CS39440 - GamePile

Major Project Report

Department of Computer Science,
Aberystwyth University

Last updated: **2nd May, 2024**

v1.0 - RELEASE

Produced by:

Kal Sandbrook
kas143@aber.ac.uk
BSc in *Computer Science - G400 BSc*

Supervised by:

Dr. Edore Akpokodje
eta@aber.ac.uk
Lecturer in Computer Science

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name: Kal Sandbrook
Date: 9th April 2024

[illegible]

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Kal Sandbrook
Date: 9th April 2024

[illegible]

Generative AI

No Generative AI tools have been used for this work.

Name: Kal Sandbrook
Date: 9th April 2024

[illegible]

Acknowledgements

With thanks to my supervisor, Dr. Edore Akpokodje, for his guidance and support throughout the development of this project. I would also like to thank my friends and family for their support throughout this project and my time at university as a whole - especially my uncle, without his support none of this would have been possible in the first place.

Abstract

The GamePile project aimed to create a native desktop application that helps users to manage their video game backlogs (lists of games they want to play) and libraries (games they own). The application allows users to add games to the library and mark them as part of their backlog, in progress or completed. GamePile also allows users to search through their games and filter them based on various attributes such as genre, platform or completion status. It also allows users to search for games to add to their library via the use of a Third-Party API. This uses a fuzzy search algorithm to allow users to search for games even if they are unsure of the exact name. The application also allows users to view detailed information about the games in their library, such as the aforementioned attributes. The project was developed using C++ and Python, with the main application using the Qt framework and the API module using the requests library. Development was based on a Kanban framework, with a weekly log being kept to document progress and any issues that arose. The project was developed with a focus on Linux compatibility, with support for other operating systems coming second. Unlike competitors, GamePile provides a native solution for managing game libraries and backlogs, offering better performance and integration with the users system without the need to open a web page and consequently without requiring an internet connection.

Contents

1 - Background, Analysis & Process 6
1.1 - Background 6
1.1.1 - Aims 6
1.1.2 - Research into Similar Tools 6
1.1.3 - Motivation 7
1.1.4 - Research of UI Design 8
1.1.5 - Options for Third-Party API 8
1.2 - Analysis 9
1.2.1 - Identification of Requirements and Objectives 9
1.2.2 - Choice of Technologies 9
1.2.3 - Alternative Approaches 10
1.3 - Process 10

2 - Requirements 11
2.1 - Functional Requirements 12
2.2 - Non-Functional Requirements 13

3 - Design 15
3.1 - Programming Language 15
3.2 - Architecture 15
3.3 - Data Structures 16
3.4 - Data Persistence 16
3.4.1 - Settings and Preferences 17
3.4.2 - Icons and Other Assets 17
3.5 - User Interface 18
3.5.1 - Main Features 19
3.5.2 - Game Edit Dialog 19
3.5.3 - Filters 19
3.5.4 - Game Details 19
3.5.5 - Model / View Programming 20
3.5.6 - Game Delegate 20
3.6 - API Integration 20
3.6.1 - API in the Main Application 21
3.7 - Algorithms 21
3.7.1 - Fuzzy Searching 21
3.8 - Class Diagram 22

4 - Implementation 23
4.1 - Development Environment 23
4.1.1 - Version Control 23
4.1.2 - CMake and Build System 23
4.1.3 - Applications and Tools 24
4.2 - Early Development 24
4.3 - Later Development 25
4.4 - Issues Encountered 25
4.4.1 - Performance 25
4.4.2 - Filters 25

4.4.3 - Availability of APIs 26
4.5 - Review against Requirements 27
4.5.1 - Review of Functional Requirements 27
4.5.2 - Review of Non-Functional Requirements 27

5 - Testing 28
5.1 - Approach 28
5.2 - Continuous Integration and Unit Testing 28
5.3 - Manual Testing 29

6 - Evaluation 30
6.1 - Achievement of Aims and Objectives 30
6.2 - Reflection on Development Process 30
6.3 - Choice of Tools and Development Environment 30
6.4 - Considerations for Future Projects 31

Bibliography 32

Appendices 34
E - Third-Party Code and Libraries 34
F - Statement of Tools Used 34
G - Class Diagram 35
H - Manual Testing Table 36

1 - Background, Analysis & Process

1.1 - Background

1.1.1 - Aims

The aim of this project was to create a native desktop application that helps users to manage their backlogs (lists of games they want to play) and libraries (games they own). This allows users to add games to the library and mark them as part of their backlog, in progress or completed. GamePile will also allow users to search through their games and filter them based on various attributes such as genre, platform or completion status.

Further to this, users will also be allowed to search for games to add to their library via the use of a Third-Party API. This will use a fuzzy search algorithm to allow users to search for games even if they are unsure of the exact name. The application will also allow users to view detailed information about the games in their library, such as the aforementioned attributes.

Some optional features that could be implemented include the ability to export a users game library graphically akin to an old-school forum signature, to facilitate sharing of game completion progress on forums and social media platforms and a recommendation system that suggests games to add to the users library based on their existing library and completion status.

1.1.2 - Research into Similar Tools

Initial research for this project involved investigation into similar tools that already exist. These tools were found by searching online for “game backlog manager” and “game library manager”. The most popular and recommended tools were found to be websites called “How Long To Beat”[1] and “Backloggery”[2]. These websites allow users to track their game completion progress and backlog, but they are web-based and do not offer a native desktop application, they are also limited to the library of games that they have in their database, not allowing for users to easily add their own games.

The research into these tools proved useful as it allowed for the mapping out of features that are essential for a backlog manager. In How Long To Beats case, the ability to track completion progress and the ability to search for games in a database were key features. Backloggery brought to light the idea of a graphical representation of a users library, which could be a unique feature to implement in this project. However, the design of the website was found to be very outdated and not visually appealing, with the sites design not having majorly changed since 2007, which could be a key area for improvement in this project, along with the performance improvements that come with a native application.

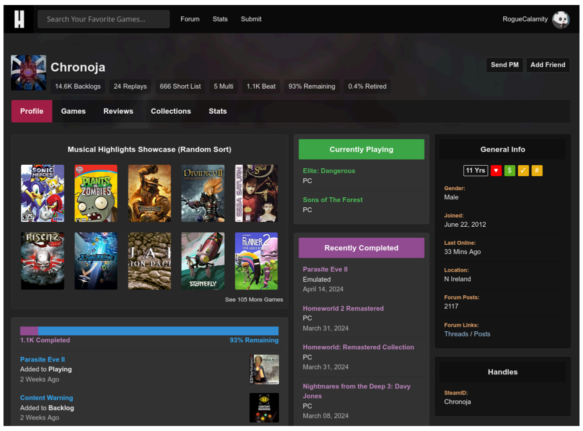


Figure 1: A screenshot of a profile on How Long To Beat. [1]

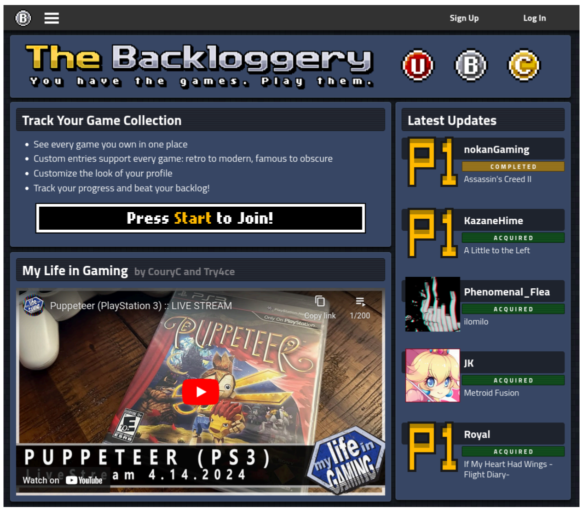

The Backloggery
You have the games. Play them.
Track Your Game Collection
See every game you own in one place
Custom entries support every game: retro to modern, famous to obscure
Customize the look of your profile
Track your progress and beat your backlog!
Press Start to Join!
Latest Updates
My Life in Gaming

Figure 2: A screenshot of the front page of The Backloggery. [2]

Another technology that was investigated during development was the "Video Game Preservation Platform" Lutris[3] - however, Lutris is more focused on game installation and management, as opposed to tracking completion and a backlog. This tool was useful for understanding how an application could be used to launch games, which could be a feature to implement in this project. The design of Lutris was also taken into account, as it contains a modern design running on a native ui framework (GTK[4]).

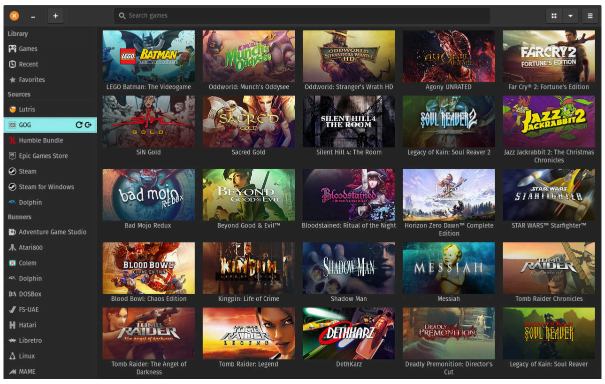


Figure 3: A screenshot of the Lutris application. [3]

1.1.3 - Motivation

The motivation to undertake this project comes from a personal interest in the topic, as efficient tracking of game libraries and backlogs is not something readily available in the market, at least, not in a native desktop setting. As existing tools are mainly web-based, they do not offer the same level of performance and integration that a native application could offer, such as being able to launch games directly from the application or being able to use the application offline.

This laid out some of the priorities for the project, such as performance improvements over existing tools, a modern and visually appealing design and the ability to be able to use the application without an internet connection. To this end, investigation into what technologies could be used to achieve these goals was undertaken.

1.1.4 - Research of UI Design

Research into UI design was conducted as a part of this project to ensure that the application was user-friendly and visually appealing. This research involved examining modern desktop applications, notably those in the KDE ecosystem, as well as inspecting the design guidelines for the Qt Framework [5] (The "4Cs": Consistency, Continuity, Context and Complementary being considered throughout design iterations.) and the KDE Human Interface Guidelines[6] (which can be summarised as "Simple by default, Powerful when needed.")

These principles promote reusing design patterns from other applications so that users can easily understand how to use the application, and that complex tasks should feel simple to the user. This research was used to inform the design of the application during development, ensuring a intuitive and user-friendly experience.

1.1.5 - Options for Third-Party API

For the API module of the project, a third-party API was required to allow users to fetch game data from the internet. Ideally, this API would allow for searching for games by name and return detailed information about the game - at the very least, the release date, genre, platform and a description of the game.

This API would also need to be free to use, and preferably not involve a complicated authentication process to access the data (such as having to sign up for an API key). Appropriate Licensing is also a consideration, as the API will be used in an educational capacity.

Two initial options were IGDB[7] (Internet Game Database) and the Steam Web API. The IGDB API was ruled out as it requires the user to have a Twitch account in order to sign up for an API key - which would require functionality that is out of scope for this project, requiring the application to sign up with the *Twitch Developers* system.

The Steam Web API does not require any authentication to access the data, and was found to be a good option for this project. The API does not support searching, but a list of games can be obtained and searched through locally. Steam Web API also provides information of a requisite level of detail for this project.

1.2 - Analysis

Whilst this project may seem simple on the surface, there are a number of considerations to be made. Firstly, the scope of the project had to be defined. The project was split into two main components - the main application and the API module. The main application would be responsible for managing the users library and backlog, whilst the API module would be designed to fetch game data from the internet. If the social features and recommendation system were to be implemented, they would be a part of the main application - although likely in seperate modules.

1.2.1 - Identification of Requirements and Objectives

The requirements for this project were identified through the research conducted in the background section. As a significant portion of the other tools investigated were web-based, there are considerations this project had to make which are not relevant for web-based apps. The key requirements for the project were as follows:

- The application must allow users to add games to their library and mark them as part of their backlog, in progress or completed.
- The application must allow users to search through their games and filter them based on various attributes.
- The application must allow users to search for games to add to their library via the use of a Third-Party API.
- The application must allow users to view detailed information about the games in their library, such as the aforementioned attributes.
- The application must be visually appealing and user-friendly, following modern design principles.
- The application must be performant, able to stand up even with a very large library of games.

These Requirements are discussed in more detail in Section 2.

1.2.2 - Choice of Technologies

The choice of language for this project was a matter of consideration up until the beginning of development. The two main languages considered were C++ and Python, and were the two languages used in the development of this project.

C++ was chosen for the main application due to its unrivalled performance, high suitability for Object-Oriented problems and, via use of the Qt[8] platform, a native and robust UI framework. Python was chosen for the API module due to its excellent networking libraries and ease of use, along with its libraries for fuzzy string matching.

The Qt Framework has two main approaches for GUI development. Qt Widgets, which is more representative of a traditional desktop application, and Qt Quick, which is a more modern approach using the QML language. Qt Quick is well suited for rapid development, as QML is a declarative language that integrates with JavaScript-like syntax. However, Qt Quick also poses two problems for this project - the first being that it is not as performant as Qt Widgets, and the second being that it introduces a level of separation between the UI and the backend that would require more work to integrate with the C++ backend. For these reasons, Qt Widgets was chosen as the more suitable option for this project.

Another advantage of using Python for the API module is that it allows for easy packaging of the module into an executable, which can be used standalone of the main program, allowing for easy testing and debugging of the module. This was achieved using the pyinstaller[9] library.

Finally, the method of persistent data storage had to be considered. Due to the nature of the data being stored, a relational database was chosen as the method of storage, in particular an SQLite database, due to its lightweight nature and ease of use. In a bigger project, a more robust database system such as PostgreSQL could be considered.

1.2.3 - Alternative Approaches

Many other languages were considered for this project, such as an entirely Python-based solution. However, Python does not have the same level of performance as C++, and would not fulfil the performance objectives of the project. Python also does not have an object-oriented system that is as robust as C++, which would make the project harder to maintain and increase the difficulty of debugging the application.

Another language that was considered for this project was the increasingly popular Rust language. Rust is known for its performance and rigorous safety requirements, which would have made it a good choice for a project such as this. However, a significant caveat to using Rust for this project is the lack of a mature UI tooling ecosystem.[1] Whilst there are bindings (bindings being a way to use a library from another language) for Qt in Rust, this would essentially involve doing the majority of the work in C++ regardless, which would defeat the purpose of using Rust in the first place.

Java and C# were also considered, but were ruled out - Java due to its performance and C# due to the impracticality of using it on Linux systems. Java also has an absence of modern UI tooling, with Swing and JavaFX being the only real options, both of which are outdated and not visually appealing. QtJambi was briefly investigated as a potential solution, but there was found to be a lack of documentation and community support for the library.

Mentioned earlier as the UI framework used for Lutris, GTK[4] was also considered as a potential UI framework for this project. However, GTK is not built for development in C++, and has to be used similarly to programming in C. GTK is also exclusive to the Linux Operating System, and whilst experimental Windows support is available, this would significantly increase any work required to make this application cross-platform.

1.3 - Process

When planning development, a Kanban framework was used to manage the project. The use of a Kanban Board is a common practice in software development, and one of the few practices that can be used for a solo project.

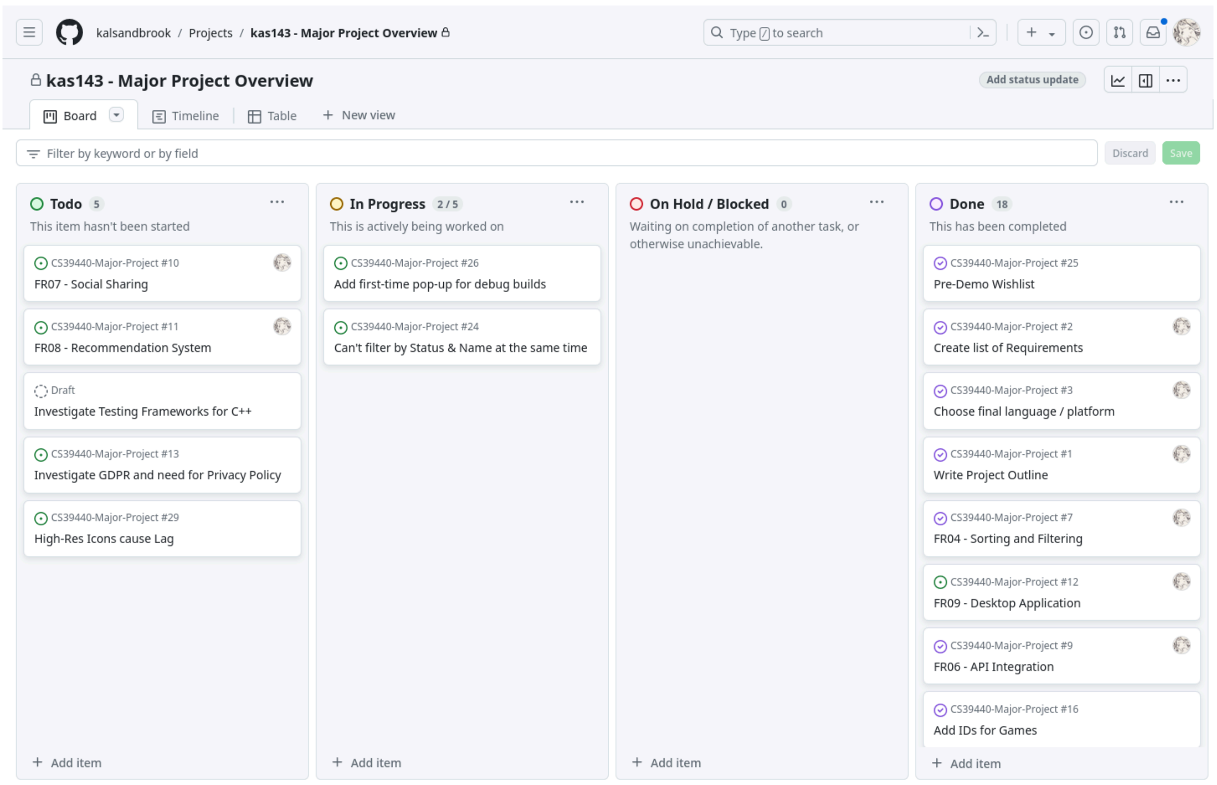
The Kanban Board was used to manage the project by decomposing the project down into smaller tasks and assigning them to one of three (although there are four columns in total) columns on the board. The columns were “To Do”, “In Progress” and “Done”. The “To Do” column contained all tasks that needed to be completed, with cards sorted by priority; the “In Progress” column contained tasks that were currently being worked on; and the “Done” column contained tasks that had been completed. A WIP Limit was set so only three tasks could be deemed to be “In Progress” at any one time, to prevent jobs from being left unfinished.

[1]See the website https://areweguiyet.com/ for a informal view on the state of GUI in Rust.

Using the "GitHub Projects" feature to host the Kanban Board, the project was able to be effectively managed. Using this feature allowed issues to be linked to cards on the board, which allowed for easy tracking of progress and completion of tasks.

When Functional Requirements were identified, they were added as issues to the GitHub repository and linked to cards on the Kanban board, serving as small milestones. Cards were also labelled with the type of task they were such as "Feature", "Bug" or "Documentation".

Certain cards were linked to larger milestones, such as the Mid-Project Demonstration, clearly showing what features I wanted to have in place by that point. This allowed for a clear plan in regards to the timing of the project, and what features were to be implemented.


kalsandbrook / Projects / kas143 - Major Project Overview
Type / to search
Add status update
Board
Timeline
Table
New view
Filter by keyword or by field
Discard
Save
Add item

kas143 - Major Project Overview

Todo 5 — This item hasn't been started
- CS39440-Major-Project #10 — FR07 - Social Sharing
- CS39440-Major-Project #11 — FR08 - Recommendation System
- Draft — Investigate Testing Frameworks for C++
- CS39440-Major-Project #13 — Investigate GDPR and need for Privacy Policy
- CS39440-Major-Project #29 — High-Res Icons cause Lag

In Progress 2 / 5 — This is actively being worked on
- CS39440-Major-Project #26 — Add first-time pop-up for debug builds
- CS39440-Major-Project #24 — Can't filter by Status & Name at the same time

On Hold / Blocked 0 — Waiting on completion of another task, or otherwise unachievable.

Done 18 — This has been completed
- CS39440-Major-Project #25 — Pre-Demo Wishlist
- CS39440-Major-Project #2 — Create list of Requirements
- CS39440-Major-Project #3 — Choose final language / platform
- CS39440-Major-Project #1 — Write Project Outline
- CS39440-Major-Project #7 — FR04 - Sorting and Filtering
- CS39440-Major-Project #12 — FR09 - Desktop Application
- CS39440-Major-Project #9 — FR06 - API Integration
- CS39440-Major-Project #16 — Add IDs for Games

Figure 4: A screenshot of the GitHub Projects board used for this project.

A weekly log has been kept throughout the project to document progress and any issues that arose. Each week was broken up into the objectives for the week, the tasks completed, the challenges faced and the plans for the upcoming week. This log was used primarily as a starting point for the weekly meetings with the project supervisor, it also served as a good way to keep track of progress and the pace of development.

2 - Requirements

The requirements for this project, identified through research detailed in the background section, underpinned the majority of the development process, dictating the features to be implemented and key design decisions. Requirements were split into two main categories: Functional Requirements and Non-Functional Requirements.

Functional Requirements are the features that the application must have, and coinciding with the Kanban methodology, serve as small milestones for the project. Non-Functional Requirements are the critical aspects that are not directly related to the features of the application, such as performance and usability, these were considered throughout development.

This project also has a set of Optional Functional Requirements, which are features that could be implemented if time allowed.

Requirement IDs are used to reference requirements throughout the document, and are formatted as "FRXX" for Functional Requirements, "NFRXX" for Non-Functional Requirements and "OFRXX" for Optional Functional Requirements.

2.1 - Functional Requirements

FR01 - Game Management

Users will be able to perform CRUD operations on games in their library. This includes adding games to the library, removing games from the library, updating the details of a game and viewing the details of a game. Games will have attributes such as title, genre, platform, release date, completion status and a description.

FR02 - Backlog Management

Users will be able to mark games in their library as part of their backlog. These games will be able to be displayed as part of a seperate list.

FR03 - Progress Tracking

The application will allow for games to have their progress tracked, with statuses such as "In Progress", "Completed", "Not Started" and "Abandoned". This will allow users to easily see which games they have completed and which they have yet to start.

FR04 - Sorting and Filtering

Users will be able to sort and filter their games based on information about the game itself (such as name, release date or genre) or meta-information about the game, such as completion status or any user-defined tags.

FR05 - Manual Game Entry

The program will allow for users to add games to their library manually, specifying data themselves, rather then fetching it via an API, for games that are not in the API's database and in the event that an internet connection is not available. This will also aid in development and testing before the API module is complete.

FR06 - API Integration

Users will be able to use a third-party API to automatically fetch game information based on a given name. This will aid usability by increasing the speed at which games can be added to the library, and will allow for more detailed information to be displayed about the game. Users will be potentially be able to pick from multiple third-party APIs, if available.

FR07 - Desktop Application

The application will be a native desktop application, allowing for good performance and offline use. This will also allow for integration with the users system, such as being able to launch games directly from the application.

OFR01 - Social Sharing (Graphical Library Export)

The application will allow users to export a graphical representation of their library, akin to an old-school forum signature. This will allow users to share the games they have completed over a period of time, and will be a unique feature of the application. A visual representation will allow for users to share their progress online, withut ties to any particular platform. This is an optional feature.

OFR02 - Recommendation System

The application will be able to suggest games to add to the users library based on their existing library and completion status. This will allow users to easily find new games to play. The implementation of this feature is unlikely, due to the complexity of recommendation systems and the time constraints of the project. This is an optional feature.

2.2 - Non-Functional Requirements

NFR01 - Usability

The application must have an intuitive, easy-to-use interface that is able to be navigated by users with limited technical knowledge. The application should also be visually appealing, following modern design principles, including the KDE Human Interface Guidelines[6].

NFR02 - Performance

The application will be performant, responding to user interaction promptly and being able to handle game data efficiently. This is an important need, especially the considering the very large volume of data the application could be dealing with.

NFR03 - Reliability

The application make sure to store data in a robust format, taking measures to tackle potential data loss or corruption. In the event of data loss, the application should be able to recover. It is also important that the application is able to handle errors gracefully, providing useful error messages to the user and avoiding crashes where possible.

NFR04 - Compatibility

Whilst the application is being developed with the Linux Operating System in mind, it should be able to easily be made to run on Windows or potentially MacOS. In order to achieve this, the application should avoid using features specific to a particular operating system and use platform-agnostic code where possible.

NFR05 - Maintainability

The codebase of the application should be well-documented and well-structured, to aid in future development and maintenance. Features should be designed in a modular manner, with the addition of new futures being kept in mind. Code should be kept in a state where somebody else could pick up the project and understand it.

NFR06 - Interoperability

As the application is using and integrating with third-party APIs, it is important that the application is able to handle changes to the API gracefully. The application should be able to handle changes to the API without crashing, and should be able to provide useful error messages to the user in the event of an API failure. Relevant standards and protocols should be adhered to.

NFR07 - Localization

Whilst the application will not be translated into multiple languages at this time, the application should be designed in a way where translation is possible in future. A theoretical translator should be able to easily translate the application into another language without a significant knowledge of programming.

3 - Design

3.1 - Programming Language

The first design decision made for this project was the choice of programming language, and deciding on the specific environment to use. C++, using the Qt platform, and Python were both chosen for the project. These decisions were explored in Section 1.2.2.

Specifically, this design can be expected to make use of: the Qt Widgets module, to provide the basic tools to create a user interface; the Qt SQL module, to provide a framework to interact with a database; and the Qt Concurrent module, to allow for concurrent programming.

Python will be used for the API module, using its requests library to make HTTP requests to the chosen API. TheFuzz library will be used for fuzzy string matching, and PyInstaller will be used to package the API module into an executable, in order to interoperate with the main application.

3.2 - Architecture

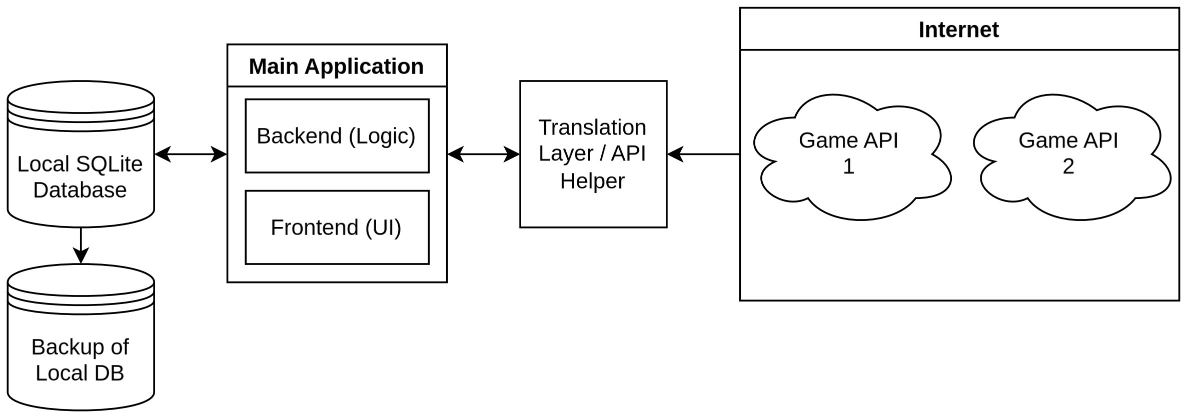


Figure 5: A diagram showing the architecture of the application.

One of the key design decisions made at the beginning of development was deciding on a sensible and robust architecture.

The main application is primarily split into two main components - the backend and the frontend. The backend is responsible for managing the data and logic of the application, whilst the frontend is responsible for the display of and user interaction with the data. This seperation of responsibilities allows for easy maintenance and shorter build times. This also aids in the proper testing of the application, as the backend logic can be tested independently of the frontend.

Data is stored locally in an SQLite database, which is accessed by the data backend. More detail can be seen in Section 3.4.

The *"Translation Layer / API Helper"* is a Python Command Line Interface (CLI) application, created as a part of the implementation, that accesses the third-party APIs based on a given name. This application returns game data to the main data backend in a compatible format.

3.3 - Data Structures

The application has a number of data structures that are used to store data about games. The main data structure is the Game class, which stores information about a game such as the title, genre, platform, release date, completion status and a description. This class is used to represent games in the library of the user. This class will not involve any complex logic or methods, as it is primarily a data structure.

The Game data structure will also include an ID, which will be used internally to uniquely identify games. This will not be displayed to the user, and be protected from user modification.

Fields that can have multiple values, such as a games genres, developers or publishers will use a unique data structure (internally referred to as an "Attribute") to store these values. Available attributes for a game will be stored as an enum
- this will allow for easy addition of new attributes in the future.

Attributes are stored in a seperate table in the database, to avoid many-to-many relationships. Filter Widgets are automatically generated based on the available attributes, allowing for easy filtering of games based on these attributes. The main motivation for this design decision is to reduce the repitition of code and maintain extendability of the application.

3.4 - Data Persistence

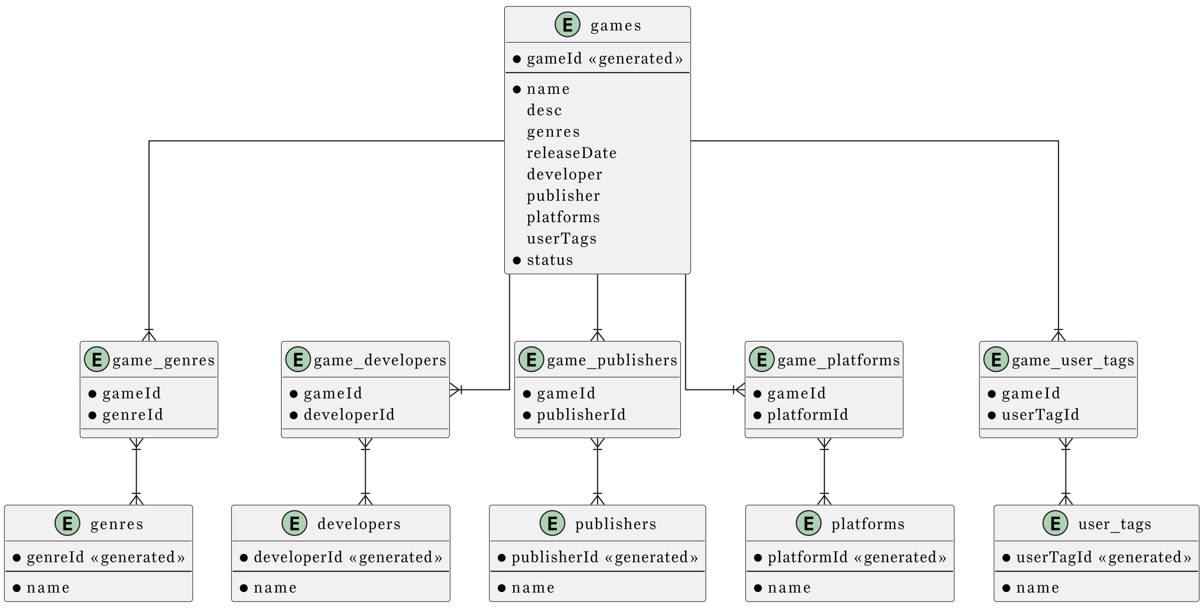


Figure 6: The database schema for the application.

GamePile requires the storage of data about games and their attributes. This data is stored in an SQLite database, which is a lightweight, file-based database system that is appropriate for use here due to its simplicity and ease of use.

The main table is the `Games` table, which contains all the stored games. Each game has a unique ID, which functions as its primary key (`gameId`). The table also contains fields for the name, description, release date and completion status of the game. Complex attributes, such as genres, developers and publishers are stored in seperate tables.

Each attribute has a table named after itself (i.e. `genres`), which contains a list of auto-generated IDs that serve as the primary key for the table. The name of each attribute is also stored in the table. Between the `games` and attribute table, there is a junction table (`game_genres`) that stores the relationship between games and their attributes. This is to ensure that the database is in the third normal form, and to avoid many-to-many relationships.

Storing data in the Third Normal Form, and as such removing transitive dependencies, is important for this project as it ensures efficient storage and retrieval of data, eliminating data duplication. As large volumes of data are likely to be encountered when storing a collection containing a high level of entries, efficient storage is an important consideration.

This data will use the QStandardPaths class to determine an appropriate location to store the database file, differing based on the operating system the application is running on. For example, on Linux, the database file be stored in the `~/.local/share/GamePile` directory - whereas on windows, a typical location would be in the `%localappdata%/GamePile` directory. Adherence to the XDG Base Directory Specification[10] is important for this project, as it ensures that best practices are being followed in regards to data storage on Linux systems.

The file the database is stored in a file named `data.sqlite`, which uses a non-ambigious file extension to ensure that the file can be easily identified. Whenever the application is opened, a backup of the database is created, to ensure that data can be restored in the event of a crash. This backup is stored in the same directory as the main database file, with the filename `data.sqlite.bak`.

3.4.1 - Settings and Preferences

When storing settings or preferences, the application will have a few options:

1. For simple settings that aren't particularly complex:
   - Key-value pairs stored in a `.ini` settings file. INI files are the standard for storing simple key-value pairs, and are easy to read and write.
2. For more complex settings that involve defaults and nesting:
   - A JSON file will be used. JSON is a widely-used object notation format that is easy to read and write, widely-used and well supported by various programming languages.

However, a likely candidate is to make use of the QSettings[2] class provided by the Qt Framework.

The QSettings class acts as an abstraction around the platform-specific settings storage system (such as the Windows Registry or property lists on MacOS). This class allows for easy storage and retrieval of settings, and is cross-platform, making it an ideal choice for this project. This class also provides plenty of fallback systems, increasing robustness and resilience to failure.

There is a potential for settings to not be entirely necessary for this project, as the scope is relatively simple and does not require many settings to be stored. Although, it would be in the interest of future development to plan for this eventuality.

3.4.2 - Icons and Other Assets

As a part of the UI, icons will be used to represent various actions and objects in the application. These icons will be sourced from the Breeze Icon Theme[11], which is the default icon theme for the KDE Plasma Desktop Environment. This icon theme is licensed under the LGPL, which allows for the use of the icons in this project. However, the use of breeze icons will introduce some dependencies on the KDE Frameworks, which introduces a dependency on

[2] https://doc.qt.io/qt-6/qsettings.html

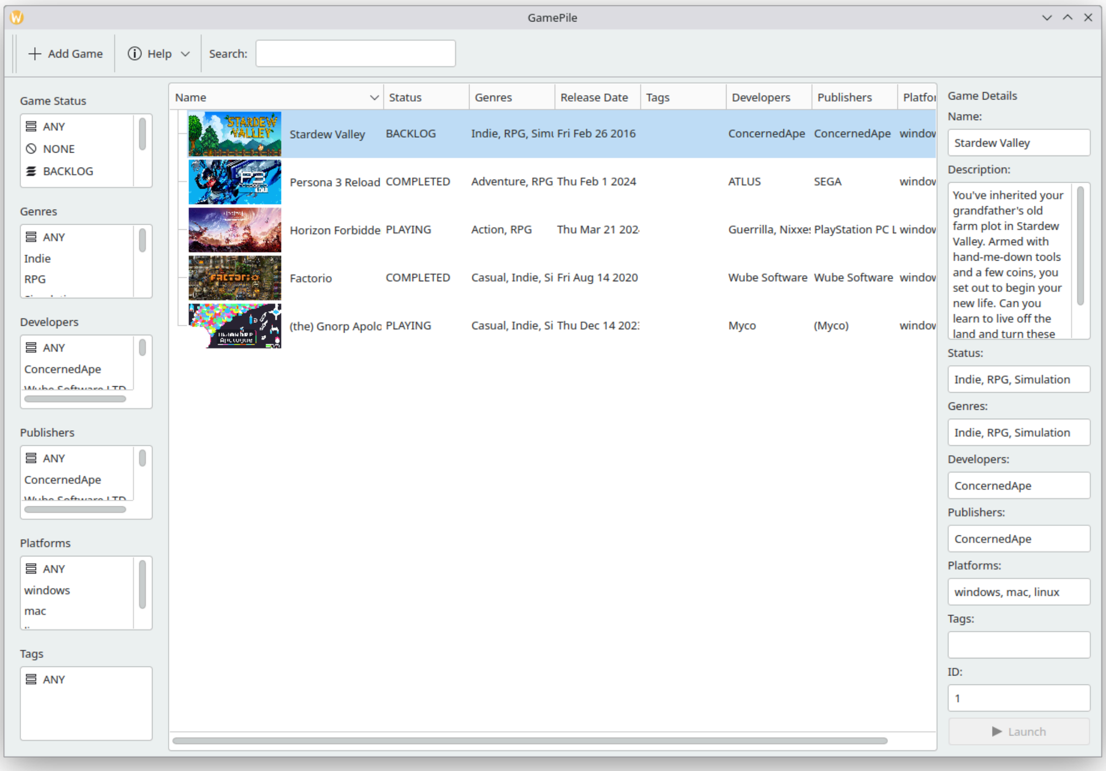
the Linux operating system. In order to mitigate this, fallback icons should be provided in the event that the Breeze Icon Theme is not available.

When storing user-generated assets, such as game icons, these are stored in a directory named `icons` in the same directory as the database file. This will allow for easy access to the icons, and will ensure that app data is mostly located inside of a single directory. Game Icons will be named accordant to the format `{gameName}-{randomIdentifier}-icon.png`, to ensure that the icons are unique and easily identifiable.

Game Icons that are no longer in use will be deleted from the directory, to ensure that unnecessary data is not being stored. This will be done by checking the database for games that no longer exist, and deleting the corresponding icon file. This check will occur whenever a game is deleted from the library, and upon startup.

In order to achieve this functionality, a class (`GameIconController`) will be created to handle the storage, retrieval and removal of icons. This class will be responsible for ensuring that the icons are stored in the correct location, and that the relevant permissions are present in order to access them when required. This class will also be responsible for deleting icons that are no longer in use.

3.5 - User Interface



Name	Status	Genres	Release Date	Tags	Developers	Publishers	Platfo
Stardew Valley	BACKLOG	Indie, RPG, Simu	Fri Feb 26 2016		ConcernedApe	ConcernedApe	window
Persona 3 Reload	COMPLETED	Adventure, RPG	Thu Feb 1 2024		ATLUS	SEGA	window
Horizon Forbidde	PLAYING	Action, RPG	Thu Mar 21 202		Guerrilla, Nixxe	PlayStation PC L	window
Factorio	COMPLETED	Casual, Indie, Si	Fri Aug 14 2020		Wube Software	Wube Software	window
(the) Gnorp Apolo	PLAYING	Casual, Indie, Si	Thu Dec 14 202		Myco	(Myco)	window

Game Details
Name: Stardew Valley
Description: You've inherited your grandfather's old farm plot in Stardew Valley. Armed with hand-me-down tools and a few coins, you set out to begin your new life. Can you learn to live off the land and turn these
Status: Indie, RPG, Simulation
Genres: Indie, RPG, Simulation
Developers: ConcernedApe
Publishers: ConcernedApe
Platforms: windows, mac, linux
Tags:
ID: 1

Figure 7: A screenshot of one of the iterations of the main window of the application.

The User Interface was one of the key focuses of the design process. It had to be designed in a way where it can be changed in an atomic fashion, without affecting other components. This was achieved by splitting code into two main packages, `data` and `ui`. The `data` package contains the data structures and logic of the application, whilst the `ui` package contains the user interface components. Nothing inside of the `data` package includes any references to the `ui` package, ensuring a one-way dependency.

3.5.1 - Main Features

The UI primarily features a main window acting as the central hub of the application. This window contains a toolbar at the top, which contains the primary actions of the application - an "Add Game" button, a "Help" button (to show information about the program) and a search bar. Buttons for settings or other key actions would also be included in this toolbar.

The main body of the window features three panes, which can be resized by the user. The left pane contains a list of filter widgets, in order to filter through the games in the library. The middle pane contains a list of games in the library, with several columns containing information about the games. This list can also be sorted by clicking on the column headers. Further development of the application could include the ability to show the list of games in a grid view, rather than a list view.

The right pane shows detailed information on the selected game, such as the description, release date and completion status. This pane also contains a button to launch the game, which will be disabled if the game does not have an executable path set.

3.5.2 - Game Edit Dialog

Another focus with designing the UI is the Edit Game dialog, which is used to add or edit games in the library. This dialog contains fields for all the information about a game. This dialog also features a button to fetch game data from the API, which will populate the fields in the dialog with the data. Users are also able to open a file dialog in order to select an icon or point at an executable for the game.

The dialog will have the ability to be initialized with a game object, which will populate the fields with the data from the game. If no game object is provided, a new game object will be created, and the fields will be empty. This will allow for easy creation of new games, and editing of existing games.

When the OK button is pressed, the data in the dialog will be validated, and if the data is valid, the dialog will close and the data will be saved to the library. If the data is invalid, an error message will be displayed to the user, and the dialog will remain open.

When the Cancel or Close Window button is pressed, the dialog will close without saving any data.

3.5.3 - Filters

The filters in the application are generated automatically based on the available attributes in the database. This allows for easy filtering of games based on these attributes. The filters are displayed in a list in the left pane of the main window, and can be toggled on and off by the user. Filters can be combined to create complex filters, allowing for fine-grained control over the games displayed in the library.

3.5.4 - Game Details

The game details pane in the main window shows detailed information about the selected game. This information includes the description, release date, completion status and any other attributes the game has. This pane also contains a button to launch the game, which will be disabled if the game does not have an executable path set. This pane is designed to be easily readable and visually appealing, following modern design principles.

This pane has a method which takes a game object as an argument, and populates the fields in the pane with the data from the game. If no game object is provided, the fields will be empty. This allows for easy updating of the pane when a new game is selected.

3.5.5 - Model / View Programming

The central widget of the main window implements a View, which is a part of the Qt's Model / View programming paradigm[12], which is a design pattern used to manage relationships between data and the way that data is presented.

With this application, the Model communicates with a source of data (the database, in this case) in order to provide an interface for other parts of the application.

The view obtains data from the model in order to display it to the user, and can also send user input back to the model. Another class called a Delegate is used to render individual items in the view (akin to the cells of a table). This allows for custom rendering of items in the view, such as displaying a progress bar for the completion status of a game or showing an icon.

This application implements a custom model, view and delegate in order to facilitate efficient and appropriate presentation. However, most of the interfacing logic is handled by a custom GameLibrary class, which acts as a single source of truth. This class is responsible for managing all the games in the library, and is used by the model, view and delegate to access and manipulate data. This is contrary to the traditional paradigm, where the model would be the single source of truth.

3.5.6 - Game Delegate

The Game Delegate is the class responsible for rendering individual items in the view. This class is used to render the cells in the list of games in the library, and is responsible for displaying the data in a visually appealing way. This class is also responsible for handling user input, such as double-clicking on a game to open the edit dialog.

Game Delegates have two context menu actions - “Edit Game” and “Delete Game”. These actions are displayed when the user right-clicks on a game in the list, and allow for easy editing and deletion of games. The “Edit Game” action will open the Edit Game dialog with the selected game, whilst the “Delete Game” action will delete the selected game from the library.

Further, the delegate is also responsible for rendering the icon of the game, if one is available. This is done by checking if the game object has an `m_iconName` variable set, and loading the icon from the `icons` folder if so.

3.6 - API Integration

The Python API Helper is used to fetch game data from the internet. This was done using the `requests` library, which is a popular library for making HTTP requests in Python. The API Helper is a Command Line Interface (CLI) application, which is used to fetch game data based on a given name.

Based on this name, the Helper makes a request to the API, returning a JSON object containing information about the game with the most similar name to the given name. As the JSON formats returned by these APIs may vary, the job of the API Helper is to translate this data into a standard JSON format that contains only the information that the main application is interested in.

The helper is designed to be easily extendable, with the ability to add new APIs with minimal changes to the code. The API can then be selected by passing an argument to the Helper, such as `--api steam`. The main program would be in charge of selecting the API to use, and passing this information to the Helper.

The API Helper also takes an `--index` argument, which allows for the selection of a specific game from the list of games returned by the API, alongside a `--number` argument, which allows for the selection of the number of games to return. This is useful for the main application, as it allows for the presentation of multiple games to the user, rather than just the top result.

3.6.1 - API in the Main Application

In the main application, a push button in the edit game dialog is used to fetch game data from the API. This button will call a function from a helper class, which will start a background thread to fetch the data. When the data is returned in the form of a JSON object, it is parsed and the fields in the edit dialog are populated with the data.

Concurrency is an important consideration for this feature, as otherwise the UI would freeze whilst waiting for the data to be fetched. In the event of a slow internet connection or API latency, this could even cause the application to time out and potentially crash. The Qt Concurrency module will be used to handle this, allowing for the application to remain responsive whilst the data is being fetched.

3.7 - Algorithms

The main application does not make use of many complex algorithms in its design.

3.7.1 - Fuzzy Searching

In the API Helper, a fuzzy search algorithm is used to find the game with the most similar name to the given name. As no known API provides a popularity ranking for games, the application can only sort based on the name.

The algorithm used is the “Token Sort Ratio” algorithm, which is a part of the RapidFuzz[13] library. This uses the Jaro-Winkler algorithm, which is a string metric which measures the edit distance between two strings, similar to the Levenshtein distance.

This variant of Jaro-Winkler is especially relevant for Games, as game names can often be misspelled or abbreviated, and the Token Sort Ratio is especially appropriate as it breaks the strings down into tokens and sorts them before comparing them. This is useful for finding games where people might search for a game using the subtitle, such as “Breath of the Wild” for the game “The Legend of Zelda: Breath of the Wild”.

3.8 - Class Diagram

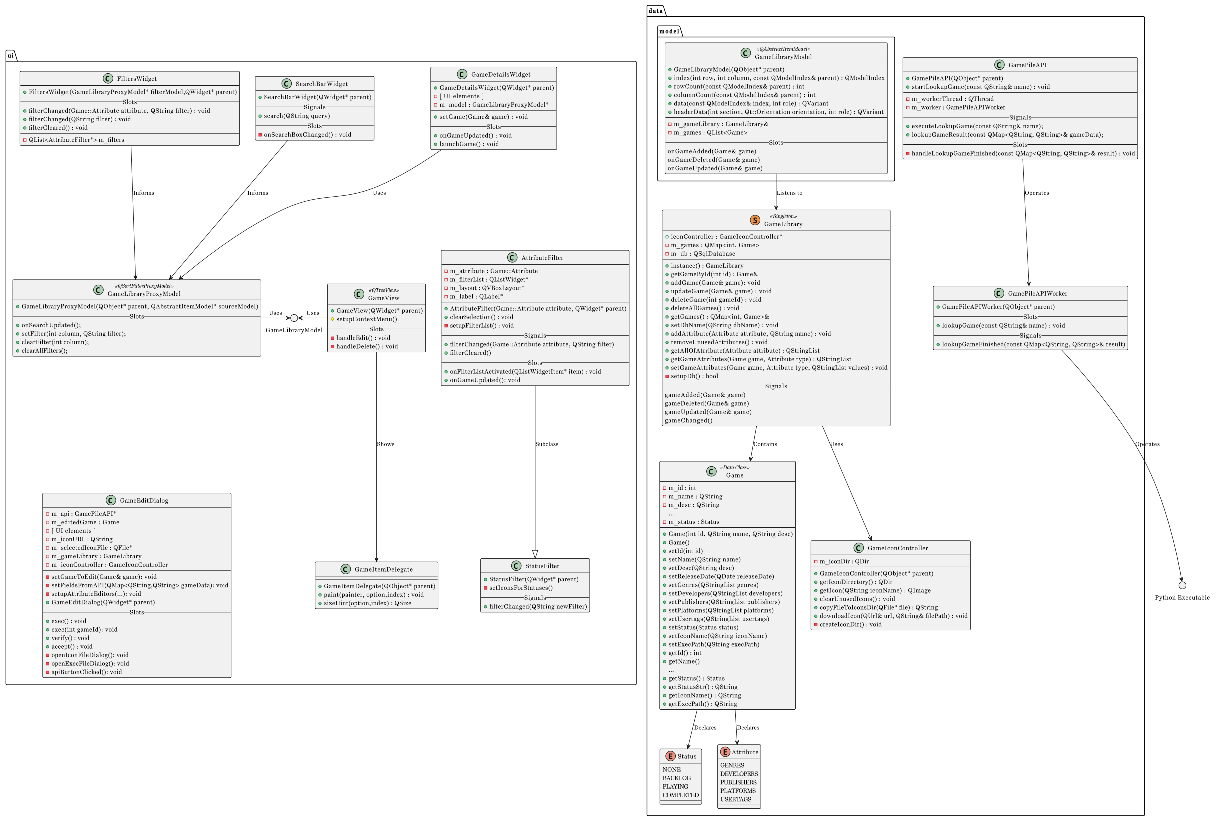


Figure 8: A class diagram showing the relationships between the classes in the application.

A landscape version of this diagram can be found in the appendices.

The python aspect of the application is not included in this diagram, as the object-oriented paradigms of Python were not used in any meaningful way.

4 - Implementation

This section will discuss the implementation of the project, including specifics on how development was carried out, the tools used, and the challenges faced.

4.1 - Development Environment

The first step of the project was to set up a development environment. For C++ Development, this typically includes: a text editor, a compiler, a build system and a version control system. For the sakes of project management, a Kanban board was also set up.

Python development also necessitates a virtual environment, some build scripts, and a seperate repository.

4.1.1 - Version Control

The project was versioned using Git. Git is a widely-used version control system that allows for tracking of changes and features such as branching and merging. Whilst most of Git's feature set is not applicable to a solo project, it is still a useful tool for tracking changes and ensuring that the project is backed up. Alternatives such as Mercurial or SVN were considered, but Git was chosen due to its ubiquity in the wider industry.

Two platforms were considered for hosting the repository: GitHub and Aberystwyth Universities GitLab[3] instance. GitHub was chosen due to the new GitHub Projects feature, which provided the Kanban board used to manage the project.

GitHub also provides a feature called "Actions", which allows for Continuous Integration (CI) testing to be set up. This was used to run unit tests on the project, ensuring that the code was functioning as expected. More detail on testing can be found in Section 5.2.

When developing the application, commits were made frequently, with each commit containing a small, atomic change. This allowed for easy tracking of changes and easy debugging in the event of an issue. For larger features that required significant changes to the codebase, a feature branch (e.g. `work/api`) was created, and changes were merged back into the `master` branch once the feature was complete. This ensures that, even in the middle of large changes, there is always a working version of the software available.

For events such as the Mid-Project Demonstration, a tag was created on the `master` branch, which is an immutable pointer to a specific commit. This creates a snapshot of the project at a specific point in time, which can be useful for observing the pace of development and the state of the project.

4.1.2 - CMake and Build System

In the C++ Community, CMake is the de-facto tool for build systems. CMake is a cross-platform build system that generates build files for a project. However, it is not a build system itself, but rather a build system generator. CMake generates build files for a specific build system, such as Make, Ninja or Visual Studio. This allows for easy cross-platform development, as the same CMakeLists.txt file can be used to generate build files for different platforms.

[3]The GitLab instance hosted by AU was not chosen as, at the inception of this project, the GitLab instance required use of a VPN. This has since changed, but development was already underway on GitHub by this point.

Setup of CMake was fairly complex, as it had to include building unit tests, building the main application and building the API Helper, which is not a C++ application. The main codebase and tests were managed by using the `add_subdirectory` command, which allows for the inclusion of other CMake projects in the main project. This meant that tests could be excluded from the main build, and the API Helper could be built as a seperate executable.

To aid in the building of the project, a `build.sh` script was produced, which accepts various arguments to run the application after building, run the tests, or install the application to the system. This script was used to automate the build process, and to ensure that the project could be built and run easily.

On the Linux Operating System, a `.desktop` file is used to create a shortcut to the application in the system menu. This file contains information about the application, such as the name, icon and executable path. This file is placed in the `~/.local/share/applications` directory, which is the standard location for `.desktop` files on Linux systems. This file is also used to set the icon of the application in the system menu.

4.1.3 - Applications and Tools

The main applications used in the development of this project were the Kate text editor and the CLion IDE. Kate was used for the majority of development, as it is a lightweight text editor that has good syntax highlighting for C++. CLion was used for debugging the C++ code, as it has a powerful debugger that integrates well with CMake. More details on the tools used can be found in the appendices.

4.2 - Early Development

With the development environment set up, development began on the project. The first step was to create the main window of the application, acting as the central hub. This window was created using the Qt Widgets module, which provides a wide range of tools for creating user interfaces. The initial version of the main window contained a button to add a game, and a list of games in the library.

After some exposure to how the Qt Widgets module works, the design was expanded to include details on some of the smaller UI components that would need to be created. Individual classes for the filter widgets, the game list, the game details and the search bar were created, each inheriting from a base class in Qt, typically `QWidget`.

It was at this stage where a database was introduced to the project and the first version of the database schema was created. However, the database was stored relative to the working directory of the application, which caused issues when the application was run from a different directory. This was resolved by using the `QStandardPaths` class to determine an appropriate location to store the database file, differing based on the operating system the application is running on.

The need for a singular source of truth was identified as extremely important during this stage, as the application was beginning to grow. To this end, the `GameLibrary` class was created, which acts as a single source of truth for the games in the library, whilst writing to and reading from the database. This class follows the singleton pattern, ensuring that only one instance of the class can be created. It is used to create, read, update and delete games in the library, and signals the model to update the view when changes are made.

Automatic code documentation via *Doxygen* was explored at this stage, but it was decided that this sort of documentation was overly technical and not particularly useful for the project at its current scope and that standard commenting practices would be sufficient.

4.3 - Later Development

During later development, the focus was on expanding the functionality of the application. The main features that were added during this stage were the ability to fetch game data from the API, the ability to filter games based on attributes, and the ability to sort games based on information about the game.

When developing the API, a Python CLI application was created to fetch game data. Using the python `requests` library, the application was able to make HTTP requests to the API and return a JSON object containing information about the game. This data was then parsed and returned to the main application in a standard format. The python application makes use of a Git submodule to include a seperate repository containing the API Helper, which allows for easy updating of the API Helper without affecting the main application. This submodule is included in the main repository, and is updated whenever changes are made to the API Helper.

This module is also available as a standalone executable, allowing for users to make use of the API helper outside of the context of the main application. This is useful for debugging and testing the API helper, as it allows for easy testing of the API without having to run the main application.

A key discovery during this stage was making use of the Virtual Environment system in Python. This allows for the isolation of dependencies for a project, ensuring that the project can be run on any system without conflicts. This means that the API Helper can be built on any system that has Python installed, without having to worry about conflicting dependencies.

4.4 - Issues Encountered

4.4.1 - Performance

During development, an issue was encountered where if a high number of games were added to the library, the application would become slow and unresponsive. This issue was specifically related to high resolution images being used as game icons. The application was loading the full-size image into memory, which was causing the application to run out of memory and become slow. In order to solve this, game icons were changed to use the `QImage` class in place of the `QIcon` class, and a scaled-down version of the image was used. This reduced the memory usage of the application and improved performance significantly.

4.4.2 - Filters

In the current version of the program, there is a bug with how the filtering system is implemented. Filters do not work as expected, and the application only applies the latest selected filter. This is due to the way that the filters are applied to the model, and the way that the model is updated when a filter is selected. This is a critical issue, as it prevents the user from using multiple filters at the same time, which is a key feature of the application.

Solving this issue would require a considerable rework of the `QSortFilterProxyModel`, which is used to display the games in the library. This would involve creating a custom proxy model that can handle multiple filters at the same time, and updating the model when a filter is selected.

4.4.3 - Availability of APIs

When looking for candidate APIs to use for the project, it was found that there were very few free APIs available that provided the data required for the project. The IGDB API[7] requires the user to login using a twitch account in order to obtain an API key, and whilst setting up an OAuth flow within the application was considered, it was decided that this was outside the scope of the project. The RAWG API[14] was also considered, but it has significant limits on its free tier. As such, the decision was made to use the Steam API[15], which is free to use and does not require an API key.

The only limitation of the Steam API is that it only provides Games that are available on the Steam platform, which may limit the number of games that can be added to the library. However, as the Steam platform is one of the largest digital distribution platforms for games, it is likely that most games will be available on the platform.

4.5 - Review against Requirements

The project was reviewed against the functional and non-functional requirements set out at the beginning of the project. A detailed breakdown of this review follows.

4.5.1 - Review of Functional Requirements

FR01 – Game Management	The application satisfies this requirement, allowing for the addition, deletion and editing of games in the library. Games have a variety of attributes.
FR02 – Backlog Management	Users are able to mark a game as part of their backlog, and the application can be filtered to only show games in the backlog.
FR03 – Progress Tracking	Games are able to marked with several completion statuses. These include "None", "Playing", "Completed" and "Abandoned".
FR04 – Sorting and Filtering	Games are able to be sorted based on any attribute, and filtered based on any attribute. However, there is a bug with the filtering system that prevents multiple filters from being applied at the same time.
FR05 – Manual Game Entry	Games are able to be added manually to the library, with the user entering the information about the game.
FR06 – API Integration	Information about games is able to be imported from an API, with the user being able to choose from five games with similar names to import.
FR07 – Desktop Application	The application is a desktop application, with a main window acting as the central hub. Games are able to be launched from the application, given an executable path is set.
OFR01, OFR02 –	Due to time constraints, neither of the Optional Features were implemented.

4.5.2 - Review of Non-Functional Requirements

Whilst Non-Functional Requirements cannot be objectively measured against in the same way as Functional Requirements, the application follows the principles set out in the requirements. GamePile performs well, doesn't crash, and is designed in a way to maximise maintainability and extendability.

Work to port the program to other operating systems and to translate the program into other languages will not require significant changes to the codebase, and would be relatively simple to achieve. The Qt translation tools `lupdate` can be used to extract strings from the application, and `lrelease` can be used to compile the translations into translation source files. This file can then be loaded by the application to provide translations.

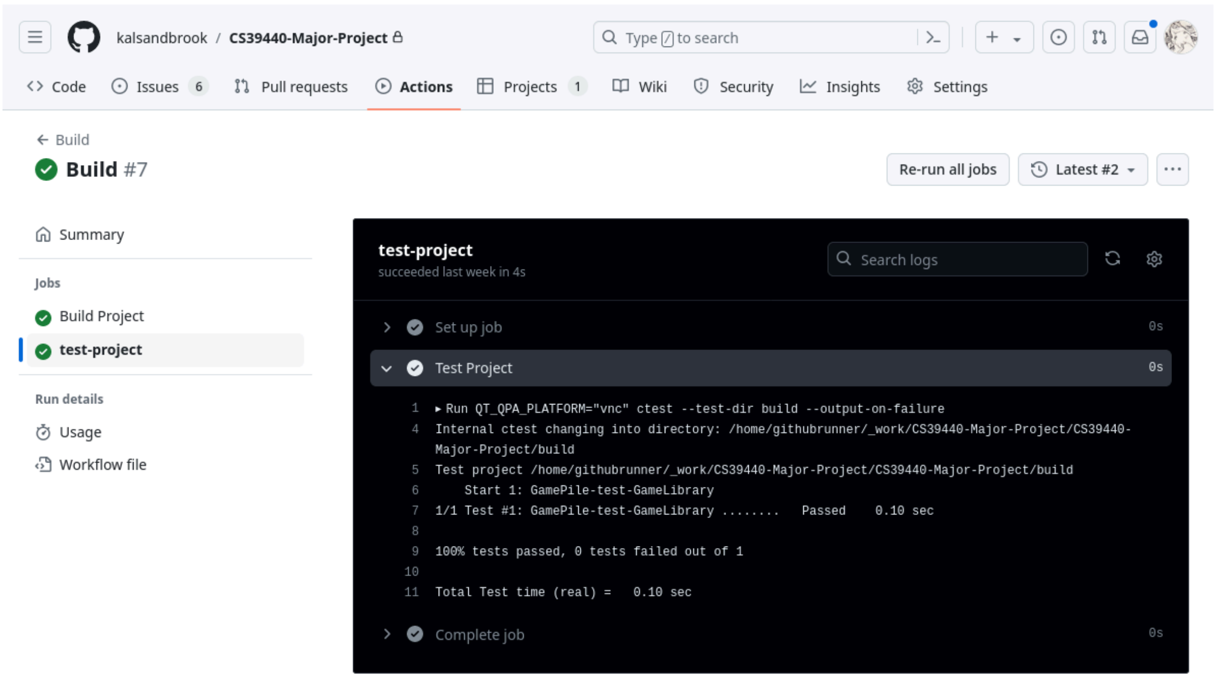
5 - Testing

5.1 - Approach

The testing of the application was done in two main ways: Unit Testing and Manual Testing. Unit Testing was done using the CTest and Qt Test framework, which is a part of CMake and the Qt Framework respectively. CTest is used to run tests alongside the build process, and to ensure that the tests are run whenever the application is built. Qt Test is used to write the tests themselves, and to provide the testing framework for the application.

Manual testing was done by running the application and using it as a user would. This involved adding games to the library, editing games, deleting games, filtering games and sorting games. This was done to ensure that the application was functioning as expected, and to identify any issues that may have been missed during development. A set of test cases was created to ensure that all aspects of the application were tested, and that the application was functioning as expected.

5.2 - Continuous Integration and Unit Testing



```
▸ Run QT_QPA_PLATFORM="vnc" ctest --test-dir build --output-on-failure
Internal ctest changing into directory: /home/githubrunner/_work/CS39440-Major-Project/CS39440-
Major-Project/build
Test project /home/githubrunner/_work/CS39440-Major-Project/CS39440-Major-Project/build
    Start 1: GamePile-test-GameLibrary
1/1 Test #1: GamePile-test-GameLibrary ........   Passed    0.10 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.10 sec
```

Figure 9: A screenshot showing the results of CI testing in GitHub Actions.

With the project using GitHub as a version control system, it was decided to make use of the GitHub Actions feature to provide Continuous Integration (CI) testing. This feature allows for the running of tests whenever the application is built, ensuring that the code is functioning as expected. This is done by creating a `.github/workflows` directory in the repository, and creating a YAML file that contains the configuration for the CI testing.

The project makes use of Continuous Integration to ensure that the application builds when changes are made to the master branch, and to run and report on the results of the unit tests. These unit tests ensure that the `GameLibrary` class is functioning as expected, and that the database is being read and written to correctly. The tests are run using the `ctest` command, which is a part of CMake, and the results are output to the console, which are logged by GitHub

Actions. Tests also exist to ensure that the API Helper is functioning as expected, and that the API is returning the correct data.

When tests fail, the build is marked as failed, and an email is sent regarding the failure. This ensures that any issues with the code are caught early, and that the code is always in a working state. This is especially important when working on a project alone, as there is no one else to catch issues with the code.

5.3 - Manual Testing

Manual testing was undertaken to ensure that the application was functioning as expected. In an ideal world, automated testing would be used more extensively to test the front-end of the application. Qt Test has the necessary capabilities to simulate testing the front-end of applications, but this was not used in this project due to the sheer complexity of writing such tests. If development was to be taken further and the application was to be expanded, automated testing for these systems would be one of the first things to be added.

If front-end testing was to be done, it would involve simulating user input and checking that the application responds as expected. This would involve simulating mouse clicks, key presses and other user interactions, and checking that the application responds correctly. Test Cases would be divided based on use-case and individual components of the UI, and would be integrated with the Continuous Integration system to ensure that the tests are run whenever the application is changed.

A testing table describing the manual tests that were carried out can be found as a part of the appendices.

6 - Evaluation

6.1 - Achievement of Aims and Objectives

I think that this project makes a reasonably good attempt at achieving the aims and objectives set out at the beginning of the project. The application is able to manage a library of games, with the ability to add, delete and edit games, and to filter and sort games based on various attributes. The application is also able to fetch game data from an API, and to mark games as part of a backlog. However, there are some issues with the application, such as the filtering system not working as expected, and the lack of optional features.

Further, I think that there are some User Experience enhancements that could be made to the application. For example, it could be made easier to change the status of a game. Currently, the user has to right-click on a game, select "Edit Game", and then change the status. This could be made easier by adding a button to the game details pane that allows the user to change the status directly. Further, I think the `QTreeView` in the main window could be enhanced by adding the option to show games in a grid view, rather than a list view. This would allow for more games to be displayed at once, whilst limiting the amount of scrolling required and potentially redundant information displayed.

It would have been good to implement the optional features, such as sharing libraries and a basic recommendation system - particularly the latter. This would have added a lot of value to the project, adding some algorithmic complexity.

6.2 - Reflection on Development Process

Kanban was not necessarily the best choice for managing the project. It was a good way to keep track of tasks, but it was not particularly useful for tracking progress or ensuring that tasks were completed on time. An iterative approach may have been better, with regular reviews of the project and the progress made. This would have allowed for changes to be made to the project as it progressed, and for issues to be caught early. As C++ was not a language I had extensive experience with, an iterative approach would have been beneficial, as it would encourage revision of the codebase and the design of the application - removing the increased technical debt accrued when working with an unfamiliar language.

It should also be considered that iterative development necessitates a significant time overhead for planning and review compared to a more linear methodology. This would have been a significant time investment, but would have likely resulted in a more polished final product.

6.3 - Choice of Tools and Development Environment

Qt Widgets was an appropriate choice for the development of the application. It provides a performant solution to creating desktop applications, and had a wide range of tools and features that allow for the creation of a robust interface.

In future, however, it may be worth considering using a more modern framework, such as Qt Quick, which is designed for more fluid interfaces. This would allow for more complex animations and transitions, and would provide a more modern look and feel to the application, closer to a modern web application.

Choosing C++ as the language for development was a good choice, but choosing a language that I did not have extensive experience with was not. C++'s famous complexity and verbosity made development more difficult than it needed to be, and the lack of experience with the language meant that development was slower than it could have been. In future, it would be better to choose a language that I am more familiar with, or practice more with a new language before starting development.

6.4 - Considerations for Future Projects

In summary, if I were to start this project again, I would likely choose a different language for development. C++ was a good choice for the project, but the lack of experience with the language made development more difficult than it needed to be. I would also consider using a more modern framework, such as Qt Quick, which would allow for more complex animations and transitions, and would provide a more modern look and feel to the application.

I would also consider using an iterative development process, with regular reviews of the project and the progress made. This would allow for changes to be made to the project as it progressed, and for issues to be caught early. This would have been particularly useful for this project, as C++ was not a language I had extensive experience with, and an iterative approach would have encouraged revision of the codebase and the design of the application.

I also believe that this project would have benefitted significantly from a larger feature-set and more time to polish the user experience. This would create room for more complex algorithms and features, adding to the technical complexity of the work, increasing the amount of work available to be done and ultimately making for a more interesting project.

All in all, I think that this project was a good learning experience, and that I have gained a lot of valuable knowledge from it. I have learned a lot about C++ development, user interface design, and I have gained a lot of experience in developing desktop applications. I think that this project has been a valuable experience, and I'm ultimately proud of what I have achieved.

Bibliography

[1] “How Long To Beat.” Accessed: Apr. 16, 2024. [Online]. Available: https://howlongtobeat.com/

One of the investigated similar applications that inspired the creation of this application. HLTB is a website that provides information on how long it takes to beat a video game, along with providing functionality to track games and create a backlog.

[2] “The Backloggery.” Accessed: Apr. 16, 2024. [Online]. Available: https://backloggery.com/

Another similar application that was investigated. The Backloggery is a website that allows users to track their video game collection and progress.

[3] “Lutris - Open Gaming Platform.” Accessed: Apr. 16, 2024. [Online]. Available: https://lutris.net/

The Lutris website, which provides an open gaming platform for Linux. It was investigated as a similar application to the one being developed.

[4] “The GTK Project - A free and open-source cross-platform widget toolkit.” Accessed: Apr. 16, 2024. [Online]. Available: https://www.gtk.org/

The GTK Project website, which provides information on the GTK toolkit used for developing graphical user interfaces. It was not used in the development of the application, but used by other applications that were investigated.

[5] “How the 4Cs of UX design benefit your software development.” Accessed: Apr. 16, 2024. [Online]. Available: https://www.qt.io/4cs-of-ux-design

An article that discusses the 4Cs of UX design and how they can benefit software development.

[6] “KDE Human Interface Guidelines.” Accessed: Apr. 16, 2024. [Online]. Available: https://develop.kde.org/hig/

The KDE Human Interface Guidelines, which provide guidance on designing user interfaces for KDE applications.

[7] “IGDB API Documentation.” Accessed: Feb. 07, 2024. [Online]. Available: https://api-docs.igdb.com/

One of the APIs considered for the application. It was not used in the final implementation, but was investigated during the planning phase.

[8] “Qt.” Accessed: Apr. 16, 2024. [Online]. Available: https://www.qt.io/

The development framework used for the application. This allows the creation of native applications across multiple platforms with ease.

[9] “PyInstaller.” Accessed: Apr. 16, 2024. [Online]. Available: https://pyinstaller.org/en/stable/

The tool used to package the application into a standalone executable.

[10] “XDG Base Directory Specification.” Accessed: Apr. 25, 2024. [Online]. Available: https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

The XDG Base Directory Specification, which provides a set of common directories for storing user-specific data.

[11] “Breeze Icons.” Accessed: Apr. 25, 2024. [Online]. Available: https://invent.kde.org/frameworks/breeze-icons

The Breeze Icons project, which provides a set of icons for use in KDE applications.

[12] “Model/View Programming.” Accessed: Apr. 25, 2024. [Online]. Available: https://doc.qt.io/qt-5/model-view-programming.html

The Qt documentation on Model/View Programming, which was used in the development of the application.

[13] “TheFuzz - Fuzzy String Matching in Python.” Accessed: Apr. 16, 2024. [Online]. Available: https://github.com/seatgeek/thefuzz

The python library that was used to perform fuzzy string matching.

[14] “RAWG Video Games Database API.” Accessed: Apr. 26, 2024. [Online]. Available: https://rawg.io/apidocs

One of the API candidates for the application. It was not used in the final implementation.

[15] “Steam Web API.” Accessed: Apr. 26, 2024. [Online]. Available: https://developer.valvesoftware.com/wiki/Steam_Web_API

The Steam Web API, which provides the ability to access Steam's data and services. It was used in the final implementation of the application to import game data from the web.

[16] “Requests - HTTP for Humans.” Accessed: Apr. 16, 2024. [Online]. Available: https://docs.python-requests.org/en/master/

The python library that was used to make HTTP requests to the API.

Appendices

E - Third-Party Code and Libraries

Qt [8]

The Qt Framework (particularly Qt Widgets) was used for the majority of development for this project. It provides a wide variety of features - notably its Qt Widgets, Qt SQL & Qt Concurrent modules.

For academic purposes, Qt Community Edition (the edition used for development) follows the *GNU Lesser General Public License* ("(L)GPL"), a copy of which can be accessed here: https://www.gnu.org/licenses/lgpl-3.0.txt

No tools such as Qt Creator or Qt Designer were used in development.

Python Libraries

For the API module of the project, Python was used due to its excellent range of libraries. The following libraries were used:

- *Requests* [16]
  - Used for making HTTP requests to the API.
- *TheFuzz* [13]
  - Used for fuzzy string matching, particularly using the jaro-winkler algorithm.
- *PyInstaller* [9]
  - Used for packaging the API module into an executable.

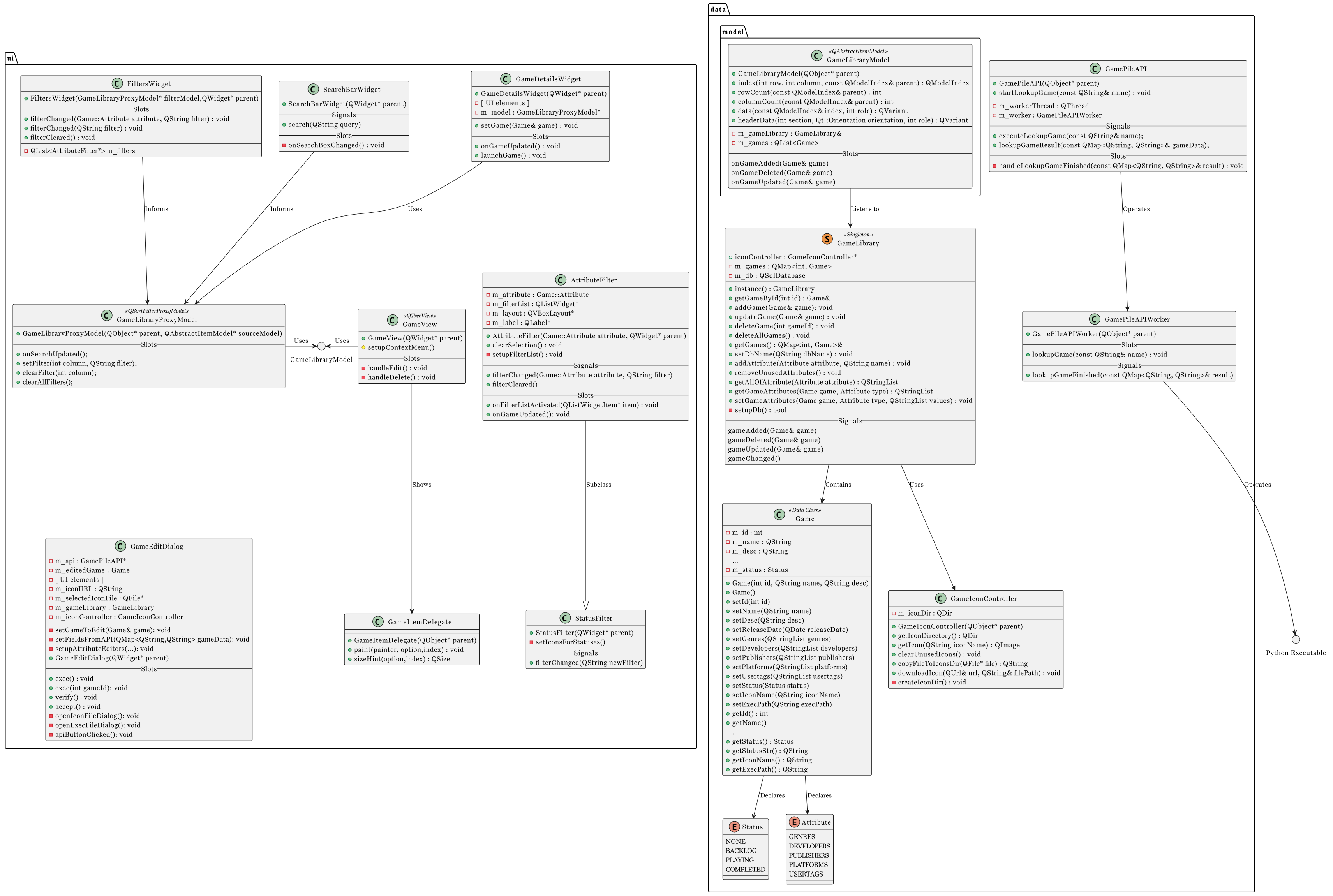
F - Statement of Tools Used

A list of key tools used in the development of this project can be found below:

Software Name	URL	Function	Notes
Kate	https://apps.kde.org/en-gb/kate/	Text Editor	Used for majority of C++ development.
CLion	https://www.jetbrains.com/clion/	IDE	Used for debugging C++ code.
Typst	https://typst.app/	Typesetting Tool	Used for documents, LaTeX alternative.
PlantUML	https://plantuml.com/	Diagramming Tool	Used for creating Class Diagram and Database Schema.
Draw.IO	https://app.diagrams.net/	Diagramming Tool	Used for creating Architecture Diagram.

Table 1: A table showing the tools used in the development of this project.

G - Class Diagram


ui
FiltersWidget
FiltersWidget(GameLibraryProxyModel* filterModel,QWidget* parent)
Slots
filterChanged(Game::Attribute attribute, QString filter) : void
filterChanged(QString filter) : void
filterCleared() : void
QList<AttributeFilter*> m_filters
SearchBarWidget
SearchBarWidget(QWidget* parent)
Signals
search(QString query)
Slots
onSearchBoxChanged() : void
GameDetailsWidget
GameDetailsWidget(QWidget* parent)
[UI elements]
m_model : GameLibraryProxyModel*
setGame(Game& game) : void
Slots
onGameUpdated() : void
launchGame() : void
Informs
Informs
Uses
«QSortFilterProxyModel»
GameLibraryProxyModel
GameLibraryProxyModel(QObject* parent, QAbstractItemModel* sourceModel)
Slots
onSearchUpdated();
setFilter(int column, QString filter);
clearFilter(int column);
clearAllFilters();
Uses
Uses
GameLibraryModel
«QTreeView»
GameView
GameView(QWidget* parent)
setupContextMenu()
Slots
handleEdit() : void
handleDelete() : void
AttributeFilter
m_attribute : Game::Attribute
m_filterList : QListWidget*
m_layout : QVBoxLayout*
m_label : QLabel*
AttributeFilter(Game::Attribute attribute, QWidget* parent)
clearSelection() : void
setupFilterList() : void
Signals
filterChanged(Game::Atrribute attribute, QString filter)
filterCleared()
Slots
onFilterListActivated(QListWidgetItem* item) : void
onGameUpdated(): void
Shows
Subclass
GameEditDialog
m_api : GamePileAPI*
m_editedGame : Game
[UI elements]
m_iconURL : QString
m_selectedIconFile : QFile*
m_gameLibrary : GameLibrary
m_iconController : GameIconController
setGameToEdit(Game& game): void
setFieldsFromAPI(QMap<QString,QString> gameData): void
setupAttributeEditors(...): void
GameEditDialog(QWidget* parent)
Slots
exec() : void
exec(int gameId): void
verify() : void
accept() : void
openIconFileDialog(): void
openExecFileDialog(): void
apiButtonClicked(): void
GameItemDelegate
GameItemDelegate(QObject* parent)
paint(painter, option,index) : void
sizeHint(option,index) : QSize
StatusFilter
StatusFilter(QWidget* parent)
setIconsForStatuses()
Signals
filterChanged(QString newFilter)
data
model
«QAbstractItemModel»
GameLibraryModel
GameLibraryModel(QObject* parent)
index(int row, int column, const QModelIndex& parent) : QModelIndex
rowCount(const QModelIndex& parent) : int
columnCount(const QModelIndex& parent) : int
data(const QModelIndex& index, int role) : QVariant
headerData(int section, Qt::Orientation orientation, int role) : QVariant
m_gameLibrary : GameLibrary&
m_games : QList<Game>
Slots
onGameAdded(Game& game)
onGameDeleted(Game& game)
onGameUpdated(Game& game)
GamePileAPI
GamePileAPI(QObject* parent)
startLookupGame(const QString& name) : void
m_workerThread : QThread
m_worker : GamePileAPIWorker
Signals
executeLookupGame(const QString& name);
lookupGameResult(const QMap<QString, QString>& gameData);
Slots
handleLookupGameFinished(const QMap<QString, QString>& result) : void
Listens to
Operates
«Singleton»
GameLibrary
iconController : GameIconController*
m_games : QMap<int, Game>
m_db : QSqlDatabase
instance() : GameLibrary
getGameById(int id) : Game&
addGame(Game& game): void
updateGame(Game& game) : void
deleteGame(int gameId) : void
deleteAllGames() : void
getGames() : QMap<int, Game>&
setDbName(QString dbName) : void
addAttribute(Attribute attribute, QString name) : void
removeUnusedAttributes() : void
getAllOfAtrribute(Attribute attribute) : QStringList
getGameAttributes(Game game, Attribute type) : QStringList
setGameAttributes(Game game, Attribute type, QStringList values) : void
setupDb() : bool
Signals
gameAdded(Game& game)
gameDeleted(Game& game)
gameUpdated(Game& game)
gameChanged()
GamePileAPIWorker
GamePileAPIWorker(QObject* parent)
Slots
lookupGame(const QString& name) : void
Signals
lookupGameFinished(const QMap<QString, QString>& result)
Contains
Uses
Operates
«Data Class»
Game
m_id : int
m_name : QString
m_desc : QString
...
m_status : Status
Game(int id, QString name, QString desc)
Game()
setId(int id)
setName(QString name)
setDesc(QString desc)
setReleaseDate(QDate releaseDate)
setGenres(QStringList genres)
setDevelopers(QStringList developers)
setPublishers(QStringList publishers)
setPlatforms(QStringList platforms)
setUsertags(QStringList usertags)
setStatus(Status status)
setIconName(QString iconName)
setExecPath(QString execPath)
getId() : int
getName()
...
getStatus() : Status
getStatusStr() : QString
getIconName() : QString
getExecPath() : QString
GameIconController
m_iconDir : QDir
GameIconController(QObject* parent)
getIconDirectory() : QDir
getIcon(QString iconName) : QImage
clearUnusedIcons() : void
copyFileToIconsDir(QFile* file) : QString
downloadIcon(QUrl& url, QString& filePath) : void
createIconDir() : void
Python Executable
Declares
Declares
Status
NONE
BACKLOG
PLAYING
COMPLETED
Attribute
GENRES
DEVELOPERS
PUBLISHERS
PLATFORMS
USERTAGS

H - Manual Testing Table

ID	Ref.	Desc.	Steps	Inputs	Expected	PASS / FAIL
TC1	FR01	Adding a game.	1. Open add game dialog. 2. Enter values for all fields. 3. Press OK.	Information about the game Minecraft.	The game is added, and shows up on the main list. The filters along the side are also populated with the relevant entries.	PASS
TC2	FR01	Deleting a game.	1. Right click on a games entry. 2. Press delete.	N/A	The game is removed from the library, and the filters are updated.	PASS
TC3	FR01	Editing a game.	1. Right click on a games entry. 2. Press edit. 3. Change the name of the game. 4. Press OK.	N/A	The game is updated with the new name, and the filters are updated.	PASS
TC4	FR02	Adding a game to the backlog.	1. Right click on a games entry. 2. Open the context menu. 3. Press edit. 4. Set the games status to "Backlog". 5. Press OK.	N/A	The game is added to the backlog.	PASS
TC5	FR02	Removing a game from the backlog.	1. Right click on a games entry. 2. Open the context menu.	N/A	The game is removed from the backlog.	PASS

ID	Ref.	Desc.	Steps	Inputs	Expected	PASS / FAIL
			3. Press edit. 4. Set the games status to "None". 5. Press OK.			
TC6	FR03	Setting a game to "Playing".	1. Right click on a games entry. 2. Open the context menu. 3. Press edit. 4. Set the games status to "Playing". 5. Press OK.	N/A	The game is set to "Playing".	PASS
TC7	FR03	Setting a game to "Completed".	1. Right click on a games entry. 2. Open the context menu. 3. Press edit. 4. Set the games status to "Completed". 5. Press OK.	N/A	The game is set to "Completed".	PASS
TC8	FR03	Setting a game to "Abandoned".	1. Right click on a games entry. 2. Open the context menu. 3. Press edit. 4. Set the games status to "Abandoned". 5. Press OK.	N/A	The game is set to "Abandoned".	PASS

ID	Ref.	Desc.	Steps	Inputs	Expected	PASS / FAIL
TC9	FR04	Sorting games by name.	1. Click on the "Name" column header.	N/A	The games are sorted by name.	PASS
TC10	FR04	Sorting games by release date.	1. Click on the "Release Date" column header.	N/A	The games are sorted by release date.	PASS
TC11	FR04	Filtering games by name.	1. Enter a name in the search bar.	N/A	The games are filtered by name.	PASS
TC12	FR04	Filtering games by multiple attributes.	1. Enter a name in the search bar. 2. Select a filter from the filter list.	N/A	The games are filtered by name and the selected filter.	FAIL
TC13	FR05	Adding a game with all fields filled.	1. Open the add game dialog. 2. Enter values for all fields. 3. Press OK.	Information about the game Minecraft.	The game is added, and shows up on the main list. The filters along the side are also populated with the relevant entries.	PASS
TC14	FR05	Adding a game with no fields filled.	1. Open the add game dialog. 2. Press OK.	N/A	An error message is displayed, and the game is not added.	PASS
TC15	FR05	Adding a game with only name filled.	1. Open the add game dialog. 2. Enter a name for the game. 3. Press OK.	N/A	The game is added, the only necessary details for a game are the name.	PASS
TC16	FR06	Fetching game data from the API.	1. Open the add game dialog.	Enter the game name "Team Fortress 2".	The input fields on the add game form are populated. The games	PASS

ID	REF.	DESC.	STEPS	INPUTS	EXPECTED	PASS / FAIL
			2. Enter a name for the game. 3. Press the "Fetch Data" button.		icon is also provided by the API.	
TC17	FR06	Fetching game data from the API with no results.	1. Open the add game dialog. 2. Enter a name for the game. 3. Press the "Fetch Data" button.	Enter the game name "This Game Does Not Exist".	An error message is displayed, and the input fields are not populated.	FAIL
TC18	FR06	Fetching game data from the API with multiple results.	1. Open the add game dialog. 2. Enter a name for the game. 3. Press the "Fetch Data" button.	Enter the game name "Team".	A list of games is displayed, and the user can select the correct game.	PASS
TC19	FR06	Fetching game data from the API with no internet connection.	1. Disconnect from the internet. 2. Open the add game dialog. 3. Enter a name for the game. 4. Press the "Fetch Data" button.	Enter the game name "Team Fortress 2".	An error message is displayed, and the input fields are not populated.	FAIL
TC20	FR07	Launching a game.	1. Select a game in the list. 2. Press launch.	N/A	The game is launched.	PASS

ID	Ref.	Desc.	Steps	Inputs	Expected	PASS / FAIL
TC21	FR07	Launching a game with no executable path set.	1. Select a game in the list. 2. Press launch.	N/A	The launch button is unable to be pressed.	PASS
TC22	FR07	Launching a game with an invalid executable path set.	1. Select a game in the list. 2. Press launch.	N/A	An error message is displayed.	PASS
TC23	FR07	Launching a game with no game selected.	1. Press launch.	N/A	The launch button is unable to be pressed.	PASS

Table 2: Test Table