Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: «Розроблення кросплатформного застосунку "Ієрархічна нотатна дошка"(Frontend).»

Виконав: студент 2 курсу, групи КНС-21
спеціальності 122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Танечник О. І.
(прізвище та ініціали)

Керівник: Крошний І. М.
(прізвище та ініціали)

Рецензент: Сторонсук О. М.
(прізвище та ініціали)

Львів – 2025

Національний лісотехнічний університет
України
(повне найменування вищого навчального закладу)

ННІ комп`ютерних наук та інформаційних технологій
Кафедра комп`ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп`ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
_________ Борецька І. Б.
"___10___" червня 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Танечнику Остапу Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення кросплатформного застосунку «Ієрархічна нотатна дошка» (Frontend)

керівник роботи *Крошний Ігор Миколайович, директор ННІ КНІТ, доцент кфедри КН, к.т.н., доцент*,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "15" листопада 2024 року № С-882

2. Термін подання студентом роботи 10 червня 2025 року

3. Вихідні дані до роботи Розробити клієнтську частину кросплатформного застосунку «Ієрархічна нотатна дошка», який даватиме змогу користувачам створювати цифрові полотна, додавати текстові блоки, зображення, файли, вкладені підполотна та графічні об`єкти. Система має забезпечити зручне додавання, редагування, видалення ієрархічно організованих елементів (текст, файли, зображення, фігури, вкладені полотна), а також збереження всієї структури у форматі JSON на диску

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
ВСТУП
РОЗДІЛ 1. Стан проблемної області
РОЗДІЛ 2. Інформаційне та математичне забезпечення
РОЗДІЛ 3. Програмне та технічне забезпечення
ВИСНОВКИ

5. Перелік графічного матеріалу (з точним зазначенням обов`язкових креслень)
Підготовка матеріалу до доповіді

6. Дата видачі завдання 18 листопада 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів проекту	Примітка
1.	Аналіз засобів побудови інтерфейсів для графічних редакторів на базі Qt/QML	18.11.24 – 08.12.24	*Виконано*
2.	Розроблення структури інтерфейсу та макетів основних візуальних компонентів	09.12.24 – 30.01.25	*Виконано*
3.	Реалізація робочої області (CanvasView) з підтримкою відображення об'єктів	31.01.25 – 09.02.25	*Виконано*
4.	Реалізація drag-and-drop, інлайн-редагування, resize та контекстного меню	10.02.25 – 19.03.25	*Виконано*
5.	Створення навігаційної панелі (ExplorerView) та системи вкладок для роботи з кількома полотнами	20.03.25 – 19.04.25	*Виконано*
6.	Інтеграція інтерфейсу з логікою C++ через CanvasManager і CanvasObjectModel	20.04.25 – 15.05.25	*Виконано*
7.	Тестування функціоналу, адаптація стилів, оформлення пояснювальної записки	16.05.25 – 08.06.25	*Виконано*

Студент **Танечник О. І.**
(підпис) (прізвище та ініціали)

Керівник роботи **Крощний І. М.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота містить 58 сторінок, 20 ілюстрацій, 5 додатків, 13 джерел.

Ця робота присвячена розробленню клієнтської (фронтенд) частини кросплатформного застосунку «Ієрархічна нотатна дошка». Застосунок забезпечує графічний інтерфейс для створення та взаємодії з цифровими полотнами, на яких можна розміщувати текстові блоки, зображення, файли, вкладені підполотна та графічні об’єкти (лінії, стрілки, фігури, вільне малювання). Інтерфейс реалізовано на основі фреймворку Qt із використанням мови QML. У роботі реалізовано підтримку вкладених полотен, drag-and-drop взаємодії, інлайн-редагування, контекстних меню, вкладок і реактивного інтерфейсу. Об’єкти синхронізуються з backend-логікою через клас CanvasManager. Інтерфейс адаптований до темної теми, а структура забезпечує гнучке масштабування та розширення.

Ключові слова: Qt, QML, фронтенд, ієрархічна нотатна дошка, CanvasView, ExplorerView, drag-and-drop, вкладені полотна, інтерфейс.

ABSTRACT

Bachelor thesis (project): explanatory note: 58 pages, 20 illustrations, 5 appendices, 13 sources.

This thesis is dedicated to the development of the client-side (frontend) of the cross-platform application "Hierarchical Note Board." The application provides a graphical interface for creating and interacting with digital canvases, where users can place text blocks, images, files, nested sub-canvases, and graphic objects (lines, arrows, shapes, free drawing). The interface is implemented using the Qt framework and QML language. The project features support for nested canvases, drag-and-drop interaction, inline editing, context menus, tabs, and a reactive user interface. All objects are synchronized with the backend logic via the CanvasManager class. The interface is adapted for a dark theme and supports flexible scaling and extensibility.

Keywords: Qt, QML, frontend, Hierarchical Note Board, CanvasView, ExplorerView, drag-and-drop, nested canvases, UI.

ТЕХНІЧНЕ ЗАВДАННЯ

У рамках дипломної роботи необхідно реалізувати програмне забезпечення у вигляді клієнтського модуля для кросплатформного застосунку «Ієрархічна нотатна дошка», призначеного для візуального відображення, взаємодії та управління графічними об'єктами на полотні з підтримкою вкладеності. Система повинна забезпечити інтуїтивно зрозумілий інтерфейс для створення, редагування, переміщення, масштабування та видалення об'єктів у графічному середовищі.

Розробка модуля включає реалізацію основних візуальних компонентів інтерфейсу: робочої області (CanvasView), яка відображає активне полотно з усіма об'єктами, що на ньому розміщені, навігаційної панелі (ExplorerView), що показує ієрархічну структуру об'єктів та дозволяє перемикатися між вкладеними полотнами та системи вкладок, що реалізує паралельну роботу з декількома полотнами.

Інтерфейс реалізовано за допомогою фреймворку Qt із використанням декларативної мови QML. Для кожного типу об'єкта передбачено візуальне представлення, інлайн-редагування тексту (TextInput), drag-and-drop переміщення, а також зміна розміру. Передбачено інтерактивну реакцію на події користувача (натискання, подвійне клацання, контекстне меню) та передачу команд до бекенду через клас CanvasManager.

Об'єкти на полотні зчитуються з моделі CanvasObjectModel, що відображає поточний стан активного полотна. Зміни, здійснені користувачем, автоматично передаються до логіки на C++ та синхронізуються з JSON-структурою.

Уся взаємодія з файлами, зображеннями, вкладеними полотнами та JSON-документами виконується через діалогові вікна (Popup, FileDialog), які також реалізовані у QML. Реалізація інтерфейсу передбачає масштабованість і можливість подальшого розширення функціональності (наприклад, додавання нових типів об'єктів або мобільної версії).

Проєкт реалізовано з використанням QML та Qt 6+, з інтеграцією C++ логіки через контекстні об'єкти та властивості. Увесь функціонал забезпечує повну інтерактивність, узгоджену структуру інтерфейсу та гнучкість при подальшому розвитку.

Зміст

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ 8
ВСТУП 9
РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 11
1.1 Проблема організації інформації 11
1.2 Аналіз засобів для створення нотаток 12
1.3 Концепція універсального нотатника 13
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 16
2.1 Проектування архітектури застосунку 16
2.2 Основні компоненти та їх взаємодія 18
2.3 JSON-модель даних 19
2.4 Особливості кросплатформності 21
2.5 Математичне та логічне забезпечення 23
2.6 Діаграма варіантів використання 24
РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ 27
3.1 Інструменти та середовище розробки 27
3.1.1 Фреймворк Qt та QML 27
3.1.2 Мова програмування C++ 28
3.1.3 Система контролю версій Git та GitHub 30
3.2 Опис програмної реалізації 31
3.2.1 Структура інтерфейсу та його основні компоненти 31
3.2.2 Панель навігації ExplorerView 33
3.2.3 Робоча область CanvasView 35
3.2.4 Діалогові вікна та FileDialog 38
3.2.5 Додаткові компоненти 39
ВИСНОВКИ 42
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 44
ДОДАТКИ 45
ДОДАТОК А 45

ДОДАТОК Б 46
ДОДАТОК В 47
ДОДАТОК Г 54
ДОДАТОК Ґ 56

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface – інтерфейс прикладного програмування.

JSON – JavaScript Object Notation – текстовий формат обміну даними між компонентами.

GUI – Graphical User Interface – графічний інтерфейс користувача.

UUID – Universally Unique Identifier – універсальний унікальний ідентифікатор об'єкта.

QML – Qt Modeling Language – декларативна мова опису інтерфейсу у Qt.

Qt – кросплатформний фреймворк для створення застосунків з графічним інтерфейсом.

JSON-файл – файл, що зберігає структуру даних у форматі JSON (для об'єктів на полотні).

Canvas – полотно, візуальна область для розміщення об'єктів (текстів, файлів, зображень тощо).

Backend – частина застосунку, яка відповідає за логіку, збереження та обробку даних (реалізована на C++).

Frontend – візуальна частина застосунку (реалізована на QML), з якою взаємодіє користувач.

Git – система контролю версій, що дозволяє відстежувати зміни у коді.

GitHub – платформа для зберігання та співпраці над Git-репозиторіями в хмарі.

QVariant – універсальний тип у Qt, який може зберігати значення будь-якого типу.

QDir, QFileInfo, QFile – класи Qt для роботи з файловою системою.

QAbstractListModel – базовий клас для моделей даних у Qt (використовується для CanvasObjectModel).

Q_PROPERTY – макрос у Qt, що дозволяє передавати властивості між C++ та QML.

Q_INVOKABLE – директива в Qt, яка дозволяє викликати методи C++ з QML.

CMake – система складання проектів, яка використовується для компіляції Qt-застосунку.

ВСТУП

У сучасному інформаційному середовищі значну роль відіграють інструментах, що забезпечують зручну візуалізацію, структурування та взаємодію з інформацією. Це актуально як для фахівців у сфері розробки програмного забезпечення, управління проєктами, дизайну, так і для освітніх закладів або звичайних користувачів, яким потрібно фіксувати ідеї, плани чи змістовні зв’язки у вигляді наочних нотаток.

Поширення підходів візуального представлення даних — таких як майндмепи, канбан-дошки, гнучке моделювання — стимулює розвиток інтерфейсів, здатних відображати складні ієрархії об’єктів: тексти, файли, зображення, вкладені блоки тощо. У центрі таких рішень перебуває саме інтерактивний інтерфейс, що дає змогу зручно керувати інформацією в межах візуального простору.

На практиці більшість популярних рішень реалізовані у вигляді веб-застосунків із хмарним зберіганням. Попри високу функціональність, такі системи мають низку обмежень: вони вимагають постійного інтернет-з’єднання, не дозволяють повністю контролювати збереження та структуру даних, обмежено підтримують вкладені елементи та можливітсь довільного розміщення їх на полотні.

Ці фактори суттєво впливають на гнучкість та ефективність роботи користувача, особливо в середовищах, де критично важливо швидко оперувати великою кількістю об'єктів, логічно структурувати їх у просторі та мати повний контроль над їх візуальним представленням і поведінкою.

Об’єктом дослідження є процес розробки інтерфейсної частини кросплатформного застосунку для роботи з ієрархічно організованими нотатками.

Метою дипломної роботи є створення фронтенд-частини застосунку «Ієрархічна нотатна дошка», яка забезпечує візуалізацію структури полотен, відображення об’єктів, взаємодію з користувачем, а також інтеграцію з внутрішньою логікою через модель CanvasObjectModel.

Предметом дослідження є програмні засоби, бібліотеки та технології, що забезпечують реалізацію інтерфейсу користувача: зокрема, мова QML, механізми реактивної прив’язки в Qt, обробка подій, підтримка вкладених структур, реалізація

елементів керування, контекстних меню, вкладок і інтерактивних сценаріїв.

До завдань, що вирішуються в рамках проєкту, належать:

- створення інтерфейсу з панеллю навігації, областю полотна та вкладками;
- візуалізація об'єктів на полотні;
- реалізація підтримки вкладених полотен та перемикання між ними;
- обробка подій вибору, переміщення, масштабування об'єктів;
- інтеграція з CanvasObjectModel для синхронізації стану з бекендом;
- підтримка вікон діалогів створення об'єктів, drag-and-drop.

Архітектура фронтенд-частини побудована з урахуванням принципів модульності, повторного використання компонентів, розширюваності, а також чіткої інтеграції з C++-частиною застосунку через модель даних.

Практична значущість роботи полягає у створенні універсального, адаптивного графічного інтерфейсу для локального настільного застосунку. Реалізований інтерфейс може бути використаний як основа для подальшого розвитку застосунку, включно з розширенням на мобільні або мережеві клієнти, а також для досліджень у сфері UX-дизайну та побудови інтерактивних середовищ роботи з інформацією.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Проблема організації інформації

Організація інформації — одна з базових задач людської діяльності, що актуальна в усі епохи. У XXI столітті, з переходом більшості робочих і освітніх процесів у цифрове середовище, ця проблема набула нових форм і масштабів. Ми зберігаємо плани, ідеї, документи, зображення, посилання, аудіо- та відеоматеріали в хмарах, локальних сховищах, месенджерах, електронній пошті тощо. Однак не менше половини цієї інформації губиться, забувається або не використовується через неструктурованість та відсутність зручного доступу.

Згідно з дослідженням IDC [1], до 2025 року обсяг глобальних цифрових даних перевищить 175 зетабайт (175 трильйонів гігабайт). Водночас, за оцінками McKinsey, працівники витрачають у середньому 19% робочого часу на пошук інформації або повторне створення вже існуючих даних [2]. Це свідчить не лише про надлишок інформації, але й про неефективність наявних підходів до її організації.

Популярні формати — списки, текстові документи, електронні таблиці — забезпечують базову структурування, однак не враховують людські когнітивні особливості: просторову пам’ять, асоціативне мислення, візуальну прив’язаність. Через це виникає розрив між тим, як ми природно мислимо (асоціативно, нелінійно), і тим, як ми змушені зберігати інформацію (лінійно, у вигляді списків або дерев).

Виникає потреба у візуально-гнучких інструментах, які дозволяють не просто «писати нотатки», а будувати просторову мапу думок, де кожна ідея, об’єкт чи файл має своє місце в координатному просторі. Такі інтерфейси не лише зручніші для сприйняття, а й краще відповідають принципам організації знань у довготривалій пам’яті людини.

Психологи виявили, що людина в середньому краще запам’ятовує об’єкти, якщо вони прив’язані до простору — навіть в уявному вигляді. Саме тому «ментальні карти» (mind maps) і діаграми значно ефективніші за списки при запам’ятовуванні й осмисленні складної інформації [3].

Тож головною проблемою сучасних застосунків для нотаток є не стільки

відсутність можливості зберігати інформацію, скільки недостатня підтримка візуальної, нелінійної, ієрархічної та асоціативної організації даних, яка краще відповідає природі людського мислення.

1.2 Аналіз засобів для створення нотаток

На ринку існує низка інструментів, що частково або повністю реалізують функціональність цифрових нотатників. Розглянемо найпопулярніші з них із погляду сильних і слабких сторін.

Серед сучасних засобів для створення цифрових нотаток особливою популярністю користується Notion. Цей застосунок пропонує гнучку систему блоків, що дозволяє працювати з текстом, списками, таблицями, чекбоксами, кодом, галереями тощо. Він підтримує вкладені сторінки та синхронізацію в реальному часі, що дає змогу спільно редагувати документи в команді. Водночас Notion не підтримує вільного просторового розміщення елементів, залежить від постійного підключення до Інтернету, а також має обмеження у візуальному масштабуванні складних структур на кшталт діаграм і ментальних мап.

Іншим поширеним рішенням є Microsoft OneNote, який зосереджується на рукописному вводі, малюванні та вільному розміщенні тексту та зображень. Завдяки глибокій інтеграції з екосистемою Microsoft, OneNote зручно використовувати в корпоративному середовищі. Проте інтерфейс не завжди інтуїтивний, а структура вкладених об’єктів іноді викликає труднощі у навігації. Крім того, OneNote не передбачає простого локального резервного копіювання.

Застосунок Miro орієнтований передусім на командну взаємодію. Він представляє собою візуальну дошку, на якій можна розміщувати нотатки, зображення, лінії, фігури та діаграми. Завдяки підтримці шаблонів і інтеграцій Miro зручно використовувати для проведення онлайн-воркшопів і мозкових штурмів. Проте застосунок працює виключно в режимі онлайн, що унеможливлює повноцінне використання у випадках відсутності підключення до мережі, а також ускладнює локальне персональне зберігання даних.

Нарешті, Obsidian є потужним інструментом для тих, хто працює з

текстовою інформацією у форматі markdown. Його основні переваги — локальне зберігання, система внутрішніх посилань, граф знань та розширювана екосистема плагінів. Obsidian ідеально підходить для особистих wiki-записів і тривалих дослідницьких проєктів. Водночас його функціональність обмежується текстовим контентом: вкладення, хоча й можливі, не мають повноцінного графічного представлення у просторі.

На прикладі існуючих програмних рішень можна виокремити низку функцій, які користувачі вважають найбільш цінними: гнучкість контенту (Notion), підтримка візуального представлення (Miro), ієрархічна організація (Obsidian), вільне позиціювання (OneNote), а також наочність структур (XMind/Freeplane). Проте кожен із цих застосунків або орієнтований лише на один з аспектів, або має певні функціональні обмеження, що заважають реалізувати повноцінну продуктивну роботу.

Таким чином, виникає потреба у створенні універсального інструменту, який би не дублював окремі рішення, а поєднував найкращі з них в одному інтуїтивному та зручному середовищі.

1.3 Концепція універсального нотатника

Очевидні технічні вимоги до сучасного програмного забезпечення — такі як кросплатформеність, стабільна робота, підтримка автозбереження, адаптивний інтерфейс і локальне або хмарне збереження — сьогодні вважаються базовими [4] характеристиками будь-якого конкурентоспроможного застосунку. Вони визначають технічну спроможність, але не формують унікальність або інтелектуальну перевагу інструменту в сфері організації знань.

Водночас, ефективність застосунку для ведення нотаток залежить не лише від технічних параметрів, а й від глибших концептуальних властивостей. Нижче наведено ключові концепції, які визначають ідеальну архітектуру інструменту для управління інформацією.

1. Мислення через структуру, а не просто текст. Інтерфейс і модель даних повинні дозволяти не лише створювати окремі записи, а й будувати логічно зв’язані

структури: вкладені сторінки, дерева, графи. Це дає змогу відображати не тільки інформаційний вміст, але й контекст, зв'язки та залежності між елементами, що суттєво підвищує якість збереження знань.

2. Адаптація до стилю мислення користувача [5]. Різні користувачі працюють з інформацією по-різному: одні віддають перевагу лінійним спискам, інші — просторовим мапам або модульним блокам. Ідеальний застосунок має забезпечувати гнучке середовище, яке підтримує перемикання між режимами візуалізації (таблиці, карти, блоки) без втрати даних або структури.

3. Контекстна взаємодія з інформацією. Застосунок має забезпечувати інтелектуальну обробку дій користувача, коли створення, редагування чи групування елементів залежить від їхнього контексту або позиції. Це включає контекстні меню, умовну поведінку блоків, автоматичну категоризацію тощо.

4. Когнітивна оптимізація інтерфейсу. Ефективне сприйняття інформації залежить від зменшення когнітивного навантаження. Застосунок має підтримувати візуальне згортання/розгортання блоків, колірне кодування, маркування, фільтрацію та швидкий доступ до актуальних фрагментів інформації. Це критично важливо при роботі з великими масивами даних.

5. Інтеграція елементів керування процесом. Нотатки не завжди є статичними — часто вони супроводжують виконання завдань, планування чи контроль прогресу. Ідеальний застосунок має підтримувати призначення статусів, дедлайнів, нагадувань, а також інтеграцію з календарями чи системами керування задачами без перевантаження основного інтерфейсу.

6. Масштабованість мислення: від мікро до макро. Користувач повинен мати змогу працювати як на локальному рівні (одна думка, запис або фрагмент), так і на глобальному (система знань, розділ, проєкт). Перехід між рівнями має бути швидким і логічним, з мінімальною втратою контексту.

7. Повторне використання та мережевість контенту. Застосунок має підтримувати механізми повторного використання даних без дублювання: внутрішні посилання, вставки, шаблони, референції. Це дозволяє будувати не лише ієрархічні, а й мережеві структури знань, які відповідають сучасним підходам до організації

інформації.

Усі зазначені концепції спрямовані на створення системи, що не лише виконує роль записника, а й виступає засобом моделювання та структурування знань відповідно до реальних пізнавальних потреб користувача. Такий підхід є критично важливим у контексті зростання інформаційного навантаження, навчального темпу та вимог до адаптивності цифрових середовищ.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Проектування архітектури застосунку

Проєктування архітектури застосунку "Ієрархічна нотатна дошка" стало ключовим етапом, що визначив внутрішню логіку, компонування модулів і загальні принципи взаємодії між елементами системи. З огляду на вимоги до автономності, локального зберігання, ієрархічної організації даних та інтерактивності інтерфейсу, було обрано компонентно-орієнтований підхід з чітким розподілом відповідальностей між модулями. Основна ідея полягала у створенні гнучкого середовища, в якому користувач може працювати з інформаційними об’єктами у вигляді блоків на полотні, з підтримкою вкладеності, збереженням історії змін, імпортом файлів та розширенням об’єктної моделі.

Застосунок реалізовано як настільну кросплатформну програму на основі Qt та QML. Архітектура побудована за класичною схемою "backend-frontend", де фронтенд-частина (QML) відповідає за рендеринг інтерфейсу, взаємодію з користувачем та відображення моделей, а бекенд (на C++) — за управління станом, логікою обробки подій, роботу з файловою системою, серіалізацією об'єктів та зберіганням.

Основні рівні архітектури:

- Frontend (QML) — візуальне представлення користувацького інтерфейсу, включає компоненти Main.qml, CanvasView.qml, ExplorerView.qml, Popup, FileDialog, TabBar.qml тощо. Використовується CanvasObjectModel для зв’язку з логікою на C++.
- Backend (C++) — реалізує класи CanvasManager, Canvas, CanvasObject, TextObject, FileObject, ImageObject, ShapeObject та інші. Відповідає за зберігання структури полотен, взаємодію з JSON-файлами, Undo/Redo, дії над об'єктами.
- JSON-диск — файлове зберігання у структурі папок: кожне полотно — окрема директорія з json-файлом і пов’язаними файлами (txt, зображення, вкладені полотна). Відсутність серверної частини забезпечує офлайн-роботу та простоту резервного копіювання.

Цей розподіл дозволив досягти гнучкості, чіткого розмежування відповідальностей і зручного масштабування. Взаємодія між рівнями реалізована через властивості, сигнали/слоти, проксі-моделі. Наприклад, зміни у QML-текстовому полі автоматично оновлюють відповідний CanvasObject через сигнал і метод updateTextProperties() на стороні C++.

Для відображення ієрархічної структури полотен передбачено рекурсивну підтримку вкладених об'єктів. Кожен Canvas може містити об’єкти типу "canvas", які при відкритті створюють нову вкладку з унікальним контекстом. Це дозволяє користувачу зберігати складні багаторівневі блок-схеми або мапи знань. Вкладки реалізовані у вигляді динамічного списку openCanvasEntries, який синхронізується з відкритими об'єктами.

Архітектура також враховує зворотну сумісність: JSON-модель має структуру, яку можна легко оновлювати без втрати існуючих даних. Кожен об'єкт зберігає свій UUID, тип, координати, розмір, специфічні властивості (текст, зображення, файл, фігура), що дозволяє уніфіковано обробляти їх на фронтенді. Такий підхід відкриває можливості для подальшої інтеграції нових типів об’єктів, наприклад, аудіо, PDF або векторної графіки.

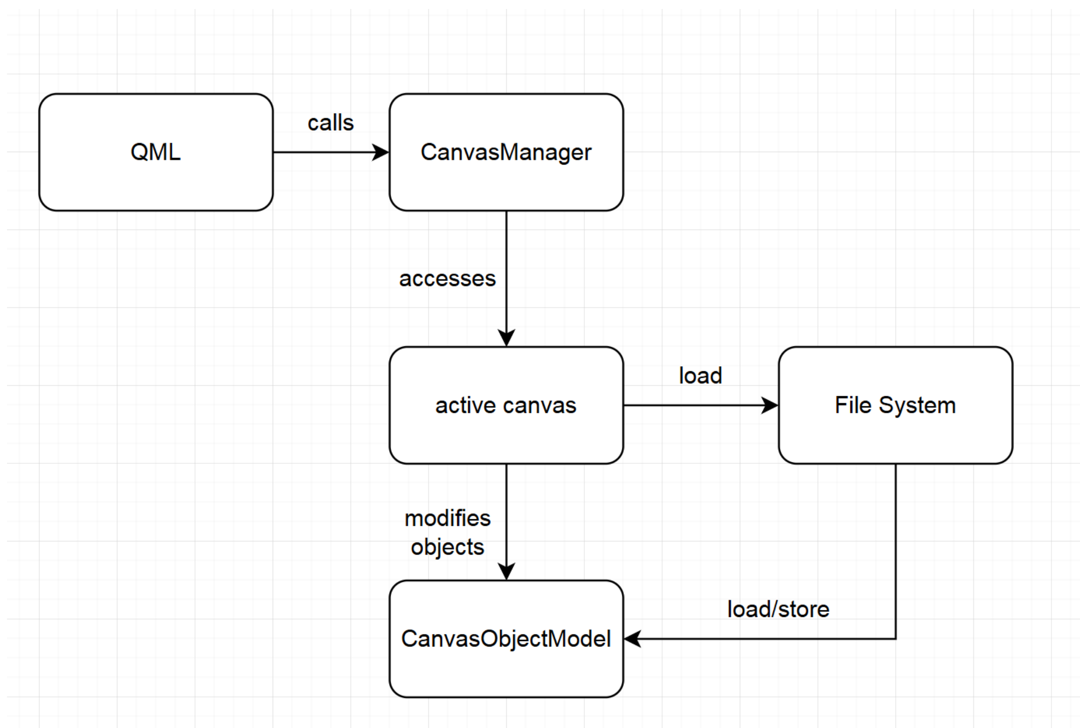


Рисунок 2.1 – Діаграма архітектури

На діаграмі архітектури (рис. 2.1) зображено загальну структуру взаємодії між основними модулями: CanvasManager виступає як центральний координатор, що обробляє виклики з QML, керує активним полотном, взаємодіє з CanvasObjectModel і здійснює завантаження/збереження у файлову систему. Усі операції над об’єктами виконуються через єдиний інтерфейс — це спрощує підтримку, логування та майбутнє тестування.

Таким чином, запропонована архітектура повністю відповідає вимогам до застосунку — автономність, підтримка вкладеності, візуальна інтерактивність, простота обміну та збереження. Вона демонструє гнучкість при розширенні, високий рівень узгодженості між модулями, а також зручність для кінцевого користувача.

2.2 Основні компоненти та їх взаємодія

На рівні архітектури система поділяється на чітко визначені компоненти: CanvasManager, Canvas, CanvasObject (та його спадкоємці), CanvasObjectModel, FileSystem, Undo/Redo менеджер, і зовнішні QML-компоненти.

CanvasManager — головний керівник застосунку. Він управляє поточним відкритим полотном, відповідає за створення, редагування, видалення об'єктів, виконує серіалізацію JSON, підтримує Undo/Redo через CanvasMemento. Усі дії з інтерфейсу проходять саме через цей об'єкт.

Canvas — внутрішній клас, що представляє одне полотно. Містить список CanvasObject, шлях до директорії з файлами, назву, ідентифікатор, JSON-модель. Може містити вкладені Canvas через об’єкти типу “canvas”.

CanvasObject — абстрактний базовий клас, який описує універсальний об’єкт на полотні: координати, розміри, UUID, тип. Він успадковується класами TextObject, FileObject, ImageObject, ShapeObject, які додають специфічні дані (текст, шлях до зображення, тип фігури, стилі тощо).

CanvasObjectModel — клас, що реалізує QAbstractListModel і передає дані в QML. Відображає поточний стан об'єктів на полотні. Усі зміни в Canvas автоматично оновлюються в цій моделі, що забезпечує реактивність інтерфейсу.

FileSystem — набір методів для створення/читання JSON-файлів, копіювання ресурсів, імпорту файлів до структури полотна. Забезпечує правильне розміщення, унікальність імен, підтримку вкладеності директорій.

CanvasMemento — реалізує шаблон Memento: зберігає попередній та наступний стан об'єктів. Підтримує Undo/Redo для операцій додавання, видалення, редагування.

QML (Main.qml, CanvasView.qml, ExplorerView.qml) — зчитують модель із CanvasObjectModel, реагують на натискання, перетягування, зміну властивостей. Викликають методи canvasManager для взаємодії з логікою. Усі QML-компоненти є реактивними й адаптовані під темну тему.

Ці компоненти тісно взаємодіють між собою і забезпечують узгоджену роботу системи. Їхня взаємозалежність зображена на діаграмі (рис. 2.2), де вказано потоки даних, напрями викликів і відображення інформації.

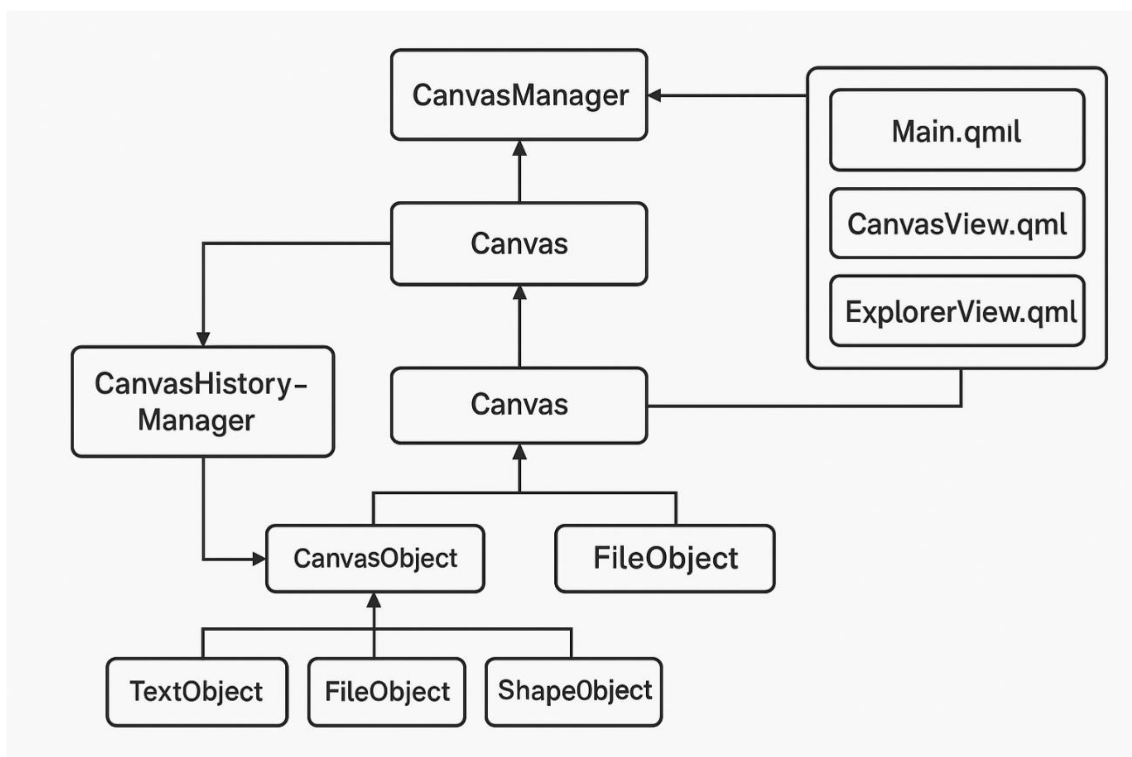


Рисунок 2.2 – Компонентна діаграма архітектури застосунку

2.3 JSON-модель даних

Оскільки система "Ієрархічна нотатна дошка" працює в автономному режимі

та не передбачає підключення до серверної бази даних, ключову роль у зберіганні інформації відіграє файлова структура, організована на основі формату JSON. Цей формат обрано через його простоту, легкість читання людиною, гнучкість у структурі та широке підтримання в інструментах серіалізації Qt. Кожне полотно (Canvas) представлено окремим JSON-файлом [6], що міститься у відповідній директорії разом із пов’язаними файлами — текстами, зображеннями, вкладеними полотнами тощо.

JSON-файл описує всі об’єкти, що розташовані на полотні, включаючи їхні типи, координати, розміри, стильові властивості та унікальні ідентифікатори [7]. Кожен об’єкт у масиві objects має таку загальну структуру (рис. 2.3).

```
"contents": {
    "canvases": [
        {
            "height": 100,
            "id": "{77d976d8-ea4a-4d1d-afcf-1c8881c0caec}",
            "name": "Вкладення",
            "type": "canvas",
            "width": 100,
            "x": 11,
            "y": 14
        },
        {
            "height": 100,
            "id": "{2a2e8e16-5a22-407a-9982-0c151c7f3fd7}",
            "name": "Джерела",
            "type": "canvas",
            "width": 100,
            "x": 820,
            "y": 531
        }
    ],
    "files": [
        {
            "file": "C++.pdf",
            "fileSize": 4804572,
            "fileType": "pdf",
            "height": 321,
            "iconPath": ":/icons/pdf.png",
            "id": "{17f5a70a-6985-464f-af3f-4f1365900cf4}",
            "type": "file",
            "width": 191,
            "x": 464,
            "y": 332
        },
```

Рисунок 2.3 – Вигляд в JSON

Для зображень (type: "image") передбачено поле source, яке містить шлях до локального ресурсу. Для вкладених полотен (type: "canvas") використовується поле canvasPath, що дозволяє рекурсивно відкрити піддошку. Усі файли, до яких посилаються об'єкти, автоматично копіюються до директорії полотна, що гарантує

цілісність структури при перенесенні.

Крім масиву об’єктів, JSON може зберігати метадані: назву полотна, дату створення, версію формату, ідентифікатор, який використовується для відкриття у вкладках. Завдяки уніфікованому формату JSON легко реалізувати функції Undo/Redo — кожен стан полотна зберігається як знімок (snapshot), і в разі необхідності система може повернутися до попередньої версії.

Таким чином, JSON виступає не лише форматом збереження, а й засобом організації структури, навігації між вкладеними рівнями та основою для переносу/імпорту інформації в інших середовищах.

2.4 Особливості кросплатформності

Одним із ключових завдань при проєктуванні "Ієрархічної нотатної дошки" була підтримка роботи застосунку на різних операційних системах — зокрема Windows, macOS та Linux. Це досягнуто завдяки використанню фреймворку Qt, який забезпечує абстракцію від ОС та пропонує єдині API для роботи з графікою, файлами, введенням та мережевими операціями.

Вся логіка, реалізована на C++, є портованою без змін на всі підтримувані платформи. QML, що використовується для побудови інтерфейсу, базується на OpenGL або Direct3D, залежно від системи, і автоматично адаптується до особливостей графічного середовища. Стилі, палітра кольорів, теми (включно з темною) працюють однаково добре на всіх системах завдяки декларативному підходу та підтримці CSS-подібної системи стилізації в QML.

Використання ресурсів (іконок, шрифтів, стилів) реалізовано через систему ресурсів Qt (.qrc), що гарантує правильне відображення незалежно від шляху чи типу ОС. Таким чином, під час компіляції застосунку всі візуальні ресурси вбудовуються у виконуваний файл, що забезпечує його автономність і відсутність залежностей від файлової структури ОС. Це особливо важливо для кросплатформного розгортання, оскільки виключає ризик втрати ресурсів при перенесенні або розпакуванні.

Оскільки дані користувача зберігаються локально у форматі файлів, структура директорій застосунку адаптується відповідно до стандартів платформи: на Windows — це, як правило, підкаталог у теці Documents або AppData, на Linux/macOS — теки у ~/.local/share, ~/Library/Application Support або аналогічні. Для доступу до них використовується API QStandardPaths::writableLocation, що гарантує коректне визначення шляхів для збереження налаштувань, проєктів, кешу тощо.

Крім того, Qt дозволяє використовувати механізм QSaveFile, що забезпечує атомарне збереження JSON-файлів незалежно від файлової системи. Це виключає ризик пошкодження даних при збоях, раптовому завершенні процесу або нестабільній роботі накопичувача.

Модульність застосунку також сприяє кросплатформності. Наприклад, взаємодія між C++ і QML реалізована через інтерфейсні класи, які не прив'язані до специфіки ОС. Зміни, зроблені у фронтенді, автоматично передаються до моделі за допомогою Q_PROPERTY та signal/slot механізмів, які реалізовані однаково на всіх системах. Тому будь-яка адаптація або доопрацювання інтерфейсу не потребує втручання у платформозалежні ділянки коду.

Крім десктопних ОС, Qt також теоретично дозволяє портування на мобільні платформи (Android, iOS) за рахунок тієї ж архітектури. Завдяки повністю QML-орієнтованому інтерфейсу та мінімальній залежності від системних бібліотек, застосунок потенційно придатний до адаптації для планшетів або великих телефонів.

Окрему роль у забезпеченні кросплатформності відіграє файл CMakeLists.txt, який контролює процес збірки застосунку. У ньому задано налаштування, що враховують специфіку різних платформ:

```
set_target_properties(appHNB PROPERTIES
    MACOSX_BUNDLE TRUE
    WIN32_EXECUTABLE TRUE
)
```

Ці параметри гарантують, що при збірці на macOS створюється повноцінний bundle (з іконкою, Info.plist тощо), а на Windows — виконуваний файл із

правильними атрибутами. Також використано опцію qt_standard_project_set, яка активує типові налаштування для сучасних версій Qt із підтримкою QML-модулів.

Ресурси включено до збірки через директиву qt_add_qml_module, що дозволяє автоматично зв'язати зображення, QML-файли та C++-джерела в один артефакт, готовий до розгортання. Встановлення також адаптовано під різні системи через install(TARGETS appHNB ...), з підтримкою бібліотек, bundle або виконуваного файлу залежно від ОС.

Таким чином, як архітектура застосунку, так і інструменти його збирання повністю орієнтовані на багатоцільове розгортання без втрати функціональності або стабільності.

2.5 Математичне та логічне забезпечення

Архітектура застосунку ґрунтується на низці внутрішніх логічних структур і алгоритмів, які забезпечують правильну організацію, узгодженість і збереження інформації. Це стосується роботи з унікальними ідентифікаторами, геометрією об'єктів, рекурсивними структурами, історією змін та операціями з файлами.

Кожен об'єкт на полотні має координати (x, y), розміри (width, height) та тип (text, file, image, shape, canvas). Для побудови взаємодії з такими об'єктами реалізовано алгоритми:

- Визначення перетинання об'єктів — при реалізації drag-and-drop або виділення об'єктів може бути використано обчислення перетину прямокутників.
- Гарантоване унікальне ID — кожен об'єкт створюється з автоматично згенерованим QUuid::createUuid(), що забезпечує відсутність конфліктів при збереженні, відновленні та Undo/Redo.
- Геометрична нормалізація координат — під час перетягування або масштабування об'єкта координати та розміри автоматично обмежуються мінімальними значеннями, щоб уникнути об'єктів з нульовою площею або негативними розмірами. Це реалізується як у QML (Math.max(...)), так і в C++ перед збереженням.

- Рекурсивна обробка вкладених полотен — при відкритті об'єкта типу "canvas" запускається пошук вкладеної директорії та десеріалізація JSON. Це здійснюється методом CanvasManager::setCurrentCanvasById(...), який використовує рекурсивну функцію findById, що проходить по дереву вкладених Canvas до потрібного ID.
- Реалізація Undo/Redo через Memento — кожна операція створення, редагування або видалення об’єкта супроводжується створенням об’єкта CanvasMemento, який зберігає beforeState і afterState у форматі JSON.

Такий підхід дає змогу зберігати повну історію дій і повертатись до попередніх станів без втрат.

- Безконфліктне збереження файлів — при імпорті файлів реалізовано перевірку на наявність однакових імен у директорії полотна.
- Побудова дерева об’єктів для ExplorerView — при відображенні вкладених полотен у провіднику кожен об’єкт типу "canvas" додається до ListView із збереженням вкладеності. У майбутньому можливо реалізувати повноцінне дерево (TreeView), що ґрунтуватиметься на ієрархії ID/parentId.

Таким чином, математичне та логічне забезпечення в системі полягає у точному визначенні координатної моделі, ідентифікації об'єктів, послідовній обробці змін та гарантуванні узгодженості структури при будь-яких діях користувача. Ці принципи закладають основу стабільної, передбачуваної і масштабованої поведінки застосунку в умовах взаємодії з великою кількістю різнорідних даних.

2.6 Діаграма варіантів використання

Для формалізації функціональних можливостей застосунку "Ієрархічна нотатна дошка" було побудовано діаграму варіантів використання, що демонструє типові сценарії взаємодії користувача з системою (рис. 2.4). Діаграма виконана за стандартом UML і слугує інструментом візуалізації основних дій, які може ініціювати актор — користувач програми.

У центрі діаграми розміщено головну дію — створення нового полотна (Create

new canvas), навколо якої згруповані пов’язані варіанти використання. Такі дії, як додавання об’єктів, редагування, видалення, збереження або відміна дій (Undo/Redo), формують основний робочий цикл користувача в межах одного полотна.

Зв’язки між варіантами дій позначені залежностями типу <<include>> та <<extend>>. Зв’язок <<include>> вказує на обов’язкове використання підфункціоналу в рамках більшої операції (наприклад, створення полотна передбачає можливість його збереження). Зв’язок <<extend>> демонструє факультативні або умовні сценарії (наприклад, редагування не є необхідним, але може бути викликане після створення об’єкта).

Опис основних варіантів використання:

- Create new canvas — ініціалізує новий робочий простір для користувача, в якому можна створювати об'єкти;
- Add object — дозволяє додати текст, файл, зображення або підполотно до поточного полотна;
- Remove object — видаляє обраний об’єкт із полотна, при цьому створюється запис в історії змін;
- Edit object — забезпечує зміну властивостей об'єкта (наприклад, тексту, шрифту, кольору);
- Undo/Redo — реалізує механізм скасування та повтору дій користувача для зручності роботи;
- Save canvas — виконує серіалізацію поточного стану полотна в JSON-файл;
- Open canvas — дозволяє завантажити збережене полотно з файлової системи для подальшого редагування.

Завдяки представленій діаграмі можливо не лише зрозуміти логіку взаємодії користувача з інтерфейсом, а й сформувати основу для майбутньої розширюваності системи (наприклад, додавання нових типів об’єктів або реалізація багатокористувацького режиму).

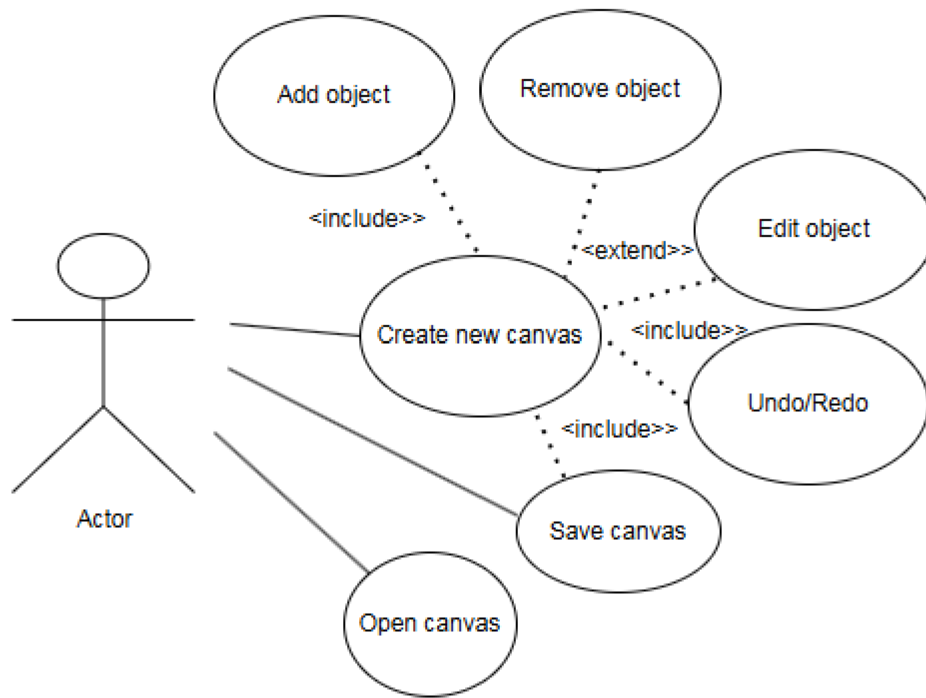


Рисунок 2.4 — Use Case діаграма взаємодії користувача із застосунком

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Інструменти та середовище розробки

3.1.1 Фреймворк Qt та QML

Для реалізації графічного інтерфейсу застосунку «Ієрархічна нотатна дошка» було обрано фреймворк Qt із модулем QML (Qt Modeling Language). Qt — це потужна бібліотека для створення кросплатформенних застосунків з графічним інтерфейсом користувача (GUI), яка підтримує розробку під Windows, macOS, Linux, Android та інші операційні системи з єдиною кодовою базою.

Однією з ключових технологій, використаних у реалізації інтерфейсу, є Qt Quick — модуль фреймворку Qt, який поєднує декларативну мову QML із високопродуктивним рушієм рендерингу, що використовує OpenGL. Qt Quick забезпечує гнучке створення адаптивного [8], анімаційного та реактивного графічного інтерфейсу користувача.

У проєкті Qt Quick дозволив реалізувати численні взаємодії на полотні: перетягування об’єктів, зміна розміру, контекстні меню, інлайн-редагування тексту, візуальне виділення, вкладені полотна тощо. Компоненти Item, Rectangle, Text, TextInput, MouseArea, Repeater, ListView, Popup, Menu — усе це частина екосистеми Qt Quick, яка дозволила швидко та інтуїтивно реалізувати всі візуальні сценарії.

Завдяки прив’язкам (bindings) і сигналам Qt, елементи інтерфейсу автоматично реагують на зміни стану моделі. Це дозволило забезпечити реальну динаміку в UI: об’єкти на полотні переміщуються, змінюють вигляд і зберігають свій стан, не потребуючи ручного оновлення відображення.

Qt Quick також підтримує стилізацію компонентів [9] (власні кольори, шрифти, межі, прозорість), що дало змогу створити темну, сучасну тему інтерфейсу, яка відповідає сучасним вимогам до UI/UX.

Використання Qt Quick стало вирішальним фактором у створенні інтерфейсу, який одночасно є простим для користувача, але здатним підтримувати складну структуру взаємодії з ієрархічними об’єктами.

У рамках проєкту застосовуються такі компоненти Qt/QML:

- ApplicationWindow — базове вікно застосунку;

- ExplorerView — ліва панель, що реалізує провідник для переходу між полотнами (canvas) та об'єктами;
- CanvasView — центральна область взаємодії, яка відображає об'єкти на полотні: текст, файли, зображення, фігури, піддошки;
- Repeater та ListView — для динамічного виводу вмісту моделі CanvasObjectModel;
- MouseArea та Menu — реалізація drag&drop, контекстного меню, натискань та редагування;
- Popup — діалоги додавання нових об'єктів;
- FileDialog — взаємодія з файловою системою.

Інтеграція між QML та C++ реалізована через контекстні об'єкти canvasManager, canvasModel, що дозволяють викликати C++-методи напряму з QML (завдяки макросу Q_INVOKABLE) і отримувати зміни через Q_PROPERTY і сигнали.

Переваги використання Qt/QML:

- Повна підтримка кросплатформності без змін у логіці інтерфейсу [10];
- Швидка побудова UI завдяки декларативному стилю QML;
- Висока продуктивність через рендеринг за допомогою OpenGL;
- Модульність і масштабованість — розділення логіки на окремі компоненти (Main.qml, ExplorerView.qml, CanvasView.qml тощо).

Це дозволило створити інтуїтивно зрозумілий, візуально гнучкий і масштабований інтерфейс для складної ієрархічної структури цифрових нотаток.

3.1.2 Мова програмування C++

У межах розробки застосунку мова C++ стала базовим інструментом для реалізації логіки взаємодії з даними, об'єктної моделі, серіалізації, збереження та управління вмістом полотна. Ця мова добре зарекомендувала себе у сфері створення продуктивного, масштабованого та кросплатформного програмного забезпечення. Висока швидкодія, потужні об'єктно-орієнтовані можливості та контроль над пам’яттю роблять її незамінною для реалізації настільних застосунків із складною

логікою.

У розробці даного застосунку C++ забезпечує функціонування об'єктної моделі: кожен елемент полотна — текстовий блок, файл, зображення, вкладене полотно або фігура — описується відповідним класом, який успадковується від спільного предка CanvasObject. Ця ієрархія дозволяє ефективно керувати атрибутами, зберігати об’єкти в універсальному форматі JSON та легко розширювати функціональність шляхом додавання нових класів.

Архітектура проєкту на базі C++ добре масштабована. Застосовані шаблони проектування, зокрема "фабрика" та "композит", забезпечують централізоване створення об’єктів і можливість побудови вкладених структур без втрати керованості. Це особливо важливо для проєкту, який дозволяє розміщувати піддошки в межах головного полотна на довільну глибину.

Ще однією перевагою є можливість точного контролю за ресурсами. В проєкті активно використовується система розумних вказівників (std::shared_ptr), що дозволяє уникати витоків пам’яті, залишаючись при цьому гнучким у роботі з об’єктами, які можуть передаватися між компонентами. Завдяки такому підходу можна ефективно працювати навіть з великими деревоподібними структурами.

Особливо важливою є інтеграція C++ із мовою QML, яка забезпечується засобами Qt. Клас CanvasManager виступає своєрідним мостом між backend-логікою на C++ та графічним інтерфейсом на QML. Через Qt-властивості (Q_PROPERTY), сигнали та публічні методи (Q_INVOKABLE) дані про об’єкти, їхню геометрію, текстовий вміст та статус оновлень передаються безпосередньо в інтерфейс, забезпечуючи його реактивність [11].

Завдяки мові C++ стало можливим реалізувати повноцінне локальне збереження даних, незалежне від серверної інфраструктури [12]. Файли, зображення та всі інші об’єкти зберігаються в структурі JSON у межах файлової системи користувача, що дозволяє не тільки працювати офлайн, а й легко переносити, архівувати та відновлювати проєкти.

Загалом, використання C++ у цьому проєкті дозволило поєднати стабільність, продуктивність і гнучкість у єдиній архітектурі, яка є основою складної ієрархічної

моделі цифрових нотаток, з можливістю масштабування функціоналу без шкоди продуктивності чи зручності для користувача.

3.1.3 Система контролю версій Git та GitHub

У процесі розробки застосунку було використано систему контролю версій Git — один із найпопулярніших інструментів для відстеження змін у кодовій базі, управління гілками, злиттям функціональностей та організації командної роботи.
Git дозволив:

- Зберігати історію змін кожного етапу проєкту — від прототипу до стабільної реалізації;
- Працювати з окремими гілками — наприклад, окремо реалізовувались механізми збереження в файл, інтеграція меню, CanvasManager, ExplorerView, а потім виконувалося злиття (merge) у основну гілку;
- Відновлювати стабільний стан застосунку у разі появи помилок або невдалих змін;
- Підключити віддалене репозиторійне зберігання (наприклад, GitHub), що дає змогу зберігати проєкт у хмарі, публікувати код і демонструвати прогрес.

GitHub відігравав роль хмарного середовища для централізованого зберігання репозиторію [13], що забезпечило доступ до коду з будь-якого пристрою, резервування, а також фіксацію віх розробки (релізів) і ведення журналу змін. Репозиторій застосунку було організовано з використанням основної (main) гілки для стабільної версії та додаткових функціональних гілок для розробки окремих компонентів. Такий підхід дозволив реалізовувати функціональність модульно та без ризику пошкодити основну версію проєкту.

Для зручності керування репозиторієм використовувався GitHub Desktop — графічний клієнт для Git, який значно спрощує роботу з гілками, комітами та злиттями, особливо в умовах персональної або невеликої командної розробки. Завдяки візуалізації історії змін, можливості швидко перемикатися між гілками та простому інтерфейсу для створення комітів, GitHub Desktop дозволив ефективно керувати життєвим циклом проєкту без необхідності вручну вводити команди у

терміналі. Це особливо зручно при щоденній роботі з великою кількістю дрібних змін та тестування функціоналу.

Інтеграція Git, GitHub і GitHub Desktop створила надійне та контрольоване середовище для розробки застосунку, де всі зміни були прозоро зафіксовані, а кожен етап міг бути відтворений або скасований при необхідності.

Основні Git-команди, що використовувались:

- git init — ініціалізація репозиторію;
- git add . + git commit -m "повідомлення" — збереження змін;
- git branch, git checkout, git merge — управління гілками;
- git revert, git reset — скасування помилкових змін;
- git log — аналіз змін і контроль прогресу.

Завдяки використанню Git та GitHub розробка залишалась структурованою, гнучкою і безпечною, дозволяючи легко повертатись до будь-якої попередньої версії проєкту, проводити експериментальні гілки та інтегрувати їх лише після перевірки стабільності.

3.2 Опис програмної реалізації

3.2.1 Структура інтерфейсу та його основні компоненти

Графічний інтерфейс користувача у застосунку Canvas Explorer поділено на три основні області, які візуально та функціонально розділяють простір головного вікна. Основу становить файл Main.qml, який задає всю структуру через компонування RowLayout (по горизонталі) та ColumnLayout (по вертикалі).

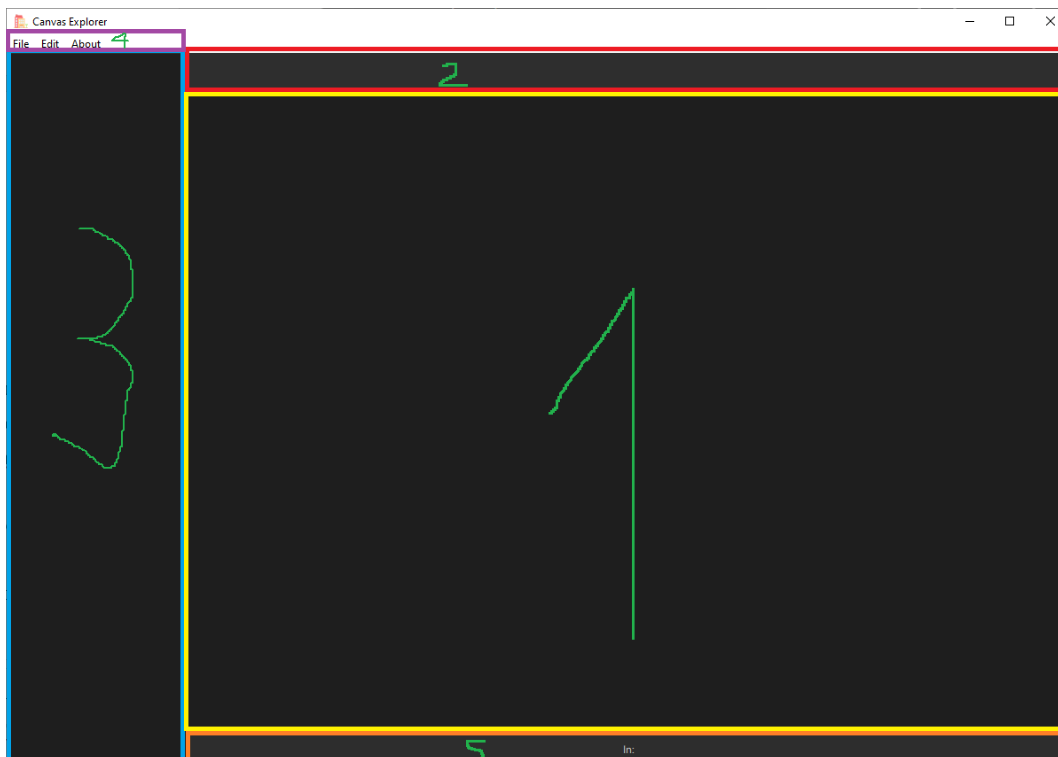


Рисунок 3.1 – Вигляд вікна програми

Інтерфейс складається з таких частин:

- Провідник (ExplorerView) — відповідає за перегляд вкладених об'єктів на полотні у вигляді списку з ієрархічною структурою. Вона дозволяє переходити між підполотнами, виділяти, відкривати та видаляти об'єкти.
- Центральна область (CanvasView) — основна робоча зона, де користувач бачить вміст активного полотна. Тут відображаються всі об'єкти: текстові блоки, файли, зображення, вкладені дошки, фігури. Вони підтримують перетягування, редагування, зміну розміру.
- Верхня панель вкладок — дає змогу перемикатися між кількома відкритими полотнами. Вкладки формуються динамічно на основі списку openCanvasEntries.
- Нижній статусбар — виводить поточну назву активного полотна. Розміщений у нижній частині Main.qml.
- Меню (MenuBar) — складається з пунктів File, Edit та About. File дозволяє створювати, відкривати та зберігати полотна. Edit передбачає підтримку Undo/Redo. About відкриває інформаційне вікно про застосунок.

3.2.2 Панель навігації ExplorerView

Компонент ExplorerView.qml реалізує логіку навігаційної панелі. Основна мета — представити структуру полотна у вигляді списку об’єктів (ListView), в якому відображаються текстові блоки, файли, вкладені полотна, фігури тощо. Кожен тип має унікальне забарвлення і відповідний підпис.

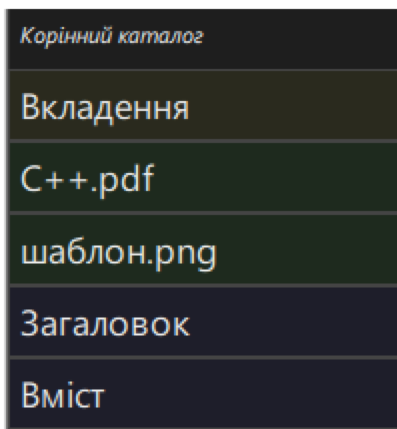


Рисунок 3.2 – Вигляд компонента ExplorerView

Панель містить:

- верхній підпис (Label), що показує поточний шлях;

```
Label {
    text: canvasManager.currentCanvasPath
    font.bold: false
    font.italic: true
    font.pixelSize: 11
    padding: 6
    elide: Text.ElideRight
    wrapMode: Text.Wrap
     color: "#f0f0f0"
}
```

Рисунок 3.3 – Компонент відображення шляху

- ListView, що відображає всі об’єкти з canvasManager.canvasModel;
- делегати, які формуються для кожного об’єкта окремо;
- обробку натискань і подвійних кліків.

```
delegate: Rectangle {
    width: explorerList.width
    height: 36
    color: {
                if (ListView.isCurrentItem)
                    return "#cc5500" // темне виділення
                else if (type === "canvas")
                    return "#2a2a1e" // темно-жовтуватий
                else if (type === "file")
                    return "#1e2a1e" // темно-зелений
                else if (type === "text")
                    return "#1e1e2a" // темно-синій
                else
                    return "#2e2e2e"
            }

    border.color: "#444"

    RowLayout {
        anchors.fill: parent
        anchors.margins: 6
        spacing: 8

        Text {
            color: "#eeeeee"
            text: {
                if (model.type === "canvas") return model.name
                if (model.type === "file") return model.fileName
                if (model.type === "text") return model.text
                return model.type
            }
            font.pixelSize: 18
            elide: Text.ElideRight
        }
    }
```

Рисунок 3.4 – Компонент формування вигляду об’єктів

Виділення реалізовано через ListView.isCurrentItem, а колір кожного елемента залежить від типу (canvas, file, text, shape).

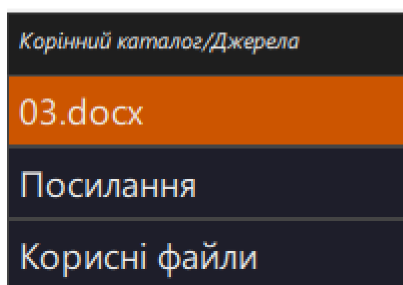


Рисунок 3.5 – Вигляд виділення в ExplorerView

Подвійний клік по елементу canvas відкриває відповідне підполотно через

метод canvasManager.setCurrentCanvasById.

Передбачено також обробку натискання клавіші Delete для видалення об'єкта.

3.2.3 Робоча область CanvasView

CanvasView — це головне візуальне полотно, де користувач взаємодіє з об'єктами. Основна логіка збудована на Repeater, який ітерує об’єкти з моделі canvasManager.canvasModel.

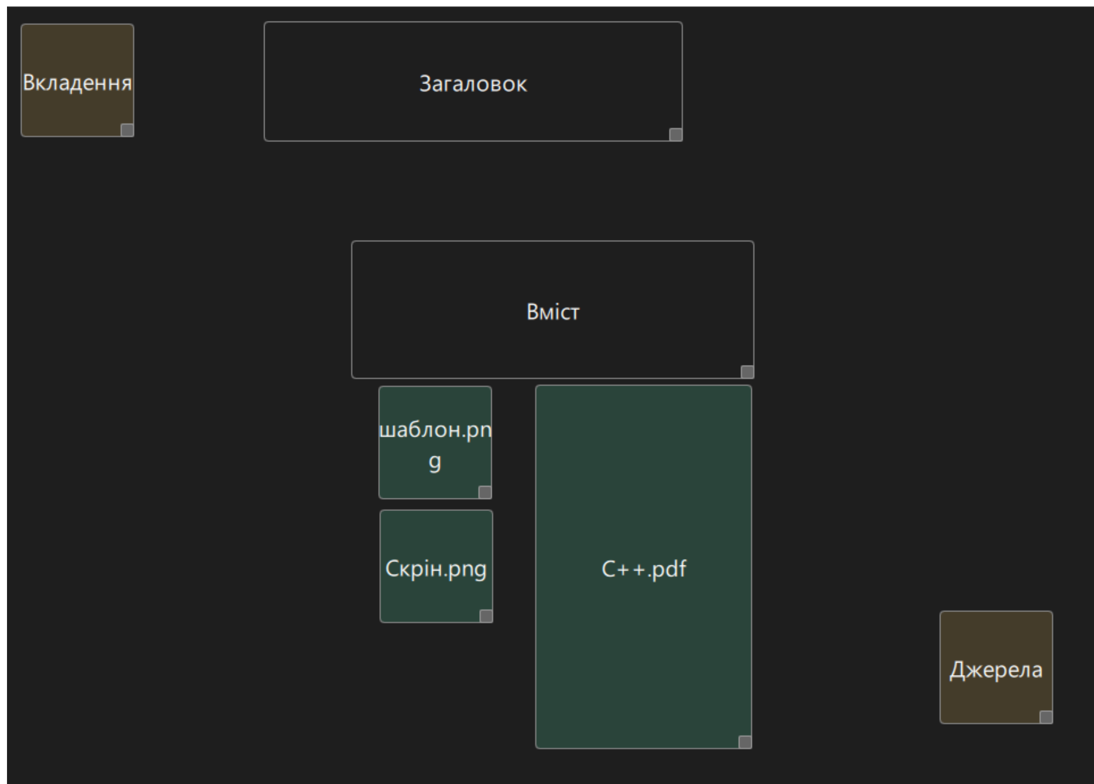


Рисунок 3.6 – Вигляд робочої зони CanvasView

Кожен об’єкт представляється через Rectangle, у якому задаються координати, розміри, колір залежно від типу.

```
Item {
    id: canvasView
    property var selectedObject
    property var onCanvasDoubleClick: function(id) {}
    width: parent.width
    height: parent.height
    clip: true

    Rectangle {
        anchors.fill: parent
        color: "#1e1e1e"

        Repeater {
            model: canvasManager.canvasModel
            delegate: Rectangle {

                property string id_: model.objectId !== undefined ? model.objectId : ""
                property string name_: model.name !== undefined ? model.name : ""
                property string type_: model.type !== undefined ? model.type : ""
                property bool editing: false

                x: model.x
                y: model.y
                width: model.width
                height: model.height
                color: model.type === "text" ? "transparent"
                          : model.type === "file" ? "#2a443a"
                          : model.type === "canvas" ? "#443c2a"
                          : "#333333"
                border.color: "#888"                                          You 2025-0
                border.width: 1
                radius: 4
```

Рисунок 3.7 – Основний компонент CanvasView

- Текстові об’єкти містять Text для перегляду та TextInput для редагування;
- Об’єкти можна переміщати мишкою (через MouseArea.drag.target);

```
// Переміщення об'єкта
MouseArea {
    id: dragArea
    anchors.fill: parent
    drag.target: parent
    drag.axis: Drag.XAndYAxis

    onPressed: {
        canvasView.selectedObject = modelData
    }

    onReleased: {
        canvasManager.updateObjectGeometry(
            model.objectId,
            parent.x,
            parent.y,
            parent.width,
            parent.height
        )
    }
```

Рисунок 3.8 – Компонент коду для переміщення

- Внизу кожного прямокутника є resizeHandle, який дозволяє змінювати розміри об'єкта;
- Подвійне натискання активує підполотно або текстовий режим редагування.

```
// 🟦 Хендл для розтягування
Rectangle {
    id: resizeHandle
    width: 12
    height: 12
    color: "#666"
    border.color: "#999"
    anchors.right: parent.right
    anchors.bottom: parent.bottom
    radius: 2
    z: 10

    MouseArea {
        anchors.fill: parent
        cursorShape: Qt.SizeFDiagCursor

        property real lastMouseX
        property real lastMouseY

        onPressed: {
            lastMouseX = mouse.x
            lastMouseY = mouse.y
        }

        onPositionChanged: {
            const dx = mouse.x - lastMouseX
            const dy = mouse.y - lastMouseY

            const newWidth = Math.max(40, parent.parent.width + dx)
            const newHeight = Math.max(40, parent.parent.height + dy)

            parent.parent.width = newWidth
            parent.parent.height = newHeight

            lastMouseX = mouse.x
            lastMouseY = mouse.y
        }

        onReleased: {
            canvasManager.updateObjectGeometry(
                model.objectId,
                parent.parent.x,
                parent.parent.y,
                parent.parent.width,
                parent.parent.height
            )
        }
    }
}
```

Рисунок 3.9 – Компонент коду для розтягування

Після зміни положення або розміру викликається метод canvasManager.updateObjectGeometry, який синхронізує зміни з C++ логікою.

Колірна палітра та стилі оформлення витримані у темній темі з відтінками сірого, темно-зеленого, коричневого та синього — залежно від типу об’єкта.

3.2.4 Діалогові вікна та FileDialog

У проєкті використано кілька типів діалогових вікон:

- Popup nameInputDialog — універсальний модальний компонент для створення тексту або підполотна. Після введення назви викликає addTextToCurrent, addCanvasToCurrent або createNewCanvas залежно від типу.

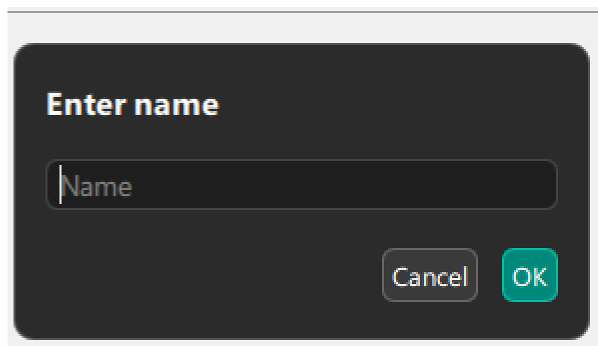


Рисунок 3.10 – Вікно для створення об’єктів

- FileDialog fileDialog — відкриває системний діалог вибору файлу. Після вибору запускає canvasManager.addFileToCurrent із вказаним шляхом до зображення та іконки.
- FileDialog openCanvasDialog — використовується для відкриття папки, яка містить структуру полотна. Після вибору викликає canvasManager.openCanvas.

```
FileDialog {
    id: openCanvasDialog
    title: "Open Canvas Folder"
    fileMode: FileDialog.SelectDirectory
    onAccepted: {
        if (openCanvasDialog.folder)
            canvasManager.openCanvas(openCanvasDialog.folder.toString().replace("file:
    }
}

FileDialog {
    id: fileDialog
    fileMode: FileDialog.OpenFile
    onAccepted: {
        canvasManager.addFileToCurrent(decodeURIComponent(file), ":/icons/file.png")
    }
}
```

Рисунок 3.11 – Компоненти FileDialog

- AboutPopup — виводить інформацію про застосунок: назву, версію, призначення.

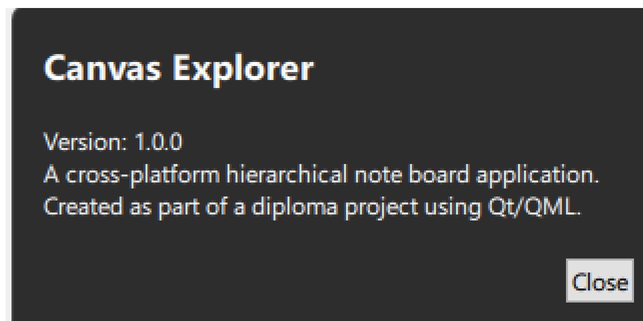


Рисунок 3.12 – Компонент About

Кожен діалог має стилізоване затемнення фону, обробку Escape та Click-outside, а також підтримку навігації з клавіатури (Enter/Escape).

3.2.5 Додаткові компоненти

До допоміжних, але важливих елементів інтерфейсу належать:

- Меню MenuBar — структура з підменю для основних дій. У File розміщено створення, відкриття, збереження. About відкриває модальне вікно.

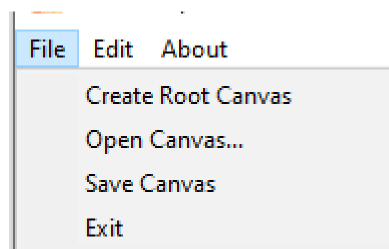


Рисунок 3.13 – Меню програми

- Система вкладок — реалізована через Repeater у верхній частині Main.qml. Кожна вкладка відповідає відкритому полотну (openCanvasEntries), з можливістю закриття через кнопку "×".

```qml
// 🟥 Вкладки
Rectangle {
    color: "#2e2e2e"
    height: 40
    Layout.fillWidth: true

    RowLayout {
        anchors.fill: parent
        spacing: 10

        Repeater {
            model: canvasManager.openCanvasEntries
            delegate: RowLayout {
                spacing: 5

                Button {
                    text: modelData.name
                    onClicked: {
                        canvasManager.setCurrentCanvasById(modelData.id)
                    }
                }

                Button {
                    text: "×"
                    onClicked: {
                        canvasManager.closeCanvasById(modelData.id)
                    }
                }
            }
        }
    }
}
```

Рисунок 3.14 – Компонент вкладки

- Статусбар — прямокутник унизу, який показує назву активного полотна через властивість canvasManager.currentCanvasName.

```qml
// 🟧 Нижній статусбар
Rectangle {
    color: "#2d2d2d"
    height: 30
    Layout.fillWidth: true
    Text {
        text: "In: " + canvasManager.currentCanvasName
        color: "#cccccc"
        anchors.centerIn: parent
    }
}
```

Рисунок 3.15 – Компонент статусбар

- Контекстне меню на CanvasView — викликається при натисканні правої кнопки. Виводить пункти: Add Text, Add File, Add Subcanvas.

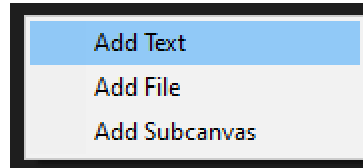


Рисунок 3.16 – Конекстне меню

Інтеграція з C++ реалізується через об’єкт canvasManager, переданий у QML як контекстна властивість. Усі методи викликаються напряму, а зміни автоматично передаються у canvasModel, з яким працюють Repeater, ListView, Text, Rectangle, TextInput та інші компоненти.

ВИСНОВКИ

У дипломній роботі було реалізовано фронтенд-частину кросплатформного застосунку «Ієрархічна нотатна дошка», призначеного для візуального структурування та локального зберігання цифрових нотаток. Основна мета полягала у створенні зручного, масштабованого та інтуїтивно зрозумілого графічного інтерфейсу, який дозволяє користувачеві працювати з вкладеними блоками інформації в інтерактивному середовищі. Роботу побудовано на основі стеку технологій Qt і QML, що забезпечує високий рівень продуктивності, гнучкість стилізації та підтримку багатоплатформності.

Під час дослідження проведено аналіз сучасних інструментів для створення цифрових нотаток та систем візуальної організації інформації. Було виявлено, що більшість із них мають обмеження щодо офлайн-доступу, структурної гнучкості або масштабованості вкладених елементів. Це підтвердило актуальність задачі побудови автономного інструменту з підтримкою просторової організації контенту та локального збереження в зрозумілому форматі.

Інтерфейс було реалізовано у вигляді багатокомпонентної системи. Головне вікно містить чітко поділені зони: панель навігації (ExplorerView), що відображає вкладеність елементів; центральну область (CanvasView), яка є інтерактивним полотном; систему вкладок для роботи з кількома полотнами одночасно; а також контекстні меню, діалогові вікна та статусну панель. Структура побудована так, щоби будь-які зміни в об'єктній моделі автоматично відображались на екрані завдяки механізмам прив’язок у QML.

Було реалізовано повноцінну інтеграцію з backend-логікою на C++ через CanvasManager — клас, що надає публічні методи для створення, редагування, збереження об’єктів та управління їх геометрією. Усі зміни передаються у модель CanvasObjectModel, яка виступає містком між логікою і представленням. Це дозволило реалізувати реактивну архітектуру з мінімальним дублюванням коду, а також забезпечити повну синхронізацію між діями користувача та внутрішнім станом програми.

Діалогові вікна (Popup, FileDialog) забезпечують зручне створення текстів,

піддошок і імпорт файлів. Контекстне меню підтримує праву кнопку миші, а також інтегрується з клавіатурними подіями (наприклад, видаленням через Delete). Вкладки дають змогу одночасно працювати з багатьма підструктурами, відкривати вкладені полотна рекурсивно та керувати ними без втрати контексту.

Окрему увагу приділено візуальному оформленню: інтерфейс витримано в темній темі з акцентними кольорами для кожного типу об’єкта, адаптивними прямокутниками з закругленими кутами, плавною взаємодією при перетягуванні й зміні розміру. Реалізовано механізм інлайн-редагування текстів, а також підтримку роботи з мишею й клавіатурою.

В результаті розробки було створено повноцінний додаток, який містить відповідає поставленим вимогам. Система дозволяє користувачеві створювати, переглядати та редагувати складні ієрархії об'єктів без потреби у серверній частині або постійному підключенні до мережі. Усі дані зберігаються локально у форматі JSON, що спрощує резервне копіювання, передачу та подальшу інтеграцію.

Практична цінність застосунку полягає в тому, що він може використовуватись як персональний інструмент для організації знань, побудови ментальних мап, планування проєктів або структурування навчальних матеріалів. Водночас його архітектура є гнучкою і допускає подальший розвиток — зокрема, реалізацію синхронізації з хмарними сервісами, підтримки колаборації та розширення типів об’єктів.

Таким чином, дипломний проєкт досяг поставленої мети. Усі задачі, пов’язані з реалізацією фронтенд-інтерфейсу для ієрархічної дошки, були виконані. Результат є працездатним, масштабованим та готовим до подальшого вдосконалення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Reinsel, D., Gantz, J., & Rydning, J. *The Digitization of the World*. IDC White Paper. – 2018. – 29 c.

2. McKinsey Global Institute. *The social economy: Unlocking value and productivity through social technologies*. – 2012. – 124 c.

3. Buzan, T. *The Mind Map Book: Unlock your creativity, boost your memory, change your life*. – 2010. – 320 c.

4. Geierhos, M., & Tophinke, D. *Visual Data Structuring in Note-Taking Applications*. In *Human-Centered Technology for Knowledge Management*. Springer. 2021. – C. 145–162.

5. Bashir, A., & Haider, S. *User Interface Design Principles for Information-Rich Applications*. *International Journal of Human–Computer Interaction*, – 2021. – Vol. 37, No. 10. – 915–927c.

6. W3C. *JSON: JavaScript Object Notation*. [Електронний ресурс] – 2022. – URL:https://www.w3.org/TR/2022/NOTE-json-20220331/

7. ISO/IEC 21778:2017. *The JSON Data Interchange Format*. (Reaffirmed 2022). – 28 c.

8. The Qt Company. *Qt 6.7 Documentation*. Retrieved from [Електронний ресурс]. – 2024. – URL: https://doc.qt.io/qt-6/

9. Johan Thelin. *Foundations of Qt Development*. Apress. – 2020. – 528 c.

10. Shen, Y., & Wang, H. *Cross-platform GUI Development Using Qt/QML*. *Journal of Software Engineering and Applications*. – 2020. – Vol. 13, No. 2. – P. 45–56.

11. Nassrat, A. *Designing Reactive QML Interfaces for Complex Applications*. *IEEE Software*. – 2021. – Vol. 38, No. 4. – P. 27–33.

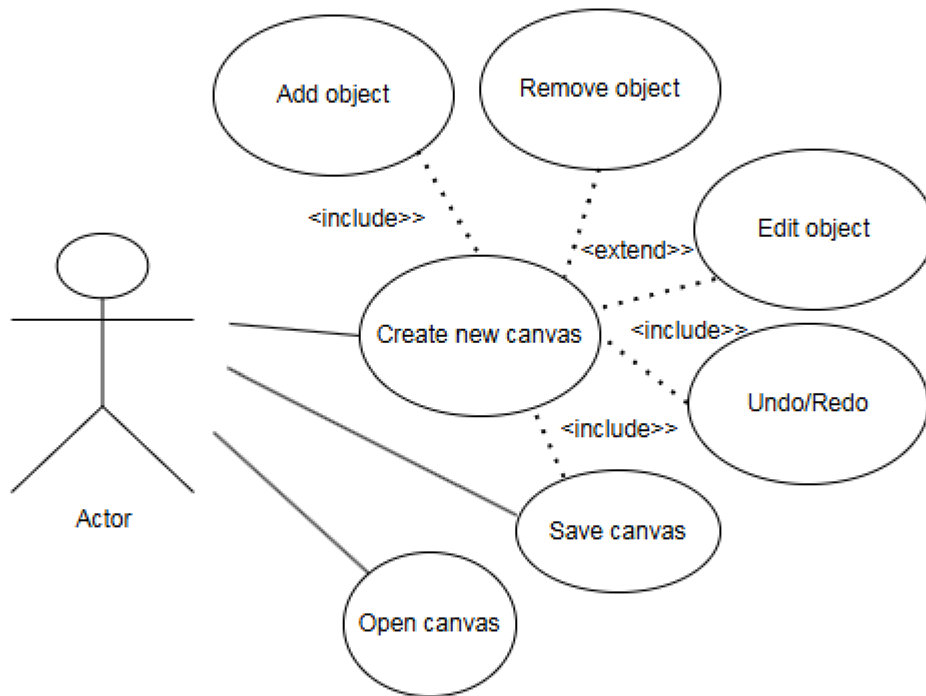
12. Blanchette, S., & Summerfield, M. *Advanced Qt Programming: Creating Great Software with C++ and Qt 6*. – Addison-Wesley, 2023. – 576 c.

13. GitHub, Inc. *Git Handbook*. [Електронний ресурс]. – 2023. – URL:https://docs.github.com/en/get-started

ДОДАТКИ

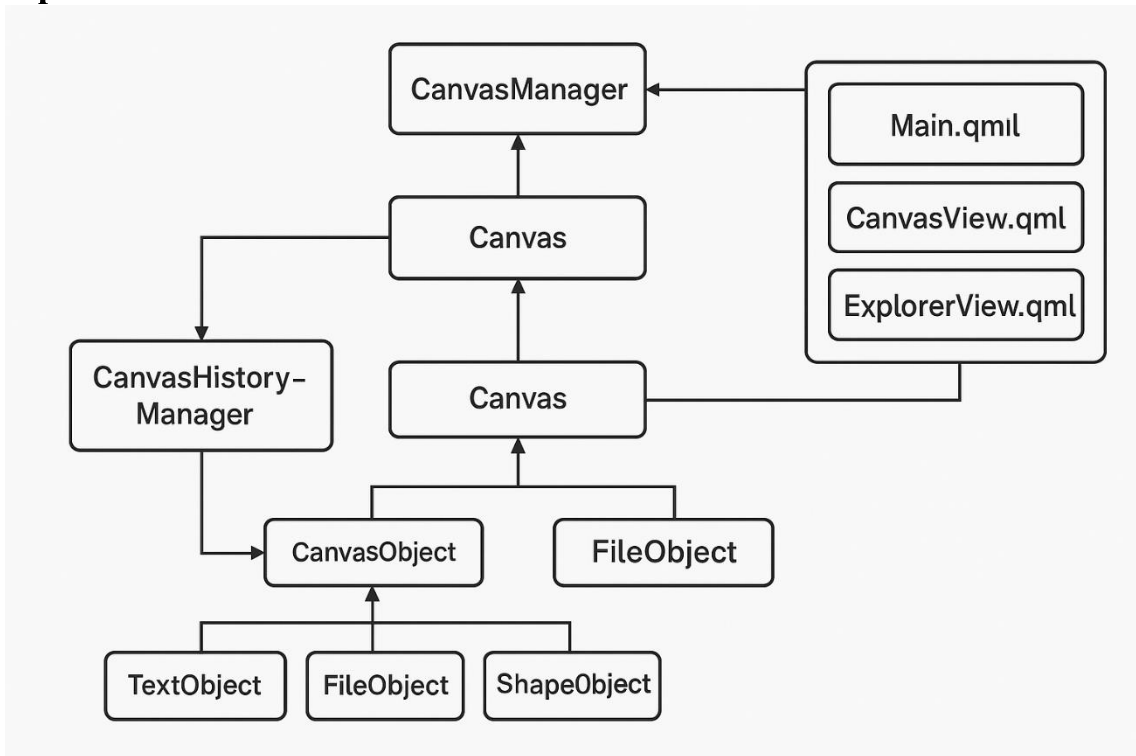
ДОДАТОК А

Use-case діаграма



ДОДАТОК Б

Uml-діаграма



ДОДАТОК В

Maim.qml

```
import QtQuick
import QtQuick.Controls 6.8
import QtQuick.Layouts
import Qt.labs.platform 1.1


ApplicationWindow {
   width: 1200
   height: 800
   visible: true
   color: "#1e1e1e"
   title: "Canvas Explorer"


   MenuBar {
      Menu {
         title: "File"

         MenuItem {
            text: "Create Root Canvas"
            onTriggered: {
               nameInputDialog.requestedType = "canvas"
               nameInputDialog.basePath = "./data"
               nameInputDialog.asRootCanvas = true
               nameInputDialog.open()
            }
         }

         MenuItem {
            text: "Open Canvas..."
            onTriggered: openCanvasDialog.open()
         }

         MenuItem {
            text: "Save Canvas"
            onTriggered: canvasManager.saveCurrentCanvas()
         }

         MenuItem {
            text: "Exit"
            onTriggered: Qt.quit()
         }
      }

      Menu {
         title: "Edit"

         MenuItem {
            text: "Undo"
            enabled: false
            onTriggered: {
            }
```

```
}

            MenuItem {
                text: "Redo"
                enabled: false
                onTriggered: {
                }
            }

            MenuItem {
                text: "Cut"
                enabled: false
                onTriggered: {
                }
            }

            MenuItem {
                text: "Copy"
                enabled: false
                onTriggered: {
                }
            }

            MenuItem {
                text: "Paste"
                enabled: false
                onTriggered: {
                }
            }
        }
        Menu {
            title: "About"

            MenuItem {
                text: "About Canvas Explorer"
                onTriggered: aboutPopup.open()
            }
        }
    }

    RowLayout {


        anchors.fill: parent

        //Навігація (ліва панель)
        ExplorerView {
            Layout.preferredWidth: 200
            Layout.fillHeight: true
        }

        // Центральна область (панель вкладок + canvas)
        ColumnLayout {
            Layout.fillWidth: true
            Layout.fillHeight: true
            spacing: 0

            //  Вкладки
            Rectangle {
                color: "#2e2e2e"
```

```
    RowLayout {
        anchors.fill: parent
        spacing: 10

        Repeater {
            model: canvasManager.openCanvasEntries
            delegate: RowLayout {
                spacing: 5

                Button {
                    text: modelData.name
                    onClicked: {
                        canvasManager.setCurrentCanvasById(modelData.id)
                    }
                }

                Button {
                    text: "×"
                    onClicked: {
                        canvasManager.closeCanvasById(modelData.id)
                    }
                }
            }
        }
    }
}

// Робоча область (Canvas View)
CanvasView {
    id: canvasView
    Layout.fillWidth: true
    Layout.fillHeight: true
    selectedObject: selectedObject
    onCanvasDoubleClick: function(id) {
        canvasManager.setCurrentCanvasById(id)
    }

    // Контекстне меню для створення об’єктів
    MouseArea {
        anchors.fill: parent
        acceptedButtons: Qt.RightButton
        onClicked: (mouse) => {
            if (mouse.button === Qt.RightButton)
                canvasContextMenu.open()
        }
    }

    //  Меню правої кнопки миші
    Menu {
        id: canvasContextMenu
        MenuItem {
            text: "Add Text"
            onTriggered: {
                nameInputDialog.requestedType = "text"
                nameInputDialog.open()
            }
        }
```

```
MenuItem {
                    text: "Add File"
                    onTriggered: fileDialog.open()
                }
                MenuItem {
                    text: "Add Subcanvas"
                    onTriggered: {
                        nameInputDialog.requestedType = "canvas"
                        nameInputDialog.asRootCanvas = false
                        nameInputDialog.open()
                    }
                }
            }
        }

        //  Нижній статусбар
        Rectangle {
            color: "#2d2d2d"
            height: 30
            Layout.fillWidth: true
            Text {
                text: "In: " + canvasManager.currentCanvasName
                color: "#cccccc"
                anchors.centerIn: parent
            }
        }
    }
}

Popup {
    id: aboutPopup
    width: 360
    height: 200
    modal: true
    focus: true
    closePolicy: Popup.CloseOnEscape | Popup.CloseOnPressOutside

    Rectangle {
        anchors.fill: parent
        color: "#2d2d2d"
        border.color: "#444"
        radius: 8
        anchors.margins: 10

        ColumnLayout {
            anchors.fill: parent
            anchors.margins: 16
            spacing: 10

            Label {
                text: "Canvas Explorer"
                color: "#ffffff"
                font.pixelSize: 18
                font.bold: true
            }

            Label {
                text: "Version: 1.0.0\nA cross-platform hierarchical note board application.\nCreated as part
of a diploma project using Qt/QML."
                color: "#ffffff"
```

```
Button {
                text: "Close"
                Layout.alignment: Qt.AlignRight
                onClicked: aboutPopup.close()
            }
        }
    }
}



// Popup для введення імені
Popup {
    id: nameInputDialog
    modal: true
    focus: true
    width: 320
    height: 180
    closePolicy: Popup.CloseOnEscape | Popup.CloseOnPressOutside

    property string requestedType: ""
    property string basePath: "./data"
    property bool asRootCanvas: false

    function confirm() {
        if (nameInputField.text.length > 0) {
            if (requestedType === "text") {
                canvasManager.addTextToCurrent(nameInputField.text, "Arial", 16, "#ffffff");
            } else if (requestedType === "canvas") {
                if (asRootCanvas)
                    canvasManager.createNewCanvas(nameInputField.text, basePath);
                else
                    canvasManager.addCanvasToCurrent(nameInputField.text);
            }
        }
        nameInputField.text = "";
        nameInputDialog.close();
    }

    function cancel() {
        nameInputField.text = ""
        nameInputDialog.close()
    }

    onOpened: nameInputField.forceActiveFocus()

    Rectangle {
        anchors.fill: parent
        color: "#2b2b2b"
        border.color: "#555"
        radius: 10
        anchors.margins: 10

        ColumnLayout {
            anchors.fill: parent
            anchors.margins: 16
            spacing: 12

            Label {
                text: "Enter name"
```

```
                id: nameInputField
                placeholderText: "Name"
                Layout.fillWidth: true
                color: "#eeeeee"
                placeholderTextColor: "#888888"
                font.pixelSize: 14

                background: Rectangle {
                    color: "#1e1e1e"
                    radius: 6
                    border.color: "#444"
                }

                Keys.onReturnPressed: nameInputDialog.confirm()
                Keys.onEscapePressed: nameInputDialog.cancel()
            }

            RowLayout {
                Layout.alignment: Qt.AlignRight
                spacing: 12

                Button {
                    text: "Cancel"
                    onClicked: nameInputDialog.cancel()
                    background: Rectangle {
                        color: "#3a3a3a"
                        radius: 6
                        border.color: "#666"
                    }
                    contentItem: Text {
                        text: qsTr("Cancel")
                        color: "#ffffff"
                        font.pixelSize: 13
                        horizontalAlignment: Text.AlignHCenter
                        verticalAlignment: Text.AlignVCenter
                    }
                }

                Button {
                    text: "OK"
                    onClicked: nameInputDialog.confirm()
                    background: Rectangle {
                        color: "#00897b"
                        radius: 6
                        border.color: "#00bfa5"
                    }
                    contentItem: Text {
                        text: qsTr("OK")
                        color: "#ffffff"
                        font.pixelSize: 13
                        horizontalAlignment: Text.AlignHCenter
                        verticalAlignment: Text.AlignVCenter
                    }
                }
            }
        }
    }
}
```

```
canvasManager.openCanvas(openCanvasDialog.folder.toString().replace("file:///", "").replace(/%20/g, "
      "))
        }
    }

    FileDialog {
        id: fileDialog
        fileMode: FileDialog.OpenFile
        onAccepted: {
            canvasManager.addFileToCurrent(decodeURIComponent(file), ":/icons/file.png")
        }
    }
}
```

ДОДАТОК Г

ExplorerView.qml

```
import QtQuick 2.15
import QtQuick.Controls 2.15
import QtQuick.Layouts 1.15

Item {
    id: explorerView
    width: 250
    property var model: canvasManager.canvasModel

    Rectangle {
        anchors.fill: parent
        color: "#1e1e1e"
        border.color: "#333"

        ColumnLayout {
            anchors.fill: parent
            spacing: 4


            Label {
                text: canvasManager.currentCanvasPath
                font.bold: false
                font.italic: true
                font.pixelSize: 11
                padding: 6
                elide: Text.ElideRight
                wrapMode: Text.Wrap
                 color: "#f0f0f0"
            }


            ListView {
                id: explorerList
                Layout.fillWidth: true
                Layout.fillHeight: true
                model: explorerView.model
                clip: true

                focus: true
                Keys.enabled: true
                    Keys.onDeletePressed: {
                        const obj = model.get(currentIndex)
                        if (!obj || !obj.objectId) return
                        console.log("[Explorer] Delete key pressed on:", obj.objectId)
                        canvasManager.removeObjectFromCurrent(obj.objectId)
                    }

                delegate: Rectangle {
                    width: explorerList.width
                    height: 36
                    color: {
                            if (ListView.isCurrentItem)
                                return "#cc5500" // темне виділення
                            else if (type === "canvas")
                                return "#2a2a1e" // темно-жовтуватий
                            else if (type === "file")
                                return "#1e2a1e" // темно-зелений
                            else if (type === "text")
```

```
return "#1e1e2a" // темно-синій
                        else
                            return "#2e2e2e"
                    }

                border.color: "#444"

                RowLayout {
                    anchors.fill: parent
                    anchors.margins: 6
                    spacing: 8

                    Text {
                        color: "#eeeeee"
                        text: {
                            if (model.type === "canvas") return model.name
                            if (model.type === "file") return model.fileName
                            if (model.type === "text") return model.text
                            return model.type
                        }
                        font.pixelSize: 18
                        elide: Text.ElideRight
                    }
                }

                MouseArea {
                    anchors.fill: parent
                    onClicked: {
                        explorerList.currentIndex = index
                        explorerList.forceActiveFocus()
                    }
                    onDoubleClicked: {
                        if (model.type === "canvas")
                            canvasManager.setCurrentCanvasById(objectId)
                        else if (model.type === "file") {
                            canvasManager.openFileObject(objectId)
                        }
                    }
                }
            }
        }
    }
}
```

ДОДАТОК Ґ

CanvasView.qml

```
import QtQuick 2.15
import QtQuick.Controls 2.15

Item {
    id: canvasView
    property var selectedObject
    property var onCanvasDoubleClick: function(id) {}
    width: parent.width
    height: parent.height
    clip: true

    Rectangle {
        anchors.fill: parent
        color: "#1e1e1e"

        Repeater {
            model: canvasManager.canvasModel
            delegate: Rectangle {

                property string id_: model.objectId !== undefined ? model.objectId : ""
                property string name_: model.name !== undefined ? model.name : ""
                property string type_: model.type !== undefined ? model.type : ""
                property bool editing: false

                x: model.x
                y: model.y
                width: model.width
                height: model.height
                color: model.type === "text" ? "transparent"
                         : model.type === "file" ? "#2a443a"
                         : model.type === "canvas" ? "#443c2a"
                         : "#333333"
                border.color: "#888"
                border.width: 1
                radius: 4

                TextInput {
                    id: inlineEditor
                    color: "#ffffff"
                    anchors.fill: parent
                    visible: editing && type === "text"
                    text: model.text
                    focus: editing
                    font.pixelSize: 24
                    wrapMode: Text.Wrap
                    horizontalAlignment: Text.AlignHCenter
                    verticalAlignment: Text.AlignVCenter
                    selectByMouse: true
                    clip: true

                    onEditingFinished: {
                        editing = false
                        canvasManager.updateTextProperties(
                            model.objectId,
                            text,
                            model.font,
                            model.fontSize,
```

```
model.color,
                    model.bold,
                    model.italic
                )
            }
        }

        Text {
            anchors.fill: parent
            color: "#ffffff"
            visible: !editing || type !== "text"
            text: type === "text" ? model.text
                 : type === "file" ? model.fileName
                 : type === "canvas" ? model.name
                 : "Object"
            font.pixelSize: 20
            wrapMode: Text.Wrap
            horizontalAlignment: Text.AlignHCenter
            verticalAlignment: Text.AlignVCenter
        }

        // Переміщення об'єкта
        MouseArea {
            id: dragArea
            anchors.fill: parent
            drag.target: parent
            drag.axis: Drag.XAndYAxis

            onPressed: {
                canvasView.selectedObject = modelData
            }

            onReleased: {
                canvasManager.updateObjectGeometry(
                    model.objectId,
                    parent.x,
                    parent.y,
                    parent.width,
                    parent.height
                )
            }

            onDoubleClicked: {
                if (model.type === "canvas") {
                    canvasView.onCanvasDoubleClick(model.objectId);
                } else if (model.type === "text") {
                    editing = true
                    inlineEditor.forceActiveFocus()
                }
            }

        }

        //  Хендл для розтягування
        Rectangle {
            id: resizeHandle
            width: 12
            height: 12
            color: "#666"
            border.color: "#999"
```

```
                MouseArea {
                    anchors.fill: parent
                    cursorShape: Qt.SizeFDiagCursor

                    property real lastMouseX
                    property real lastMouseY

                    onPressed: {
                        lastMouseX = mouse.x
                        lastMouseY = mouse.y
                    }

                    onPositionChanged: {
                        const dx = mouse.x - lastMouseX
                        const dy = mouse.y - lastMouseY

                        const newWidth = Math.max(40, parent.parent.width + dx)
                        const newHeight = Math.max(40, parent.parent.height + dy)

                        parent.parent.width = newWidth
                        parent.parent.height = newHeight

                        lastMouseX = mouse.x
                        lastMouseY = mouse.y
                    }

                    onReleased: {
                        canvasManager.updateObjectGeometry(
                            model.objectId,
                            parent.parent.x,
                            parent.parent.y,
                            parent.parent.width,
                            parent.parent.height
                        )
                    }
                }
            }
        }
    }
}
}
```