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" (Backend).

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

Тимчак Д. О.
(прізвище та ініціали)

Керівник: Яцишин С.І.
(прізвище та ініціали)

Рецензент Процик Ю.С
(прізвище та ініціали)

Львів – 2025 року

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

Тимчаку Дмитру Олеговичу
(прізвище, ім'я, по батькові)

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

керівник роботи *Яцишин Світлана Іванівна, зав. каф. ІПЗ, доцент, к.т.н.*,
(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 серверну 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 роботи	Примітка
1	Аналіз архітектурних підходів до побудови ієрархічних графічних редакторів	18.11.24 – 08.12.24	Виконано
2	Формалізація структури об'єктної моделі полотна та сценаріїв роботи з об'єктами	09.12.24 – 30.12.24	Виконано
3	Проєктування та реалізація класів Canvas, CanvasObject, їхніх нащадків і збереження JSON	31.12.24 – 23.01.25	Виконано
4	Реалізація функцій додавання, оновлення, видалення ієрархічних об'єктів з підтримкою UUID	24.01.25 – 22.02.25	Виконано
5	Підтримка збереження стану (Undo/Redo) за допомогою CanvasMemento та історії	23.02.25 – 26.03.25	Виконано
6	Інтеграція бекенду з QML (через CanvasManager, CanvasObjectModel)	27.03.25 – 01.05.25	Виконано
7	Тестування функціоналу, оформлення пояснювальної записки	02.05.25 – 08.06.25	Виконано

Студент _____ (підпис) **Тимчак Д.О.** (прізвище та ініціали)

Керівник роботи _____ (підпис) **Яцишин С.І.** (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота містить 47 сторінок, 7 ілюстрацій, 12 додатки, 18 джерел.

Ця робота присвячена розробленню серверної частини кросплатформного застосунку «Ієрахічна нотатна дошка». Застосунок дозволяє користувачам створювати цифрові полотна, додавати текстові блоки, зображення, файли, вкладені підполотна та графічні об’єкти (лінії, стрілки, вільне малювання), реалізуючи повну ієрархічну структуру та історію змін (undo/redo). Бекенд реалізований мовою програмування C++ з використання фреймворку QT. В роботі розроблено систему збереження у форматі JSON, уніфіковану модель об’єктів (CanvasObject), підтримку унікальних ідентифікаторів (UUID), а також механізми взаємодії з графічним інтерфейсом через клас CanvasManager.

Ключові слова: QT, C++, Ієрархічна нотатна дошка, undo/redo, JSON, CanvsObject, QML, бекенд, структура даних

ABSTRACT

The Bachelor's thesis consists of 47 pages, 7 illustrations, 12 appendices, and 18 sources.

This work is dedicated to the development of the backend part of a cross-platform application called «Hierarchical Note Board». The application enables users to create digital canvases, add text block, images, files, nested sub-canvases, and graphical object (lines, arrows, free drawing), implementing a full hierarchical structure along with an action history system (undo/redo). The backend is implemented in the C++ programing language using the Qt framework. The system features a unified object model (CanvasObject), JSON-based data storage, support for unique object identifiers (UUIDs), and integration with the user interface layer via the CanvasManager class.

Keywords: Qt, C++ Hierarchical Note Board, udno/redo, JSON, CanvasObject, QML, backend, data structure.

ТЕХНІЧНЕ ЗАВДАННЯ

У рамках дипломної роботи необхідно реалізувати програмне забезпечення у вигляді бекенд-модуля для кросплатформного застосунку «Ієрархічна нотатна дошка», призначеного для збереження, модифікації та управління структурованими графічними об’єктами на полотні з підтримкою вкладеності та історії змін. 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.

Розробка програмного модуля включає реалізацію наступних функціональних складових. По-перше, створено універсальну об’єктну модель (CanvasObject), що лягає в основу всіх типів елементів: TextObject, FileObject, ImageObject, ShapeObject, Canvas. По-друге, реалізовано систему збереження даних у файловій системі у форматі JSON із підтримкою вкладеної структури (папка полотна містить власний JSON-файл та підпапки з дочірніми полотнами). Усі об’єкти мають унікальні ідентифікатори (UUID), що забезпечує їх однозначну ідентифікацію та обробку.

Особливу увагу приділено підтримці історії змін. Реалізовано механізм збереження попередніх і нових станів об’єктів через структуру CanvasMemento, що дозволяє реалізувати операції undo() та redo() для будь-яких змін, включно з додаванням, оновленням та видаленням об’єктів. Історія зберігається у вигляді стека дій для кожного полотна окремо.

Інтеграція з графічним інтерфейсом (QML) здійснюється через клас-посередник CanvasManager, який забезпечує зв’язок між внутрішньою логікою та UI. Він дозволяє створювати, відкривати та перемикати полотна, а також виконувати всі основні дії над об’єктами. Усі об’єкти передаються у вигляді моделей для QML через CanvasObjectModel.

Конфігурація та структура даних є самодостатніми та не вимагають зовнішньої бази даних. Уся інформація зберігається у вигляді локальних файлів, що забезпечує простоту переносу та розгортання проєкту. Проєкт реалізовано мовою C++ із використанням середовища Qt 6+.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface – інтерфейс прикладного програмування.

JSON – JavaScript Object Notation – формат обміну даними.

GUI – Graphical User Interface – графічний інтерфейс користувача.

UUID – Universally Unique Identifier – універсальний унікальний ідентифікатор.

SVG – Scalable Vector Graphics – масштабовані векторні зображення.

QML – Qt Modeling Language – мова опису інтерфейсів у середовищі Qt.

Qt – бібліотека для розробки кросплатформного ПЗ з GUI.

JSON-файл – файл, що зберігає структуру даних у форматі JSON.

Canvas – полотно, елемент інтерфейсу для розміщення об’єктів.

CRUD – Create, Read, Update, Delete – базові операції над даними.

Undo/Redo – операції скасування та повторення дії.

Backend – серверна частина застосунку, що відповідає за логіку і зберігання.

Git – система контролю версій.

QVariant – універсальний тип у Qt, що може містити будь-яке значення.

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

ВСТУП

У сучасних умовах цифрової трансформації зростає потреба в інструментах, що дозволяють зручно структурувати, зберігати й обробляти інформацію. Особливо це стосується фахівців у сферах управління проєктами, освіти, дизайну, розробки програмного забезпечення, а також звичайних користувачів, яким необхідно фіксувати ідеї, думки чи плани у вигляді логічно впорядкованих нотаток. Поширення моделей візуального мислення, таких як mind-mapping, канбан-дошки, гнучке проектування тощо, сприяє розвитку інтерактивних графічних застосунків, здатних зберігати складні структури з різнорідними об'єктами: текстами, зображеннями, файлами, підрозділами.

Більшість існуючих рішень у цій галузі — зокрема, Miro, Notion, Excalidraw, Milanote — зосереджені на веб-інтерфейсах з хмарним зберіганням даних. Такі застосунки, незважаючи на високу функціональність, мають низку обмежень: вони вимагають постійного інтернет-з'єднання, не дозволяють повністю локалізувати дані, а також не завжди підтримують вкладені структури та повноцінну систему відстеження змін (undo/redo). Ці обмеження суттєво знижують гнучкість і конфіденційність використання, особливо в освітньому або корпоративному середовищі.

У цьому контексті актуальним стає створення локального кросплатформного застосунку, який дозволяє працювати з ієрархічно організованими нотатками без прив'язки до серверної частини чи зовнішніх API. Такий застосунок повинен надавати можливість створення вкладених полотен (canvas), підтримку різних типів об'єктів (текст, файли, зображення), збереження структури у відкритому форматі (наприклад, JSON), а також реалізацію механізму скасування/повторення дій.

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

Метою дипломної роботи є розроблення бекенд-частини кросплатформного застосунку «Ієрархічна нотатна дошка», яка відповідає за внутрішню логіку зберігання, обробки та управління об’єктами на полотні.

Предметом дослідження є програмні засоби, бібліотеки та технології, які

забезпечують реалізацію функціональних компонентів системи: зокрема, засоби для моделювання об'єктів, серіалізації в JSON, керування унікальними ідентифікаторами (UUID), шаблони проєктування (наприклад, Memento, Factory), реактивне оновлення моделі та інтеграція з QML через CanvasObjectModel.

До **завдань**, що вирішуються в рамках проєкту, належать:

- побудова моделі даних для представлення об'єктів з унікальними ідентифікаторами (UUID);
- збереження структури полотна з вкладеними об’єктами у форматі JSON;
- реалізація базових операцій над об'єктами (створення, оновлення, видалення);
- підтримка функціональності undo/redo для зворотного відновлення стану;
- забезпечення взаємодії з інтерфейсом через модель CanvasObjectModel для Qt/QML.

Архітектура бекенд-модуля орієнтована на розширюваність, ізоляцію логіки від графічного інтерфейсу та можливість подальшої інтеграції з іншими платформами або синхронізаційними механізмами.

Практична значущість роботи полягає у створенні універсального та розширюваного ядра для локального застосунку, який може використовуватися як в освітньому процесі, так і в персональних або командних середовищах. Застосунок може слугувати платформою для подальших досліджень у сфері UX-дизайну, архітектури ПЗ, або як основа для розширення у вигляді мобільного чи мережевого клієнта.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Загальна характеристика проблеми організації цифрових нотаток.

У добу цифровізації обсяг інформації, що щодня обробляється людиною, стрімко зростає. Це зумовлює потребу у зручних і гнучких інструментах для фіксації, структурування, збереження та візуалізації інформації. Цифрові нотатки стали невід'ємною частиною робочого процесу для широкого кола користувачів — від студентів і викладачів до розробників, менеджерів проєктів та дизайнерів.

Проте, попри широку доступність простих текстових застосунків, таких як стандартні нотатки на мобільних пристроях або десктопні текстові редактори, користувачі дедалі частіше зіштовхуються з проблемою обмеженої структури та функціональності. Звичайні текстові записи не дозволяють логічно групувати дані, створювати вкладені підрозділи, прикріплювати додаткові об'єкти (зображення, файли, коментарі) чи візуально уявляти зв'язки між елементами. Це призводить до втрати контексту, зниження зручності користування та продуктивності.

Ще однією проблемою є відсутність універсального формату подання. Деякі інструменти дозволяють створювати графічні нотатки, проте зберігають їх у закритих, важкодоступних форматах. Інші, навпаки, мають відкриту структуру, але обмежені лише лінійними текстовими записами. Користувачеві часто доводиться обирати між візуальністю та структурністю, між гнучкістю форматування і простотою збереження.

Також виникає потреба в локальному контролі над даними. Велика частина сучасних інструментів зосереджена на хмарному зберіганні, що вимагає постійного підключення до інтернету та не завжди гарантує конфіденційність. Для деяких категорій користувачів (наприклад, освітян, держустанов або незалежних розробників) критично важливо зберігати інформацію на локальному пристрої без передачі її стороннім сервісам.

Таким чином, проблема організації цифрових нотаток полягає у відсутності універсального інструменту, який одночасно:

- підтримує різнорідні типи об'єктів (текст, зображення, файли);
- дозволяє ієрархічне структурування;

- працює офлайн і зберігає дані локально;
- забезпечує зручну візуалізацію і можливість відновлення змін (undo/redo).

1.2. Аналіз сучасних засобів створення цифрових нотаток

Сучасний ринок пропонує велику кількість інструментів для створення, редагування та організації цифрових нотаток. Ці рішення можна умовно поділити на кілька категорій: текстові редактори, органайзери з базами даних, візуальні дошки та інтегровані системи управління знаннями (рис 1.1).

До найпопулярніших інструментів належать Notion, Obsidian, Miro, Excalidraw, Milanote, Joplin та інші. Кожен із них має свої переваги та недоліки залежно від цільової аудиторії та особливостей реалізації.

Notion - Сервіс, орієнтований на створення структурованих документів з підтримкою блоків: тексту, списків, таблиць, зображень тощо. Підтримує вкладені сторінки, має просту візуалізацію, однак не є повністю офлайн-рішенням. Дані зберігаються у хмарі.

Obsidian - Файл-орієнтований Markdown-редактор з підтримкою графа зв’язків між сторінками. Працює локально, зберігає дані у вигляді .md файлів, має потужне розширення через плагіни, проте не орієнтований на графічну візуалізацію або вкладені об'єкти з різними типами (наприклад, файли чи canvas). Miro / Excalidraw - Онлайн-дошки для спільної візуальної роботи. Дозволяють створювати блок-схеми, діаграми, розміщувати об'єкти у вільному просторі. Не підтримують вкладені дошки або структуровану ієрархію. Крім того, залежні від наявності підключення до інтернету.

Milanote - Поєднує візуальне представлення з блоками тексту, медіа, лінками. Призначений для дизайнерів і креативних команд. Має обмежену підтримку структури, відсутність локального зберігання, а також не підтримує undo/redo на рівні даних.

Інструмент	Кросплатформність	Синхронізація	Додаткові можливості
	Windows, macOS, Android, iOS, Web	Через хмару	Теги, шаблони, вбудоване зображенн
	Windows, macOS, Android, iOS, Web	Через хмару	Організація в блокноти, вілна розмітка сторінок
	Android, iOS, Web	Через хмару	Списки, нагадування, розпізнавання тексту
	Windows, macOS, Android, iOS, Web	Через хмару	Підтримка версійностіі
Simplenote			

Рисунок 1.1 – Порівняння популярних сервісів для цифрових нотаток

1.3. Потреба у кросплатформності та локальному збереженні без сервера

У сучасному цифровому світі питання організації, структурування та збереження особистої та професійної інформації набуває все більшого значення. Швидкий темп життя, різноманітні джерела даних, робота над кількома проєктами одночасно — усе це вимагає зручного, надійного та гнучкого інструмента для нотаток, який буде доступним у будь-який момент, на будь-якому пристрої, без прив’язки до стабільності інтернет-з’єднання або наявності серверної інфраструктури.

Більшість сучасних рішень для цифрових нотаток, таких як Notion, Google Keep, Evernote, Obsidian або Microsoft OneNote, базуються на моделі зберігання даних у хмарі. Такий підхід має очевидні переваги: синхронізація між пристроями, централізоване резервне копіювання, спільний доступ до інформації в реальному часі. Проте така модель не позбавлена суттєвих недоліків, особливо в контексті безпеки, приватності, автономності та контролю над власними даними.

Проблеми залежності від хмари. Однією з ключових проблем є повна або часткова втрата контролю над даними. У випадку використання сторонніх серверів усі нотатки, файли, вкладення та історія змін зберігаються на ресурсах компанії-постачальника. Це означає, що користувач фактично делегує відповідальність за

збереження інформації, її захист, а також стабільність доступу до неї.

До цього додається питання конфіденційності. Незважаючи на політики захисту даних, жоден хмарний сервіс не може гарантувати абсолютної безпеки від витоків, атак чи зловживань. Для державних установ, наукових організацій, бізнесу та приватних осіб, що працюють з чутливою інформацією, це може бути критичним фактором.

Ще одна проблема — нестабільність доступу. У випадках, коли інтернет-з'єднання відсутнє або перебуває в зоні з обмеженим покриттям (наприклад, у сільській місцевості, під час подорожей або в польових умовах), користувач повністю втрачає можливість працювати з даними. Таким чином, доступність сервісу стає залежною від зовнішніх чинників.

Переваги локального зберігання:

На відміну від хмарних рішень, локальне збереження даних дозволяє зберігати всю інформацію безпосередньо на пристрої користувача. Це забезпечує:

- Повну автономність: жодних обмежень у доступі до даних — працювати з ними можна офлайн у будь-який момент;
- Безпеку: дані зберігаються лише локально і не передаються стороннім серверам, що виключає ризик витоку інформації;
- Простоту резервного копіювання: користувач може архівувати дані на флешці, зовнішньому диску або у власному зашифрованому сховищі;
- Швидкість роботи: відсутність необхідності завантаження та синхронізації пришвидшує взаємодію з нотатками.

Крім локального збереження, сучасне програмне забезпечення повинно бути кросплатформним, тобто підтримувати роботу на різних операційних системах: Windows, macOS, Linux. Універсальність застосунку дає змогу:

- уникнути прив'язки до одного типу пристрою чи ОС;
- легко передавати проєкти між колегами з різними технічними конфігураціями;
- використовувати єдине середовище для роботи вдома, в офісі або в дорозі;
- мінімізувати витрати часу на адаптацію інтерфейсу.

Особливо важливо це для проєктів із відкритим вихідним кодом, інструментів для дослідників, студентів, а також невеликих команд розробників, яким потрібно однакове ПЗ на різних машинах. Вибір таких технологій, як Qt/QML, дозволяє забезпечити однакову логіку, вигляд та функціональність застосунку на різних платформах без потреби переписування коду під кожну ОС окремо.

Рисунок 1.2 – порівняння "Хмарні сервіси vs Локальний додаток".

На думку фахівців з безпеки, локальне зберігання даних значно зменшує ризики витоків, дозволяючи користувачу зберігати повний контроль над інформацією [1].

Сукупність вищезгаданих факторів обґрунтовує актуальність створення локального кросплатформного застосунку без серверної частини, який би дозволяв працювати з цифровими нотатками у вигляді вкладених структур, зберігати всі дані у зрозумілому форматі (наприклад, JSON + файли) та не залежати від зовнішніх чинників. Такий підхід не лише гарантує стабільність та приватність, а й забезпечує повну гнучкість у роботі, адаптації та розширенні застосунку під потреби конкретного користувача.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Постановка задачі

У сучасних умовах інформаційного перевантаження користувачі потребують інструментів, які дозволяють ефективно організовувати цифрові нотатки, зберігати їх у зручному форматі, структурувати ієрархічно та мати доступ незалежно від наявності інтернету. Особливої актуальності набувають локальні кросплатформні рішення, які дозволяють працювати з особистими або робочими даними без серверної інфраструктури, забезпечуючи максимальний рівень приватності, гнучкості та незалежності.

На цьому тлі постає завдання створити програмний засіб — "Ієрархічна нотатна дошка", який дозволить:

- створювати необмежену кількість ієрархічно вкладених "полотен" (canvas), кожне з яких є самостійною структурною одиницею;
- додавати на полотна різні типи об’єктів: текстові блоки, зображення, файли, підполотна, графічні фігури;
- зберігати стан кожного полотна у вигляді локальної файлової структури (JSON-файл + вкладені ресурси);
- забезпечити повну кросплатформну роботу програми (Linux, Windows, macOS);
- реалізувати редагування та збереження об'єктів у реальному часі;
- підтримувати зміну геометрії об'єктів (позиція, розміри), їх властивостей (текст, шрифт, колір тощо);
- реалізувати систему undo/redo, яка дозволяє повертати або повторювати дії користувача;
- вбудувати об’єктну модель та алгоритми, що дозволяють динамічно створювати, оновлювати або видаляти об’єкти;
- забезпечити легке резервне копіювання, перенесення або синхронізацію даних без втрати ієрархії.

Таким чином, метою задачі є розробка ядра застосунку, яке обробляє всі операції із даними, зберігає логіку об'єктної моделі, відповідає за маніпуляції з файлами та забезпечує цілісність даних при кожній дії користувача.

Для вирішення поставленої задачі передбачається застосування принципів об'єктно-орієнтованого проєктування, патернів проектування (Factory, Command/Memento), а також використання структури даних на базі JSON для забезпечення зберігання та відновлення інформації.

Результатом вирішення задачі має стати багатоплатформний програмний модуль, який повністю реалізує вищезазначені функції, а також може бути вбудований у більший інтерфейсний застосунок на базі QML.

2.2. Побутова об’єктно-орієнтованої моделі

2.2.1. Діаграма класів

Розробка програмного забезпечення для роботи з ієрархічними нотатками потребує ретельного проєктування структури класів, які б відповідали принципам об’єктно-орієнтованого підходу, забезпечували повторне використання коду, легкість у підтримці та розширенні функціональності. На основі цих вимог було побудовано діаграму класів (рис. 2.1), що відображає основні сутності системи, їх зв’язки, наслідування та взаємодію між ними.

Ключовим елементом архітектури є абстрактний базовий клас CanvasObject, який об'єднує спільні характеристики всіх об’єктів, що можуть бути розміщені на полотні: це координати розташування, розміри, тип об’єкта та унікальний ідентифікатор. Такий підхід дозволяє розглядати всі візуальні елементи (тексти, файли, зображення, підполотна, фігури) як уніфіковані сутності та обробляти їх спільними методами.

На основі CanvasObject побудовано спеціалізовані класи, кожен з яких реалізує певну функціональність:

- TextObject — відповідає за текстові нотатки, які мають стилізацію, шрифт, колір, жирність та курсив;

- FileObject — дозволяє зберігати інформацію про прикріплені файли, їх тип, розмір, піктограму;
- ImageObject — описує графічні зображення, які додаються користувачем;

ShapeObject — виступає проміжним абстрактним класом для фігур (лінії, кола, стрілки), які використовуються для графічних позначень чи акцентів. Його нащадки — LineObject, ArrowObject, CircleObject, FreeDrawObject.

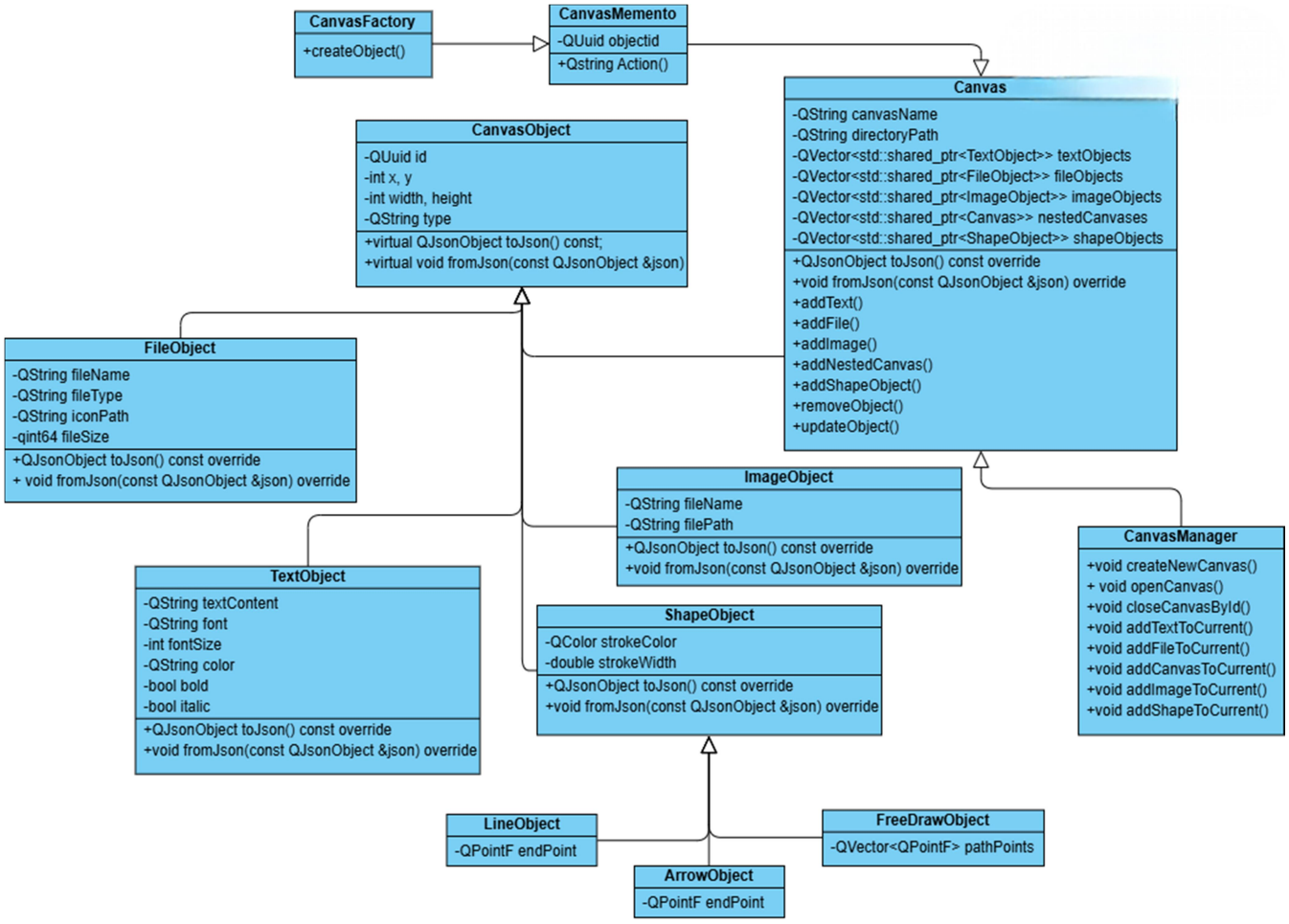


Рисунок 2.1 – UML діаграма класів

Особливої уваги заслуговує клас Canvas, який також є нащадком CanvasObject. Це дозволяє будувати ієрархію вкладених полотен, де кожне з них, будучи об’єктом, може містити інші полотна. Таким чином, реалізується структура типу "дерево", де гілками можуть виступати як звичайні об’єкти, так і підполотна. Canvas виступає головним контейнером, який оперує колекціями текстів, файлів, зображень, фігур та інших полотен.

Для управління структурою проєкту в цілому використовується клас CanvasManager. Він забезпечує відкриття, створення, закриття та перемикання між полотнами. CanvasManager взаємодіє з Canvas, делегуючи основні операції, і виступає як зв’язкова ланка між бекенд-логікою та графічним інтерфейсом користувача (QML-компонентами). Менеджер також підтримує список відкритих полотен та визначає поточне активне полотно, з яким ведеться робота.

З метою забезпечення зручного збереження та відновлення даних, усі класи підтримують механізм серіалізації у формат JSON. Це дозволяє легко зберігати стан полотна та окремих об’єктів на диск, а також виконувати операції undo/redo. Для реалізації цих функцій було впроваджено спеціалізований клас CanvasMemento, який інкапсулює інформацію про зміну: попередній і поточний стани об’єкта (у форматі QJsonObject), його унікальний ідентифікатор, а також тип виконаної дії (додавання, видалення, оновлення). Клас Canvas відповідально зберігає ці об’єкти у двох стеках — undoStack та redoStack, що дає змогу реалізувати покрокове скасування або повторення змін, ініційованих користувачем.

Крім того, в моделі передбачено CanvasFactory — фабричний клас, який забезпечує створення об’єктів відповідного типу на основі інформації, отриманої з JSON. Це дозволяє реалізувати відновлення об’єктів після завантаження з диску або при виконанні дій undo/redo, не прив’язуючись до конкретних класів у момент виконання.

Загалом така структура класів забезпечує:

- модульність (кожен тип об’єкта має окрему відповідальність);
- масштабованість (можна легко додавати нові типи об’єктів);
- повторне використання коду (через наслідування);
- зручне збереження та відновлення стану.

Дана модель є стійкою до змін та може ефективно застосовуватись для реалізації як десктопного, так і мобільного інтерфейсу користувача, зберігаючи при цьому повну підтримку локального зберігання, офлайн-режиму та гнучкого управління структурою нотаток.

2.2.2. Use case діаграма

Use Case діаграма відображає основні функціональні сценарії взаємодії користувача з системою. Вона дозволяє сформулювати вимоги до системи з точки зору зовнішнього користувача (Actor), який взаємодіє із застосунком через інтерфейс (рис. 2.2).

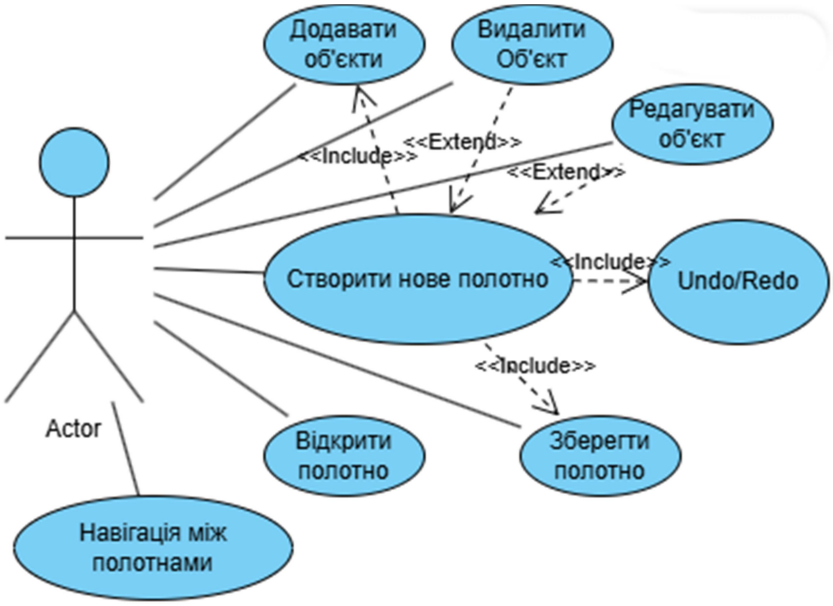


Рисунок 2.2 – Use Case діагра

На наведеній діаграмі Actor — це користувач, який взаємодіє з системою для керування цифровими нотатками та ієрархією полотен.

Основні сценарії (Use Cases):

Створити нове полотно — базовий сценарій, з якого зазвичай починається робота з додатком. Полотно виступає в ролі контейнера для об’єктів.

Має включення (<<include>>) до:

- Додати об'єкти — об’єкти (тексти, файли, зображення, фігури) додаються одразу після створення полотна.
- Undo/Redo — дозволяє відміняти або повторювати дії після створення об’єктів.
- Зберегти полотно — кожне нове полотно передбачає серіалізацію в файл.

- Редагувати об'єкт — після створення часто виконується редагування.

Додати об'єкти - може включати додавання різних типів об’єктів (текстів, зображень, файлів, фігур). Реалізується за допомогою відповідних методів у CanvasManager і Canvas.

Видалити об'єкт - дозволяє видалити будь-який об’єкт із поточного полотна. Має зв’язок <<extend>> з базовим сценарієм Створити нове полотно, оскільки це необов’язкова, але поширена операція після створення або відкриття.

Редагувати об'єкт - зміна тексту, позиції, розміру, кольору або геометрії об’єкта. Також є розширенням сценарію Створити нове полотно.

Undo/Redo - вбудований механізм для скасування та повтору останніх дій. Включається в базову функцію створення об'єкта, оскільки підтримка змін — невід'ємна частина роботи з полотном.

Відкрити полотно - дає змогу завантажити існуючі полотна з диска. Може ініціювати подальшу навігацію або редагування.

Зберегти полотно - Включається в усі ключові сценарії взаємодії. Важлива функція для забезпечення збереження внесених змін.

Навігація між полотнами - Унікальна можливість переходити між вкладеними підполотнами (subcanvas). Дає змогу будувати ієрархічні структури та переходити між ними за допомогою інтерфейсу користувача.

2.2.3. Архітектура збереження даних

Однією з ключових вимог до системи “Ієрархічна нотатна дошка” є можливість локального збереження даних без залучення серверної інфраструктури. Це дозволяє працювати в автономному режимі, зберігаючи повну функціональність навіть при відсутності підключення до мережі. Для реалізації цієї мети обрана файлова модель збереження даних, яка базується на використанні формату JSON.

Формат JSON є одним з найпоширеніших форматів обміну даними, оскільки легко читається як людьми, так і машинами. Як зазначено у Mozilla Developer Network [2], JSON забезпечує просту структуру даних у вигляді ключ-значення.

Формат подання. Кожне полотно зберігається у вигляді окремої директорії,

яка містить:

- файл canvas.json, що є основним дескриптором полотна;
- усі додані файли, зображення тощо, які фізично копіюються в цю папку;
- вкладені підпапки (вкладені полотна), кожна з яких має власний canvas.json.

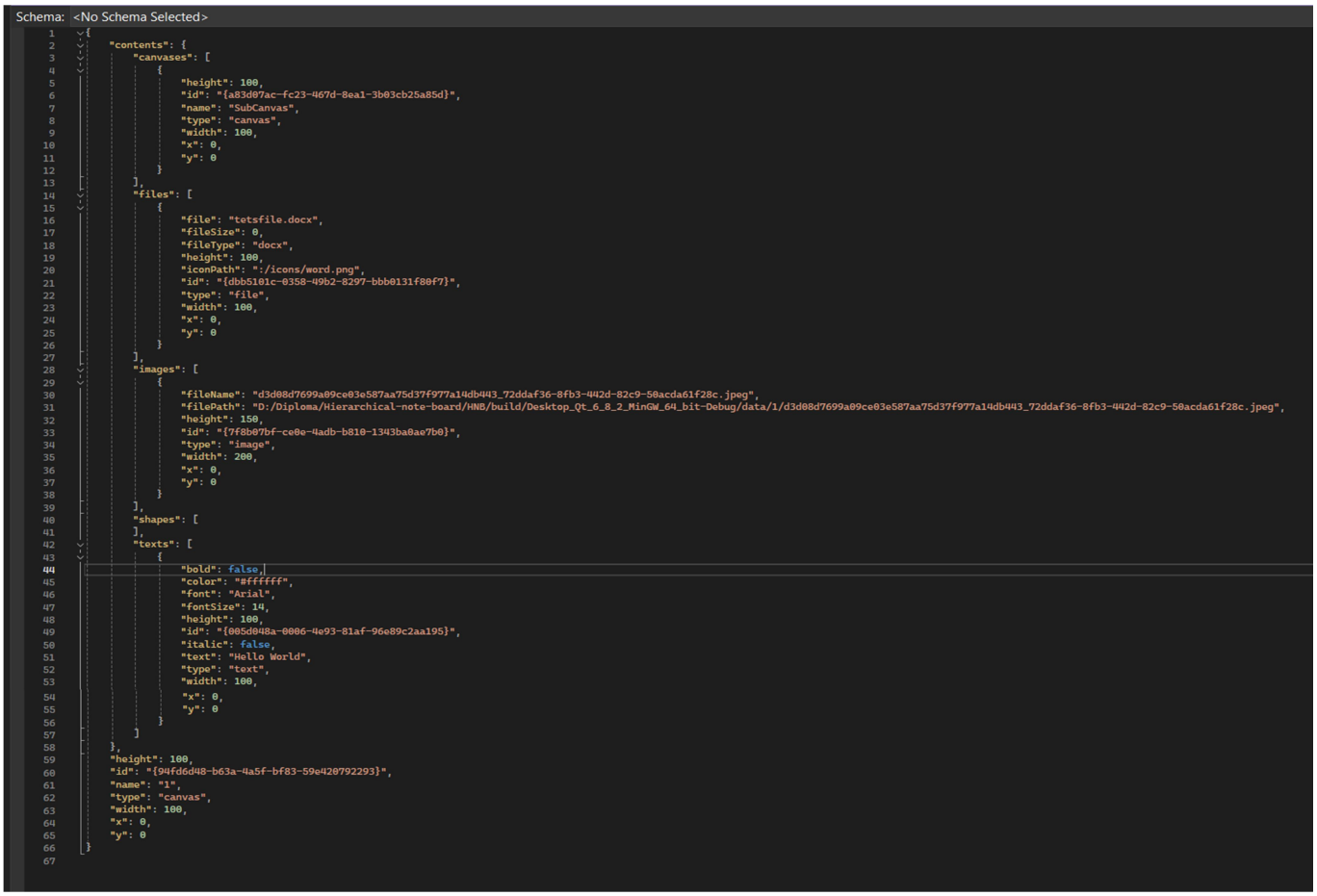


Рисунок 2.3 – Вигляд JSON файлу з мета інформацією.

Файл canvas.json містить:

- загальні метадані полотна (назва, координати, розміри);
- масиви об’єктів відповідних типів: texts, files, images, shapes, canvases;
- для кожного об’єкта — серіалізовану інформацію у форматі JSON, зокрема координати, розміри, тип, специфічні параметри (шрифт, колір, шлях до файлу, тощо).

Це дозволяє структурувати вміст полотна як ієрархію, де будь-яке вкладене полотно є самостійною сутністю, яка зберігається окремо, але логічно включається до структури батьківського.

Усі об’єкти, що розміщуються на полотні, наслідують від базового класу CanvasObject та реалізують методи:

- toJson() — для перетворення об’єкта у формат JSON;
- fromJson() — для створення об’єкта з JSON-представлення.

Цей підхід дозволяє зручно:

- зберігати всі об’єкти на диск у вигляді зрозумілої структури;
- легко відновлювати повний стан полотна при відкритті;
- виконувати операції на рівні об’єктів (пошук, оновлення, undo/redo) без потреби в складних базах даних.

Кожне полотно (Canvas) може містити інші полотна як об’єкти (CanvasObject). При серіалізації у батьківське canvas.json зберігається лише коротка метаінформація про вкладене полотно, а повна структура і вміст — у відповідній вкладеній папці.

Цей підхід реалізує дерево полотен, яке можна проходити рекурсивно при завантаженні або маніпуляції даними, при цьому фізична структура на диску повністю відображає логічну ієрархію системи.

Переваги обраної моделі:

- Прозорість і зручність — JSON легко читати, редагувати, тестувати.
- Кросплатформність — формат підтримується всіма ОС, і не вимагає встановлення БД.
- Надійність — файли зберігаються локально, без залежності від мережі.
- Масштабованість — можливо реалізувати синхронізацію в майбутньому, не змінюючи структуру.

2.3. Алгоритмічне забезпечення (undo/redo, пошук, оновлення)

Для реалізації функціоналу відміни й повтору дій у системі “Ієрархічна нотатна дошка” використано патерн Memento. Memento широко використовується для реалізації механізму undo/redo, зберігаючи внутрішній стан об’єкта без порушення інкапсуляції [3]. Основна ідея — фіксація змін об'єктів у вигляді знімків стану до і після дії.

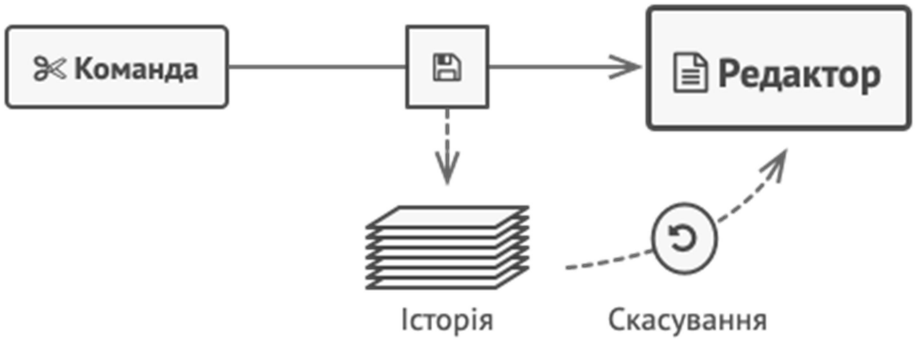


Рисунок 2.4 – Ілюстрація роботи патерна memento

Кожна взаємодія з об’єктом (додавання, видалення, редагування) зберігається у вигляді структури CanvasMemento, що включає:

- тип дії ("add", "remove", "update"),
- унікальний ідентифікатор об’єкта,
- JSON-стан до і після зміни.

Всі знімки поміщаються в стек undoStack, а після виконання відміни — у redoStack.

Алгоритм:

- undo() — відкат останньої дії, переміщення мементо у redoStack;
- redo() — повторення останньої відміни, повернення у undoStack.

Цей підхід дозволяє повністю відновлювати навіть складні дії без втрати точності.

Система підтримує швидкий пошук об’єкта за його UUID. UUID використовується як унікальний ідентифікатор для забезпечення несуперечливості іменування об’єктів у системі, навіть при злитті або переміщенні між системами [4]. Для цього метод getObjectById(QUuid) перебирає всі об’єкти поточного полотна. У разі вкладеної структури — пошук можна реалізувати рекурсивно.

Це забезпечує точність при роботі з:

- редагуванням властивостей;
- оновленням позиції чи розміру;
- викликами undo/redo.

Після пошуку об’єкта за UUID, його новий стан передається у функцію updateObject(), яка виконує:

- перевірку відповідності типу,
- оновлення посилання в контейнері об’єктів (через dynamic_pointer_cast),
- збереження полотна на диск.

Кожне оновлення також реєструється у CanvasMemento, щоб його можна було відкликати.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Інструменти та середовище розробки

3.1.1 Мова програмування

Мова програмування C++ є одним із найпотужніших інструментів розробника для створення складних програмних систем, що потребують високої продуктивності, надійності та масштабованості. Це мова з багатою історією, яка застосовується у критично важливих сферах, таких як авіаційне програмне забезпечення, фінансові системи, системи керування пам’яттю в операційних системах, графічні рушії тощо.

З моменту свого створення C++ став однією з найбільш широко використовуваних мов програмування у світі. Грамотно сконструйовані програми мовами C++ швидкі та ефективні. Мова програмування C++ вирізняється високою продуктивністю та гнучкістю, що дозволяє реалізовувати як низькорівневу, так і об’єктно-орієнтовану логіку. За словами Б. Страуструпа, сучасне програмування на C++ базується не лише на синтаксисі, а й на правильному розумінні інваріантів, абстракцій і безпечних практик. Воно дозволяє розробникам “писати програмне забезпечення, яке можна ефективно масштабувати” [5]. C++ надає стандартні бібліотеки із високим рівнем оптимізації. Він забезпечує доступ до апаратних функцій низького рівня, щоб максимально збільшити швидкість та скоротити споживання пам'яті. C++ може створювати практично будь-який вид програми: ігри, драйвери пристроїв, HPC, хмара, настільний комп'ютер, впроваджені та мобільні програми та багато іншого. Навіть бібліотеки та компілятори для інших мов програмування пишуться C++.

Однією з початкових вимог для C++ є зворотна сумісність з мовою C. В результаті програми на C++ завжди можна писати в стилі C: з необробленими покажчиками, масивами, символьними рядками із завершальним нулем та іншими функціями. Це може забезпечити високу продуктивність, але може призводити до помилок і збільшення складності. Еволюція C++ концентрується на можливостях, які значно знижують необхідність використання ідіом у стилі C. Старі об'єкти C-програмування, як і раніше, там, коли вам потрібні. Однак у сучасному коді C++ їх

потрібно менше та менше. Сучасний код на C++ простіше, безпечніше, елегантніше і так само швидко, як і раніше.

Причини вибору C++ для бекенд проєкту "Ієрархічна нотатна дошка":

- Висока швидкодія та ефективність використання ресурсів. Завдяки прямому доступу до апаратного рівня, C++ дозволяє писати продуктивний код без втрат продуктивності, що особливо важливо при обробці графіки, збереженні даних, побудові undo/redo-стека та інших внутрішніх механізмів. У сучасному C++ важливу роль у цьому відіграє заміна ручного керування пам’яттю на смарт-пойнтери (std::shared_ptr, std::unique_ptr), що значно підвищує безпечність та підтримуваність коду [6].

- Гнучкість парадигм. Мова підтримує процедурне, об’єктно-орієнтоване, шаблонне програмування та елементи функціонального стилю, що дозволяє створити гнучку архітектуру.

- Підтримка сучасних стандартів. C++ постійно оновлюється: з C++11/14/17 і до C++20 додаються нові можливості, які роблять мову ще більш потужною та зручною. Наприклад, у C++20 з'явилися потужні нововведення, зокрема діапазони (ranges), корутини (coroutines) та концепти (concepts), які роблять мову більш виразною та гнучкою [7].

Серед альтернатив могли б розглядатися C# або Java, однак вони або мають вищі системні вимоги, або тісніше прив’язані до середовища виконання (CLR/JVM), що ускладнює створення легкого кросплатформного застосунку з мінімальними залежностями.

Крім того, C++ повністю сумісна з фреймворком Qt, що дозволяє використовувати всю глибину API цього середовища та будувати як внутрішню логіку (модель Canvas, управління об’єктами, історія дій), так і зовнішню взаємодію з UI.

Таким чином, вибір C++ як основної мови реалізації бекенду пояснюється необхідністю прямого контролю над ресурсами, ефективною роботою з файлами, підтримкою об'єктної моделі, високою продуктивністю та відмінною сумісністю з Qt.

3.1.2 Qt-фреймворк та модуль QML

Qt — це широко визнаний кросплатформний фреймворк, який надає потужний інструментарій для розробки настільних і мобільних додатків. Qt створений для поєднання сучасного C++ із гнучкістю користувацького інтерфейсу, що забезпечує надійність і кросплатформність застосунків [8]. Qt надає можливість створювати застосунки для Windows, Linux, macOS з однаковою кодовою базою, що забезпечує високу кросплатформність [9]. Базується на C++ та використовує власну мета-об’єктну модель, яка дозволяє створювати динамічні сигнали/слоти, властивості та взаємодію між об’єктами.

QML — декларативна мова програмування, що дозволяє створювати адаптивні та анімовані інтерфейси в екосистемі Qt [10]. Попри те, що QML часто використовується з JavaScript або Python, рушій QML і метаоб’єктна система Qt є незалежними від мови, тому повністю сумісні з C++, що підтверджується у [11]. У нашому проєкті QML відповідає за всю візуальну частину — від структури вікон і вкладок до обробки подій на полотні.

Переваги Qt/QML у проєкті:

- Повноцінне розділення інтерфейсу та логіки. Увесь інтерфейс зосереджено в QML, а логіку реалізовано у C++ — це дозволяє паралельно розвивати обидві частини проєкту.
- Гнучкий UI. Завдяки QML можна легко реалізувати анімації, адаптивне масштабування, реакцію на кліки/перетягування та інші інтерактивні елементи.
- Підтримка моделей. За допомогою QAbstractListModel реалізовано двосторонню прив'язку між внутрішньою структурою Canvas і відображенням її у списках QML. У Qt модель QAbstractListModel дозволяє ефективно працювати з динамічними колекціями об’єктів, які відображаються у QML-компонентах, таких як ListView, Repeater або GridView [12].
- Масштабованість. Qt дозволяє легко масштабувати інтерфейс, додаючи нові компоненти або змінюючи зовнішній вигляд без значного переписування коду.

- Інструменти Qt Designer, Qt Creator і Qt Quick дозволяють швидко створювати макети, тестувати вигляд компонентів та організовувати компоненти в єдину архітектуру. Це сприяє швидкій ітерації над UI частиною і надає широкі можливості для кастомізації.
- У Qt класи QDir, QFileInfo та QFile забезпечують зручну роботу з файловою системою на будь-якій платформі, дозволяючи керувати шляхами, перевіряти існування файлів, читати та записувати дані [13].

Завдяки Qt Framework та QML, було можливо розробити багатофункціональний, візуально зручний та легкий у використанні застосунок із підтримкою ієрархічної структури полотен, вкладеного редагування, drag-and-drop та збереження даних у локальній файловій системі.

3.1.3 Система контролю версій Git

Git — це система контролю версій, яка стала стандартом де-факто для розробки програмного забезпечення. Вона забезпечує збереження історії змін, підтримку гілкування, ефективну командну роботу, об'єднання коду, перевірку змін, відкат помилкових версій тощо. Git є розподіленою системою контролю версій, яка дозволяє зберігати історію змін та підтримувати паралельну розробку [14].

Переваги використання Git:

- Локальна незалежність. Git працює як локально, так і з віддаленими репозиторіями, дозволяючи вести розробку без постійного з’єднання з сервером.
- Незалежність гілок. У проєкті активно використовувалися окремі гілки для реалізації нових можливостей (undo, draw, export), багфіксів, експериментів.
- Інтеграція з GitHub. Дозволяє організовувати code review, створювати pull-requests, розглядати зміни у вигляді діфів.
- Збереження стабільності. Завдяки гілкуванню розробка нового функціоналу не впливає на основну стабільну версію (main або release).

У рамках цього проєкту Git використовувався для:

- збереження контрольних точок перед реалізацією нових модулів (наприклад, додавання undo/redo);
- відновлення після експериментальних змін;
- логування ходу розробки;
- командної роботи з іншими членами команди UI;
- створення архіву версій для можливого релізу та документації.

Git сприяє надійності розробки, захисту коду, спрощує відкат змін та дозволяє ефективно управляти проєктом протягом усього життєвого циклу.

3.2. Опис програмної реалізації

3.2.1. Структура класів CanvasObject і спадкоємців

Одним із ключових принципів побудови програмного забезпечення у проєкті «Ієрархічна нотатна дошка» є використання об’єктно-орієнтованого підходу, що забезпечує гнучкість, повторне використання коду та легку масштабованість. Базовим елементом усіх об’єктів, що розміщуються на полотні (Canvas), є абстрактний клас CanvasObject. Поняття "canvas" активно використовується в UI/UX-проєктуванні для позначення простору, на якому користувач може розміщувати об’єкти. У HTML5 та Qt це елемент вільного розміщення [15]. Нащадки CanvasObject реалізують конкретну поведінку та зберігають специфічні дані (наприклад, для текстів, зображень, фігур тощо). Згідно з принципами SOLID, кожен клас повинен мати лише одну відповідальність, що сприяє підтримуваності системи [16].

На рисунку 3.1 представлено узагальнену візуалізацію ієрархічної структури полотен у вигляді стилізованої схеми, яка демонструє взаємозв’язок між об’єктами типу Canvas, підполотнами (Subcanvas) та вкладеними елементами — текстами й файлами. Такий підхід ілюструє логіку вкладеності та взаємодії об’єктів у системі.

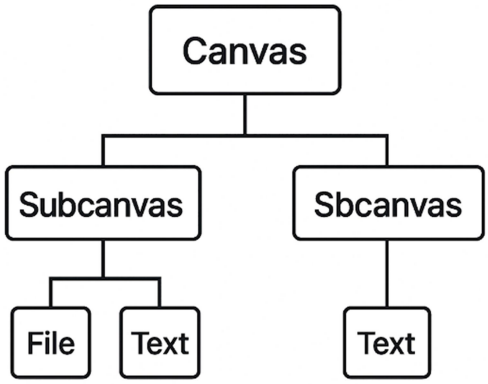


Рисунок 3.1 – ілюстрація логіки вкладеності полотна

CanvasObject — абстрактний базовий клас, що містить загальні властивості для всіх об’єктів: координати, розміри, тип, унікальний ідентифікатор, а також методи серіалізації/десеріалізації до формату JSON.

```
class CanvasObject {
public:
    CanvasObject();
    CanvasObject(const QString &type, int x, int y, int width = 100, int height = 100);
    virtual ~CanvasObject() = default;

    virtual QString getType() const;
    int getX() const;
    int getY() const;
    int getWidth() const;
    int getHeight() const;
    QUuid getId() const;

    void setX(int newX);
    void setY(int newY);
    void setWidth(int w);
    void setHeight(int h);

    virtual QJsonObject toJson() const;
    virtual void fromJson(const QJsonObject &json);

protected:
    QUuid id;
    int x, y;
    int width, height;
    QString type;
};
```

TextObject додає до CanvasObject параметри, необхідні для відображення тексту: текстовий вміст, шрифт, розмір, колір, жирність, курсив. Усе це серіалізується в JSON та дозволяє налаштування в UI.

```
class TextObject : public CanvasObject {
public:
    TextObject() = default;
    TextObject(const QString &text, int x, int y, const QString &font, int size, const QString &color);

    QString getText() const;
    void setText(const QString &text);

    QString getFont() const;
    void setFont(const QString &font);

    int getFontSize() const;
    void setFontSize(int size);

    QString getColor() const;
    void setColor(const QString &color);

    bool isBold() const;
    void setBold(bool value);

    bool isItalic() const;
    void setItalic(bool value);


    QJsonObject toJson() const override;
    void fromJson(const QJsonObject &json) override;

private:
    QString textContent;
    QString font;
    int fontSize;
    QString color;
    bool bold = false;
    bool italic = false;

};
```

FileObject зберігає інформацію про прикріплений файл: його ім’я, тип, розмір і шлях до піктограми.

```
class FileObject : public CanvasObject {
public:
    FileObject() = default;

        FileObject(const QString &fileName,
                const QString &fileType,
                qint64 fileSize,
                int x, int y, int width, int height,
                const QString &iconPath);


    QString getFileName() const;
    QString getFileType() const;
    qint64 getFileSize() const;
    QString getIconPath() const;

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject &json) override;

private:
    QString fileName;
    QString fileType;
    qint64 fileSize;
    QString iconPath;
};
```

ImageObject репрезентує зображення, прикріплене до полотна. Містить локальний шлях до зображення, його ім’я та розміри.

```
class ImageObject : public CanvasObject {
public:
    ImageObject() = default;
    ImageObject(const QString &fileName, const QString &filePath, int x, int y, int width, int height);

    QString getFileName() const;
    QString getFilePath() const;

    void setFileName(const QString &name);
    void setFilePath(const QString &path);

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject &json) override;

private:
    QString fileName;
    QString filePath;
};
```

Клас Canvas є також спадкоємцем CanvasObject. Він містить інші об’єкти та має функції для їх додавання, редагування, серіалізації та збереження у файл. Полотно може містити вкладені підполотна (тобто рекурсивну структуру).

```
class Canvas : public CanvasObject {
public:
    Canvas(const QString &name, int x, int y, const QString &directoryPath);

    QString getCanvasName() const;
    void setCanvasName(const QString &name);
    QString getAbsoluteDirectoryPath() const;

    void addText(const QString &textContent, const QString &font, int size, const QString &color);
    void addFile(const QString &sourcePath, const QString &iconPath);
    void addImageObject(const QString &sourceFilePath);
    void addShape(const QString &type, int x1, int y1, int x2, int y2, double radius,
                  const QVariantList &points, const QString &strokeColor, double strokeWidth);
    void addNestedCanvas(const QString &nestedCanvasName);

    QVector<std::shared_ptr<CanvasObject>> getAllObjects() const;
    const QVector<std::shared_ptr<Canvas>> &getNestedCanvases() const;
    const QVector<std::shared_ptr<ShapeObject>> &getShapeObjects() const;

    std::shared_ptr<CanvasObject> getObjectById(const QUuid &id) const;
    bool removeObjectById(const QUuid &id);
    bool updateObject(std::shared_ptr<CanvasObject> updated);
    bool updateObjectGeometry(const QUuid& id, int x, int y, int width, int height);
    bool updateTextProperties(const QUuid& id, const QString& text, const QString& font,
                              int fontSize, const QString& color, bool bold, bool italic);
    bool updateShapeProperties(const QUuid& id, int x, int y, int x2, int y2, float radius,
                               const QVariantList& points, const QString& color, float strokeWidth);

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject &json) override;
    void saveToDisk() const;
    bool loadFromDisk();



private:
    QString canvasName;
    QString directoryPath;
    QVector<std::shared_ptr<TextObject>> textObjects;
    QVector<std::shared_ptr<FileObject>> fileObjects;
    QVector<std::shared_ptr<ImageObject>> imageObjects;
    QVector<std::shared_ptr<Canvas>> nestedCanvases;
    QVector<std::shared_ptr<ShapeObject>> shapeObjects;

    QString generateUniqueFilename(const QString &originalName) const;
};
```

ShapeObject — абстрактний клас для геометричних фігур. Містить колір та товщину лінії.

```
class ShapeObject : public CanvasObject {
public:
    ShapeObject();
    virtual ~ShapeObject() = default;

    void setStrokeColor(const QColor& color);
    QColor getStrokeColor() const;

    void setStrokeWidth(double width);
    double getStrokeWidth() const;

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject& json) override;

protected:
    QColor strokeColor = Qt::black;
    double strokeWidth = 2.0;
};
```

Його спадкоємцями є LineObject, ArrowObject і FreeDrawObject, кожен із яких реалізує власну форму та логіку серіалізації.

```
class LineObject : public ShapeObject {
public:
    LineObject();
    LineObject(int x, int y, int x2, int y2);

    QPointF getEndPoint() const;
    void setEndPoint(const QPointF& point);

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject& json) override;

private:
    QPointF endPoint;
};

class ArrowObject : public ShapeObject {
public:
    ArrowObject();
    ArrowObject(int x, int y, int x2, int y2);

    QPointF getEndPoint() const;
    void setEndPoint(const QPointF& point);

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject& json) override;

private:
    QPointF endPoint;
};
```

```
class FreeDrawObject : public ShapeObject {
public:
    FreeDrawObject();
    FreeDrawObject(const QVector<QPointF>& points);

    QVector<QPointF> getPoints() const;
    void setPoints(const QVector<QPointF>& pts);

    QJsonObject toJson() const override;
    void fromJson(const QJsonObject& json) override;

private:
    QVector<QPointF> pathPoints;
};
```

Ієрархія класів у цій системі дозволяє гнучко керувати різнорідними об’єктами на полотні, зберігати їх у локальному файлі, редагувати їхні властивості через UI та підтримувати історію змін. Реалізація через спільний базовий клас CanvasObject спрощує обробку візуальних елементів, а шаблон проектування "композит" дає змогу реалізувати вкладені підполотна без додаткових складнощів.

3.2.2. Клас CanvasManager і зв’язок з GUI

Клас CanvasManager є ключовою ланкою між об’єктною моделлю застосунку та його графічним інтерфейсом користувача, реалізованим у QML. Він реалізує шаблон "контролер" і забезпечує зв’язок між даними та UI, а також дозволяє керувати кількома відкритими полотнами одночасно.

```
class CanvasManager : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString currentCanvasName READ getCurrentCanvasName NOTIFY currentCanvasChanged)
    Q_PROPERTY(QVariantList openCanvasEntries READ getOpenCanvasEntries NOTIFY canvasListChanged)
    Q_PROPERTY(CanvasObjectModel* canvasModel READ canvasModel NOTIFY canvasModelChanged)

public:
    explicit CanvasManager(QObject *parent = nullptr);

    Q_INVOKABLE void createNewCanvas(const QString &name, const QString &directoryPath);
    Q_INVOKABLE void openCanvas(const QString &path);
    Q_INVOKABLE void closeCanvasById(const QString &id);
    Q_INVOKABLE bool setCurrentCanvasById(const QString &id);
    Q_INVOKABLE QString getCurrentCanvasName() const;
     Q_INVOKABLE QVariantList getOpenCanvasEntries() const;


    Q_INVOKABLE void addTextToCurrent(const QString &text, const QString &font, int size, const QString &color);
    Q_INVOKABLE void addFileToCurrent(const QString &filePath, const QString &iconPath);
    Q_INVOKABLE void addCanvasToCurrent(const QString &name);
    Q_INVOKABLE void addImageToCurrent(const QString &filePath);
    Q_INVOKABLE void addShapeToCurrent(
        const QString &type,
        int x1, int y1,
        int x2 = 0, int y2 = 0,
        double radius = 0.0,
        const QVariantList &points = {},
        const QString &strokeColor = "#000000",
        double strokeWidth = 2.0
        );

    Q_INVOKABLE bool updateObjectGeometry(const QString &id, int x, int y, int width, int height);

    Q_INVOKABLE bool removeObjectFromCurrent(const QString &id);
    Q_INVOKABLE bool updateTextProperties(const QString &id, const QString &text, const QString &font, int
fontSize, const QString &color, bool bold, bool italic);
    Q_INVOKABLE QStringList listOpenCanvases() const;
    Q_INVOKABLE bool updateShapeProperties(
        const QString &id,
        int x, int y, int x2, int y2,
        float radius,
        const QVariantList &points,
        const QString &color,
        float strokeWidth
        );

    CanvasObjectModel* canvasModel() const;

signals:
    void currentCanvasChanged();
    void canvasListChanged();
    void canvasModelChanged();

private:
    QVector<std::shared_ptr<Canvas>> openCanvases;
    std::shared_ptr<Canvas> currentCanvas;
    CanvasObjectModel* m_canvasModel = nullptr;
    std::shared_ptr<CanvasObject> findObjectById(const QUuid &id) const;
};
```

CanvasManager відповідає за створення, відкриття, закриття та переключення між різними полотнами. У класі підтримується список відкритих полотен openCanvases, а також активне полотно currentCanvas. Кожне полотно зберігається на диску у вигляді JSON-структури, а взаємодія з ним відбувається через методи класу.

Приклад створення нового полотна:

```
Q_INVOKABLE void createNewCanvas(const QString &name, const QString &directoryPath);
```

Завдяки властивостям Q_PROPERTY, об'єкти CanvasManager зручно використовуються в QML через прив'язки. Наприклад, властивість currentCanvasName автоматично оновлюється в інтерфейсі при зміні активного полотна:

```
Q_PROPERTY(QString currentCanvasName READ getCurrentCanvasName NOTIFY
currentCanvasChanged)
```

Клас має декілька Q_INVOKABLE методів для виклику з інтерфейсу:

```
Q_INVOKABLE void addTextToCurrent(const QString &text, const QString &font, int size, const QString &color);
Q_INVOKABLE bool updateTextProperties(const QString &id, const QString &text, const QString &font, int
fontSize, const QString &color, bool bold, bool italic);
```

CanvasManager також містить вказівник на модель CanvasObjectModel, яка є проксі-шаром між внутрішніми об'єктами полотна та візуальним представленням у QML. Коли полотно змінюється або додається новий об'єкт, модель оновлюється:

```
if (m_canvasModel && currentCanvas)
    m_canvasModel->setObjects(currentCanvas->getAllObjects());
emit canvasModelChanged();
```

Особливу увагу приділено методові setCurrentCanvasById, який дозволяє рекурсивно знаходити вкладене полотно за ID, відкривати його та робити його активним.

```
Q_INVOKABLE bool setCurrentCanvasById(const QString &id)
```

Інтерфейс CanvasManager дозволяє з QML додавати різні об'єкти:

- зображення

  ```
  Q_INVOKABLE void addImageToCurrent(const QString &filePath)
  ```

- канви

  ```
  Q_INVOKABLE void addCanvasToCurrent(const QString &name);
  ```

- фігури

```
    Q_INVOKABLE void addShapeToCurrent(
        const QString &type,
        int x1, int y1,
        int x2 = 0, int y2 = 0,
        double radius = 0.0,
        const QVariantList &points = {},
        const QString &strokeColor = "#000000",
        double strokeWidth = 2.0
);
```

Ці методи додають об’єкти на поточне полотно, оновлюють модель і зберігають зміни на диск через saveToDisk().

CanvasManager відіграє центральну роль у реалізації логіки керування даними, поєднуючи внутрішні структури класів із графічним інтерфейсом.

Він дає змогу маніпулювати об’єктами на полотні в реальному часі, зберігати зміни та підтримує зручну інтеграцію з QML-компонентами. Завдяки модульній реалізації та використанню інструментів Qt (Q_PROPERTY, signals/slots), цей клас забезпечує гнучкість, масштабованість та ефективну взаємодію між рівнями архітектури програми.

```
signals:
    void currentCanvasChanged();
    void canvasListChanged();
  void canvasModelChanged();
```

Механізм "сигналів і слотів" у Qt забезпечує зручну асинхронну взаємодію між об'єктами, особливо у GUI-додатках [17].

3.2.3. Реалізація undo/redo (CanvasMemento)

Однією з ключових функцій зручності користувача є підтримка скасування та повтору дій — механізм undo/redo. У даному застосунку ця функціональність реалізована за допомогою класу CanvasMemento та двох стеків: undoStack і redoStack, які зберігають історію змін об'єктів.

CanvasMemento:

```
class CanvasMemento {
public:
    CanvasMemento() = default;
    CanvasMemento(const QString& action, const QUuid& id,
                  const QJsonObject& before, const QJsonObject& after);

    QString getAction() const;
    QUuid getObjectId() const;
    QJsonObject getBeforeState() const;
    QJsonObject getAfterState() const;

private:
    QString action;
    QUuid objectId;
    QJsonObject beforeState;
    QJsonObject afterState;
};
```

Об'єкт класу зберігає:

- дію, яка була виконана (add, remove, update);
- ідентифікатор об'єкта;
- стан об'єкта до та після операції.
- Захоплення змін у Canvas
- При кожній зміні створюється відповідний CanvasMemento:

```
CanvasMemento m("update", newObj->getId(), oldObj->toJson(), newObj->toJson());
undoStack.push(m);
redoStack.clear();
```

Аналогічно викликаються методи для objectAdd() та objectRemove().

Переваги реалізації:

- Гнучкість — всі об'єкти серіалізуються у форматі JSON, що дозволяє зберігати та відновлювати стан незалежно від типу.
- Розширюваність — нові дії або об'єкти легко інтегруються через додавання підтримки в CanvasMemento.
- Зручність інтеграції з GUI — достатньо викликати undo() або redo() із кнопки

в інтерфейсі.

3.2.4. Збереження та відновлення об'єктів (фабрика)

Шаблон проектування Factory дає змогу створювати об’єкти без прямої прив’язки до конкретного класу, що підвищує гнучкість системи [18]. Щоб забезпечити коректне відновлення об’єктів після завантаження JSON-структури, в системі реалізовано механізм фабрики. Його завдання — автоматично створювати об'єкти потрібного типу (TextObject, FileObject, ImageObject, ShapeObject тощо) залежно від вказаного типу об'єкта в JSON.

Це дозволяє інкапсулювати логіку створення об'єктів в одному місці і спростити код завантаження:

```
#include <QJsonObject>
#include <memory>
#include "canvasobject.h"
#include "textobject.h"
#include "fileobject.h"
#include "imageobject.h"
#include "lineobject.h"
#include "arrowobject.h"
#include "circleobject.h"
#include "freedrawobject.h"

class CanvasFactory {
public:
    // Створює CanvasObject на основі JSON-опису
    static std::shared_ptr<CanvasObject> createFromJson(const QJsonObject& json);
};
```

Цей підхід дозволяє централізовано керувати створенням і розширювати підтримку нових типів об’єктів без дублювання коду в інших частинах проєкту.

Під час завантаження JSON-файлу відповідні масиви (texts, files, images, shapes) проходяться у циклі, і для кожного елемента викликається CanvasFactory::createFromJson(json), після чого результат додається до відповідного списку об’єктів у Canvas.

Таким чином, фабрика виступає важливим елементом архітектури, що сприяє

підтримуваності та масштабованості коду.

3.2.5. Взаємодія з QML через CanvasObjectModel

Для забезпечення повноцінної інтеграції логіки застосунку з інтерфейсом, реалізованим у QML, у системі використовується клас CanvasObjectModel, який є нащадком QAbstractListModel. Цей клас виступає мостом між C++-поданням об'єктів на полотні та їх візуальним відображенням у QML-компонентах.

Модель CanvasObjectModel дає змогу QML-інтерфейсу відображати та взаємодіяти з динамічним списком об'єктів (CanvasObject) на активному полотні (Canvas). Вона забезпечує:

- передачу властивостей (тип, координати, розмір, ідентифікатор, текст, колір тощо);
- оновлення інтерфейсу при зміні полотна або його об'єктів;
- можливість зв’язування (binding) у QML з такими компонентами як Repeater, ListView, GridView, тощо.

  Основні методи та ролі:

```
class CanvasObjectModel : public QAbstractListModel {
    Q_OBJECT

public:
    enum Roles {
        ObjectIdRole = Qt::UserRole + 1,
        TypeRole,
        XRole,
        YRole,
        WidthRole,
        HeightRole,
        FileNameRole,
        FilePathRole,
        FileTypeRole,
        FileSizeRole,
        IconPathRole,
        TextRole,
        FontRole,
        FontSizeRole,
        ColorRole,
        BoldRole,
        ItalicRole,
        NameRole,
        StrokeColorRole,
        StrokeWidthRole,
        X2Role,
        Y2Role,
        RadiusRole,
        PointsRole
    };


    explicit CanvasObjectModel(QObject *parent = nullptr);

    int rowCount(const QModelIndex &parent = QModelIndex()) const override;
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const override;
    QHash<int, QByteArray> roleNames() const override;

    void setObjects(const QVector<std::shared_ptr<CanvasObject>> &objects);
    std::shared_ptr<CanvasObject> getObject(int index) const;
    Q_INVOKABLE QVariantMap get(int index) const;

private:
    QVector<std::shared_ptr<CanvasObject>> m_objects;
```

Модель використовує перелік ролей, які дозволяють QML отримати відповідні властивості.

Ось короткий опис значення кожної ролі в CanvasObjectModel:

Основні ролі:

- ObjectIdRole - унікальний ідентифікатор об'єкта (QUuid).
- TypeRole - тип об'єкта (наприклад: "text", "file", "image", "line").
- XRole - координата X розташування об'єкта.
- YRole - координата Y розташування об'єкта.
- WidthRole - ширина об'єкта.
- HeightRole - висота об'єкта.

Ролі для файлів:

- FileNameRole - назва прикріпленого файлу.
- FilePathRole - повний шлях до файлу (для ImageObject).
- FileTypeRole - тип файлу (наприклад: "pdf", "jpg").
- FileSizeRole - розмір файлу в байтах.
- IconPathRole - шлях до іконки, яка візуалізує файл.

Ролі для текстових об'єктів:

- TextRole - вміст текстового блоку.
- FontRole - назва шрифту.
- FontSizeRole - розмір шрифту.
- ColorRole - колір тексту (у hex-форматі: #000000).
- BoldRole - жирність шрифту (true/false).
- ItalicRole - курсивність шрифту (true/false).

Ролі для фігур (ShapeObject):

- StrokeColorRole - колір обводки фігури.
- StrokeWidthRole - товщина лінії фігури.
- X2Role - координата X кінцевої точки (для LineObject, ArrowObject).
- Y2Role - координата Y кінцевої точки.
- RadiusRole - радіус фігури (CircleObject).

- PointsRole - список точок (QVector<QPointF>) для вільного малювання (FreeDrawObject).

Клас CanvasManager зберігає об’єкт типу CanvasObjectModel і оновлює його щоразу, коли змінюється поточне полотно:

```
if (m_canvasModel && currentCanvas)
        m_canvasModel->setObjects(currentCanvas->getAllObjects());
emit canvasModelChanged();
```

Це дає змогу автоматично оновити відображення в UI після додавання, видалення чи зміни об'єктів на полотні.

Приклад використання в QML:

```
ListView {
    model: canvasManager.canvasModel
    delegate: Rectangle {
        width: model.width
        height: model.height
        Text {
            text: model.text
        }
    }
}
```

У цьому прикладі ListView автоматично показує всі об’єкти, а Text відображає вміст текстових об'єктів.

Переваги підходу

- Модульність — логіка моделі не прив'язана до інтерфейсу.
- Автоматичне оновлення UI — при зміні вмісту CanvasObjectModel.
- Гнучкість — через підтримку ролей можна легко розширити інтерфейс новими властивостями.

Використання CanvasObjectModel забезпечує масштабовану й реактивну взаємодію між C++-базованою логікою застосунку та QML-інтерфейсом. Кожен об'єкт моделі презентується як окрема сутність з чітко визначеними параметрами, що дозволяє гнучко й ефективно реалізовувати відображення, редагування та маніпуляції візуальними об'єктами в користувацькому інтерфейсі.

ВИСНОВКИ

У процесі розробки кросплатформного застосунку “Ієрархічна нотатна дошка” було проведено ґрунтовне дослідження сучасного стану цифрових інструментів для створення й збереження нотаток. Аналіз існуючих рішень дозволив виявити ключові обмеження, серед яких — відсутність повної автономності, слабка підтримка локального збереження без серверної складової та складність роботи з вкладеними структурами. Ці фактори визначили напрямок розробки системи, орієнтованої на зручність, гнучкість і незалежність користувача від зовнішніх сервісів.

Архітектура застосунку була побудована з урахуванням об'єктно-орієнтованого підходу, що дозволило забезпечити високу модульність, масштабованість та простоту розширення. Центральним елементом структури став базовий клас CanvasObject, від якого наслідуються всі типи об’єктів на полотні — тексти, файли, зображення, фігури, вкладені полотна. Завдяки цьому забезпечено уніфікований механізм серіалізації об'єктів, їх збереження та відновлення у форматі JSON.

Особливу увагу приділено розробці класу CanvasManager, який виступає сполучною ланкою між логікою програми та інтерфейсом користувача. Через зв'язок із QML через CanvasObjectModel, реалізовано динамічне оновлення UI при будь-яких змінах у структурі полотна. Також реалізовано повну підтримку операцій undo/redo з використанням патерну “memento”, що дозволяє зберігати історію змін та віднавлювати попередні стани об'єктів.

Для забезпечення зручної взаємодії з користувачем застосунок підтримує створення вкладених полотен, додавання графічних елементів, імпорт зображень і файлів, а також збереження всіх даних у локальну файлову систему. Це робить систему повністю автономною — вона не потребує постійного інтернет-з'єднання чи зовнішніх серверів.

У ході розробки було використано сучасні інструменти: мову програмування C++ та Qt-фреймворк з модулем QML, що дозволило досягти високої продуктивності, кросплатформності та сучасного інтерфейсу.

Система Git забезпечувала контроль версій, що дало змогу підтримувати гнучкий і надійний процес розробки.

Підсумовуючи, реалізований застосунок "Ієрархічна нотатна дошка" є потужним інструментом для ведення локальних ієрархічних записів із гнучкою структурою та широкими можливостями модифікації. Він може ефективно застосовуватись для персонального планування, творчих проєктів, організації знань та замінити складніші хмарні аналоги у випадках, де потрібна автономність, простота й контроль над даними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Local Data Storage and Security [Електронний ресурс] – OWASP. – https://owasp.org/www-project-mobile-top-10

[2] JSON: JavaScript Object Notation [Електронний ресурс] – Mozilla. – Режим доступу: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

[3] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994.

[4] RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace – IETF. – https://datatracker.ietf.org/doc/html/rfc4122

[5] Stroustrup B. – Programming: Principles and Practice Using C++ (2024)

[6]Gregoire M. – *Professional C++* (2024)

[7]Deitel P. C++20 for Programmers. – Deitel & Associates, 2020. – 1056 p.

[8] Qt Company Docs (2023)

[9] Introduction to Qt [Електронний ресурс] – Qt Company. – Режим доступу: https://doc.qt.io/

[10] Qt Quick and QML [Електронний ресурс] – Qt Company. – Режим доступу: https://doc.qt.io/qt-6/qmlapplications.html

[11]Fitzpatrick M. – *Create GUI Applications with Qt* (2024)

[12] QAbstractListModel Class [Електронний ресурс] – Qt Documentation – https://doc.qt.io/qt-6/qabstractlistmodel.html

[13] QFile Class Reference [Електронний ресурс] – Qt Documentation. – https://doc.qt.io/qt-6/qfile.html

[14] Chacon S., Straub B. Pro Git. – Apress, 2014. – https://git-scm.com/book/en/v2

[15] HTML5 Canvas API [Електронний ресурс] – MDN Web Docs. – https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

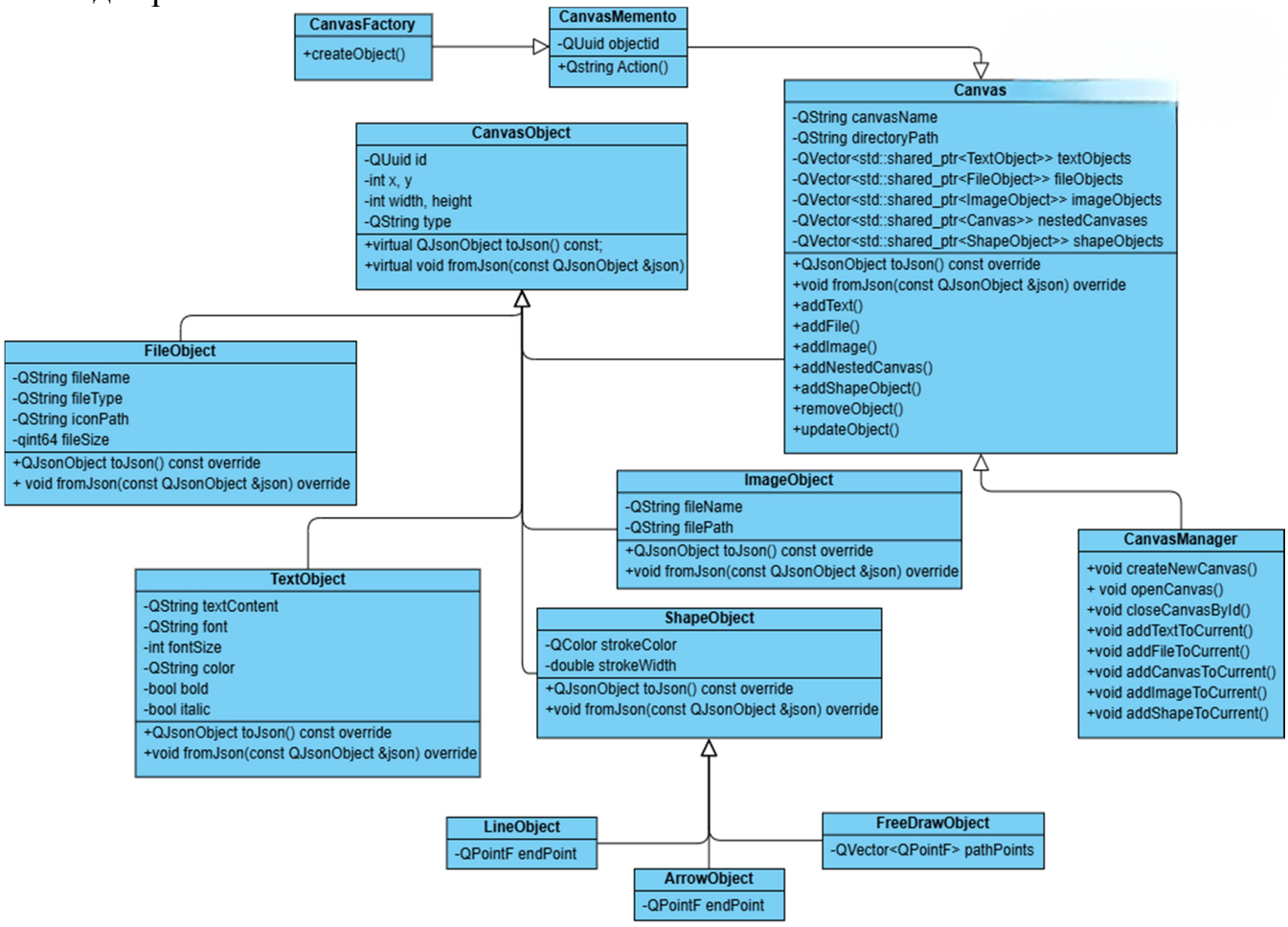
[16] Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017.

[17] Signals and Slots in Qt [Електронний ресурс] – https://doc.qt.io/qt-6/signalsandslots.html

[18] Freeman E., Robson E. Head First Design Patterns. – O'Reilly Media, 2021

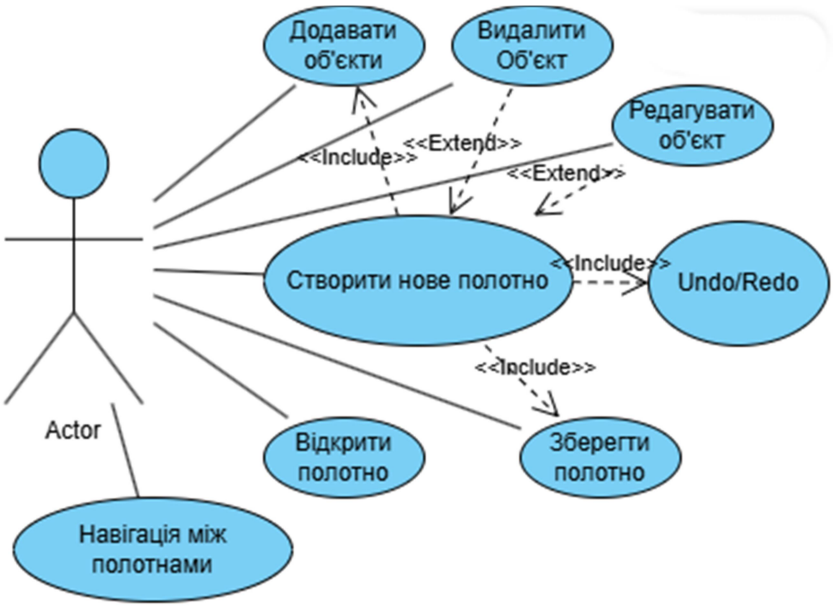
ДОДАТОК А

UML діаграма класів:



ДОДАТОК Б

Use case діаграма:



ДОДАТОК В

Реалізація класу CanvasObject:

```
#include "canvasobject.h"

CanvasObject::CanvasObject()
    : id(QUuid::createUuid()), x(0), y(0), width(100), height(100), type("unknown") {}

CanvasObject::CanvasObject(const QString &type, int x, int y, int width, int height)
    : id(QUuid::createUuid()), x(x), y(y), width(width), height(height), type(type) {}

QString CanvasObject::getType() const { return type; }
int CanvasObject::getX() const { return x; }
int CanvasObject::getY() const { return y; }
int CanvasObject::getWidth() const { return width; }
int CanvasObject::getHeight() const { return height; }
QUuid CanvasObject::getId() const { return id; }

void CanvasObject::setX(int newX) { x = newX; }
void CanvasObject::setY(int newY) { y = newY; }
void CanvasObject::setWidth(int w) { width = w; }
void CanvasObject::setHeight(int h) { height = h; }

QJsonObject CanvasObject::toJson() const {
    QJsonObject obj;
    obj["id"] = id.toString();
    obj["type"] = type;
    obj["x"] = x;
    obj["y"] = y;
    obj["width"] = width;
    obj["height"] = height;
    return obj;
}

void CanvasObject::fromJson(const QJsonObject &json) {
    if (json.contains("id")) id = QUuid::fromString(json["id"].toString());
    if (json.contains("type")) type = json["type"].toString();
    if (json.contains("x")) x = json["x"].toInt();
    if (json.contains("y")) y = json["y"].toInt();
    if (json.contains("width")) width = json["width"].toInt();
    if (json.contains("height")) height = json["height"].toInt();
}
```

ДОДАТОК Г

Реалізація класу TextObject:

```
#include "textobject.h"

TextObject::TextObject(const QString &text, int x, int y, const QString &font, int size, const QString &color)
    : CanvasObject("text", x, y), textContent(text), font(font), fontSize(size), color(color) {}

QString TextObject::getText() const { return textContent; }
void TextObject::setText(const QString &text) { textContent = text; }

QString TextObject::getFont() const { return font; }
void TextObject::setFont(const QString &f) { font = f; }

int TextObject::getFontSize() const { return fontSize; }
void TextObject::setFontSize(int size) { fontSize = size; }

QString TextObject::getColor() const { return color; }
void TextObject::setColor(const QString &c) { color = c; }

bool TextObject::isBold() const {    return bold;}

void TextObject::setBold(bool value) {bold = value;}

bool TextObject::isItalic() const {return italic;}

void TextObject::setItalic(bool value) {italic = value;}

QJsonObject TextObject::toJson() const {
    QJsonObject obj = CanvasObject::toJson();
    obj["text"] = textContent;
    obj["font"] = font;
    obj["fontSize"] = fontSize;
    obj["color"] = color;
    obj["bold"] = bold;
    obj["italic"] = italic;

    return obj;
}
void TextObject::fromJson(const QJsonObject &json) {
    CanvasObject::fromJson(json);
    if (json.contains("text")) textContent = json["text"].toString();
    if (json.contains("font")) font = json["font"].toString();
    if (json.contains("fontSize")) fontSize = json["fontSize"].toInt();
    if (json.contains("color")) color = json["color"].toString();
    if (json.contains("bold")) bold = json["bold"].toBool();
    if (json.contains("italic")) italic = json["italic"].toBool();
```

ДОДАТОК Ґ

Реалізація класу FileObject:

```
#include "fileobject.h"
#include <QFileInfo>

FileObject::FileObject(const QString &fileName,
                       const QString &fileType,
                       qint64 fileSize,
                       int x, int y, int width, int height,
                       const QString &iconPath)
    : CanvasObject("file", x, y, width, height),
    fileName(fileName),
    fileType(fileType),
    fileSize(fileSize),
    iconPath(iconPath){}



QString FileObject::getFileName() const { return fileName; }
QString FileObject::getFileType() const { return fileType; }
qint64 FileObject::getFileSize() const { return fileSize; }
QString FileObject::getIconPath() const { return iconPath; }

QJsonObject FileObject::toJson() const {
    QJsonObject obj = CanvasObject::toJson();
    obj["file"] = fileName;  // тільки ім’я файлу
    obj["fileType"] = fileType;
    obj["fileSize"] = static_cast<int>(fileSize);
    obj["iconPath"] = iconPath;
    return obj;
}

void FileObject::fromJson(const QJsonObject &json) {
    CanvasObject::fromJson(json);
    if (json.contains("file")) fileName = json["file"].toString();
    if (json.contains("fileType")) fileType = json["fileType"].toString();
    if (json.contains("fileSize")) fileSize = json["fileSize"].toInt();
    if (json.contains("iconPath")) iconPath = json["iconPath"].toString();
}
```

ДОДАТОК Д

Реалізація класу ImageObject:

```
#include "imageobject.h"

ImageObject::ImageObject(const QString &fileName, const QString &filePath, int x, int y, int width, int height)
    : CanvasObject("image", x, y, width, height),
    fileName(fileName),
    filePath(filePath)
{}

QString ImageObject::getFileName() const { return fileName; }
QString ImageObject::getFilePath() const { return filePath; }

void ImageObject::setFileName(const QString &name) { fileName = name; }
void ImageObject::setFilePath(const QString &path) { filePath = path; }

QJsonObject ImageObject::toJson() const {
    QJsonObject obj = CanvasObject::toJson();
    obj["fileName"] = fileName;
    obj["filePath"] = filePath;
    return obj;
}

void ImageObject::fromJson(const QJsonObject &json) {
    CanvasObject::fromJson(json);
    if (json.contains("fileName")) fileName = json["fileName"].toString();
    if (json.contains("filePath")) filePath = json["filePath"].toString();
}
```

ДОДАТОК Е

Реалізація класу Canvas:

```
#include "canvas.h"
#include <QJsonArray>
#include <QDir>
#include <QFile>
#include <QTextStream>
#include <QJsonDocument>
#include <QFileInfo>
#include <QUuid>
#include "lineobject.h"
#include "arrowobject.h"
#include "circleobject.h"
#include "freedrawobject.h"

Canvas::Canvas(const QString &name, int x, int y, const QString &directoryPath)
    : CanvasObject("canvas", x, y), canvasName(name), directoryPath(directoryPath) {
    QDir dir(directoryPath);
    if (!dir.exists()) {
        dir.mkpath(".");
    }
}

QString Canvas::getCanvasName() const {
    return canvasName;
}

void Canvas::setCanvasName(const QString &name) {
    canvasName = name;
}

QString Canvas::generateUniqueFilename(const QString &originalName) const {
    QString baseName = QFileInfo(originalName).completeBaseName();
    QString extension = QFileInfo(originalName).suffix();
    QString uniqueName = baseName + "_" + QUuid::createUuid().toString(QUuid::WithoutBraces) + "." + extension;
    return uniqueName;
}

void Canvas::addShape(const QString &type, int x1, int y1, int x2, int y2, double radius, const QVariantList &points,
const QString &strokeColor, double strokeWidth) {
    std::shared_ptr<ShapeObject> shape = nullptr;
```

```
    if (type == "line") {
        shape = std::make_shared<LineObject>(x1, y1, x2, y2);
    } else if (type == "arrow") {
        shape = std::make_shared<ArrowObject>(x1, y1, x2, y2);
    } else if (type == "circle") {
        shape = std::make_shared<CircleObject>(x1, y1, radius);
    } else if (type == "freedraw") {
        QVector<QPointF> pts;
        for (const QVariant &val : points) {
            QVariantMap m = val.toMap();
            pts.append(QPointF(m["x"].toDouble(), m["y"].toDouble()));
        }
        shape = std::make_shared<FreeDrawObject>(pts);
    }

    if (shape) {
        shape->setStrokeColor(QColor(strokeColor));
        shape->setStrokeWidth(strokeWidth);
        shapeObjects.append(shape);
    }
}
const QVector<std::shared_ptr<ShapeObject>>& Canvas::getShapeObjects() const {
    return shapeObjects;
}

void Canvas::addFile(const QString &sourcePath, const QString &iconPath) {
    QFileInfo info(QUrl(sourcePath).toLocalFile());
    if (!info.exists() || !info.isFile()) {
        qWarning() << "[Canvas::addFile] Source file does not exist:" << sourcePath;
        return;
    }

    QString extension = info.suffix().toLower();
    QString destFileName = info.fileName();
    QString destPath = getAbsoluteDirectoryPath() + "/" + destFileName;

    if (!QFile::copy(info.absoluteFilePath(), destPath)) {
        qWarning() << "[Canvas::addFile] Failed to copy file to:" << destPath;
        return;
    }

    QString resolvedIcon = iconPath;
```

if

```
(extension == "pdf")
        resolvedIcon = ":/icons/pdf.png";
    else if (extension == "docx" || extension == "doc")
        resolvedIcon = ":/icons/word.png";
    else if (extension == "png" || extension == "jpg")
        resolvedIcon = ":/icons/image.png";
    else
        resolvedIcon = ":/icons/file.png";

    auto fileObj = std::make_shared<FileObject>(
        destFileName, extension, info.size(),
        0, 0, 100, 100,
        resolvedIcon
        );
    fileObjects.append(fileObj);

    qDebug() << "[Canvas::addFile] File added to canvas:" << destFileName;
}



void Canvas::addText(const QString &textContent, const QString &font, int size, const QString &color) {
    auto text = std::make_shared<TextObject>(textContent, x, y, font, size, color);
    textObjects.append(text);
}


void Canvas::addNestedCanvas(const QString &nestedCanvasName) {
    QString nestedPath = directoryPath + "/" + nestedCanvasName;
    auto nestedCanvas = std::make_shared<Canvas>(nestedCanvasName, x, y, nestedPath);
    nestedCanvases.append(nestedCanvas);
}


void Canvas::addImageObject(const QString &sourceFilePath) {
    QString localPath = sourceFilePath;
    if (localPath.startsWith("file:///")) {
        localPath = QUrl(localPath).toLocalFile();
    }

    QFileInfo srcInfo(localPath);
    if (!srcInfo.exists() || !srcInfo.isFile()) {
```

```
        qWarning() << "[Canvas::addImageObject] Source file does not exist:" << localPath;
        return;
    }

    QString uniqueFileName = generateUniqueFilename(srcInfo.fileName());
    QString destDir = getAbsoluteDirectoryPath();
    QString destPath = destDir + "/" + uniqueFileName;

    if (!QFile::copy(srcInfo.absoluteFilePath(), destPath)) {
        qWarning() << "[Canvas::addImageObject] Failed to copy image to:" << destPath;
        return;
    }

    auto imageObj = std::make_shared<ImageObject>(uniqueFileName, destPath, 0, 0, 200, 150);
    imageObjects.append(imageObj);
}

QJsonObject Canvas::toJson() const {
    QJsonObject canvasJson = CanvasObject::toJson();
    canvasJson["name"] = canvasName;

    QJsonArray filesJson, textsJson, canvasesJson, imagesJson, shapesJson;

    // Файли
    for (const auto &file : fileObjects)
        filesJson.append(file->toJson());

    // Текстові блоки
    for (const auto &text : textObjects)
        textsJson.append(text->toJson());

    // Вкладені полотна (тільки як посилання)
    for (const auto &canvas : nestedCanvases) {
        QJsonObject meta = canvas->CanvasObject::toJson(); // не рекурсивно!
        meta["name"] = canvas->getCanvasName();
        canvasesJson.append(meta);
    }
    // images
    for (const auto &img : imageObjects) {
        imagesJson.append(img->toJson());
    }
```

```
    for (const auto& shape : shapeObjects)
        shapesJson.append(shape->toJson());

    QJsonObject contents;
    contents["files"] = filesJson;
    contents["texts"] = textsJson;
    contents["canvases"] = canvasesJson;
    contents["images"] = imagesJson;
    contents["shapes"] = shapesJson;

    canvasJson["contents"] = contents;
    return canvasJson;
}


void Canvas::fromJson(const QJsonObject &json) {
    CanvasObject::fromJson(json);

    if (json.contains("name"))
        canvasName = json["name"].toString();

    if (json.contains("contents")) {
        QJsonObject contents = json["contents"].toObject();

        // --- FILES ---
        if (contents.contains("files")) {
            for (const QJsonValue &val : contents["files"].toArray()) {
                auto obj = std::make_shared<FileObject>();
                obj->fromJson(val.toObject());
                fileObjects.append(obj);
            }
        }

        // --- TEXTS ---
        if (contents.contains("texts")) {
            for (const QJsonValue &val : contents["texts"].toArray()) {
                auto obj = std::make_shared<TextObject>();
                obj->fromJson(val.toObject());
                textObjects.append(obj);
            }
        }
```

```
// --- CANVASES ---
if (contents.contains("canvases")) {
    for (const QJsonValue &val : contents["canvases"].toArray()) {
        QJsonObject meta = val.toObject();
        QString name = meta["name"].toString();
        QString nestedPath = directoryPath + "/" + name;

        auto nestedCanvas = std::make_shared<Canvas>(name, 0, 0, nestedPath);
        nestedCanvas->fromJson(meta);  // для позиції, id, розмірів
        nestedCanvas->loadFromDisk();  // для структури

        nestedCanvases.append(nestedCanvas);
    }
}
// --- IMAGE---
if (contents.contains("images")) {
    for (const QJsonValue &val : contents["images"].toArray()) {
        auto imgObj = std::make_shared<ImageObject>();
        imgObj->fromJson(val.toObject());
        imageObjects.append(imgObj);
    }
}
if (contents.contains("shapes")) {
    for (const QJsonValue& val : contents["shapes"].toArray()) {
        QJsonObject obj = val.toObject();
        QString type = obj["type"].toString();

        std::shared_ptr<ShapeObject> shape = nullptr;
        if (type == "line") {
            shape = std::make_shared<LineObject>();
        } else if (type == "arrow") {
            shape = std::make_shared<ArrowObject>();
        } else if (type == "circle") {
            shape = std::make_shared<CircleObject>();
        } else if (type == "freedraw") {
            shape = std::make_shared<FreeDrawObject>();
        }

        if (shape) {
            shape->fromJson(obj);
            shapeObjects.append(shape);
        }
```

```
}
        }

    }
}


void Canvas::saveToDisk() const {
    QJsonDocument doc(toJson());
    QFile file(directoryPath + "/canvas.json");
    if (file.open(QIODevice::WriteOnly)) {
        file.write(doc.toJson());
        file.close();
    }

    for (const auto &canvas : nestedCanvases)
        canvas->saveToDisk();
}

bool Canvas::loadFromDisk() {
    // завантаження JSON
    QFile file(directoryPath + "/canvas.json");
    if (!file.open(QIODevice::ReadOnly)) {
        qWarning() << "Не вдалося відкрити файл:" << file.fileName();
        return false;
    }

    QByteArray data = file.readAll();
    file.close();

    QJsonDocument doc = QJsonDocument::fromJson(data);
    if (!doc.isObject()) {
        qWarning() << "Невірний формат JSON";
        return false;
    }

    fromJson(doc.object());
    return true;
}


QVector<std::shared_ptr<CanvasObject>> Canvas::getAllObjects() const {
```

```
    QVector<std::shared_ptr<CanvasObject>> all;
    for (const auto &f : fileObjects) all.append(f);
    for (const auto &t : textObjects) all.append(t);
    for (const auto &c : nestedCanvases) all.append(c);
    for (const auto &i : imageObjects) all.append(i);
    for (const auto& s : shapeObjects) all.append(s);

    return all;
}

std::shared_ptr<CanvasObject> Canvas::getObjectById(const QUuid &id) const {
    for (const auto &obj : getAllObjects()) {
        if (obj->getId() == id)
            return obj;
    }
    return nullptr;
}

bool Canvas::removeObjectById(const QUuid &id) {
    auto obj = getObjectById(id);
    if (!obj) return false;

    QString basePath = getAbsoluteDirectoryPath();
    QString type = obj->getType();

    // --- FILE ---
    if (type == "file") {
        for (int i = 0; i < fileObjects.size(); ++i) {
            if (fileObjects[i]->getId() == id) {
                QFile::remove(basePath + "/" + fileObjects[i]->getFileName());
                fileObjects.removeAt(i);
                return true;
            }
        }
    }

    // --- TEXT ---
    if (type == "text") {
        for (int i = 0; i < textObjects.size(); ++i) {
            if (textObjects[i]->getId() == id) {
                textObjects.removeAt(i);
                return true;
```

```
}
        }
    }

    // --- IMAGE ---
    if (type == "image") {
        for (int i = 0; i < imageObjects.size(); ++i) {
            if (imageObjects[i]->getId() == id) {
                QFile::remove(basePath + "/" + imageObjects[i]->getFileName());
                imageObjects.removeAt(i);
                return true;
            }
        }
    }

    // --- NESTED CANVAS ---
    if (type == "canvas") {
        for (int i = 0; i < nestedCanvases.size(); ++i) {
            if (nestedCanvases[i]->getId() == id) {
                QDir dir(nestedCanvases[i]->getAbsoluteDirectoryPath());
                if (dir.exists())
                    dir.removeRecursively();
                nestedCanvases.removeAt(i);
                return true;
            }
        }
    }

    // --- SHAPE ---
    if (type == "shape") {
        for (int i = 0; i < shapeObjects.size(); ++i) {
            if (shapeObjects[i]->getId() == id) {
                shapeObjects.removeAt(i);
                return true;
            }
        }
    }

    return false;
}
```

```
QString Canvas::getAbsoluteDirectoryPath() const {
    return QDir(directoryPath).absolutePath();
}

bool Canvas::updateObject(std::shared_ptr<CanvasObject> updated) {
    const QUuid id = updated->getId();

    // Файли
    for (auto &ptr : fileObjects) {
        if (ptr->getId() == id) {
            ptr = std::dynamic_pointer_cast<FileObject>(updated);
            return true;
        }
    }

    // Тексти
    for (auto &ptr : textObjects) {
        if (ptr->getId() == id) {
            ptr = std::dynamic_pointer_cast<TextObject>(updated);
            return true;
        }
    }

    // Зображення
    for (auto &ptr : imageObjects) {
        if (ptr->getId() == id) {
            ptr = std::dynamic_pointer_cast<ImageObject>(updated);
            return true;
        }
    }

    // Вкладені полотна
    for (auto &ptr : nestedCanvases) {
        if (ptr->getId() == id) {
            ptr = std::dynamic_pointer_cast<Canvas>(updated);
            return true;
        }
    }
    // --- Update Shape ---
    for (auto& ptr : shapeObjects) {
        if (ptr->getId() == id) {
            ptr = std::dynamic_pointer_cast<ShapeObject>(updated);
```

```
return true;
        }
    }

    return false;
}

const QVector<std::shared_ptr<Canvas>> &Canvas::getNestedCanvases() const {
    return nestedCanvases;
}


bool Canvas::updateTextProperties(const QUuid& id, const QString& text, const QString& font, int fontSize, const
QString& color, bool bold, bool italic) {
    auto obj = getObjectById(id);
    if (!obj || obj->getType() != "text") return false;

    auto oldState = obj->toJson();

    auto textObj = std::dynamic_pointer_cast<TextObject>(obj);
    if (!textObj) return false;

    textObj->setText(text);
    textObj->setFont(font);
    textObj->setFontSize(fontSize);
    textObj->setColor(color);
    textObj->setBold(bold);
    textObj->setItalic(italic);

    return updateObject(textObj);
}


bool Canvas::updateShapeProperties(const QUuid& id, int x, int y, int x2, int y2, float radius, const QVariantList&
points, const QString& color, float strokeWidth) {
    auto obj = getObjectById(id);
    if (!obj || obj->getType() != "shape") return false;

    auto oldState = obj->toJson();

    auto shape = std::dynamic_pointer_cast<ShapeObject>(obj);
    if (!shape) return false;
```

```
    shape->setX(x);
    shape->setY(y);
    shape->setStrokeColor(QColor(color));
    shape->setStrokeWidth(strokeWidth);

    if (auto line = std::dynamic_pointer_cast<LineObject>(shape)) {
        line->setEndPoint(QPointF(x2, y2));
    } else if (auto arrow = std::dynamic_pointer_cast<ArrowObject>(shape)) {
        arrow->setEndPoint(QPointF(x2, y2));
    } else if (auto circle = std::dynamic_pointer_cast<CircleObject>(shape)) {
        circle->setRadius(radius);
    } else if (auto free = std::dynamic_pointer_cast<FreeDrawObject>(shape)) {
        QVector<QPointF> newPoints;
        for (const auto &val : points) {
            QVariantMap map = val.toMap();
            if (map.contains("x") && map.contains("y"))
                newPoints.append(QPointF(map["x"].toDouble(), map["y"].toDouble()));
        }
        free->setPoints(newPoints);
    }
    return updateObject(shape);
}

bool Canvas::updateObjectGeometry(const QUuid& id, int x, int y, int width, int height) {
    auto obj = getObjectById(id);
    if (!obj) return false;

    auto oldState = obj->toJson();

    obj->setX(x);
    obj->setY(y);
    obj->setWidth(width);
    obj->setHeight(height);

    return updateObject(obj);
}
```

ДОДАТОК Є

Реалізація класів ShapeObject:

```
#include "shapeobject.h"
ShapeObject::ShapeObject() : CanvasObject() {}

void ShapeObject::setStrokeColor(const QColor& color) { strokeColor = color; }
QColor ShapeObject::getStrokeColor() const { return strokeColor; }

void ShapeObject::setStrokeWidth(double width) { strokeWidth = width; }
double ShapeObject::getStrokeWidth() const { return strokeWidth; }

QJsonObject ShapeObject::toJson() const {
    QJsonObject obj = CanvasObject::toJson();
    obj["strokeColor"] = strokeColor.name();
    obj["strokeWidth"] = strokeWidth;
    return obj;
}
void ShapeObject::fromJson(const QJsonObject& json) {
    CanvasObject::fromJson(json);
    if (json.contains("strokeColor"))
        strokeColor = QColor(json["strokeColor"].toString());
    if (json.contains("strokeWidth"))
        strokeWidth = json["strokeWidth"].toDouble();
}
```

lineObject:

```
#include "lineobject.h"

LineObject::LineObject() : ShapeObject() {
    type = "line";
}

LineObject::LineObject(int x, int y, int x2, int y2)
    : ShapeObject(), endPoint(x2, y2) {
    this->x = x;
    this->y = y;
    this->width = 0;
    this->height = 0;
    this->type = "line";
}

QPointF LineObject::getEndPoint() const {
    return endPoint;
}

void LineObject::setEndPoint(const QPointF& point) {
    endPoint = point;
}

QJsonObject LineObject::toJson() const {
    QJsonObject obj = ShapeObject::toJson();
    obj["type"] = "line";
    obj["x2"] = endPoint.x();
    obj["y2"] = endPoint.y();
    return obj;
}

void LineObject::fromJson(const QJsonObject& json) {
    ShapeObject::fromJson(json);
    if (json.contains("x2")) endPoint.setX(json["x2"].toDouble());
    if (json.contains("y2")) endPoint.setY(json["y2"].toDouble());
}
```

arrowObject:

```
#include "arrowobject.h"

ArrowObject::ArrowObject() : ShapeObject() {
    type = "arrow";
}

ArrowObject::ArrowObject(int x, int y, int x2, int y2)
    : ShapeObject(), endPoint(x2, y2) {
    this->x = x;
    this->y = y;
    this->width = 0;
    this->height = 0;
    this->type = "arrow";
}

QPointF ArrowObject::getEndPoint() const {
    return endPoint;
}

void ArrowObject::setEndPoint(const QPointF& point) {
    endPoint = point;
}

QJsonObject ArrowObject::toJson() const {
    QJsonObject obj = ShapeObject::toJson();
    obj["type"] = "arrow";
    obj["x2"] = endPoint.x();
    obj["y2"] = endPoint.y();
    return obj;
}

void ArrowObject::fromJson(const QJsonObject& json) {
    ShapeObject::fromJson(json);
    if (json.contains("x2")) endPoint.setX(json["x2"].toDouble());
    if (json.contains("y2")) endPoint.setY(json["y2"].toDouble());
}
```

freedrawObject:

```
#include "freedrawobject.h"
#include <QJsonArray>


FreeDrawObject::FreeDrawObject() : ShapeObject() {
    type = "freedraw";
}

FreeDrawObject::FreeDrawObject(const QVector<QPointF>& points)
    : ShapeObject(), pathPoints(points) {
    type = "freedraw";
}

QVector<QPointF> FreeDrawObject::getPoints() const {
    return pathPoints;
}

void FreeDrawObject::setPoints(const QVector<QPointF>& pts) {
    pathPoints = pts;
}

QJsonObject FreeDrawObject::toJson() const {
    QJsonObject obj = ShapeObject::toJson();
    obj["type"] = "freedraw";

    QJsonArray arr;
    for (const QPointF& p : pathPoints) {
        QJsonObject pt;
        pt["x"] = p.x();
        pt["y"] = p.y();
        arr.append(pt);
    }
    obj["points"] = arr;
    return obj;
}

void FreeDrawObject::fromJson(const QJsonObject& json) {
    ShapeObject::fromJson(json);
    if (json.contains("points")) {
        QJsonArray arr = json["points"].toArray();
        pathPoints.clear();
        for (const auto& v : arr) {
            QJsonObject pt = v.toObject();
            pathPoints.append(QPointF(pt["x"].toDouble(), pt["y"].toDouble()));
```

ДОДАТОК Ж

Реалізація класу CanvasManager:

```
#include "canvasmanager.h"
#include <QFileInfo>
#include <QDebug>
#include <QJsonDocument>
#include <QJsonObject>

CanvasManager::CanvasManager(QObject *parent) : QObject(parent) { m_canvasModel = new
CanvasObjectModel(this);}

void CanvasManager::createNewCanvas(const QString &name, const QString &directoryPath) {
    QString fullPath = directoryPath + "/" + name;
    auto canvas = std::make_shared<Canvas>(name, 0, 0, fullPath);
    canvas->saveToDisk(); // створює папку і canvas.json

    openCanvases.append(canvas);
    currentCanvas = canvas;
    if (m_canvasModel && currentCanvas)
        m_canvasModel->setObjects(currentCanvas->getAllObjects());
    emit canvasModelChanged();
    emit currentCanvasChanged();
    emit canvasListChanged();
}

CanvasObjectModel* CanvasManager::canvasModel() const {
    return m_canvasModel;
}

void CanvasManager::openCanvas(const QString &path) {
    QFileInfo info(path);
    QString name = info.fileName();
    auto canvas = std::make_shared<Canvas>(name, 0, 0, path);
    canvas->loadFromDisk();
    openCanvases.append(canvas);
    currentCanvas = canvas;
    if (m_canvasModel && currentCanvas)
        m_canvasModel->setObjects(currentCanvas->getAllObjects());
    emit canvasModelChanged();
    emit currentCanvasChanged();
    emit canvasListChanged();
}
```

```
void CanvasManager::closeCanvasById(const QString &id) {
    QUuid uuid(id);
    openCanvases.erase(std::remove_if(openCanvases.begin(), openCanvases.end(),
                                      [&](const std::shared_ptr<Canvas> &c) {
                                          return c->getId() == uuid;
                                      }), openCanvases.end());

    if (currentCanvas && currentCanvas->getId() == uuid) {
        currentCanvas = openCanvases.isEmpty() ? nullptr : openCanvases.first();
        if (m_canvasModel && currentCanvas)
            m_canvasModel->setObjects(currentCanvas->getAllObjects());
        emit canvasModelChanged();
        emit currentCanvasChanged();
    }
    emit canvasListChanged();

}



QString CanvasManager::getCurrentCanvasName() const {
    return currentCanvas ? currentCanvas->getCanvasName() : QString();
}

void CanvasManager::addTextToCurrent(const QString &text, const QString &font, int size, const QString &color) {
    if (currentCanvas) {
        currentCanvas->addText(text, font, size, color);
        if (m_canvasModel && currentCanvas)
            m_canvasModel->setObjects(currentCanvas->getAllObjects());
        emit canvasModelChanged();
        currentCanvas->saveToDisk();
    }
}

void CanvasManager::addFileToCurrent(const QString &filePath, const QString &iconPath) {
    if (!currentCanvas) {
        qWarning() << "[addFileToCurrent] No current canvas!";
        return;
    }

    currentCanvas->addFile(filePath, ":/icons/file.png");
    if (m_canvasModel && currentCanvas)
```

```
m_canvasModel->setObjects(currentCanvas->getAllObjects());
    emit canvasModelChanged();
    currentCanvas->saveToDisk();

}

void CanvasManager::addShapeToCurrent(const QString &type, int x1, int y1, int x2, int y2,
                                      double radius, const QVariantList &points,
                                      const QString &strokeColor, double strokeWidth) {
    if (!currentCanvas) return;

    currentCanvas->addShape(type, x1, y1, x2, y2, radius, points, strokeColor, strokeWidth);

    if (m_canvasModel)
        m_canvasModel->setObjects(currentCanvas->getAllObjects());
    emit canvasModelChanged();
    currentCanvas->saveToDisk();
}



void CanvasManager::addCanvasToCurrent(const QString &name) {
    if (currentCanvas) {
        currentCanvas->addNestedCanvas(name);
        if (m_canvasModel && currentCanvas)
            m_canvasModel->setObjects(currentCanvas->getAllObjects());
        emit canvasModelChanged();
        currentCanvas->saveToDisk();
    }
}

void CanvasManager::addImageToCurrent(const QString &sourceFilePath) {
    if (!currentCanvas) {
        qWarning() << "[addImageToCurrent] No current canvas!";
        return;
    }

    currentCanvas->addImageObject(sourceFilePath);

    if (m_canvasModel && currentCanvas) {
        m_canvasModel->setObjects(currentCanvas->getAllObjects());
        emit canvasModelChanged();
```

```
    }

    currentCanvas->saveToDisk();
}



bool CanvasManager::removeObjectFromCurrent(const QString &id) {
    if (!currentCanvas) return false;

    QUuid uuid(id);
    auto obj = currentCanvas->getObjectById(uuid);
    if (!obj) return false;

    const QString type = obj->getType();

    bool removed = currentCanvas->removeObjectById(uuid);

    if (removed) {
        if (type == "canvas") {
            QString deletedPath = currentCanvas->getAbsoluteDirectoryPath() + "/" +
                                  std::dynamic_pointer_cast<Canvas>(obj)->getCanvasName();

            openCanvases.erase(std::remove_if(openCanvases.begin(), openCanvases.end(),
                                              [&](const std::shared_ptr<Canvas> &c) {
                                                  return c->getAbsoluteDirectoryPath().startsWith(deletedPath);
                                              }), openCanvases.end());

            if (currentCanvas && currentCanvas->getAbsoluteDirectoryPath().startsWith(deletedPath)) {
                currentCanvas = openCanvases.isEmpty() ? nullptr : openCanvases.first();
                emit currentCanvasChanged();
            }

            emit canvasListChanged();
        }

        currentCanvas->saveToDisk();

        if (m_canvasModel && currentCanvas)
            m_canvasModel->setObjects(currentCanvas->getAllObjects());

        emit canvasModelChanged();
                                                                                    }
```

```
    return removed;
}



QStringList CanvasManager::listOpenCanvases() const {
    QStringList list;
    for (const auto &c : openCanvases) {
        list.append(c->getCanvasName());
    }
    return list;
}


bool CanvasManager::setCurrentCanvasById(const QString &id) {
    QUuid uuid(id);

    std::function<std::shared_ptr<Canvas>(std::shared_ptr<Canvas>, const QString &)> findById;
    findById = [this, &uuid, &findById](std::shared_ptr<Canvas> root, const QString &basePath) ->
std::shared_ptr<Canvas> {
        if (root->getId() == uuid)
            return root;

        for (const auto &obj : root->getAllObjects()) {
            if (obj->getId() == uuid && obj->getType() == "canvas") {
                auto canvasPtr = std::dynamic_pointer_cast<Canvas>(obj);
                if (canvasPtr) {
                    QString subPath = basePath + "/" + canvasPtr->getCanvasName();
                    auto nested = std::make_shared<Canvas>(canvasPtr->getCanvasName(), obj->getX(), obj->getY(),
subPath);
                    if (nested->loadFromDisk()) {
                        if (nested->getId() == uuid) {

                            auto existing = std::find_if(openCanvases.begin(), openCanvases.end(),
                                                         [&](const std::shared_ptr<Canvas>& c) {
                                                             return c->getId() == uuid;
                                                         });

                            if (existing != openCanvases.end()) {
                                qDebug() << "[setCurrentCanvasById] Canvas already open (deep):" << (*existing)-
>getCanvasName() << (*existing)->getId().toString(QUuid::WithoutBraces);
                                return *existing;
                            }
```

```
                openCanvases.append(nested);
                emit canvasListChanged();
                return nested;
            }
        }
    }
}


for (const auto &obj : root->getAllObjects()) {
    if (obj->getType() == "canvas") {
        auto canvasPtr = std::dynamic_pointer_cast<Canvas>(obj);
        if (canvasPtr) {
            QString subPath = basePath + "/" + canvasPtr->getCanvasName();
            auto nested = std::make_shared<Canvas>(canvasPtr->getCanvasName(), obj->getX(), obj->getY(),
subPath);
            if (nested->loadFromDisk()) {
                if (nested->getId() == uuid) {
                    auto existing = std::find_if(openCanvases.begin(), openCanvases.end(),
                                                 [&](const std::shared_ptr<Canvas>& c) {
                                                     return c->getId() == uuid;
                                                 });

                    if (existing != openCanvases.end()) {
                        qDebug() << "[setCurrentCanvasById] Canvas already open (deep):" << (*existing)-
>getCanvasName() << (*existing)->getId().toString(QUuid::WithoutBraces);
                        return *existing;
                    }

                    openCanvases.append(nested);
                    emit canvasListChanged();
                    return nested;
                } else {
                    auto result = findById(nested, subPath);
                    if (result)
                        return result;
                }
            }
        }
    }
}
```

```
        return nullptr;
    };

    for (const auto &canvas : openCanvases) {
        auto found = findById(canvas, canvas->getAbsoluteDirectoryPath());
        if (found) {
            currentCanvas = found;
            if (m_canvasModel && currentCanvas)
                m_canvasModel->setObjects(currentCanvas->getAllObjects());
            emit canvasModelChanged();
            emit currentCanvasChanged();
            qDebug() << "[setCurrentCanvasById] Set current canvas to:" << found->getCanvasName() << found-
>getId().toString(QUuid::WithoutBraces);
            return true;
        }
    }

    qWarning() << "[setCurrentCanvasById] Canvas not found for id:" << id;
    return false;
}




QVariantList CanvasManager::getOpenCanvasEntries() const {
    QVariantList list;
    for (const auto &canvas : openCanvases) {
        QVariantMap item;
        item["name"] = canvas->getCanvasName();
        item["id"] = canvas->getId().toString(QUuid::WithoutBraces);
        list.append(item);
    }
    qDebug() << "Open canvases count:" << openCanvases.size();
    return list;
}

bool CanvasManager::updateObjectGeometry(const QString &id, int x, int y, int width, int height) {
    if (!currentCanvas) return false;

    bool updated = currentCanvas->updateObjectGeometry(QUuid(id), x, y, width, height);
```

if

```
(updated)
        currentCanvas->saveToDisk();

    return updated;
}
bool CanvasManager::updateTextProperties(const QString &id, const QString &text, const QString &font, int
fontSize, const QString &color, bool bold, bool italic) {
    if (!currentCanvas) return false;

    bool updated = currentCanvas->updateTextProperties(QUuid(id), text, font, fontSize, color, bold, italic);

    if (updated)
        currentCanvas->saveToDisk();

    return updated;
}
bool CanvasManager::updateShapeProperties(const QString &id, int x, int y, int x2, int y2, float radius, const
QVariantList &points, const QString &color, float strokeWidth) {
    if (!currentCanvas)
        return false;

    bool updated = currentCanvas->updateShapeProperties(QUuid(id), x, y, x2, y2, radius, points, color, strokeWidth);

    if (updated)
        currentCanvas->saveToDisk();

    if (m_canvasModel && currentCanvas)
        m_canvasModel->setObjects(currentCanvas->getAllObjects());
    emit canvasModelChanged();

    return updated;
}
```

ДОДАТОК З

Реалізація класу CanvasMemento:

```
#include "canvasmemento.h"

CanvasMemento::CanvasMemento(const QString& action, const QUuid& id,
                             const QJsonObject& before, const QJsonObject& after)
    : action(action), objectId(id), beforeState(before), afterState(after)
{}

QString CanvasMemento::getAction() const {
    return action;
}

QUuid CanvasMemento::getObjectId() const {
    return objectId;
}

QJsonObject CanvasMemento::getBeforeState() const {
    return beforeState;
}
QJsonObject CanvasMemento::getAfterState() const {
    return afterState;
}
```

ДОДАТОК И

Реалізація класу CanvasFactory:

```
#include "canvasfactory.h"

std::shared_ptr<CanvasObject> CanvasFactory::createFromJson(const QJsonObject& json) {
    QString type = json["type"].toString();

    std::shared_ptr<CanvasObject> obj = nullptr;

    if (type == "text") {
        obj = std::make_shared<TextObject>();
    } else if (type == "file") {
        obj = std::make_shared<FileObject>();
    } else if (type == "image") {
        obj = std::make_shared<ImageObject>();
    } else if (type == "line") {
        obj = std::make_shared<LineObject>();
    } else if (type == "arrow") {
        obj = std::make_shared<ArrowObject>();
    } else if (type == "circle") {
        obj = std::make_shared<CircleObject>();
    } else if (type == "freedraw") {
        obj = std::make_shared<FreeDrawObject>();
    } else if (type == "canvas") {
        obj = std::make_shared<Canvas>(); // якщо Canvas теж є об'єктом
    } else {
        qWarning() << "Unknown object type in JSON:" << type;
        return nullptr;
    }

    if (obj) {
        obj->fromJson(json); // виклик завантаження з JSON
    }

    return obj;
}
```

ДОДАТОК І

Реалізація класу CanvasObjectModel:

```
#include "canvasobjectmodel.h"
#include "fileobject.h"
#include "textobject.h"
#include "canvas.h"
#include "shapeobject.h"
#include "lineobject.h"
#include "arrowobject.h"
#include "circleobject.h"
#include "freedrawobject.h"

CanvasObjectModel::CanvasObjectModel(QObject *parent) : QAbstractListModel(parent) {}

int CanvasObjectModel::rowCount(const QModelIndex &) const {
    return m_objects.size();
}

QVariant CanvasObjectModel::data(const QModelIndex &index, int role) const {
    if (!index.isValid() || index.row() < 0 || index.row() >= m_objects.size())
        return {};

    const auto &obj = m_objects[index.row()];
    if (!obj)
        return {};

    if (role == ObjectIdRole) return obj->getId().toString(QUuid::WithoutBraces);
    if (role == TypeRole) return obj->getType();
    if (role == XRole) return obj->getX();
    if (role == YRole) return obj->getY();
    if (role == WidthRole) return obj->getWidth();
    if (role == HeightRole) return obj->getHeight();

    if (auto file = std::dynamic_pointer_cast<FileObject>(obj)) {
        if (role == FileNameRole) return file->getFileName();
        if (role == FileTypeRole) return file->getFileType();
        if (role == FileSizeRole) return file->getFileSize();
        if (role == IconPathRole) return file->getIconPath();
    }

    if (auto image = std::dynamic_pointer_cast<ImageObject>(obj)) {
```

```
    if (role == FileNameRole) return image->getFileName();
    if (role == FilePathRole) return image->getFilePath();
}

if (auto text = std::dynamic_pointer_cast<TextObject>(obj)) {
    if (role == TextRole) return text->getText();
    if (role == FontRole) return text->getFont();
    if (role == FontSizeRole) return text->getFontSize();
    if (role == ColorRole) return text->getColor();
    if (role == BoldRole) return text->isBold();
    if (role == ItalicRole) return text->isItalic();
}

if (auto canvas = std::dynamic_pointer_cast<Canvas>(obj)) {
    if (role == NameRole) return canvas->getCanvasName();
}
// Спільне для всіх ShapeObject
if (auto shape = std::dynamic_pointer_cast<ShapeObject>(obj)) {
    if (role == StrokeColorRole) return shape->getStrokeColor().name();
    if (role == StrokeWidthRole) return shape->getStrokeWidth();
}

// LineObject / ArrowObject
if (auto line = std::dynamic_pointer_cast<LineObject>(obj)) {
    if (role == X2Role) return line->getEndPoint().x();
    if (role == Y2Role) return line->getEndPoint().y();
}
if (auto arrow = std::dynamic_pointer_cast<ArrowObject>(obj)) {
    if (role == X2Role) return arrow->getEndPoint().x();
    if (role == Y2Role) return arrow->getEndPoint().y();
}

// CircleObject
if (auto circle = std::dynamic_pointer_cast<CircleObject>(obj)) {
    if (role == RadiusRole) return circle->getRadius();
}

// FreeDrawObject
if (auto draw = std::dynamic_pointer_cast<FreeDrawObject>(obj)) {
    if (role == PointsRole) {
        QVariantList pointList;
        for (const QPointF& pt : draw->getPoints()) {
```

```
QVariantMap m;
            m["x"] = pt.x();
            m["y"] = pt.y();
            pointList.append(m);
        }
        return pointList;
    }
}


return {};
}


QHash<int, QByteArray> CanvasObjectModel::roleNames() const {
    return {
        {ObjectIdRole, "objectId"},
        {TypeRole, "type"},
        {XRole, "x"},
        {YRole, "y"},
        {WidthRole, "width"},
        {HeightRole, "height"},
        {FileNameRole, "fileName"},
        {FileTypeRole, "fileType"},
        {FileSizeRole, "fileSize"},
        {IconPathRole, "iconPath"},
        {TextRole, "text"},
        {FontRole, "font"},
        {FontSizeRole, "fontSize"},
        {ColorRole, "color"},
        {BoldRole, "bold"},
        {ItalicRole, "italic"},
        {NameRole, "name"},
        {StrokeColorRole, "strokeColor"},
        {StrokeWidthRole, "strokeWidth"},
        {X2Role, "x2"},
        {Y2Role, "y2"},
        {RadiusRole, "radius"},
        {PointsRole, "points"}
    };
}
```

```
void CanvasObjectModel::setObjects(const QVector<std::shared_ptr<CanvasObject>> &objects) {
    beginResetModel();
    m_objects = objects;
      endResetModel();
}

std::shared_ptr<CanvasObject> CanvasObjectModel::getObject(int index) const {
    if (index < 0 || index >= m_objects.size())
        return nullptr;
    return m_objects[index];
}
QVariantMap CanvasObjectModel::get(int index) const {
    QVariantMap map;
    if (index < 0 || index >= m_objects.size())
        return map;

    const auto& obj = m_objects[index];
    map["type"] = obj->getType();
    map["x"] = obj->getX();
    map["y"] = obj->getY();
    map["width"] = obj->getWidth();
    map["height"] = obj->getHeight();
    map["objectId"] = obj->getId().toString(QUuid::WithoutBraces);

    // ↓ Можеш доповнити залежно від типу
    if (auto shape = std::dynamic_pointer_cast<ShapeObject>(obj)) {
        map["strokeColor"] = shape->getStrokeColor().name();
        map["strokeWidth"] = shape->getStrokeWidth();
    }

    if (auto line = std::dynamic_pointer_cast<LineObject>(obj)) {
        map["x2"] = line->getEndPoint().x();
        map["y2"] = line->getEndPoint().y();
    }

    if (auto arrow = std::dynamic_pointer_cast<ArrowObject>(obj)) {
        map["x2"] = arrow->getEndPoint().x();
        map["y2"] = arrow->getEndPoint().y();
    }

    if (auto circle = std::dynamic_pointer_cast<CircleObject>(obj)) {
        map["radius"] = circle->getRadius();
                                                                                    }
```

```
    if (auto draw = std::dynamic_pointer_cast<FreeDrawObject>(obj)) {
        QVariantList points;
        for (const QPointF& pt : draw->getPoints()) {
            QVariantMap m;
            m["x"] = pt.x();
            m["y"] = pt.y();
            points.append(m);
        }
        map["points"] = points;
    }

    return map;
}
```