Project Report for Object Oriented Programming Course
Artinian Luca mat. 2052772
Virtual Library

A - Introduction

Virtual Library is a multimedia library developed in C++ used for the dynamic management of four different media types, using an interactive user interface developed with the Qt framework. The media types supported by *Virtual Library* are *Audio*, *Video*, *EBook*, and *Image*.

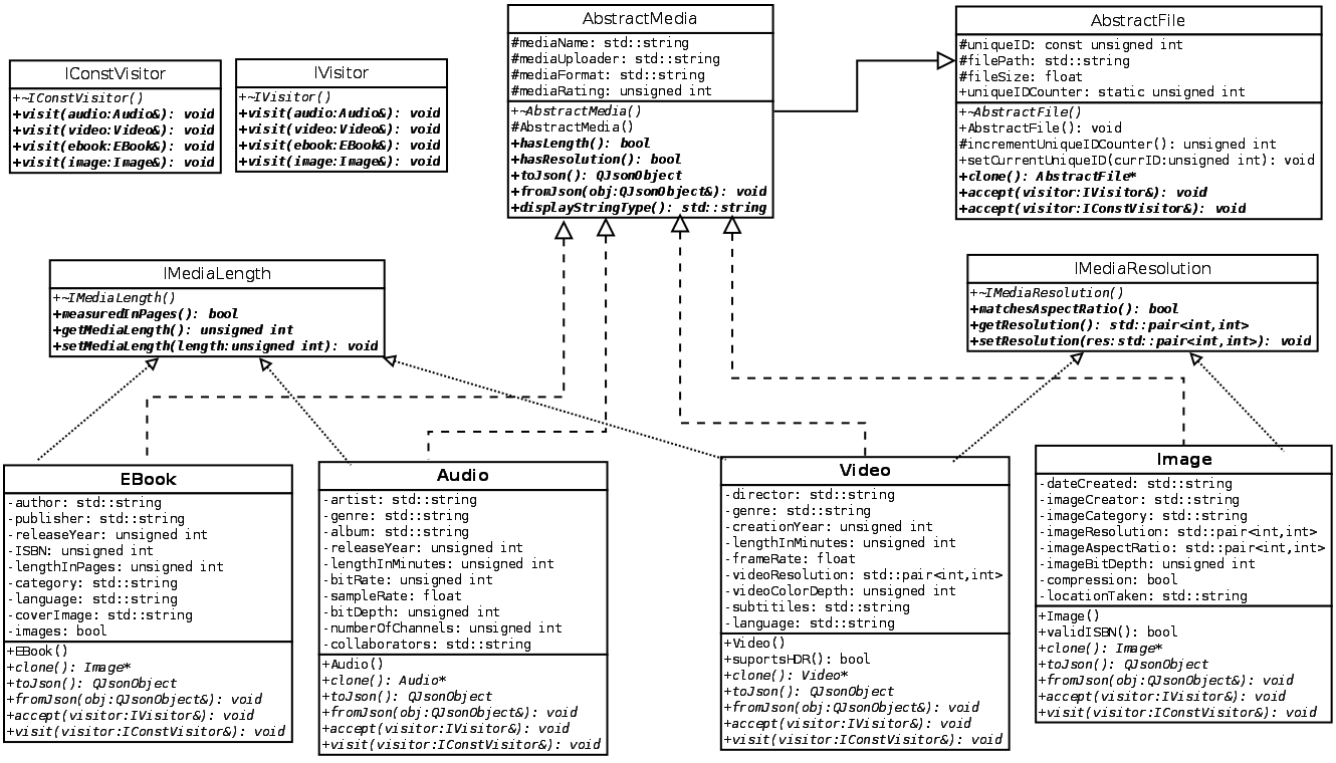
The main objective of the project was to model each media type in a way that reflects its real-world counterpart as closely as possible. I focused on both their informational and technical aspects, implementing the logic, behaviors, and attributes specific to each one, drawing inspiration from the actual media files they represent.
A secondary objective was to create an interactive user interface, which took slightly longer than anticipated due to being new to the Qt framework and its classes. The user interface is described in more detail in section (**e**) but allows users to create, view, edit, search, and remove the media items using custom widgets for each media type while trying to keep a user-friendly UI.

It follows the *MVC* (*Model-View-Controller*) design pattern to keep the logical model completely independent of the view. The logical model is entirely written in C++, except for the methods regarding data persistence.
I chose to use JSON to implement data persistence, given how Qt provides dedicated classes for it, along with classes that handle I/O operations. This allowed me to implement operations for the serialization and deserialization of the complete library to and from a JSON file. I placed a lot of focus on scalability and extensibility, to allow new media types or behaviors to be introduced without changing the underlying class structures. To achieve this, I used well-known design patterns such as *Visitor*, *Builder*, *Factory*, and *Command*.

Note: The code is commented in Italian, although I chose to write the thesis in English for clarity, as English is my native language.

B - Logical Model

The logical model is divided into three main categories: (1) the representation of the media types, (2) the classes that handle the media collection, and (3) the service classes used. The foundation of the logical model is represented by the inheritance hierarchy shown in the UML below.



Note: the UML only shows the key methods, omitting standard getters/setters, overloaded operators, and inherited methods that are not redefined by the class. This is done to maintain clarity.

1 - Representation of Media Types

The abstract base class *AbstractFile* is the foundation for all media types and represents attributes that are common to all files (path, identifier, size). The identifier is assigned at creation and remains constant for the session, backed by a static counter with an adjustable starting value. The identifier is considered to be "session-immutable", meaning that if a media is saved in one session and then loaded in another, the media will be assigned a new identifier, while keeping all other attributes unchanged. This design choice was made to simplify the management of the media items and to always ensure that each media item has a unique ID at creation. *AbstractFile* also declares the pure virtual methods *clone*, used for polymorphic copying and the two *accept* methods, used to implement the *const* and *non-const* versions of the *Visitor* design pattern. This is further discussed in section (**c**).

AbstractMedia derives from *AbstractFile* and expands the concept of a multimedia file by adding other attributes common to all media types (name, uploader, format, rating). In addition to these new attributes, *AbstractMedia* declares the pure virtual methods *toJson* and *fromJson*, which are redefined in each concrete media class to implement JSON data persistence. Each class defines static constants for default values and valid attribute ranges. The default values are used to ensure that each media type is always default-initialized with realistic default values, while the attribute ranges are used to check the validity of each media type. These ranges where chosen based on typical usage scenarios and generally accepted standards, though in some cases (e.g. constraining *Video* media items to a 16:9 aspect ratio) they reflected project-specific choices rather than universal real-world limitations.

To ensure that each media type only implements behaviours that are logical to it, I implemented an inheritance hierarchy that makes use of two abstract interfaces: *IMediaLength* and *IMediaResolution*. These are used to indicate and handle the presence or absence of a duration and resolution, respectively. For example, only the media types with a measurable duration implement the *IMediaLength* interface, and only those with a meaningful resolution implement *IMediaResolution*. This is done to further differentiate the media types, while also trying to keep their representations realistic.

All concrete media types (*Audio*, *Video*, *EBook*, *Image*) derive from *AbstractMedia*, ensuring a consistent foundation for polymorphic behaviour. This hierarchy of classes allowed me to model the media types with domain-specific attributes as well as behaviours meaningful to the media they represent. For example, *Audio* represents a song, with a duration measured in minutes. *Video* represents a movie, with a duration in minutes and a resolution constrained to reflect the 16:9 aspect ratio. *EBook* represents an eBook, with its duration measured by page count. *Image* represents a picture, characterized by its resolution and other image-specific attributes, with no concept of duration.

2 - Media Collection Management

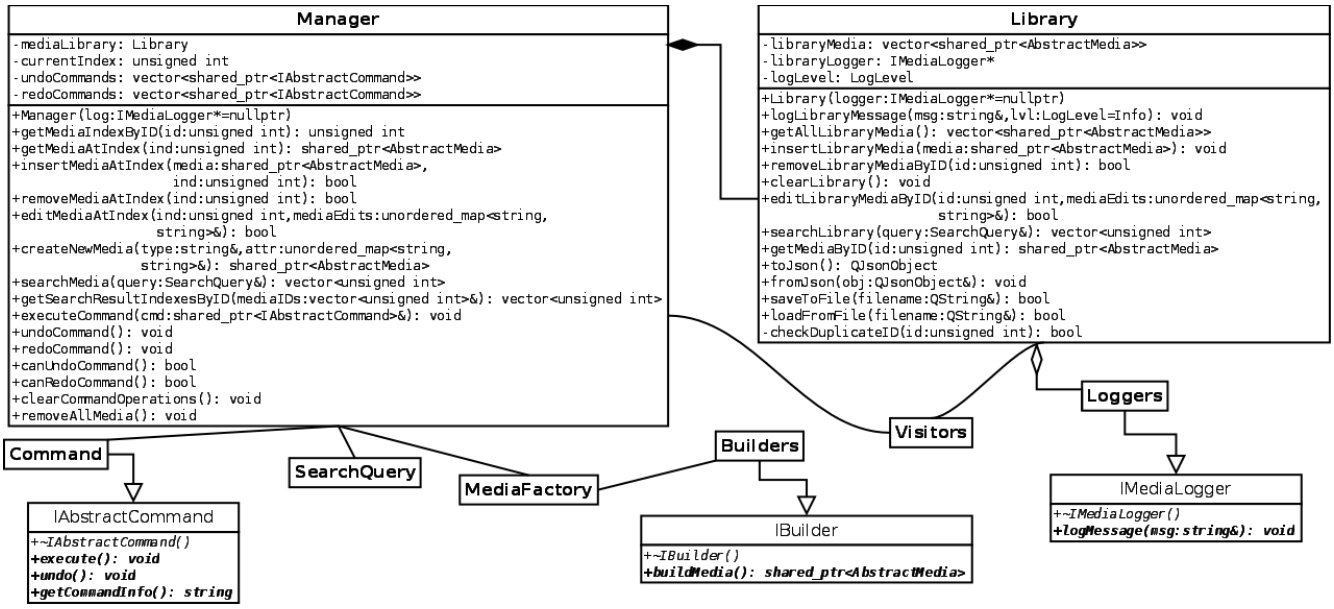
After defining the media hierarchy and core attributes, I needed some way to manage the media items themselves.
The UML on the following page shows an overview of the core logical model, management, and service classes used.

The *Library* class represents a container class for the media items, storing them in a vector of polymorphic smart pointers to the base class (*std::shared_ptr<AsbtractMedia>*).
I chose to use smart pointers in order to ensure safe memory management.
The *Library* class defines methods for the insertion, deletion, editing, viewing, fetching and scoring of the media items based on their unique identifiers. It also defines methods for saving and loading the entire library to/from a JSON file, which is explained in more detail in section (**d**).
In order to help with debugging and general testing, I decided to allow the library to be associated with a *Logger*, which logs all significant events that occur within the library (either to file with *IFileLogger* or directly to the terminal, via *IConsoleLogger*). The logging is done based on a severity level defined in the enum class *LogLevel*, and methods for setting the log level as well as the type of logger, are implemented in the *Library* class.

The *Manager* class extends the *Library* class and serves as a bridge between the logical model and the view.
It supports all the operations provided by *Library*, extending them to work based on a current index.
It also introduces more complex methods, such as saving and loading the library from a JSON file specified by the user, creating new media items (using *MediaFactory*), and searching the media collection using one or more search parameters (using *SearchQuery* and the concrete visitor *SearchVisitor*).

Manager also implements the Undo/Redo operations through the use of the *Command* design pattern, encapsulating each operation done by *Manager* as a single object with its own information and state. This allowed me to implement Redo/Undo operations for the insert, remove, and edit operations performed by the user while providing information on what the Redo/Undo operation is relating to.
To achieve this I used two vectors of smart pointers of type *std::shared_ptr<IAsbtractCommand>*, which store the operations that allow a possible Redo/Undo request made by the user along with two informational strings that are updated with the last Undo/Redo operation available. The *Command* design pattern is explained in more detail in section (**c**).

Note: In the UML below, only key associations and abstract interfaces are shown for clarity (concrete implementations are omitted).



3 - Service and Utility Classes

Since each media type declares a long list of attributes, I decided to adopt the *Builder* design pattern to separate the construction of each media from its actual representation. I decided to use a concrete builder for each media type, all of which share a common abstract interface, *IBuilder*, which declares the pure virtual method *buildMedia*.
Each concrete builder handles the construction of its specified type, ensuring that each media is either constructed with valid default values or values valid for its given type. The construction operation is implemented as an edit operation, as each concrete builder implements its operation starting from either a default-initialized media type or an existing media type passed by constant reference.
The media's attributes are represented as key-value pairs in a *std::unordered_map<std::string, std::string>*, corresponding to the media's attribute names and values. Each concrete builder declares a static constant vector of strings, corresponding to the attribute's names and is declared *public* as it is also used in other operations such as creation, editing, and the creation of the custom widgets in the *View*.

Manager implements the creation of new media items using the *MediaFactory* class, which implements the *Factory* design pattern. *MediaFactory* accepts a string indicating the type of media to create and the map of attributes with which to create it. The string indicating the media type is used for simplicity, as it is typically provided by user input or read from a JSON file, and is not a substitute for polymorphism. *MediaFactory* then delegates the creation operation to the appropriate concrete builder.

Manager implements the search operation using the struct *SearchQuery* which is used to set either common or type-specific search parameters. The search results are stored in a vector containing the identifiers of the media items found, which can then be easily converted into a vector containing their corresponding indexes so that *Manager* can operate on them.

C - Polymorphism

Each concrete media class is essentially a "data container" that encapsulates all the necessary attributes, getters/setters, and minimal logic required to represent its specific type. Because each media type is unique, I needed a way to implement type-specific behaviours and operations, without modifying their underlying structure. This is why I chose to use the *Visitor* design pattern, which allowed me to decouple the logic for operations such as validation, editing, viewing, scoring, and searching. Rather than implementing these directly in the media types, I used dedicated visitor classes.

As mentioned above, the concrete media types implement the pure virtual methods *clone*, *hasLength*, *hasResolution* and those defined in the *IMediaLength* and *IMediaResolution* interfaces (where applicable).
While these are used to implement polymorphism, such as polymorphic copying and differentiating media items based on their capabilities, they represent a more "trivial" use of polymorphism and will therefore not be explained further.

The "non-trivial" use of polymorphism in *Virtual Library* is mostly seen in the *Visitor* design pattern. Each concrete media type implements two *accept* methods, which are used to implement the *Visitor* design pattern via the two abstract interfaces *IVisitor* and *IConstVisitor*. Specifically, *IConstVisitor* is used by dedicated visitor classes that perform read-only operations, ensuring const-correctness and leaving the media unchanged. Conversely, *IVisitor* is implemented by dedicated visitors that may perform operations which modify the media.

Both abstract interfaces define four pure virtual *visit* methods, which are then implemented in the concrete visitor classes according to the operation and media type. This pattern improved the extensibility and scalability of *Virtual Library* since adding new behaviours now only required a new visitor class, with no changes needed to existing media types.
It also avoids having to use *RTTI* operators (such as *dynamic_cast* or *typeid*), relying instead on double-dispatch to select the correct operation at runtime.

The *Visitor* design pattern is deeply integrated into *Virtual Library's* architecture and enables most of the advanced operations expected by the media types. Some examples of concrete visitors include:

- *ConcisePrinter* and *DetailedPrinter* (*IConstVisitor*): print the media item information directly to the terminal in either a concise or detailed format. These are not actively used in the project but I chose to keep them as examples of the *Visitor* design pattern.

- *MediaValidator* (*IConstVisitor*): checks the validity of the media items before placing them in the library by enforcing both general and type-specific constraints. This visitor plays an important role within the project as most operations performed by other visitors assume that the media items they act on are valid, having already been validated by *MediaValidator*. It is also where the custom exception *MediaValidatorException* is defined, which is the primary exception thrown when validation errors occur in a create or edit operation, providing descriptive messages on the potential issues encountered.

- *MediaEditor* (*IVisitor*): enables editing of media items. It makes use of the *Builder* design pattern, selecting the appropriate concrete builder depending on the media's type and using *MediaValidator* after the edit operation is complete to ensure the edited media is still valid.

- *ScoreVisitor* (*IConstVisitor*): enables a "scoring" mechanism for the media types by calculating a numeric score based on their overall "quality" as perceived by the user. Each media type is evaluated with type-specific logic, depending on the attributes most relevant to the "quality" of its given type.

- *SearchVisitor* (*IConstVisitor*): implements the search operation using *SearchQuery*, supporting both general and type-specific search parameters. The search operation is implemented as being case-insensitive, using both exact and partial matches in its implementation.

In short, by using the *Visitor* design pattern I was able to enrich the media types with specific behaviours and operations unique to their type, providing a "non-trivial" use of polymorphism in their implementation.
More information on each concrete visitor can be found in the code comments.

Another "non-trivial" use of polymorphism appears in the *Command* design pattern used by the *Manager* class.
Each operation handled by *Manager* is encapsulated as an object implementing the *IAbstractCommand* interface, which declares the pure virtual methods *execute*, *undo* and *getCommandInfo*. The classes *InsertCommand*, *EditCommand*, and *RemoveCommand* redefine these to implement the specific logic for each operation, allowing *Manager* to uniformly handle different commands via polymorphic pointers. This enabled *Manager* to implement Undo/Redo functionality using backups of the media items in case of a "rollback" situation, using the *clone* method.

Logger represents a more straightforward use of polymorphism. It allows for different concrete loggers but does not leverage the kind of double-dispatch or behavioural extension found in *Visitor* or *Command*.

Finally, although not part of the logical model, the *Visitor* design pattern is also leveraged in the *View* to dynamically generate the appropriate UI widgets for each media type (e.g. *MediaViewerFactory*, *MediaEditorFactory*).
This is further explained in section (**e**).

D - Data Persistence

As mentioned earlier, each media type implements its own *toJson* and *fromJson* methods, first defined as pure virtual methods in *AbstractMedia*. These are responsible for serializing and deserializing media objects by mapping both their common and type-specific attributes to simple key-value pairs in JSON format.

To distinguish between different media types, each JSON object that represents a single media item always includes a "mediaType" key to indicate its type.

Serialization and deserialization of the entire library is handled by the *toJson* and *fromJson* methods defined in the *Library* class. The entire media library is represented as a JSON object containing an array of media items under the "media" key, with each element in the array being a JSON object produced by the respective media's *toJson* method.
This functionality is extended by the *saveToFile* and *loadFromFile* methods in *Library*, which allow the user to serialize or deserialize the library from a specified path (typically chosen through a dialog popup in the UI).

I chose to use the JSON format as Qt provides classes such as *QJsonObject*, *QJsonArray*, and *QJsonDocument*, along with other classes dedicated to I/O operations. It took some time to study their documentation, but they facilitated the data persistence mechanism implemented in *Virtual Library*.

The JSON file "example_library.json" includes an example library with three media items for each type.

E - Additional Functionalities

The additional functionalities implemented in *Virtual Library* can be categorized into two categories: those implemented in the logical model, and those implemented in the user interface.

The additional functionalities implemented in the logical model are as follows:

- Management of four distinct media types: *Audio*, *Video*, *EBook*, *Image*
- Safe memory management by using smart pointers (*std::shared_ptr*)
- Serialization and deserialization of the entire library to/from a JSON file
- Validation of media items using *MediaValidator*, providing descriptive feedback on potential issues/errors
- Search operations using type-specific search parameters (*SearchQuery* + *SearchVisitor*) with case-insensitive search supporting both exact and partial matches
- Scoring of the "quality" of different media types based on their specific attributes (*ScoreVisitor*)
  - Implemented in the custom *Viewer* widgets described in the next category
- Undo/Redo for each insert, edit, and remove operation in *Manager* using the *Command* design pattern
- Possibility to associate a *Logger* to log all significant operations and results

The additional functionalities implemented in the user interface are as follows:

- User interface containing menus, toolbar, and a splitter to resize the two main panels
- Use of status bar to provide descriptive messages on the operations performed
- Keyboard shortcuts and tooltips to help user navigate the application

- Ability to view the entire library as a list of selectable media items in the left panel (*ConciseViewer*)
- Custom widgets for viewing (*Viewer*) each media type with buttons for scoring and other type-specific operations
- Custom widgets for creating (*Creator*) each media type with required fields displayed in bold
- Custom widgets for editing (*Editor*) each media type
- Custom widget for searching (*SearchWidget*) using type-specific parameters (*SearchWidget*)
- Ability to save/load the entire library to a specified path
- Current index counter using a spin box to keep track of the selected media item
- Automatic enabling/disabling of buttons based on current operation
- File dialogs with extension filters during saving/loading

F - Projected vs Effective Hours

Activity	Projected Hours	Effective Hours
Studying the Qt framework	10	14
Modeling the logical mode	3	3
Developing logical model and service classes	5	5
Implementing data persistence with JSON	2	4
Developing the GUI	12	14
Developing the GUI custom widgets	8	10
Test and debugging	3	2
Project Report	2	3
total	45 hours	**55** hours

I took longer than anticipated to develop the GUI and learn the Qt framework, especially in regards to implementing data persistence using the classes provided by Qt.
In particular, it took more time that expected to develop the custom widgets in the GUI for each operation and for each media type, as this meant reading up on the Qt classes and their documentation. The custom widgets in the GUI were not part of the project specifics, but I decided to implement them in order to create a user-friendly interface.
Finally, it took longer than expected to write up this project report, as I initially meant to write it in Italian.
However, as English is my native language, I made the switch around halfway through.