Tampereen yliopisto

DESIGN DOCUMENT: MovieDemographics Explorer

COMP.SE.110 Software Design

Jonna Hartikka
Elias Penkkimäki
Lotte Karlsson
Aarni Heikkilä

CONTENTS

1. APPLICATION DESCRIPTION

MovieDemographics Explorer is an application that integrates movie information with global demographic statistics to provide visual insights into trends. The user can select a country, movie genre, indicator and time period to explore how many movies are produced or linked to that country and compare that data with the population demographics of that country. Additionally, the user can compare the data of two countries with each other.

Through a user interface, users can select a country from a dropdown menu to analyze movie-making trends and demographic data specific to that region. They can then choose a movie genre, such as action, drama, or comedy, to focus their analysis on specific types of movies. Next, they can select a population indicator (e.g., total population, population change, or life expectancy) to compare trends with the movie data. Finally, users select a time range (start year and end year) to explore how movie genre preferences or demographic conditions have evolved over specific periods. Once the "fetch data" button is clicked, the application retrieves data from backend APIs and presents the results graphically.

When the data for one country has been fetched, the user can press the 'Compare' button. The user selects the country and movie genre they want to compare the data with. After that, when the user presses the 'Fetch Comparison Data' button, both data sets are displayed. The comparison data can be removed by pressing the 'Remove Comparison' button.

The user can select what data is displayed in the user interface. Movie data is always displayed in the UI. Using checkboxes, the user can choose which population data to display based on the selected indicator. If desired, the user can view only movie data or movie data together with population data. Additionally, if the user selects "Total Population," they can also choose "Show Population by Age Groups”, which displays the population by age groups.

The user can save and view favorites in the user interface. By pressing "Add Favourite," the user can save the current selections as a favorite. By clicking the "Favourites" button, the user can view previously saved favorites, delete saved favorites, or activate one of the saved favorites.

The application integrates two external APIs to gather data based on user preferences. Movie data is sourced from The Movie Database (TMDB) API, while population data comes from the United Nations World Population Prospects (UN WPP) API.

1.1 Technology stack

The technology stack for this project involves a modern and robust combination of frontend, backend, and external tools to facilitate data visualization between movie genres and demographic data.

On the frontend, **React** serves as the main JavaScript library for building the user interface. React's component-based architecture allows for efficient rendering and supports dynamic, interactive elements such as forms, charts, and navigation. **Vite** is used as the build tool for the frontend, providing fast development speeds. The frontend environment is managed with **Node.js**, which runs build processes, while **NPM** (Node Package Manager) handles the project's dependencies, including React, Vite, and other essential libraries.

On the backend, **Java** is utilized for writing server-side logic, with the **Java Development Kit (JDK)** providing the necessary development tools. The backend is built using **Spring Boot**, a powerful framework that simplifies RESTful API development and enables seamless communication between the frontend and backend through HTTP requests. **Maven** is employed to manage dependencies, ensuring smooth integration of necessary libraries into the backend. The backend handles all business logic, retrieves data from external sources, and coordinates communication between various components of the system.

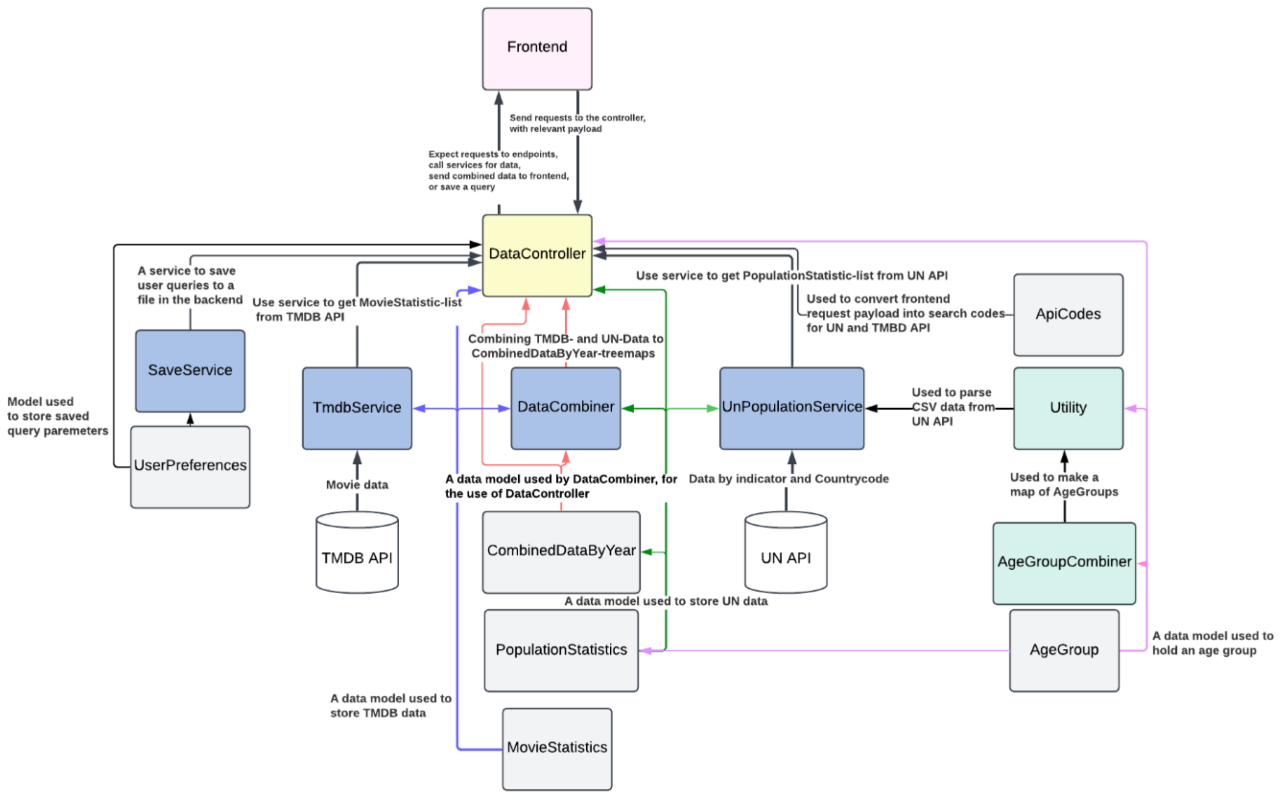
2. BOUNDARIES AND INTERFACES

We chose to document our interfaces using the internal interfaces diagram and Javadoc comments which are found above our components and methods in the code. The Javadoc can also be accessed by opening the **index.hml** file in the web browser and provides navigable documentation.

The file can be found in the following location of your downloaded copy of the app: ./movie-popularity-backend/apidocs/**index.html**

2.1 Internal Interfaces Diagram

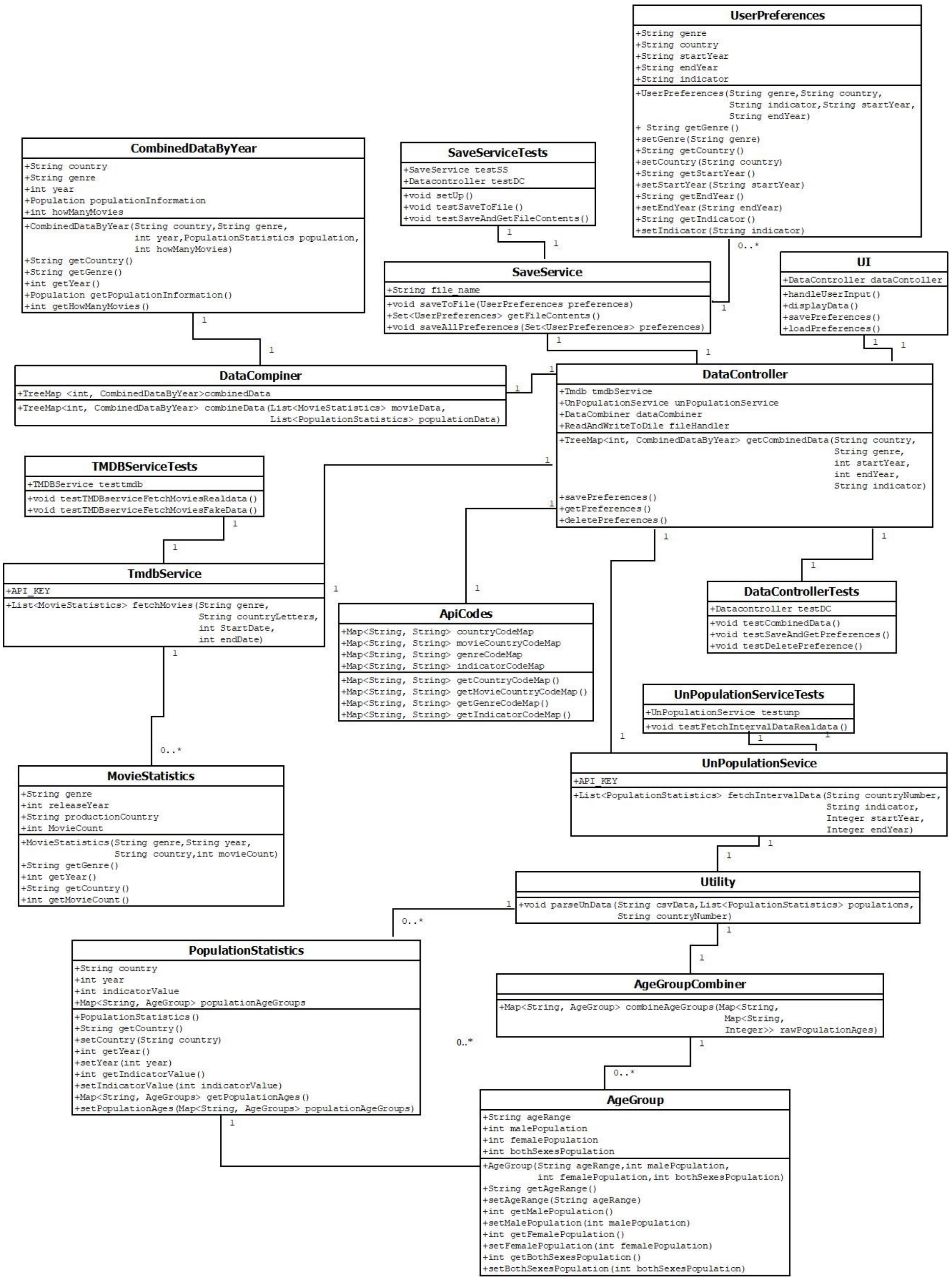
We have also created an internal interfaces diagram, to clarity the interfaces, and how information flows between them.


Frontend
Send requests to the controller, with relevant payload
Expect requests to endpoints, call services for data, send combined data to frontend, or save a query
DataController
A service to save user queries to a file in the backend
Use service to get MovieStatistic-list from TMDB API
Use service to get PopulationStatistic-list from UN API
Used to convert frontend request payload into search codes for UN and TMBD API
ApiCodes
Combining TMDB- and UN-Data to CombinedDataByYear-treemaps
SaveService
Model used to store saved query paremeters
UserPreferences
TmdbService
DataCombiner
UnPopulationService
Used to parse CSV data from UN API
Utility
Movie data
TMDB API
A data model used by DataCombiner, for the use of DataController
Data by indicator and Countrycode
Used to make a map of AgeGroups
CombinedDataByYear
UN API
AgeGroupCombiner
A data model used to store UN data
PopulationStatistics
AgeGroup
A data model used to hold an age group
A data model used to store TMDB data
MovieStatistics

Picture 1 Internal Interfaces Diagram

3. COMPONENTS AND RESPONSIBILITIES

3.1 UML class diagram


UserPreferences
+String genre
+String country
+String startYear
+String endYear
+String indicator
+UserPreferences(String genre,String country, String indicator,String startYear, String endYear)
+ String getGenre()
+setGenre(String genre)
+String getCountry()
+setCountry(String country)
+String getStartYear()
+setStartYear(String startYear)
+String getEndYear()
+setEndYear(String endYear)
+String getIndicator()
+setIndicator(String indicator)
CombinedDataByYear
+String country
+String genre
+int year
+Population populationInformation
+int howManyMovies
+CombinedDataByYear(String country,String genre, int year,PopulationStatistics population, int howManyMovies)
+String getCountry()
+String getGenre()
+int getYear()
+Population getPopulationInformation()
+int getHowManyMovies()
SaveServiceTests
+SaveService testSS
+Datacontroller testDC
+void setUp()
+void testSaveToFile()
+void testSaveAndGetFileContents()
SaveService
+String file_name
+void saveToFile(UserPreferences preferences)
+Set<UserPreferences> getFileContents()
+void saveAllPreferences(Set<UserPreferences> preferences)
UI
+DataController dataContoller
+handleUserInput()
+displayData()
+savePreferences()
+loadPreferences()
DataCompiner
+TreeMap <int, CombinedDataByYear>combinedData
+TreeMap<int, CombinedDataByYear> combineData(List<MovieStatistics> movieData, List<PopulationStatistics> populationData)
DataController
+Tmdb tmdbService
+UnPopulationService unPopulationService
+DataCombiner dataCombiner
+ReadAndWriteToDile fileHandler
+TreeMap<int, CombinedDataByYear> getCombinedData(String country, String genre, int startYear, int endYear, String indicator)
+savePreferences()
+getPreferences()
+deletePreferences()
TMDBServiceTests
+TMDBService testtmdb
+void testTMDBserviceFetchMoviesRealdata()
+void testTMDBserviceFetchMoviesFakeData()
TmdbService
+API_KEY
+List<MovieStatistics> fetchMovies(String genre, String countryLetters, int StartDate, int endDate)
ApiCodes
+Map<String, String> countryCodeMap
+Map<String, String> movieCountryCodeMap
+Map<String, String> genreCodeMap
+Map<String, String> indicatorCodeMap
+Map<String, String> getCountryCodeMap()
+Map<String, String> getMovieCountryCodeMap()
+Map<String, String> getGenreCodeMap()
+Map<String, String> getIndicatorCodeMap()
DataControllerTests
+Datacontroller testDC
+void testCombinedData()
+void testSaveAndGetPreferences()
+void testDeletePreference()
UnPopulationServiceTests
+UnPopulationService testunp
+void testFetchIntervalDataRealdata()
UnPopulationSevice
+API_KEY
+List<PopulationStatistics> fetchIntervalData(String countryNumber, String indicator, Integer startYear, Integer endYear)
MovieStatistics
+String genre
+int releaseYear
+String productionCountry
+int MovieCount
+MovieStatistics(String genre,String year, String country,int movieCount)
+String getGenre()
+int getYear()
+String getCountry()
+int getMovieCount()
Utility
+void parseUnData(String csvData,List<PopulationStatistics> populations, String countryNumber)
PopulationStatistics
+String country
+int year
+int indicatorValue
+Map<String, AgeGroup> populationAgeGroups
+PopulationStatistics()
+String getCountry()
+setCountry(String country)
+int getYear()
+setYear(int year)
+int getIndicatorValue()
+setIndicatorValue(int indicatorValue)
+Map<String, AgeGroups> getPopulationAges()
+setPopulationAges(Map<String, AgeGroups> populationAgeGroups)
AgeGroupCombiner
+Map<String, AgeGroup> combineAgeGroups(Map<String, Map<String, Integer>> rawPopulationAges)
AgeGroup
+String ageRange
+int malePopulation
+int femalePopulation
+int bothSexesPopulation
+AgeGroup(String ageRange,int malePopulation, int femalePopulation,int bothSexesPopulation)
+String getAgeRange()
+setAgeRange(String ageRange)
+int getMalePopulation()
+setMalePopulation(int malePopulation)
+int getFemalePopulation()
+setFemalePopulation(int femalePopulation)
+int getBothSexesPopulation()
+setBothSexesPopulation(int bothSexesPopulation)
0..*
1

Picture 2 UML class diagram

The UML class diagram above (Picture 2) describes the class division of the application, the attributes within the classes, the public methods and the relationships between the classes. The **UI** class is responsible for interacting with the user. It handles user input, saves and loads preferences, and displays the data to the user. The UI class interacts with the DataController to handle business logic and data retrieval.

DataController class is the main control class of the system. It coordinates the flow of data between the user interface, the APIs, and the data combiner. It handles user requests, retrieves data, and returns results. It also handles communication between frontend and backend. The React frontend and the Spring Boot backend communicate via REST APIs. The React application sends HTTP requests (GET, POST, etc.) to the Spring Boot backend to retrieve or send data. The Spring Boot backend provides a set of RESTful endpoints that the frontend can interact with.

Class **UserPreferences** captures the preferences of a user, such as the country, genre, indicator and date range they are interested in. DataController class interacts with the **SaveService** Class, which handles reading from and writing to files, specifically for saving and loading user preferences. When the user closes the program, the preferences are saved to the disk. When starting the application again, previously saved preferences can still be found from the application.

DataController also communicates with **TmdbService** and **UnPopulationService** classes to retrieve data from the respective APIs. The **ApiCodes** class, used by DataController, is responsible for converting codes related to countries, movies, genres, and indicators into human-readable values. TmdbService class retrieves data from TMDB movie API based on user's input selection about genre, country and time period. It generates **MovieStatistics** objects based on user's input and returns a list of created MovieStatistics objects. UnPopulationSevice class retrieves population data from the United Nations World Population Prospects (UN WPP) API based on a user's input. UnPopulationSevice uses **Utility** and **AgeGroupCombiner** classes to parse and categorize data and to create **PopulationStatistics** objects based on user's input. After that it returns a list of created PopulationStatistics objects to DataController. DataController uses **DataCombiner** class to combine received data into **CombinedDataByYear** objects. DataController then sends data back to the UI to display it to the user.

Test classes **DataControllerTests**, **SaveServiceTests**, **TMDBServiceTests** and **UnPopulationServiceTests** are responsible to test the key components of the program (DataController, SaveService, TmdbService and UnPopulationService) to ensure their functionality and reliability.

4. DESIGN DECISIONS

4.1 Model-View-Controller (MVC) Architecture

The application follows the MVC (Model-View-Controller) architecture pattern, which aims for clean, maintainable, and scalable code via separation.

Model (Active): In this pattern, the MovieStatistics, PopulationStatistics, AgeGroup, CombinedDataByYear, and UserPreferences, classes represent the Model, holding the data structures and handling logic. These models are the core representation of the data retrieved from external APIs and processed in the application.

View (Dynamic Frontend): The View is represented by the React frontend. It dynamically renders user interfaces, such as charts and forms, based on user interactions. The Chart.js enables graphical data representation, allowing users to explore trends visually.

Controller: The Controller is embodied in the DataController class. It handles user requests, communicates with the backend, processes the data, and returns it to the frontend. It also coordinates API calls through TmdbService and UnPopulationService classes, ensuring seamless integration between the frontend and backend.

This architecture adds complexity for small applications but enables expanding the application more easily in the future.

4.2 RESTful API Architecture

The RESTful API design is achieved by using JAVA backend with Spring Boot. This provides a way to communicate between the frontend and backend. Each API endpoint corresponds to a specific operation, such as fetching user set movie or population data, making the system extensible to future requirements (e.g., adding more endpoints for different data sources).

Resource-Based: The RESTful architecture provides easy-to-understand endpoints such as T /movies, /population or /combinedData

Separation: The frontend and backend can be separately developed and even completely separated for the future use of this program. This also allows independent testing.

4.3 Data Transfer Object Pattern

Data transfer object pattern (DTOP) is a pattern in which Data transfer objects (DTOs) carry data between processes to reduce the amount of method calls. In practice this is visible in nearly all our classes in the Model folder: they only include data and have no internal logic to them. They are passed between different parts of the program to save and transfer data but are internally just data holders. As an example, DataController and SaveService have the business logic for saving user preferences using UserPreferences – objects, which hold only the data of a preference: genre, country, indicator, startyear and endyear.

4.4 Single responsibility principle

Single Responsibility Principle (SRP) refers to a concept in Object oriented programming and stated that a class should be responsible for only one part of the functionality in the software. It also applies to how modules and functions should be designed. The advantages this design pattern provide are that code should be easier to understand, maintain and test.

We have used this pattern in the application and have to our best understanding divided the code into separate classes. This is visible in cases such as the Controller handling requests made to the app, the TMDB- and UnPopulationServices fetching the data, DataCombiner forming the data to be sent back, and the CombinedDataByYear forming the data structure of individual data from one year, along with other supporting data structures like MovieStatistics which have their own class instead of being a part of a larger class.

4.5 Singleton beans

Singleton refers to a design pattern where only a single instance of a class is created in the application, and is used repeatedly instead of creating more. We wanted to implement this pattern in our application, and it led to us choosing a favorable backend framework, Java Spring Boot, which has inbuilt support for it in the form of Singleton Beans. Beans are components automatically managed by Spring, and are usually marked with annotations like @Service, @Component and @RestController. Spring Boot Beans are singleton by default, and benefit from the singleton pattern upsides such as controlled access to the instance, testing ease, and possible performance improvements. This feature influenced our choice in using Java Spring Boot.

5. DIVISION OF WORK

Each member of the group actively contributed to brainstorming the project and designing the structure, architecture, and design patterns of the project. Lotte created the prototype using Figma. Regarding documentation, Jonna wrote chapters 1, 3, 5, and 7.1 and created the UML class diagram. Lotte wrote chapter 2, 4.4, 4.5 and drew the Internal Interfaces Diagram. Elias wrote chapters 4.1 and 4.2, while Aarni wrote chapters 4.3, and 6. Additionally, all group members collaborated on writing chapter 7.2.

On the coding side, Elias implemented the whole frontend using React. For the backend, Jonna implemented the classes AgeGroup.java, ApiCodes.java, CombinedDataByYear.java, DataCombiner.java, MovieStatistics.java, PopulationStatistics.java, and AgeGroupCombiner.java. Lotte implemented the classes TmdbService.java, UnPopulationService.java, and Utility.java. Aarni implemented the classes UserPreferences.java, SaveService.java, and the test classes (DataControllerTests.java, SaveServiceTests.java, TMDBServiceTests.java, UnPopulationServiceTests.java). The DataController.java class was implemented collaboratively by Jonna and Elias.

6. SELF EVALUATION

During the development, we faced some obstacles, but for the most part, the implementation went according to plan, and in the mid-term even better than planned, as we did not expect the app to have a core functionality completely functional until the very end of mid-term but finished it 1.5 weeks early. This is due to successful planning, communication, general know-how and good work by the team.

Changes to the original design document are utility classes in the utility-folder, for parsing data from the UN api and reforming the age groups in AgeGroupCombiner. Other functionality was mostly unchanged, other than multiple renamings of Classes and functions. The biggest examples being ApiPopulation, ApiMovies and ReadAndWriteToFile being changed into unPopulationService, TmdbService and SaveService. Some individual methods may also have different names, but the main functionalities are mostly unchanged other than in minor ways.

A quirk of the UN API, which sped up the development even further, was the fact that the UN API returns different data in the Totalpopulation field for the different indicators. This number can be used to display different indicators in the front end without any changes to the model to change between indicators, which we expected we would have to do. This means that the addition of more indicators was far easier than previously thought, and the model classes had to be renamed to reflect this.

The country code to language conversion had to be hard coded into the app, as the movie API did not have a functional “query by country name”-ability that was mentioned in its documentation. Instead, it offered only a language query. We had to resort to a sub-optimal solution because of the poor implementation of the API. It was unfortunate, as we partially chose the API because in its documentation it featured the country name query, but it was revealed to not have been supported for years. We chose to make the best of what we had available.

The design patterns and technologies we used made the development a lot smoother than expected, and fixing bugs was relatively straightforward.

7. THE USE OF AI

7.1 Idea for the application

We used the help of ChatGPT(4.0) to develop the idea for our application. First, we presented the group assignment requirements to ChatGPT and asked it to list various APIs we could use. ChatGPT gave a list of various APIs. Two APIs caught our interest: The Movie Database (TMDB) API and the United Nations World Population Prospects (UN WPP) API. We then asked ChatGPT if these APIs could be combined in some way. ChatGPT aswered this way: ”*At first glance, these APIs provide data from completely different domains: one focuses on global population statistics, while the other provides movie and entertainment data. However, with some creativity, you can combine these two datasets in a way that generates interesting insights*”. ChatGPT gave us different options to combine data, and we selected these suggestions:

- Idea: You could explore a relationship between global or regional population changes and the number of movies produced, watched, or rated highly in certain countries or globally. For instance, how does the population growth or demographic shifts in a particular country or region relate to:

    - The number of movies produced
    - Genre preferences or changing audience interests over time
- Compare the population data with movie production or popularity to generate insights such as:
    - Is there a correlation between population growth in certain countries and the types of movies they consume (e.g., genres)?
- The user can select parameters such as:
    - Time period (e.g., 1990–2023) for both movie data and population data.
    - Specific countries or regions for the population data, then filter movie data by the same region.
    - Type of insights they want to explore (e.g., movie production trends, genre popularity).
- You'll need to retrieve and clean both datasets, ensuring they align in terms of country codes, time periods, etc. Merging population data with movie data on common keys like country and year will enable meaningful comparisons.
- You could create visualizations such as:
    - Line charts comparing population growth and the number of movies produced or watched in a specific country or region over time.
    - Bar charts showing movie genre popularity vs. age demographics.
- Allow user interactions to filter by country, time period, genre, etc., and dynamically update the graphs.

7.2 Other use of AI Tools

ChatGPT-4 and **CoPilot** were used for the following purposes:

Assistance in writing the design document: A picture of the UML class diagram was sent to the ChatGPT, and it was asked to describe the relationships between the classes. The response was used as an aid in describing the structure of the program. Additionally, the ChatGPT was asked for help in writing the application overview and technology stack sections and proofreading the text.

Creating Javadoc comments for the following classes: PopulationStatistics, MovieStatistics, CombinedDataByYear, AgeGroupCombiner, AgeGroup, ApiCodes and DataController. The entire code for each class was copied and provided to the ChatGPT, and it was asked to add Javadoc comments to the code.

Assistance with code writing: In the AgeGroupCombiner class, the ChatGPT was asked to help implement the combineAgeGroups function. The ChatGPT was given an explanation of what the function was supposed to do, and it provided the implementation. Afterward, small modifications were made to the code.

In the SaveService class ideas for the structure of the implementation were got from CoPilot, but significant changes were made as describing the whole design pattern was difficult, and there were some errors in its initial design and use of the UserPreferences class. The saving as a set was also a later change, as the AI gave only ideas based on the temporarily implemented saving model of the time.

The data parsing method for CSV data of Utility.utility.parseUnData() was strongly written with CoPilot help, as it was a very time consuming parsing function to write. The end result is very functional and fulfills the requirement but is not the cleanest possible implementation.

Frontend development: Ai was used to debug REACT syntax errors and to suggest a suitable library (chart.js) for creating the visualization. Mostly questions were asked in the ChatGPT chat and in some cases code and error messages were pasted to the chat and given response was analyzed and asked further questions if the response did not make any sense. Usually some of the parts were used as is but in most cases they needed to be adapted to the existing code.