Software Design
(COMP.SE.110)

Design Document for:
MediaFinder Application

By

Sami Peltola, Niko Toskala, Eero Laine, Saurabh Chauhan

Table of Contents

Introduction

The Mediafinder is a software application designed to supply personalized movie and book recommendations to users based on keyword searches. This document outlines the design and architecture of the desktop application, supplying a comprehensive view of its components, functionalities, and technical specifications.

Link to the figma project

Purpose

The purpose of this document is to supply a detailed description of the software design for the Mediafinder App. It serves as a reference for developers involved in the project, ensuring a mutual understanding of the system's architecture and functionality.

Scope

The Mediafinder Desktop App will include the following key features:

- A simple user registration and log-in.
- Keyword-based search for movies and books.
- Filtering search results by genre and release year.
- Detailed information about the books and movies, including ratings.
- Integration with External APIs (TMDb API, Google Books API)

Architecture Overview

The application will follow the MVC with passive architecture, as shown below:

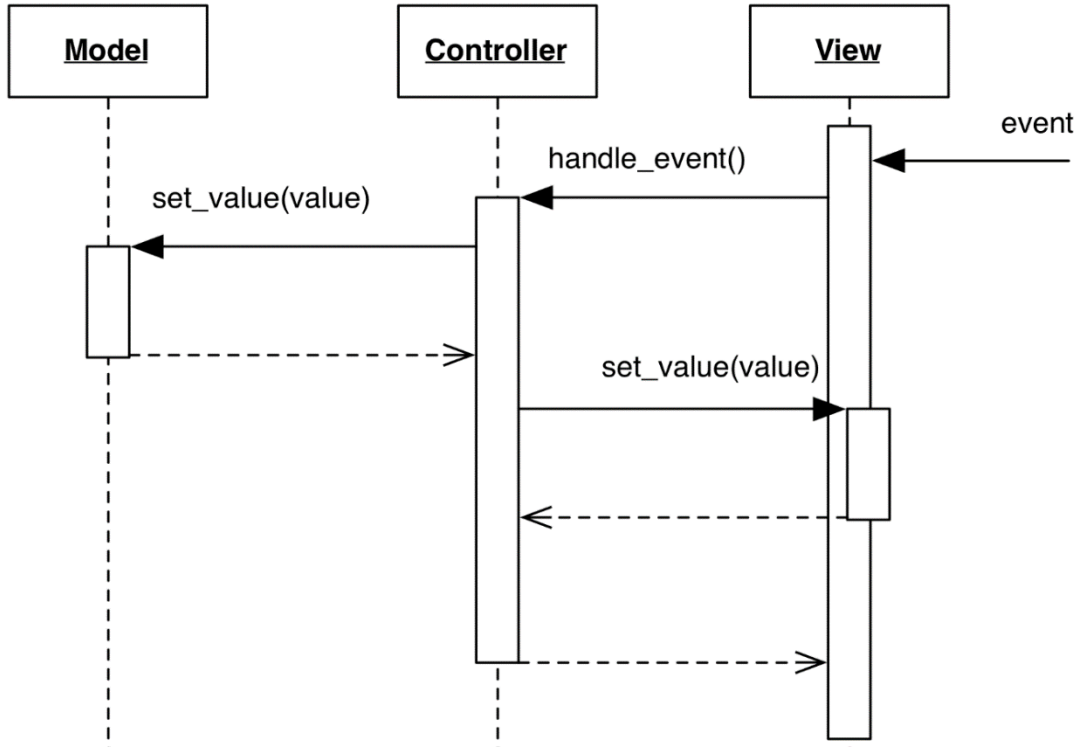

Image 1: Architectural View

- **View**: The View stands for the user interface (UI) of the application. It displays data from the Model to the user and sends user commands to the Controller.
    - **Responsibilities**:
        - Display movie and book information in a user-friendly manner.
        - Supply UI elements for user interactions, such as search bars, filters, and detailed view pages.
        - Reflect any changes in the Model to ensure the displayed data is always current.

- **Controller**: The Controller acts as an interface between Model and View. It takes the user input from the View, processes it (with possible updates to the Model), and returns the display output to the View.
    - **Responsibilities**:
        - Handle user input from the View, such as keyword searches or filter selections.
        - Interact with the Model to retrieve or update data based on user input.

        - Update the View to reflect changes in the Model or to display new data to the user.
    - **Model**: The Model stands for the data and the business logic of the application. It directly manages the data, the logic, and the rules of the application.
        - **Responsibilities**:
            - Communicate with external APIs (TMDb API, Google Books API) to fetch updated movies and book data.
            - Process data according to business rules, such as filtering search results based on genre and release year.

Why MVC Passive Model?

- **Separation of Concerns**: MVC enforces a clear separation of concerns in your application. The Model represents the data and business logic, the View handles the user interface, and the Controller acts as an intermediary that decouples the Model from the View. This separation makes it easier to manage and maintain the codebase.

- **Testability**: MVC is known for its testability. Since the model is passive and doesn't have direct knowledge of the view, it's easier to write unit tests for the presentation logic in the Controller. This separation makes it possible to test the application's behavior without a graphical user interface.

- **Reusability**: MVC promotes reusability. Because the business logic is contained within the Controller and the UI components are in the View, you can potentially reuse the same Controller with different Views, making it more adaptable to changing user interfaces or platforms.

System Components

The major components of the Movie and Book Recommendation Desktop App include:

- **Data Classes and Interfaces**

In the system, the **Book** and **Movie** classes, as shown represent data for storing information about books and movies. They both implements **Media** interface.

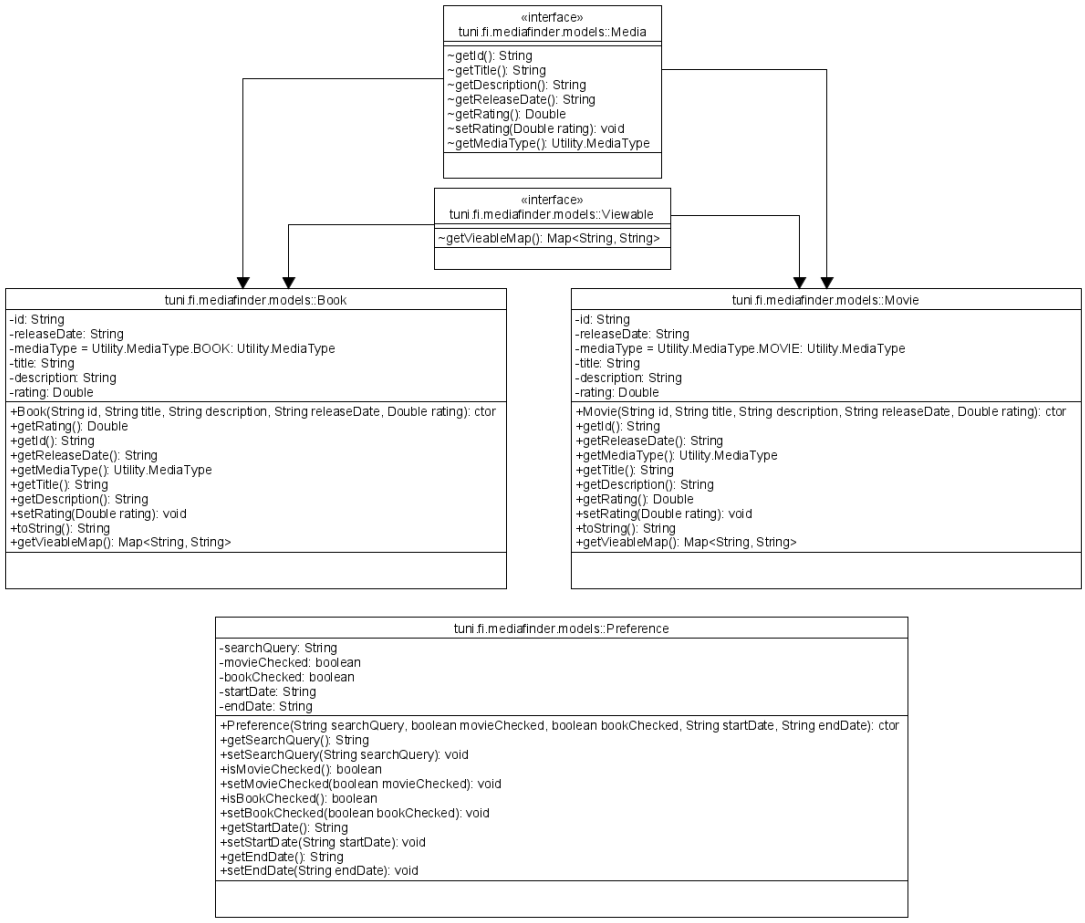

Image 2: MVC Classes

By defining a common interface **Media**, we can treat objects of different classes uniformly for example when we will provide sorted or filtered list of multiple types of media items to user, we can process them under same roof. This promotes polymorphism, where we can use objects of different types interchangeably.

Any code that relies on the **Media** interface can work with both **Book** and **Movie** objects without needing to know the specific implementation details of each class. This promotes code reusability and reduces the need for duplicated code.

Having a common interface enforces consistency in the methods that are expected to be available in implementing classes. This can make our code more predictable and easier to maintain.

If we want to add more types of media in the future, you can simply create new classes that implement the **Media** interface, and they will seamlessly integrate with the existing code that relies on the interface.

Another model class, the **Preference** class is like a blueprint for storing information about what a user wants when searching for media. It includes details like the search query (what they are looking for), whether they want movies or books, and the release time they are interested in. The class has methods to set and get these details. Additionally, it's set up to easily convert this information to and from a format called JSON, which is commonly used for data exchange in the app.

- **Controller**

The Controller is responsible for handling UI events and interacting with the model. It bridges the gap between the UI and the data.

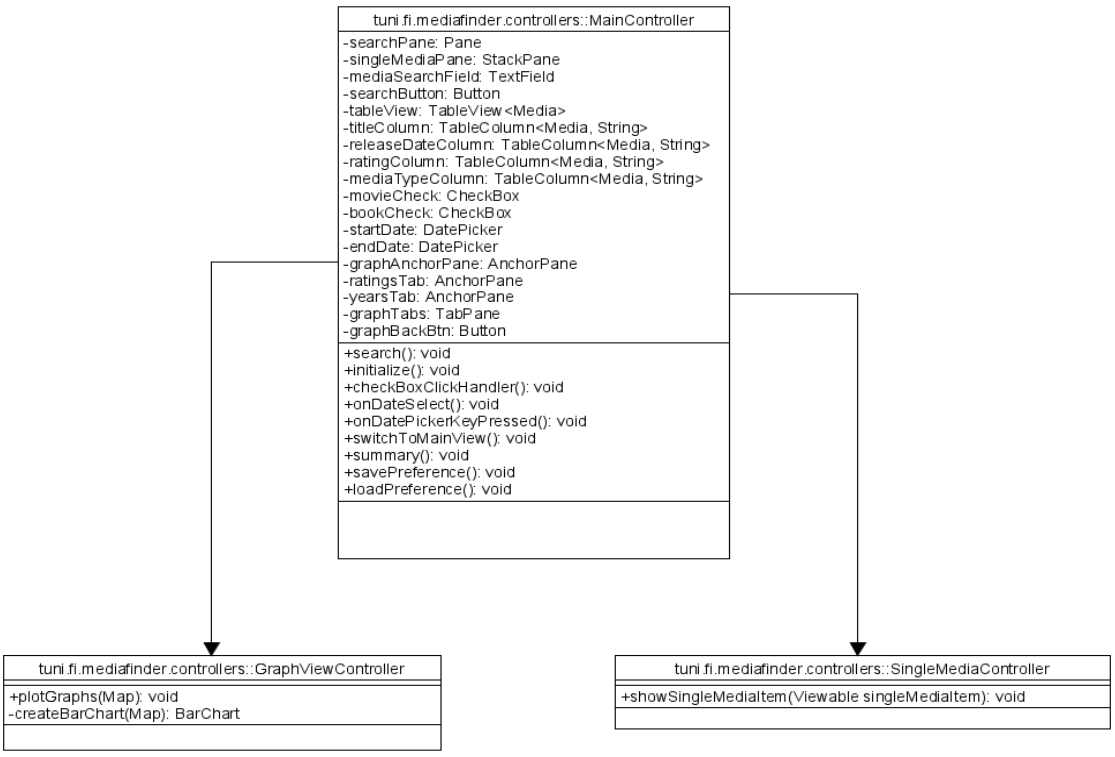

Image 3: Controller Classes

It is responsible for handling events triggered by UI components, such as buttons, text fields, and other interactive elements. This includes defining event handlers for actions like button clicks, mouse events, and keyboard input.

It also communicates with the application's data models or services. The controller retrieves data from models and updates the View with new data when changes are made through the UI.

Mainly, it can change text, enable or disable buttons, update lists or tables, and manage transitions between different views or scenes.

- **MainView**

The MainView FXML file is the view of the program and is therefore responsible for displaying the graphical elements of the program’s main page to the user. The MainView displays the data given to it by the Controller and it does not modify the data in any way.

- **API Manager and External API Integrations**:

The API manager serves as a central component for handling communication with external APIs, such as the Google Books API and MovieDB API, in our application. Its main responsibilities include making HTTP requests, processing API responses, and providing methods for other parts of your application to query these APIs.

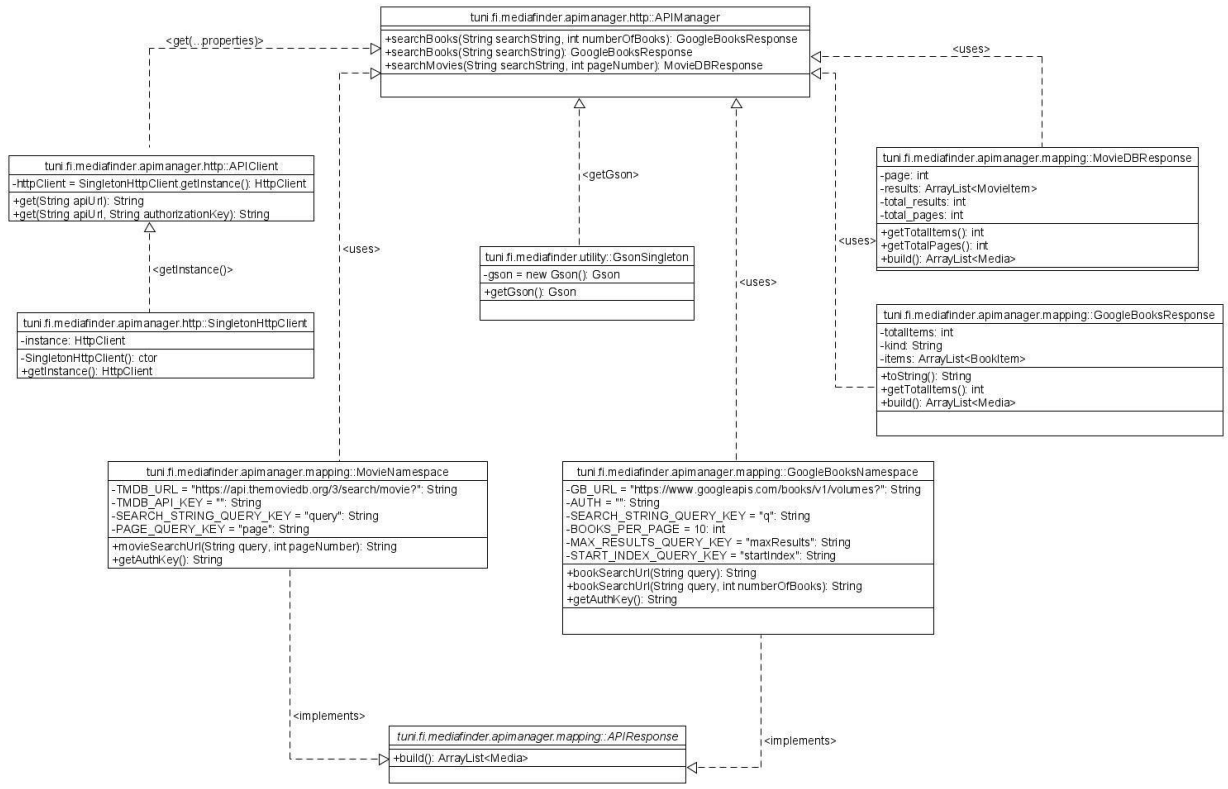

Image 4: API Model Classes

The API manager can interact with API namespace classes (**GoogleBookNameSpace** and **MovieNameSpace**) to obtain essential information, such as API endpoints and API keys. These classes help keep the API-related constants organized and centralized.

The API manager uses GSON for deserialization, which allows it to convert JSON responses from the APIs into Java objects. This simplifies working with the API data in your application, making it more manageable and type safe. It uses a GSON singleton instance to promote efficiency and consistency across your application. Moreover, Creating and initializing a GSON instance can be an expensive operation, so we do not use customized GSON with specific configurations (e.g., date formats, type adapters). The mapping is all done by primitive types. Furthermore, by using a singleton, we ensure that only one GSON instance is created and shared throughout your application's lifecycle. This reduces memory consumption and initialization overhead.

Response classes **GoogleBookResponse and MovieDBRespnse** define the structure of data received from the API. They mirror the JSON structure of the API response, making it easier to parse the data into Java objects.

API manager can convert JSON responses into these response objects, making it easier for the rest of your application to work with the data.

To perform its operation, it uses **SingletonHttpClient** and **APIClient** classes.
The **SingletonHttpClient** class manages the HTTP client instance and provides a consistent, configured client to **APIClient** class.

The **APIClient** is responsible for constructing and sending HTTP requests. It uses the singleton HTTP client to ensure that all requests are made using the same client configuration.

The API manager then utilizes the API client to make requests to external APIs. It focuses on the higher-level logic of handling responses, processing data, and returning results to other parts of your application.

The singleton HTTP client ensures that all HTTP requests made by your API manager are consistent in terms of settings, connection pools, and authentication, enhancing predictability and reducing the likelihood of subtle issues due to differing client configurations.

- **Dependencies, libraries, third-party components:**

GSON

GSON provides a simpler and efficient way to convert JSON to Java objects. This is essential when working with APIs that communicate in JSON format. When the API manager makes a request to an external API (Google Books API or MovieDB API), it receives JSON data as the response.
The API manager uses GSON to parse the JSON response into Java objects. It does this by creating response classes that mirror the structure of the JSON data, as discussed earlier.

JavaFXML and SceneBuilder

JavaFX with FXML is used because it allows for a clean separation between the UI design and application logic, making it easier to work collaboratively and maintain UI code. Since we are using MVC pattern it already solves the concern of separating Controller and UI.

It also allows us to use SceneBuilder, which is a visual layout tool, to design and create the UI faster and more easily, rather than manually create each element in the code.

- **Overall structure of the app**

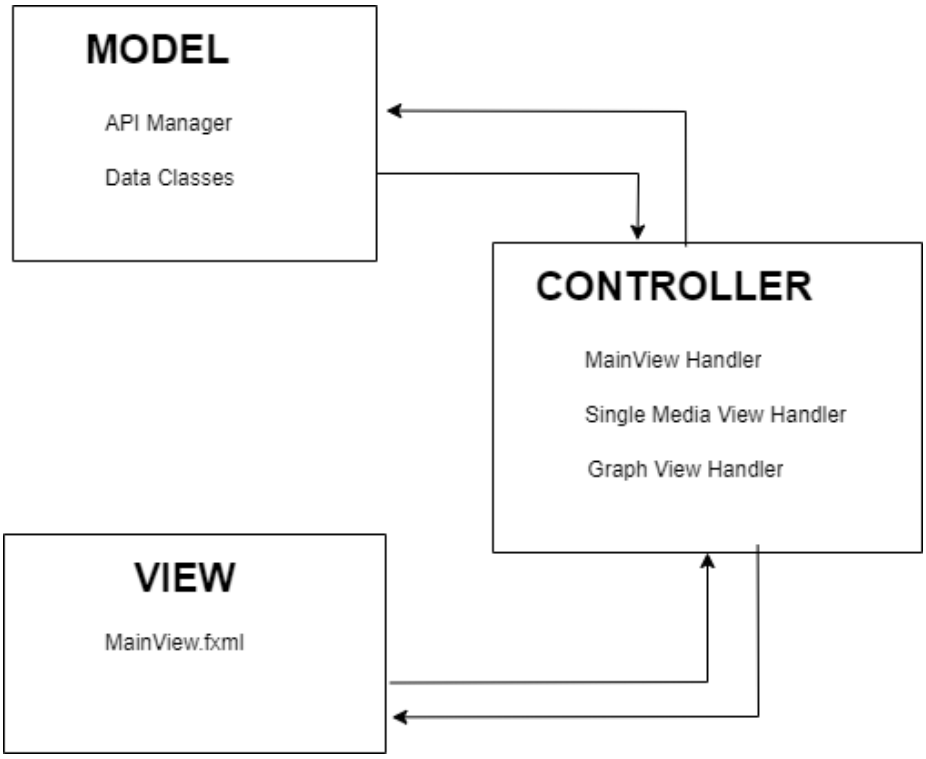

Image 5: MVC File Structure

- The user interacts with the View by clicking buttons, entering text, or performing other actions.
- The Controller listens for these user interactions and decides what actions to take.
- The Controller communicates with the Model, invoking methods and updating the Model's data.
- The Model processes the data and updates its state if needed.
- The Model notifies the Controller of changes, which may involve firing events or using data binding mechanisms.
- The Controller updates the View to reflect the changes in the Model's data.

Simple sequence diagram for making search query:

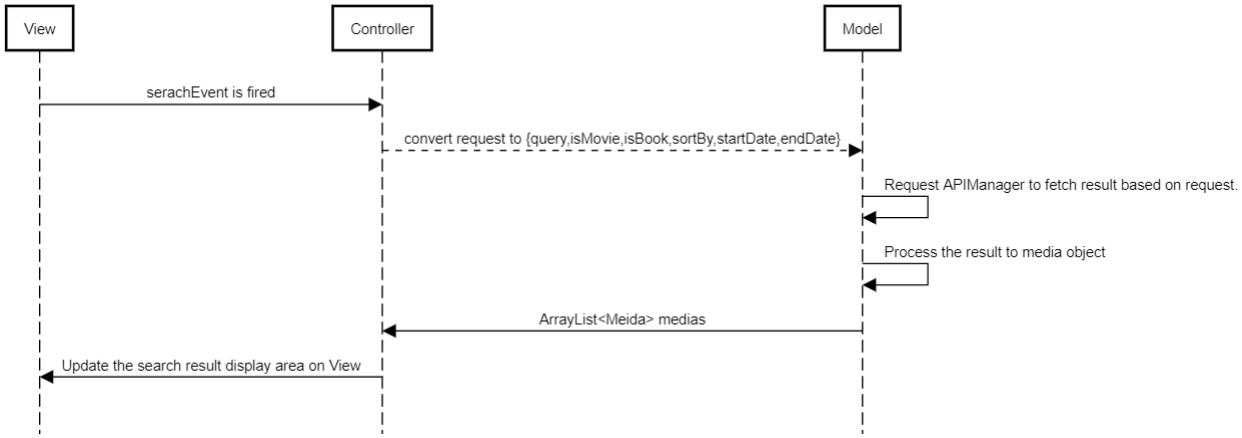

Image 6: Event Hierarchy

SOLID Principles

Single Responsibility Principle (SRP):

- **APIManager Class:** The **APIManager** class is responsible for managing API communication and does not have unrelated responsibilities. It interacts with the **APIClient** for handling HTTP requests and the other response classes **for parsing JSON responses. Each method in APIManager** has a specific responsibility related to API communication.
- **APIClient Class:** The **APIClient** class has a single responsibility of handling HTTP requests using java HttpClient.
- **Media, Book, and Movie Classes:** Each of these classes represents a media entity, and they have the responsibility of holding information related to media. They do not directly deal with API communication or parsing.

Open/Closed Principle (OCP):

- **Media, Book, and Movie Classes:** These classes and interface are open for extension. You can easily introduce new classes that implement the **Media** interface without modifying the existing classes.
- **APIManager Class:** The **APIManager** class is open for extension as well. If you want to add support for new APIs or new features related to API communication, you can do so by creating new methods or classes without modifying existing ones.

Liskov Substitution Principle (LSP):

- **Media, Book, and Movie Classes:** Instances of **Book** and **Movie** can be substituted for instances of **Media** without affecting the correctness of the program. They adhere to the Liskov Substitution Principle by inheriting from the **Media** interface.

Interface Segregation Principle (ISP):

- **Media, Book, and Movie Classes:** The **Media** interface is focused on media-related methods, and the **Viewable** interface is focused on viewing-related methods. Classes implementing **Media** and **Viewable** only need to implement the methods relevant to their responsibilities.

Dependency Inversion Principle (DIP):

- **APIManager Class:** The **APIManager** class depends on abstractions **(APICLient and Namespace classes)** rather than concrete implementations. This allows for flexibility in changing or extending the behaviour of API communication without modifying the **APIManager** class.

User Interface

The desktop application will provide a user-friendly graphical interface for users to interact with the system. Key features will encompass:

- **Keyword Search Bar**: A search bar that allows the user to search for books and movies through keywords.
- **Search Results Display**: Displays the 10 first results retrieved from the APIs. 5 books and 5 movies. More can be displayed if the user wishes so.

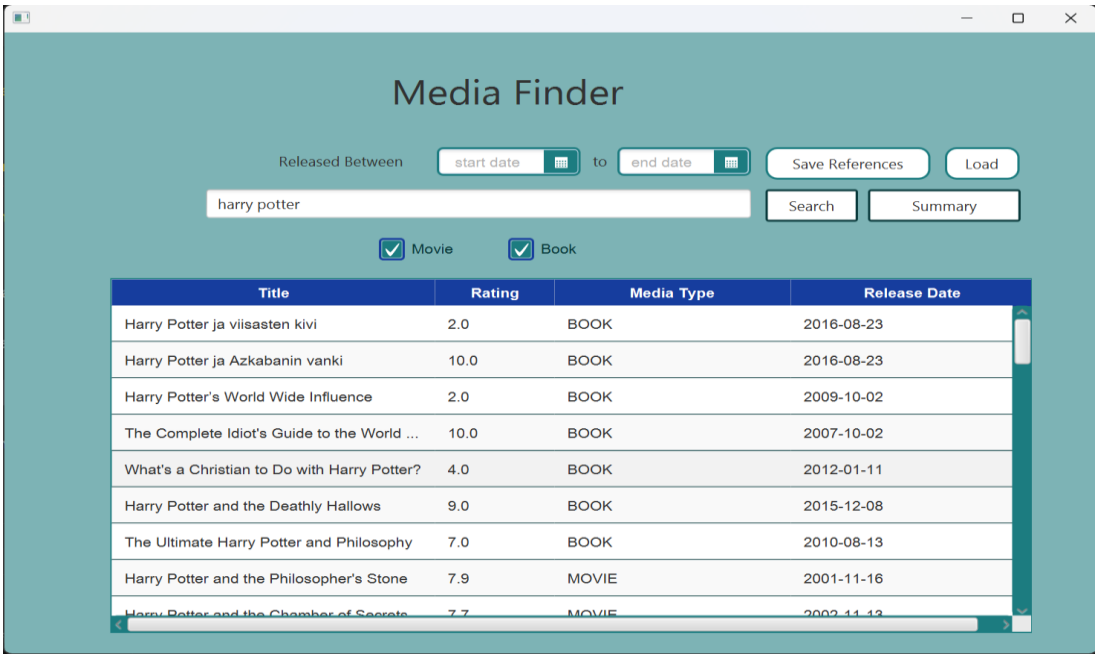

Image 7: Search results for "Harry Potter"

- **Sorting by Rating**: Users will have the option to sort search results by movie or book ratings, allowing them to quickly find the most highly rated content.
- **Sorting by Title**: User will have option to sort the media items by titles.
- **Sorting by Release Date**: User will have option to sort media items by Release Date.
- **Filtering by Release Date**: Users will be able to apply filters to search results based on the release date of movies or books, refining their results to include content released within a specified period.

Image 8: Rating-based Graph

Image 9: Year-based Graph

- **Summary Analysis with Bar Graphs**: Users will have the ability to visualize the summary of their search results using bar graphs. These graphs will supply visual insights into factors such as ratings, release dates, and other relevant data, enabling users to make more informed decisions about their selections.

Search Algorithm

The search engine will use a combination of keyword matching to return relevant movie and book results. It will consider factors such as title, description and release year.
After the result has been fetched from the API sources it will be processed according to filter and sort selection made by the user and then it will be displayed.
There are few differences between API responses which we need to consider while displaying the search result.
For e.g., Movie API returns rating out of 10 while Books API returns rating out of 5 as a result, we need to map the ratings on correct scale to sort the result in terms of ratings.

Performing a search

You can perform a volume/book search by sending an HTTP GET request to the following URI:
https://www.googleapis.com/books/v1/volumes?q=search+terms
This request has a single required parameter:

- q - Search for volumes that have this text string. There are special keywords you can specify in the search terms to search fields, such as:

Request

Here is an example of searching for Daniel Keyes' "Flowers for Algernon":

GET https://www.googleapis.com/books/v1/volumes?q=flowers+inauthor:keyes&key=APIKey

Response

If the request succeeds, the server responds with a 200 OK HTTP status code and the volume results:


```
{
"kind": "books#volumes",
"items": [
 {
  "kind": "books#volume",
  "id": "_ojXNuzgHRcC",
  "selfLink": "https://www.googleapis.com/books/v1/volumes/_ojXNuzgHRcC",
  "volumeInfo": {
   "title": "Flowers",
   "authors": [
    "Vijaya Khisty Bodach"
   ],
  ...
 },
 {
  "kind": "books#volume",
  "id": "RJxWIQOvoZUC",
  "selfLink": "https://www.googleapis.com/books/v1/volumes/RJxWIQOvoZUC",
  "volumeInfo": {
   "title": "Flowers",
   "authors": [
    "Gail Saunders-Smith"
   ],
   ...
 },
 ],
 "totalItems": 3
}
```

- TMDb API

Performing a search

You can perform movie search by sending an HTTP GET request to the following URI:
https://api.themoviedb.org/3/search/movie?query=moviename&page=number
This request has a single required parameter:

- query - Search for movies that have this text string.:

Request

Here is an example of searching for “horror” :
GET https://api.themoviedb.org/3/search/movie?query=horror&page=1

Response

If the request succeeds, the server responds with a 200 OK HTTP status code and the movie results:


```
{
 "page": 1,
 "results": [
  {
   "adult": false,
   "backdrop_path": "/q0q0nISnX4Z8V2wIfhIYfzaVX3b.jpg",
   "genre_ids": [
    18,
    9648,
    27,
    53
   ],
   "id": 301325,
   "original_language": "en",
   "original_title": "#Horror",
 },
  ...
 ],
 "total_pages": 58,
 "total_results": 1142

}
```

id	name
28	Action
12	Adventure
16	Animation
35	Comedy
80	Crime
99	Documentary
18	Drama
10751	Family
14	Fantasy
36	History
27	Horror
10402	Music
9648	Mystery
10749	Romance
878	Science Fiction
10770	TV Movie
53	Thriller
10752	War
37	Western

Image 10: Movie Genres

We query in movie database like: https://api.themoviedb.org/3/discover/movie?with_genres=27. The genres are with the ids and there is set number of them. You can also just query all genres at once from the database. From Google Books Api, we query like: https://www.googleapis.com/books/v1/volumes?q=horror. The query key is the name of the genre in the movie database. The Google query system looks if the word is mentioned in the title or description of the book and returns matching copies. We are not able to, for example, search Lord of the Rings movies and tie them to the Lord of the Rings books somehow. The problem is that the Google books Api does not offer an enforced genre system. There are parameters such as categories, but there are unknown number of them and even for same theme.
By default, let's say we show 20 items on one page. Half, meaning ten would come from the books side, the other half from movies side. In the case the other Api does not return enough books, we query more from other side. These Apis supports offset parameters, so we can query items for each page separately and not have too much memory used at one time.
The problem is with drawing graphs. It would probably be best to allow the user the option to choose data set size. For example, the default 20, 200 or 2000

thousand. We don't really have it possible to compare all books and movies, because we cannot really query all that. If we had a separate server and database, we could query everything once there and make prebaked statistics to show the user. But by our design, the project has no database, but all data is processed and discarded in runtime.
The system needs the Api authorization tokens to work, so we should be extra careful to not commit them to Gitlab! Maybe we can send the teacher a text file with the tokens they can copy to their cloned copy folder, or they have to run build from us?
The data we get from the two Apis differs, so we need some work to map them together. Generally, we will use name, subtitle, cover image, description, release date, author / director, language, number of votes and votes average. Google Api seems to have far less votes also they rank the votes by 1-5 and movies Api 1-10. We will use 1 to 5 rating as it is more reliable to try to multiply the more unprecise data from Google Books up, to instead just divide each point in movies Api by two. We will show the votes count in short int, meaning if there are thousands show just “2.3k” and not “2394”. Also, under that, so number of stars from 1 to 5 or similar visually. The votes are shown on the movie page. The elements shown on the search list are just name, author / director, genre and the cover image.

Performance

Performance improvements will be made as necessary to ensure reasonable efficiency and ease of use.

Self-evaluation

We believe that we have been able to stick to our original design quite well. Some filters for the search functionality have been streamlined to remove unnecessary complexity and accommodations for the APIs parameter limitations have been made. We have improved the file structure of the program so that it better adheres to the MVC design pattern. Overall, the design has remained largely unchanged, only having been expanded upon and having details added to it, as our project progressed.
So far, we have been able to implement all the features in the program according to our original plan. Some features have turned out to be more complex than our initial design predicted but no major changes have been made to the design to implement these.

The program does mostly adhere to quality and the ability to add more APIs and functionality to program is available easily enough.
Since the previous iteration of our program, we have changed the way we populate the search results grid to present the results in a more visually pleasing manner and allowing us to manipulate the search results more directly and easily. This change also allows us to display all the search results on one page. Likewise, we have improved our API implementation to allow the users to use and combine filters while searching for books and movies.

Usage of AI

AI was used quite liberally during our project. We used ChatGPT. In general, the prompts given to ChatGPT were questions on how to implement something or having ChatGPT analyze snippets of code and explain what is wrong with it or how it works. The first draft of the design document was also generated by ChatGPT. All in all, the usage of AI was ubiquitous throughout the project.
We consider AI to have been very useful during the implementation process. As stated above, AI was used a lot during the implementation and helped with the creation of most of the code. The usefulness of AI in the design process was much less, although the initial features and the design provided by ChatGPT were used as the base for our project. This base changed quite a bit since the template provided by ChatGPT was very barebone and required many changes.
As for advantages of AI in software design, I would say that using AI can speed up the process, especially during the implementation. On the other hand, there is the danger of misinformation and misunderstanding on the AI's side. There is a chance that, for example, ChatGPT will offer outdated or faulty information. ChatGPT could also suggest design patterns that won't really work in your program. Especially dangerous is a situation where the suggestion appears to be valid, but has some slight errors, that won't be detected until later in the project.

Conclusion

This Software Design Document provides an overview of the architecture and functionality of the Movie and Book Recommendation Desktop App. It serves as a reference for the development and maintenance of the application, ensuring that it meets the needs and expectations of its users in a desktop environment.

Weekly meetings were held on almost all Fridays. Few exceptions were made when a meeting was deemed to be unnecessary. There was also no meeting on the exam week of period 1.

Hours (Eero)

Who	What	When	How long
Eero	11.9.	Working for more visually pleasing finished	3h
Eero	22.9.	Looking at the Api	1h
Eero	29.9	Some API development	5h
Eero	4.11.	Fix bug on the search updating every time	1h
Eero	13.11.	Trying to learn SceneBuilder	2h
Eero	23.11.	Trying to implement genres, did not work	1h
Eero	29.11.	Working on toast messages and saving references	5h

Hours (Sami)

Who	What	When	How long
Sami	Writing the design document	14.9	1h
Sami	Writing the design document	22.9	0,5h
Sami	Create single media item view	13.10	2h
Sami	Refactor controllers and views	29.10	2h
Sami	Refactor plotting and develop working plotter	22.11	2h
Sami	Refactor previous plotter with latest query function. Finish diagram plotting. Removed unused code, cleaned code.	23.11	2h

Hours (Niko)

Who	What	When	How long
Niko	Writing the design document	22.9	1h
Niko	Creating the main view of the program	11.10	6h
Niko	Completing the search function in the main view	24.10.-25.10.	2h
Niko	Fixed a bug in the mainViewController	27.10.2023	10 min
Niko	Writing the design document	29.10.2023	2h
Niko	Creating the graph view	3.11.2023	4-5 h
Niko	Writing the AI section on the design document	23.11.2023	1 h

Hours (Saurabh Chauhan)

Who	Task	How long
Saurabh	Design Document: Created initial design document, wrote about MVC, APIs, API Manager, Controller and Models. Also created class diagrams and other illustration required for the document.	10 hours
Saurabh	Meetings: Organized and scheduled meetings in first period of course so that everyone is on same page. Attended all the meeting organized by team.	8 h
Saurabh	Work Items: created various issues and tracked their progress with team members. Created pull requests, Reviewed pull requests, resolved bugs and merged PRs.	6h
Saurabh	Code: Added Api manager logic, refactored search media screen, refactored API response handling, created utility function to manage medias, GSON and other common logic required for the app. Created model classes and interface like media, book, movie and preference.	24h