Qt C++ Library Desktop App

Edoardo De Piccoli - 2101055

May 2025

This project involves developing a fully functional Library Management desktop application using Qt and C++. The app must follow the MVC pattern and support polymorphic behavior using an advanced technique such as the visitor pattern. The user will be able to perform full CRUD operations on at least three item types (in this particular case they are a book, movie, and article) using a simple and user-friendly Qt GUI.

1 Introduction and Project Description

Qt Library is a desktop application that allows users to quickly and easily categorize their books, movies, and articles through a simple, clean, and intuitive interface. Each item has its own set of attributes.

The project is written in C++ using the Qt framework, which I used for widgets and helpful types like custom strings and JSON classes. It is structured following the MVC pattern and uses the Visitor pattern to handle polymorphism in a non-trivial way.

In this, I took a lot of inspiration from the Ruby on Rails web development framework, which uses the RESTful pattern for routing to create a battle-tested, convention-oriented system.

Although my project does not use routing in the same way (being a desktop app), I applied the same philosophy to how views and features are structured. Every feature is built with that kind of organization in mind, resulting in a consistent structure and user experience. This results in a "routing system" that looks like this:

Action	**View**	**View Transition**
List all items (INDEX)	IndexView	→ IndexView
Show item (SHOW)	ShowItemView	IndexView → ShowItemView
New item form (NEW)	NewItemView	IndexView → NewItemView
Edit item form (EDIT)	EditItemView	ShowItemView → EditItemView
Create item (CREATE)	NewItemView	NewItemView → IndexView
Update item (UPDATE)	EditItemView	EditItemView → IndexView
Delete item (DESTROY)	ShowItemView	ShowItemView → IndexView

Table 1: View transitions for CRUD actions

I chose this architecture because it is familiar to me from working with Ruby on Rails. I like it, it is low maintenance and makes for a straightforward user experience.

I also thought that fully embracing the Visitor pattern, for debugging, creating JSON objects, and generating Qt widgets, was a good way to use polymorphism in a modular and interesting way.

2 Model

2.1 High-Level Overview

At the core of the application is the Library class, which handles all operations related to managing different types of items, books, movies, and articles. The backend is designed to function independently of the graphical user interface, resulting in a loosely coupled, highly maintainable system. For example, to understand or modify how `searchItems(query)` works, one can directly inspect the implementation in the Library class without needing to adjust any part of the UI: the views will automatically reflect changes in the logic.

2.2 The Library

The model is composed of five main classes: `Library`, `Item`, and three subclasses of `Item` — namely, `Book`, `Movie`, and `Article`. The `Library` class handles all model-level logic and acts as the sole point of interaction for modifying items, ensuring a centralized and consistent approach to data management.

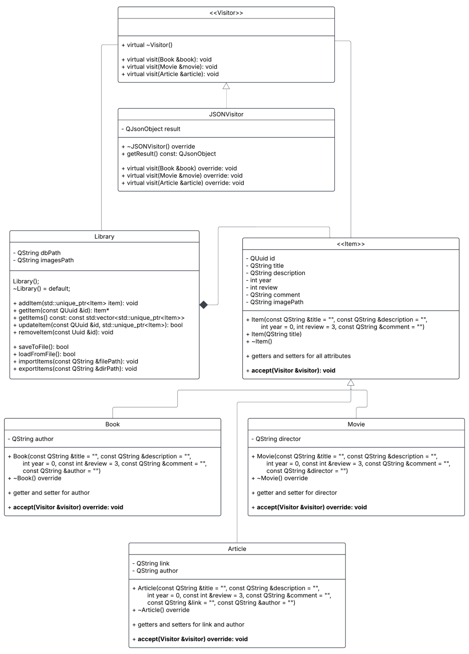
<<Visitor>>
+ virtual ~Visitor()
+ virtual visit(Book &book): void
+ virtual visit(Movie &movie): void
+ virtual visit(Article &article): void
JSONVisitor
- QJsonObject result
+ ~JSONVisitor() override
+ getResult() const: QJsonObject
+ virtual visit(Book &book) override: void
+ virtual visit(Movie &movie) override: void
+ virtual visit(Article &article) override: void
Library
- QString dbPath
- QString imagesPath
Library();
~Library() = default;
+ addItem(std::unique_ptr<Item> item): void
+ getItem(const QUuid &id): Item*
+ getItems() const: const std:vector<std::unique_ptr<Item>>
+ updateItem(const QUuid &id, std::unique_ptr<Item>): bool
+ removeItem(const Uuid &id): void
+ saveToFile(): bool
+ loadFromFile(): bool
+ importItems(const QString &filePath): void
+ exportItems(const QString &dirPath): void
<<Item>>
- QUuid id
- QString title
- QString description
- int year
- int review
- QString comment
- QString imagePath
+ Item(const QString &title = "", const QString &description = "", int year = 0, int review = 3, const QString &comment = "")
+ Item(QString title)
+ ~Item()
+ getters and setters for all attributes
+ accept(Visitor &visitor): void
Book
- QString author
+ Book(const QString &title = "", const QString &description = "", int year = 0, const int &review = 3, const QString &comment = "", const QString &author = "")
+ ~Book() override
+ getter and setter for author
+ accept(Visitor &visitor) override: void
Movie
- QString director
+ Movie(const QString &title = "", const QString &description = "", int year = 0, const int &review = 3, const QString &comment = "", const QString &director = "")
+ ~Movie() override
+ getter and setter for director
+ accept(Visitor &visitor) override: void
Article
- QString link
- QString author
+ Article(const QString &title = "", const QString &description = "", int year = 0, const int &review = 3, const QString &comment = "", const QString &link = "", const QString &author = "")
+ ~Article() override
+ getters and setters for link and author
+ accept(Visitor &visitor) override: void

Figure 1: Model UML with the most important classes, attributes and methods

This design simplifies debugging, enforces uniform behavior across the application, and speeds up development and testing.

Internally, the `Library` class maintains a `std::vector` of `std::unique_ptr<Item>`, allowing it to manage a polymorphic collection of items. It exposes a full set of CRUD methods, returning meaningful results such as status codes, pointers to specific items, or collections of items. The interface is intentionally frontend-agnostic, meaning it supports both GUI and CLI frontends without requiring changes to the underlying logic.

3 Data Persistence and the Visitor Pattern

The `Library` class is also responsible for managing data persistence. All data is saved locally in a `data.json` file located in the `db` folder, along with file paths to associated images. When a user adds or updates an item and includes an image, the application automatically creates a local copy of that image. This ensures that the item's visual representation remains intact even if the original image file is deleted or moved.

To support polymorphic serialization, the application uses the **Visitor Pattern**. A dedicated `JSONVisitor` class handles the conversion of items to JSON format by defining a specific `visit()` method for each concrete item type—`Book`, `Movie`, and `Article`. This allows items stored as base `Item` pointers to be processed uniformly without type checking or casting. The result is clean, modular code that keeps the `Library` class focused on orchestration rather than format-specific logic.

3.1 The Item Class

The `Item` class serves as the abstract base for all item types in the application. It defines a common set of attributes, including `id`, `title`, `description`, `year`, `review`, `comment`, and `imagePath`. These properties represent the shared metadata across all item types. The class also provides standard getter and setter methods to ensure controlled access and modification of these fields.

3.2 Book, Movie, and Article

The `Book`, `Movie`, and `Article` classes inherit from `Item` and add fields specific to their content type. The `Book` class introduces an `author` field, while the `Movie` class includes a `director` field. The `Article` class extends the base with both an `author` and a `link` field. These subclasses are deliberately minimal, including only the attributes necessary to distinguish each type, thus maintaining a clear and lightweight class hierarchy.

4 View and Controller

4.1 Qt Framework and Initial Challenges

The view layer of the application proved to be the most difficult part to implement. Initially, development began without much planning, which quickly revealed the complexity of the Qt framework. Qt is a powerful and feature-rich environment with extensive documentation and multiple ways to approach most tasks. While this flexibility is valuable, it can be overwhelming for beginners.

One of the first ideas was to build the UI using reusable components. For instance, I tried to create a single form component that could handle both item creation and editing, requiring only minimal differences in behavior. Although this seemed like a good strategy, my limited experience made it difficult to execute properly—particularly in terms of memory management. Improper pointer usage and unclear ownership semantics frequently led to segmentation faults.

After investing more than 20 hours into an initial implementation, I eventually abandoned the project out of frustration. Several months later, I revisited it with a deeper understanding of Qt and a more deliberate approach to architectural planning. This shift in mindset allowed me to build a more stable and maintainable UI layer.

4.2 Key Architectural Questions

While designing the user interface, I encountered several foundational questions that significantly influenced the final architecture. These included decisions about where the core application logic should reside—centrally in a controller-like structure or distributed across individual widgets. Managing and preserving application state was also a major concern, especially when using pointers to share data between components.

Ownership became a key design consideration. If pointers were passed to share state, who would be responsible for managing their lifetime? Should the `Library` class maintain ownership, or should each form be responsible for the items it edits? This raised further questions about communication between components: for example, if a form modifies an item, should it notify the `MainWindow`, or should the logic be centralized?

Additionally, I had to determine how to handle view transitions cleanly and ensure that deleted views properly cleaned up their widgets to avoid memory leaks or dangling references. Another important consideration was how to integrate widgets generated by the UI visitor classes into the overall view structure.

These questions played a critical role in shaping the final design of the application's view architecture, guiding key decisions around responsibility delegation, memory management, and component integration.

4.3 The View Layer

4.3.1 MainWindow

The `MainWindow` serves as the Controller for the entire application. It acts as the central hub that connects the model—represented by the `Library` class—to all views and UI components. To maintain a consistent application state, the `MainWindow` holds a `std::unique_ptr` to the main `Library` instance. All create, update, and delete operations initiated by views are funneled through the `MainWindow`, ensuring that logic and state management remain centralized and predictable.

4.3.2 IndexView

The `IndexView` is responsible for displaying and searching through the list of items. It invokes the `library->searchItems()` method to fetch results. If the search query is empty, the method returns the full list of items; otherwise, it searches relevant attributes and returns the matching entries. Each retrieved item is passed to an `ItemCardVisitor`, which generates a widget in card format, appropriate to the specific item type. Clicking the "View" button on a card transitions the interface to the `ShowItemView` for that item.

4.3.3 ShowItemView

The `ShowItemView` closely resembles the `IndexView` but focuses on presenting detailed information about a single item. It uses an `ItemShowVisitor` to construct a widget that renders all details of the item in a structured and visually appealing format. In addition to displaying information, it provides buttons for editing and deletion. The delete action is routed through the `MainWindow` to ensure proper state handling, while the edit button transitions to the `EditItemView`.

4.3.4 EditItemView

The `EditItemView` provides a pre-filled form that allows users to edit an existing item. It receives a pointer to the selected item and uses the `ItemFormVisitor` to generate the corresponding form. Any changes made in this form are passed back through the `MainWindow`, ensuring that updates are applied consistently and safely across the application.

4.3.5 NewItemView

The `NewItemView` introduces more dynamic behavior. It is responsible for rendering an input form that changes based on the item type selected by the user. By default, it loads the form for a `Book`. When a different type is selected via a `QComboBox`, the view deletes the current form to prevent memory leaks and then renders a new one appropriate to the selected type. This dynamic switching is done safely to avoid dangling pointers.

Image handling in this view added additional complexity. Since this feature was implemented later in development, the logic for managing image paths was handled directly within the `NewItemView`, rather than being offloaded to visitor-generated widgets. While this somewhat diverged from the modular visitor pattern, it offered a practical balance between flexibility and maintainability.

4.3.6 The Widgets

A natural question that emerges is how these views render the appropriate UI components. This is where the **Visitor Pattern** proves essential. By delegating widget generation to specialized visitor classes—such as `ItemCardVisitor`, `ItemShowVisitor`, and `ItemFormVisitor`—the application maintains a clean separation between data structures and their visual representations. This design promotes extensibility and makes it easy to adapt the UI to different item types without duplicating code.

5 Polymorphism and the Visitor Pattern

A central requirement of the project was to implement polymorphism in a meaningful and scalable way. To ensure the architecture remained modular and extensible, the chosen solution was the **Visitor Pattern**. This design pattern offered a structured way to handle behavior that depends on item types, while preserving the separation between data and presentation layers.

5.1 The Problem

The simplest approach would have been to define UI functionality directly within each item class. For instance, the `Book`, `Movie`, and `Article` classes could have implemented a virtual `getWidget()` method to return the appropriate UI component. While this strategy works for small applications, it introduces tight coupling between the model and the view.

This coupling becomes problematic when UI requirements change. If the models are responsible for rendering their own views, any UI refactor requires editing the core data classes. This violates the principle of separation of concerns, resulting in fragile and hard-to-maintain code over time.

5.2 The Solution

The `Visitor Pattern` offers a way to cleanly decouple the model from the view logic. Each item class implements a single `accept()` method, which takes a visitor as its argument. The visitor then executes logic specific to the item type.

This pattern creates an intermediary layer between the models and the views. As a result, UI components can be developed and updated independently of the item logic. The code becomes easier to extend, more maintainable, and adheres more closely to the MVC architecture.

A typical visitor works by visiting a specific item type (such as `Book`, `Movie`, or `Article`), executing logic based on that type, storing the result—such as a widget or a data object—and exposing it through a public method. The view then retrieves the result and deletes the visitor after use.

5.2.1 JSONVisitor

This is the most straightforward visitor. Its role is to serialize an item into a `QJsonObject` that includes all relevant properties. The `JSONVisitor` is used primarily for saving and loading data from the local `data.json` file.

5.2.2 DebugVisitor

The `DebugVisitor` was especially helpful during development. It outputs item details using `qDebug()`, making it easy to verify backend behavior and test data structures through the command line interface.

5.2.3 ItemCardVisitor

The `ItemCardVisitor` generates visual card widgets for each item displayed in the `IndexView`. Each card includes basic details and a "View" button. When this button is clicked, a signal is emitted, which is handled by the `IndexView` and forwarded to the `MainWindow`, allowing a transition to the detailed item view.

5.2.4 ItemShowVisitor

This visitor builds a detailed display widget for a single item. It determines the item type and adds appropriate UI elements such as image previews and metadata. It also includes two action buttons: **Edit** and **Delete**. These buttons emit signals that are captured by the `MainWindow`, ensuring consistency with other parts of the interface.

5.2.5 ItemFormVisitor

The `ItemFormVisitor` is the most complex of all the visitors. It generates a form for either creating a new item or editing an existing one. Depending on which `visit()` method is called—`visit(Book*)`, `visit(Movie*)`, or `visit(Article*)`—the visitor dynamically constructs the corresponding form fields.

Once the form is submitted, the visitor builds a new item instance using the entered data and emits a signal with this item. The `MainWindow` captures the signal and delegates the creation or update operation to the `Library`. A boolean flag within the visitor distinguishes between creation and editing modes, allowing the same codebase to serve both purposes and reducing redundancy.

With the visitor system in place, the application's MVC structure is complete. This architecture ensures flexibility, encourages modular design, and prepares the application for future expansion or refactoring. The next component of the system focuses on data persistence and retrieval.

6 Data Storage and Persistence

6.1 Data Persistence

Data is stored locally in a dedicated `db` folder, which contains two key components: the `data.json` file, where all item properties and their relative image paths are maintained, and the `/images` directory, which holds working copies of all user-uploaded images.

6.1.1 How It Works

To ensure that storage remains synchronized with application state, all CRUD operations are handled by the `Library` class. Whenever an item is created, updated, or deleted, the corresponding internal methods automatically persist these changes to the local files. This centralizes persistence logic and frees the UI layer from having to manage any save or sync operations.

6.1.2 Handling Images

When a user uploads an image, the application copies it into the `/images` folder. This guarantees that the image remains available even if the original file is removed from the user's system. If no image is provided during item creation, a default image—bundled within the `db` folder—is assigned based on the item type (book, movie, or article), preserving visual consistency.

6.1.3 File Operations

The core persistence logic resides in two primary `Library` methods: `loadFromFile()`, which reads existing data from `data.json` at startup, and `saveToFile()`, which writes the current library state back to `data.json`. By invoking `loadFromFile()` during initialization and `saveToFile()` after any data-modifying operation, the application maintains an up-to-date, consistent state with minimal UI involvement.

This approach keeps the UI logic simple and focused, since views only need to interact with the `Library` without worrying about persistence details.

6.2 Import/Export Functionalities

The application supports importing and exporting items through both the GUI and CLI interfaces. This functionality enables users to share their libraries or back up their data efficiently.

6.2.1 Implementation Details

Import/export operations are implemented directly within the `Library` class, preserving separation of concerns by delegating all data operations to the model layer. The implementation accounts for both item data and associated images:

- During import, relative image paths are resolved against the location of the import file.
- During export, images are copied to a dedicated `images` subdirectory.
- Image paths in the exported JSON file are updated to be relative to the export directory.
- Default images are retained and not mistakenly deleted.

6.2.2 GUI Integration

In the GUI, users can access import/export functionality via toolbar buttons:

- **Import Items**: Opens a file dialog for selecting a JSON file.
- **Export Items**: Opens a directory dialog for selecting the export location.

The flow follows a clear structure:

1. User actions are managed by the `MainWindow` controller.
2. File dialogs are implemented using Qt's native APIs for consistency.
3. After successful operations, a confirmation message is shown.
4. The view refreshes automatically post-import to reflect new items.

6.2.3 CLI Integration

The CLI provides equivalent commands for consistent feature access:

- `import`: Prompts the user for a JSON file path.
- `export`: Prompts for a target directory.

This ensures users can manage their data using either interface as preferred.

6.2.4 Data Structure

The exported data is structured as follows:

```
export_dir/
 data.json
 images/
     image1.png
     image2.jpg
     ...
```

This layout keeps data organized and ensures all relative paths remain valid, making re-imports straightforward and consistent.

6.2.5 Error Handling

Basic error handling mechanisms are included to support a smooth user experience. These mechanisms validate paths, check file existence, and gracefully handle malformed data files.

This feature rounds out the application's data management capabilities, enabling reliable sharing and backup of library data while maintaining architectural clarity and separation between the model and view layers.

7 User Flow, UI/UX

7.1 User Interface and UX Flow

When the application launches, the user is greeted with a clean and minimal interface: a blank screen with a search bar at the top, and two main toolbar buttons—**New Item** and **Index**.

To add a new entry, the user clicks **New Item** and is presented with a form to input relevant details. To browse existing entries, clicking **Index** displays a searchable list of all saved items.

Once the database contains data, users navigate items following a flow inspired by conventional RESTful design. The search bar enables real-time filtering of all items. Clicking **View** next to any entry opens the **Show Item View** for that specific item (book, movie, or article).

7.1.1 Show Item View

This view presents detailed information of the selected item:

- On the **left**, the item's image is displayed.
- On the **right**, at the top, are all relevant details (title, year, description, etc.).
- Below the details, two buttons are provided:
  - **Edit**: navigates to the Edit Item View.
  - **Delete**: removes the item and returns the user to the Index.

7.1.2 Edit Item View

The **Edit View** closely resembles the New Item form with one key difference: the user cannot change the item's type. They may modify other properties and submit changes by clicking the **Update** button. After updating, the user is returned to the Index view.

7.1.3 Image Upload

Users can add or update images by clicking an upload button within the form. This opens a file dialog to select an image. The application copies the selected image as a working copy, ensuring availability even if the original file is deleted from the user's device.

7.1.4 Design Philosophy

The navigation flow is deliberately simple and intuitive, with only **one way** to perform each major action. This reduces user confusion and keeps development and UX consistent and maintainable. The interface avoids unnecessary toolbars or redundant buttons.

Although a polished UI was not a formal requirement, the design prioritizes clarity and usability. The result is functional, clean, and easy to use, rather than flashy but with a poor UX.

8 Total Work Hours

8.1 Total Estimated Time: around 60 hours

The project was initially scoped for about 40 hours. However, due to a steep learning curve with Qt, memory management in C++, and scrapping the first attempt, the actual total time exceeded this estimate significantly. Nevertheless, the end result is a modular application that I fully understand.

9 Lessons Learned

9.1 Qt-Specific Takeaways

Qt provides helpful defaults that simplify memory management, such as automatically preventing the deletion of null widgets and cleaning up child widgets when the parent is deleted. These behaviors reduce the risk of memory errors. I also learned that planning before coding is essential with Qt; taking time upfront to explore its components and tools would have saved development time and reduced complexity.

Topic	Hours	Description
Scrapped First Version	15-20 hours	Initial full build attempt; ultimately discarded, but some backend reused.
Backend Logic	4 hours	Core logic: Library class, Item types, and main structure (excluding storage).
Planning (Qt, UI/UX)	1–2 hours	Project design, structure planning, and defining user interaction flow.
C++ Virtual Refresher	1–2 hours	Quick review of virtual methods and polymorphism essentials.
Storage + JSON & Debug Visitors	4 hours	Persistent data storage system and basic visitors for debug/output.
Command Line Interface	3 hours	CLI layer to interact with and test backend components.
MainWindow + Qt Research	2 hours	Built initial MainWindow and explored Qt widget system and APIs.
Views + Visitors (Index, Show, Edit, New)	16 hours (4 per view)	All major UI views with their corresponding visitor classes.
Debugging & Refactoring	6 hours	Memory management fixes, segmentation fault resolution, general cleanup.
Image Upload + Path Logic	3 hours	Integrated image selector and implemented fallback/default image logic.
Import and Export functionalities	3 hours	Implemented import/export functionalities for enhanced data portability.
Code Cleanup for Maintainability	2 hour	Structural refactor for clarity and reuse.
Final Report Writing	3 hours	Documentation of project structure, decisions, and outcomes.

Table 2: Summary of time spent on project components

9.2 C++ Language Insights

Virtual methods in C++ are fundamental for enabling polymorphism and flexibility, despite seeming complex at first. Memory management constitutes about half the work in reliable C++ applications; avoiding dangling pointers and using clear ownership conventions are critical. Smart pointers like `unique_ptr` and `shared_ptr` greatly simplify ownership logic, and I plan to use them as a default in future projects to reduce manual memory handling. Pointer logic becomes manageable with practice and is a powerful way to efficiently manage object lifetimes. Additionally, platform differences matter: on macOS uninitialized pointers often default to `nullptr`, while on Ubuntu they may point to garbage memory, causing elusive bugs. This taught me to always initialize pointers explicitly to `nullptr`.

9.3 Design & Architecture

Applying RESTful patterns outside the web context helped bring clarity to view routing and maintain modularity across UI components.

9.4 General Reflection

The project took longer than originally expected but was worth the time investment. The additional effort significantly deepened my understanding of both Qt and C++, making the learning experience valuable overall.