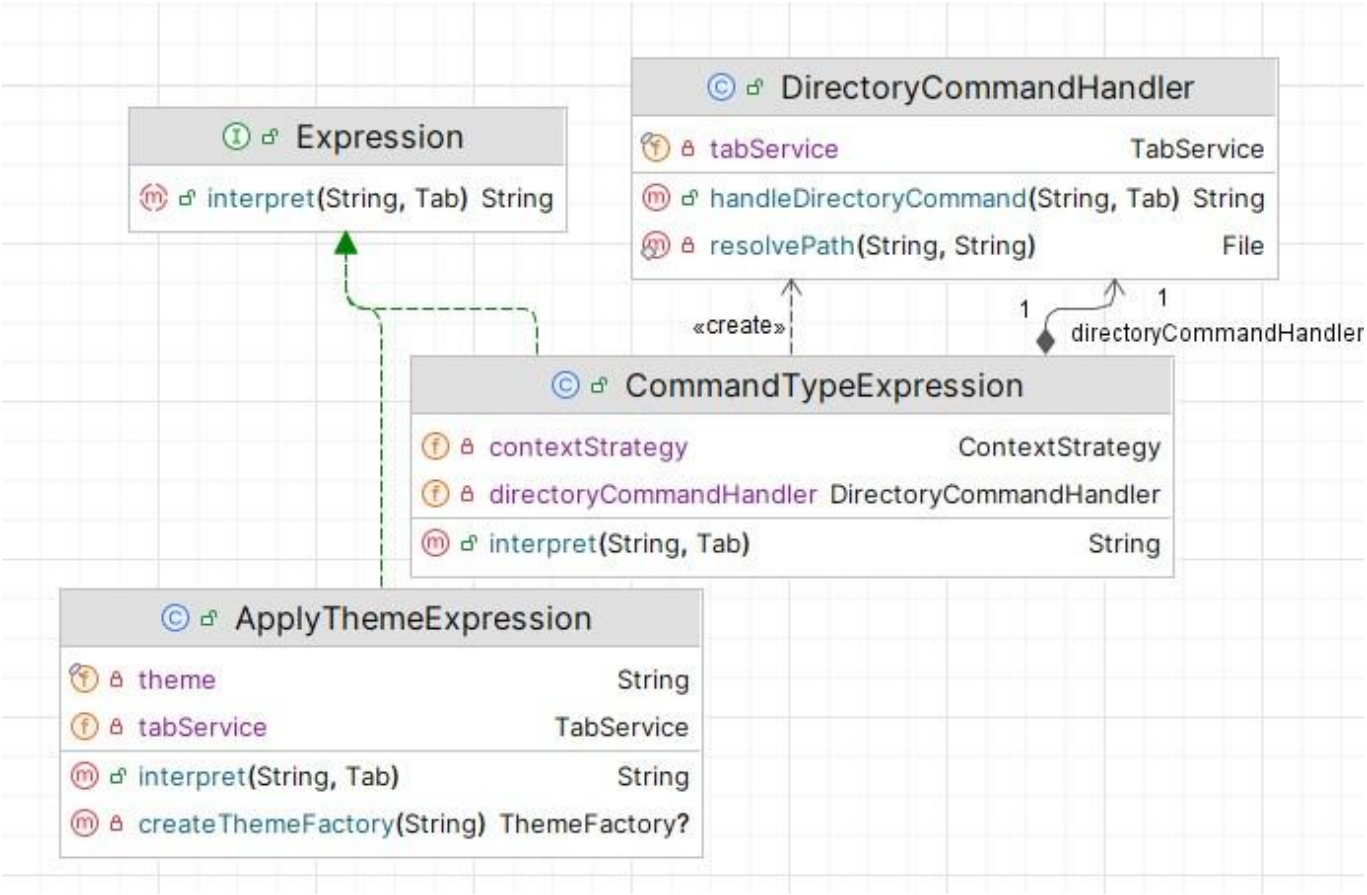

Рисунок 2.2.2.4 – Діаграма класів реалізації шаблону Interpreter

CommandTypeExpression — клас, що реалізує патерн Interpreter. Він відповідає за інтерпретацію команд, переданих в контексті. Залежно від типу команди (директорії чи виконуваний файл), цей клас делегує виконання відповідним стратегіям через патерн Strategy. Якщо команда стосується зміни директорії, вона обробляється за допомогою DirectoryCommandHandler, а якщо це команда для виконання файлу або PowerShell команди, використовуються стратегії для виконання команд.

DirectoryCommandHandler — клас, що обробляє команди, пов'язані з директоріями, такими як зміна директорії або перехід до нового диска. Він приймає команду та перевіряє, чи є директорія або диск доступними. Якщо так, він викликає метод зміни

директорії через TabService. Цей клас також забезпечує перевірку правильності команд і повідомляє користувача про помилки, якщо вони виникають.

ApplyThemeExpression — клас для обробки команд зміни теми терміналу. Якщо команда починається з "applyTheme", клас зчитує назву теми та застосовує її через відповідну фабрику тем (ThemeFactory). Він викликає методи TabService для зміни вигляду терміналу відповідно до вибраної теми (темна чи світла).

Expression — інтерфейс, що визначає метод interpret, який реалізується в класах, що інтерпретують різні типи команд. Це дозволяє зручно додавати нові типи команд і забезпечує гнучкість у зміні функціональності без зміни існуючого коду.

Створення системи з використанням архітектури “Client-server”.

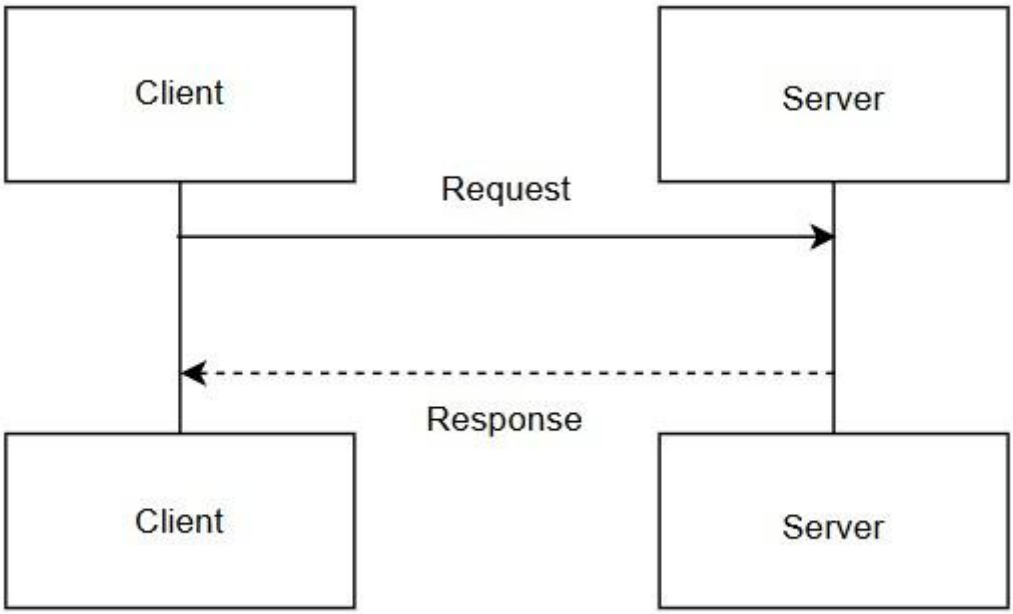

Рисунок 2.2.2.5 – Архітектура системи client-server

Для мого проєкту з терміналом PowerShell клієнт-серверна архітектура є дуже обґрунтованим вибором з кількох причин:

- Розподіл навантаження та підтримка масштабованості: Використання клієнтсерверної архітектури дозволяє серверу зосередитися на обробці запитів і виконанні команд, тоді як клієнт відповідає лише за взаємодію з користувачем (відображення вкладок, налаштування кольорів, введення команд). Це дає змогу ефективно обробляти великий обсяг запитів і масштабувати систему в разі потреби, наприклад, за допомогою додавання нових серверів для обробки команд.

- Логічний розподіл функцій: Клієнтський додаток займається лише інтерфейсом користувача (UI), що спрощує розробку та тестування. Це дозволяє зробити код клієнта легким для підтримки та модифікацій, оскільки більшість логіки виконання команд і керування вкладками обробляється на сервері. Сервер, в свою чергу, займається виконанням команди PowerShell та зберіганням даних про вкладки.
- Безпека і контроль доступу: Сервер може ефективно управляти доступом до виконання команд і ресурсів. Ви можете централізовано обробляти будь-які запити від клієнтів, що дозволяє контролювати, хто може виконувати які команди, а також верифікувати параметри запитів перед їх обробкою.

- Підтримка декількох клієнтів: Архітектура клієнт-сервер дозволяє підтримувати одночасну роботу кількох користувачів. Кожен клієнт взаємодіє з сервером через REST API, що дозволяє додавати нові функціональності без зміни клієнтського коду. - Покращення обслуговування та розвитку: Клієнт і сервер можуть розвиватися незалежно один від одного. Якщо потрібно змінити бізнес-логіку виконання команд, сервер можна оновити без потреби змінювати код клієнта. Це полегшує тестування та деплой нових версій програми.

- Розширення можливостей: Клієнт-серверна архітектура дає змогу з часом додавати нові можливості для серверної частини (наприклад, інтеграція з іншими сервісами для виконання команд або обробки даних). Це дозволяє терміналу адаптуватися до майбутніх вимог та нових технологій.

У моєму випадку, сервер обробляє виконання команд, зберігає стан вкладок і керує кольорами, а клієнт лише відображає цю інформацію, дозволяючи користувачам зручно взаємодіяти з терміналом. Така архітектура оптимальна для розробки складних і масштабованих систем, зокрема для мого терміналу PowerShell. На рисунку 2.2.2.5 зображено архітектуру client-server.

2.3 Інструкція користувача

Коли користувач відкриває застосунок він бачить термінал, де виводиться поточна директорія, є кнопка додавання вкладок, кнопки зміни кольору синтаксичних конструкцій та фону та поле введення де ми можемо писати наші команди. На рисунку 2.3.1 зображення даного терміналу.

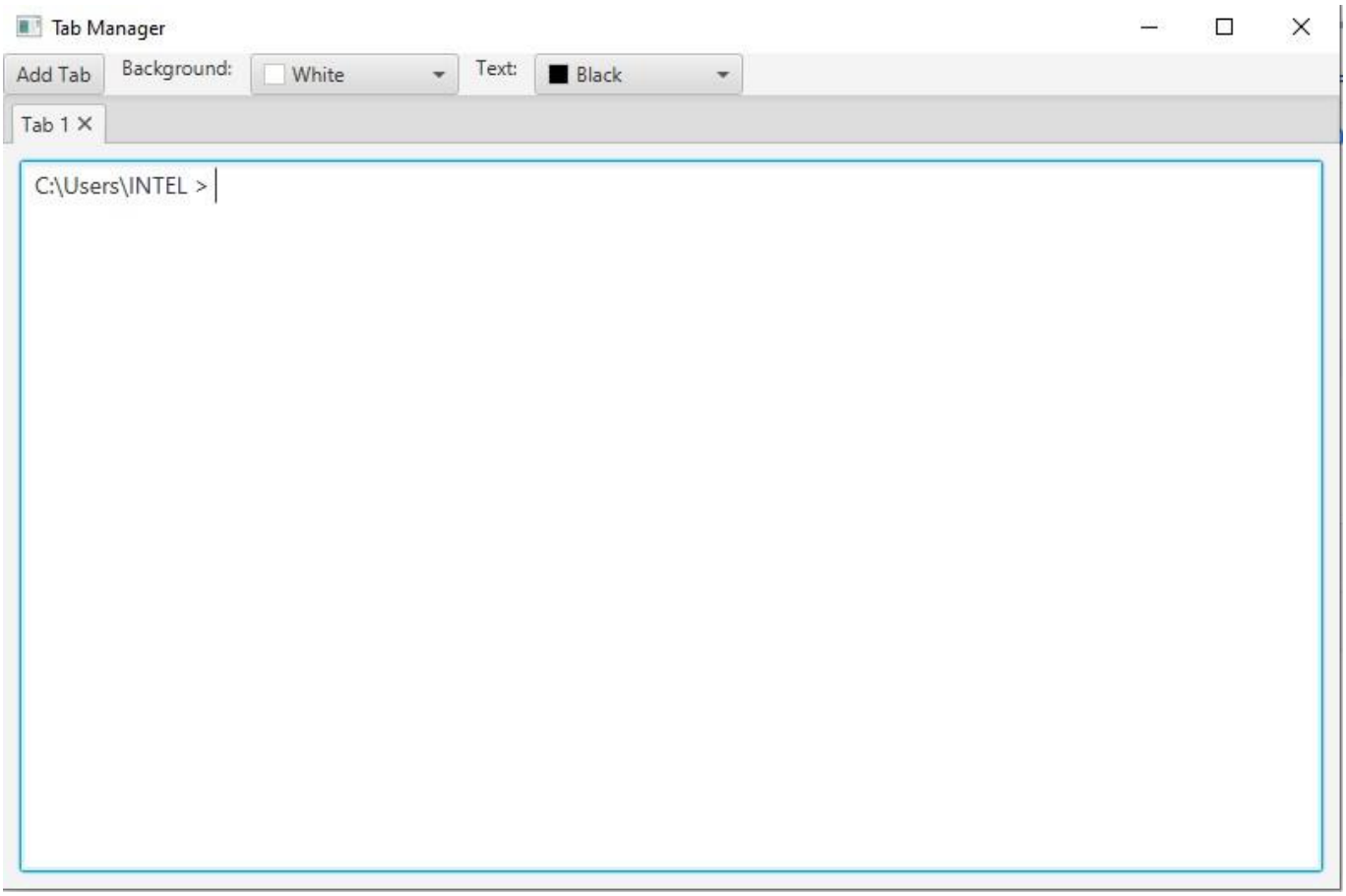

Рисунок 2.3.1 – Зображення терміналу при відкритті

За допомогою кнопки “Add Tab” можна додавати вкладки. На рисунку 2.3.2 зображено термінал з декількаами вкладками. На рисунку 2.3.3 зображено термінал після зміни фону та кольору тексту. На рисунку 2.3.4 – зображення терміналу після зміни теми кастомною командою та виведення результату команди.

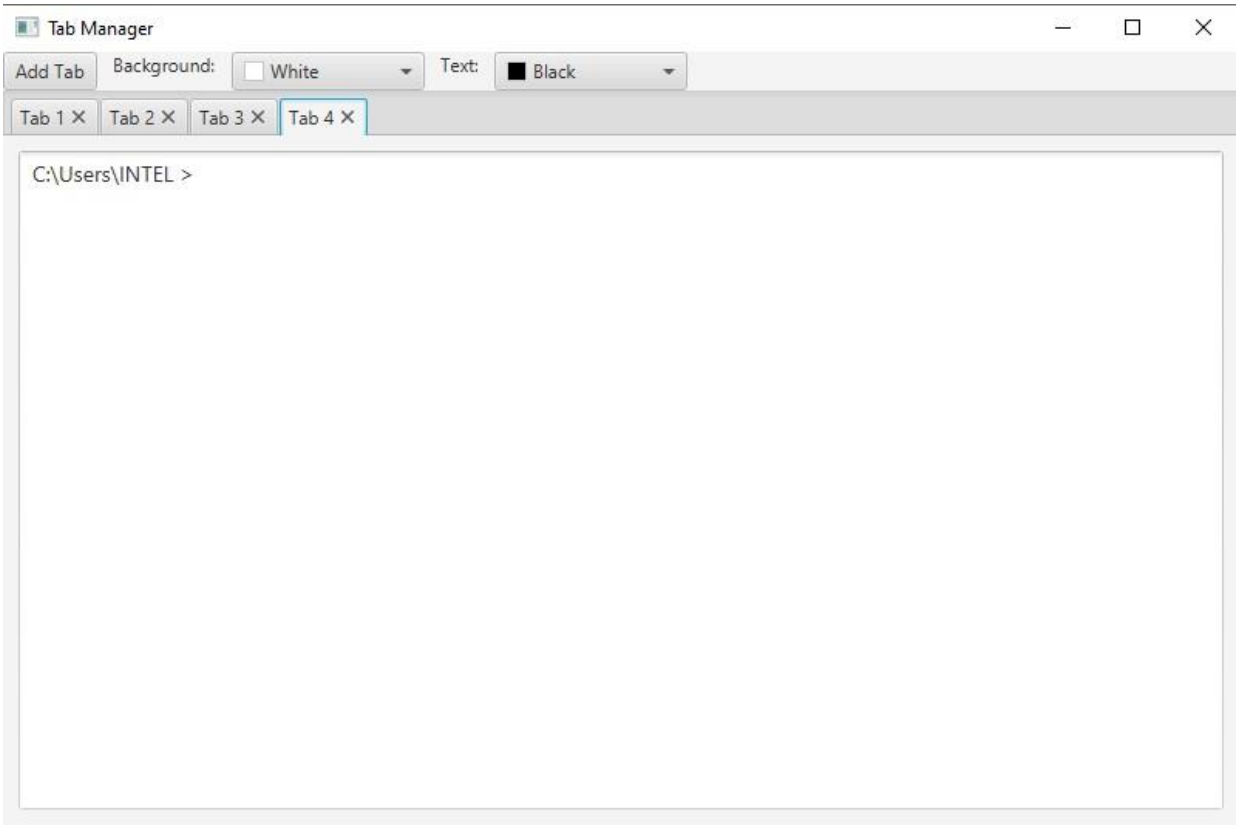

Рисунок 2.3.2 – Зображення терміналу з декількаами вкладками

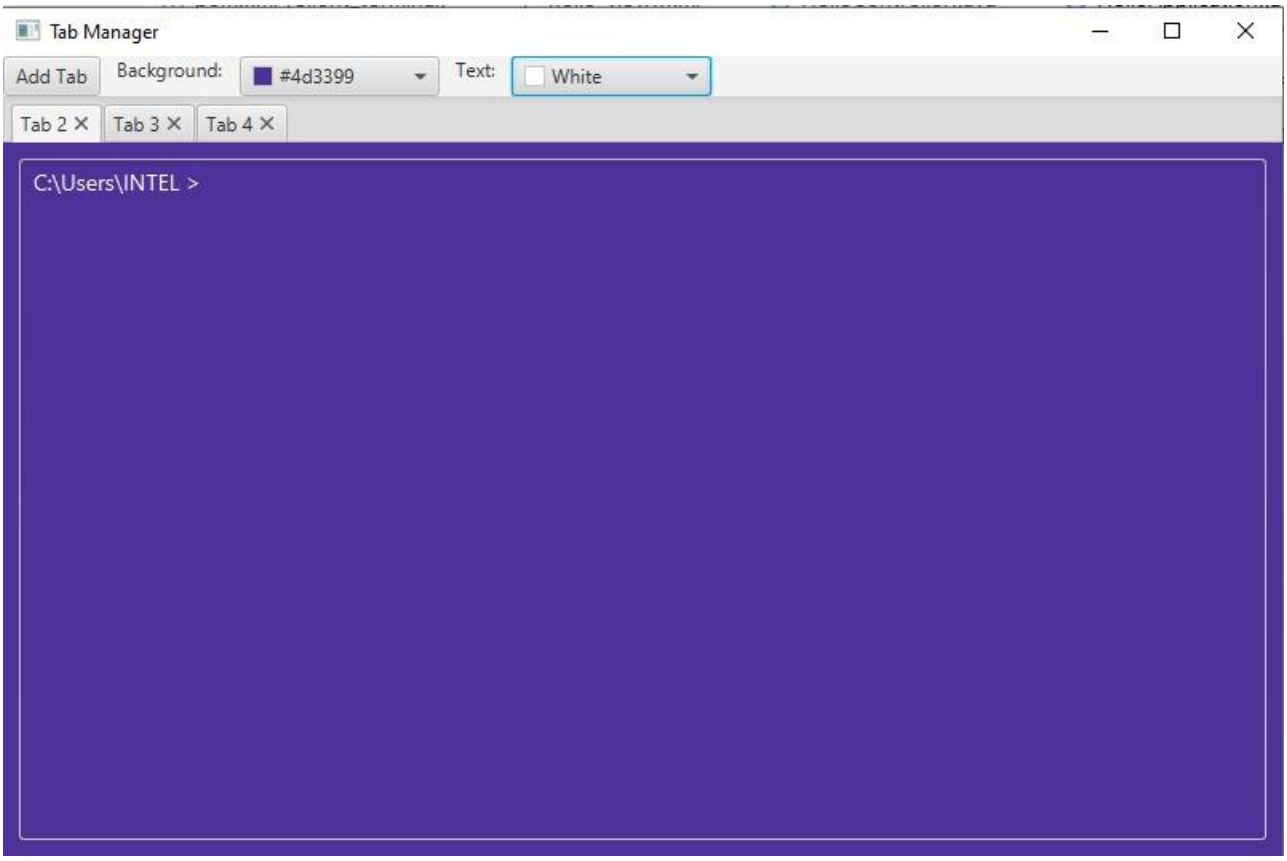

Рисунок 2.3.3 – Зображення терміналу після зміни фону та кольору тексту

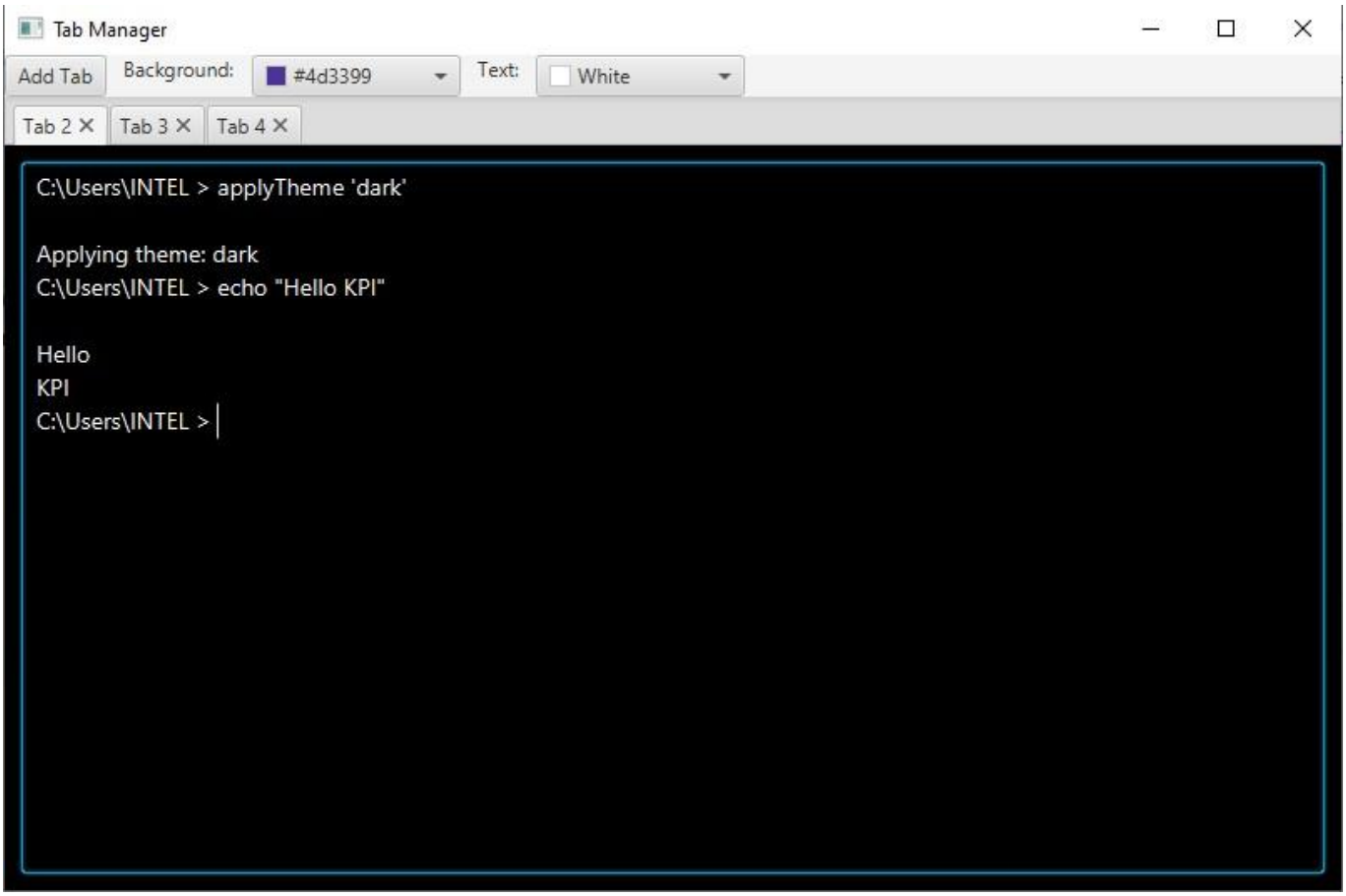

Рисунок 2.3.4 – Зображення терміналу після зміни теми кастомною командою та виведення результату команди

ВИСНОВКИ

У результаті виконання курсової роботи був успішно розроблений термінал для PowerShell з використанням патернів проектування, таких як Strategy, Command, Abstract Factory, Interpreter та Client-Server. Цей термінал має зручний інтерфейс, який дозволяє користувачам налаштовувати кольори синтаксичних конструкцій, змінювати розмір і фон вікна, а також працювати з кількома вкладками одночасно. Така функціональність значно покращує взаємодію з PowerShell, підвищуючи ефективність виконання адміністративних завдань та автоматизації процесів. Завдяки застосуванню патернів проектування, система отримала гнучкість і можливість для подальшого розширення. Патерн Strategy дозволив реалізувати різні стратегії виконання команд і скриптів, що забезпечило підтримку різних сценаріїв роботи з PowerShell. Патерн Command дозволив ефективно обробляти команди користувача, а Abstract Factory забезпечив можливість створення різних тем оформлення інтерфейсу з підтримкою різних стилів, а Interpreter допоміг при розробці парсера для обробки введених команд.

Модель Client-Server забезпечила архітектуру терміналу, в якій клієнтська частина взаємодіє з серверною через REST API, що дозволяє ефективно обробляти запити і виконувати команди на сервері.

Розроблений термінал дозволяє користувачам ефективно працювати з PowerShell у декількох вікнах одночасно, що підвищує продуктивність і зручність роботи з великими обсягами команд та скриптів. Застосування патернів проектування допомогло створити модульну та масштабовану систему, яка може бути адаптована до нових вимог і розширень.

У підсумку, виконана курсова робота продемонструвала важливість використання шаблонів проектування для створення гнучких та ефективних програмних рішень, які задовольняють потреби користувачів та покращують взаємодію з програмними інструментами, такими як PowerShell.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Стратегія. *Refactoring and Design Patterns*. URL: https://refactoring.guru/uk/designpatterns/strategy.
2. Команда. *Refactoring and Design Patterns*. URL: https://refactoring.guru/uk/designpatterns/command.
3. Учасники проектів Вікімедіа. Абстрактна фабрика – Вікіпедія. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Абстрактна_фабрика.
4. Учасники проектів Вікімедіа. Інтерпретатор (шаблон проєктування) – Вікіпедія. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Інтерпретатор_(шаблон_проєктування).
5. REST with Spring Tutorial. https://www.baeldung.com/rest-with-spring-series.
6. PowerShell Documentation. https://learn.microsoft.com/en-us/powershell/
7. UML 2.0 in a Nutshell: A Desktop Quick Reference. https://www.oreilly.com/library/view/uml-20-in/0596007957/.
8. Spring Framework Documentation. https://spring.io/docs
9. IntelliJ IDEA Documentation. https://www.jetbrains.com/idea/documentation/.
10. PowerShell Commands. https://www.pdq.com/powershell/
11. Design Patterns: Elements of Reusable Object-Oriented Software. https://www.pearson.com/us/higher education/program/Gamma-Design-PatternsElements-of-Reusable-Object OrienteSoftware/PGM310633.html.

ДОДАТКИ

ДОДАТОК А

Програмний

код Клієнт:

```
public class TerminalApplication extends Application {

    private final RestTemplate restTemplate = new RestTemplate();
    private final String serverUrl = "http://localhost:8080/tabs";

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

    @Override
    public void start(Stage primaryStage) {
        TabPane tabPane = new TabPane();
        tabPane.setTabClosingPolicy(TabPane.TabClosingPolicy.ALL_TABS);

        ColorPicker backgroundColorPicker = new ColorPicker(Color.WHITE);
        ColorPicker textColorPicker = new ColorPicker(Color.BLACK);

        backgroundColorPicker.setOnAction( ActionEvent event -> {
            Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
            if (selectedTab != null) {
                changeBackgroundColor(selectedTab, backgroundColorPicker.getValue());
            }
        });
```

```
        textColorPicker.setOnAction( ActionEvent event -> {
            Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
            if (selectedTab != null) {
                changeTextColor(selectedTab, textColorPicker.getValue());
            }
        });

        // Buttons panel
        HBox controls = new HBox( v: 10);
        Button createTabButton = new Button( s: "Add Tab");
        controls.getChildren().addAll(createTabButton, new Label( s: "Background:"), backgroundColorPicker, new Label(

        createTabButton.setOnAction( ActionEvent e -> createTab(tabPane));

        tabPane.getSelectionModel().selectedItemProperty().addListener(( ObservableValue<extends Tab> obs, Tab oldTab,
            if (newTab != null) {
                Map<String, String> tabStyles = fetchTabStyles(Long.parseLong(newTab.getId()));
                backgroundColorPicker.setValue(Color.valueOf(tabStyles.get("background")));
                textColorPicker.setValue(Color.valueOf(tabStyles.get("textColor")));
            }
        });
```

```
        BorderPane root = new BorderPane();
        root.setTop(controls);
        root.setCenter(tabPane);

        Scene scene = new Scene(root, v: 800, v1: 500);
        primaryStage.setTitle("Tab Manager");
        primaryStage.setScene(scene);
        primaryStage.show();

        createTab(tabPane);
    }

    private void createTab(TabPane tabPane) {
        Map<String, Object> createdTab = restTemplate.postForObject( url: serverUrl + "/create", request: null, Map.class);
        if (createdTab != null) {
            Tab tab = new Tab((String) createdTab.get("title"));
            tab.setId(String.valueOf(createdTab.get("id"))); // Set ID to identify the tab on the server

            VBox tabContent = new VBox( v: 10);
            tabContent.setPadding(new Insets( v: 10));
            tabContent.setFillWidth(true);

            String currentDirectory = (String) restTemplate.getForObject( url: serverUrl + "/get-directory?tabId=" + tab.getId(), S
            TextArea commandInputOutput = new TextArea(currentDirectory);


            // Listen for changes and ensure the user cannot modify protected text
            commandInputOutput.textProperty().addListener(( ObservableValue<extends String> obs, String oldText, String newText) -> {
                if (newText.length() < editableStart[0]) {
                    commandInputOutput.setText(oldText); // Revert to old text if protected area is modified
                } else if (!newText.startsWith(currentDirectory)) {
                    commandInputOutput.setText(currentDirectory); // Ensure text always starts with the current directory
                }
            });

            // Listen for key presses
            commandInputOutput.setOnKeyPressed( KeyEvent event -> {
                if (event.getCode() == KeyCode.ENTER) {
                    String[] lines = commandInputOutput.getText().split( regex: "\n");
                    String lastLine = lines[lines.length - 1];
                    if (lastLine.startsWith(currentDirectory)) {
                        String commandText = lastLine.substring(currentDirectory.length()).trim();
                        if (!commandText.isEmpty()) {
                            // Execute the command
                            Map request = Map.of( k1: "name", commandText, k2: "tabId", Long.parseLong(tab.getId()));
                            String result = (String) restTemplate.postForObject( url: serverUrl + "/execute", request, Map.class).ge

                            // Apply theme if command is 'applyTheme'
```

```
        });

        tabContent.getChildren().add(commandInputOutput);
        tab.setContent(tabContent);

        tabPane.getTabs().add(tab);
    }
}

private void changeBackgroundColor(Tab tab, Color color) {
    restTemplate.postForObject( url: serverUrl + "/change-background-color",
            Map.of( k1: "tabId", Long.parseLong(tab.getId()), k2: "newBackgroundColor", toHexString(color)), String.class);

    VBox tabContent = (VBox) tab.getContent();
    tabContent.setStyle("-fx-background-color: " + toHexString(color) + ";");

    tabContent.getChildren().forEach( Node node -> {
        if (node instanceof TextField || node instanceof TextArea) {
            String textColor = extractStyleValue(node.getStyle(), property: "-fx-text-fill", toHexString(Color.BLACK)); // Save
            node.setStyle(String.format("-fx-control-inner-background: %s; -fx-text-fill: %s; -fx-font-size: 14;",
                    toHexString(color), textColor));
        }
    });
}

private void changeTextColor(Tab tab, Color color) {
    restTemplate.postForObject( url: serverUrl + "/change-text-color",
            Map.of( k1: "tabId", Long.parseLong(tab.getId()), k2: "newColor", toHexString(color)), String.class);

    VBox tabContent = (VBox) tab.getContent();

    tabContent.getChildren().forEach( Node node -> {
        if (node instanceof Label || node instanceof TextField || node instanceof TextArea) {
            String backgroundColor = extractStyleValue(node.getStyle(), property: "-fx-control-inner-background", toHexString(Co
            node.setStyle(String.format("-fx-control-inner-background: %s; -fx-text-fill: %s; -fx-font-size: 14;",
                    backgroundColor, toHexString(color)));
        }
    });
}

private String extractStyleValue(String style, String property, String defaultValue) {
```

```
    private String extractStyleValue(String style, String property, String defaultValue) {
        if (style.contains(property)) {
            try {
                return style.split( regex: property + ":")[1].split( regex: ";")[0].trim();
            } catch (Exception e) {
                return defaultValue;
            }
        }
        return defaultValue;
    }

    private Map<String, String> fetchTabStyles(Long tabId) {
        return restTemplate.getForObject( url: serverUrl + "/get-styles?tabId=" + tabId, Map.class);
    }

    private String toHexString(Color color) {
        return String.format("#%02X%02X%02X",
                (int) (color.getRed() * 255),
                (int) (color.getGreen() * 255),
                (int) (color.getBlue() * 255));
    }
}
```

Сервер:

```
@RestController
@RequestMapping("/tabs")
public class TabController {

    private final TabService tabService;
    private final CommandInvoker commandInvoker;


    public TabController(TabService tabService) {
        this.tabService = tabService;
        this.commandInvoker = new CommandInvoker();

    }

    @PostMapping("/create")
    public ResponseEntity<Tab> createTab() {
        Command createTabCommand = new CreateTabCommand(tabService);
        commandInvoker.setCommand(createTabCommand);
        Tab createdTab = commandInvoker.executeAndReturn();
        return ResponseEntity.ok(createdTab);
    }

    @PostMapping("/close")
    public ResponseEntity<String> closeTab(@RequestBody Map<String, Long> request) {
```

```
    }

    @PostMapping("/change-text-color")
    public ResponseEntity<String> changeTextColor(@RequestBody Map<String, Object> request) {
        Long tabId = Long.valueOf(request.get("tabId").toString());
        String newColor = request.get("newColor").toString();
        Command changeColorCommand = new ChangeColorTextCommand(tabService, tabId, newColor);
        commandInvoker.setCommand(changeColorCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Text color changed to: " + newColor);
    }

    @PostMapping("/close-all")
    public ResponseEntity<String> closeAllTabs() {
        tabService.closeAllTabs();
        return ResponseEntity.ok( body: "All tabs closed.");
    }
```

```
@PostMapping("/change-background-color")
public ResponseEntity<String> changeBackgroundColor(@RequestBody Map<String, Object> request) {
    Long tabId = Long.valueOf(request.get("tabId").toString());
    String newBackgroundColor = request.get("newBackgroundColor").toString();
    Command changeBackgroundColorCommand = new ChangeColorBackgroundCommand(tabService, tabId, newBackgroundColo
    commandInvoker.setCommand(changeBackgroundColorCommand);
    commandInvoker.executeCommand();
    return ResponseEntity.ok( body: "Background color changed to: " + newBackgroundColor);
}

@GetMapping("/get-styles")
public ResponseEntity<Map<String, String>> getStyles(@RequestParam Long tabId) {
    Tab tab = tabService.findTabById(tabId);
    if (tab == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(Map.of(
            k1: "background", tab.getWindowBackground(),
            k2: "textColor", tab.getSyntaxColor()
    ));
}
```

```
    @GetMapping("/get-directory")
    public ResponseEntity<String> getCurrentDirectory(@RequestParam Long tabId) {
        Tab tab = tabService.findTabById(tabId);
        return ResponseEntity.ok(tab.getCurrentDirectory());
    }

    @PostMapping("/execute")
    public ResponseEntity<Map<String, String>> executeCommand(@RequestBody Map<String, String> request) {
        String commandText = request.get("name");
        Long tabId = Long.valueOf(request.get("tabId").toString());

        Tab tab = tabService.findTabById(tabId);
        if (tab == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(Map.of( k1: "output",  v1: "Tab not found"));
        }

        ApplyThemeExpression themeExpression = new ApplyThemeExpression(commandText, tabId, tabService);
        CommandTypeExpression typeExpression = new CommandTypeExpression(new ContextStrategy(), tabService);

        String themeResult = themeExpression.interpret(commandText, tab);
        if (!themeResult.equals("Unknown command")) {

        }

        ApplyThemeExpression themeExpression = new ApplyThemeExpression(commandText, tabId, tabService);
        CommandTypeExpression typeExpression = new CommandTypeExpression(new ContextStrategy(), tabService);

        String themeResult = themeExpression.interpret(commandText, tab);
        if (!themeResult.equals("Unknown command")) {
            return ResponseEntity.ok(Map.of( k1: "output", themeResult));
        }

        String result = typeExpression.interpret(commandText, tab);

        return ResponseEntity.ok(Map.of( k1: "output", result,  k2: "currentDirectory", tab.getCurrentDirectory()));
    }
}
```