Movie Recommendation System
Design Document

Authors:
Alex Horton
David Ho
Marko Kosoric

Group: 9

The purpose of this document is to provide you with a guideline for writing the software design document for your project.

Points to remember:

- *Content is important, not the volume.* ***Another team should be able to develop this system from only this document.***
- *Pay attention to details.*
- *Completeness and consistency will be rewarded.*
- *Readability is important.*

This page intentionally left blank.

Document Revision History

Date	Version	Description	Author
11/25/2025	1.0	First final version	Alex Horton, David Ho, Marko Kosoric

This page intentionally left blank.

Contents

1 Introduction 6
1.1 Purpose 6
1.2 System Overview 6
1.3 Design Objectives 6
1.4 References 6
1.5 Definitions, Acronyms, and Abbreviations 7
2 Design Overview 7
2.1 Introduction 7
2.2 Environment Overview 7
2.3 System Architecture 7
2.4 Constraints and Assumptions 10
3 Interfaces and Data Stores 11
3.1 System Interfaces 11
3.2 Data Stores 11
4 Structural Design 12
4.1 Class Diagram 12
4.2 Classes in the Movie Recommendation System 12
5 Dynamic Model 23
5.1 Scenarios 23
6 Non-functional requirements 25
7 Recommendation Algorithm Pseudocode 25

1 Introduction

1.1 Purpose

The purpose of this Software Design Document is to describe the design and architecture of the Movie Recommendation System. This document explains how the system's main components collaborate, how data is stored and processed, and how the system achieves its functional and non-functional requirements. It is intended for developers who will implement, extend, or review the system. It also serves an educational purpose within the COMPSCI 2ME3:Introduction to Software Development course by providing experience with the Waterfall and Agile software processes. The document is structured into sections including an introduction, design overview, interfaces and data stores, structural design, dynamic model, non-functional requirements, and recommendation algorithm pseudocode.

1.2 System Overview

The Movie Recommendation System allows users to review movies and receive movie recommendations based on their past ratings. The system runs through a command-line interface and processes two primary datasets: a movie database and a set of user ratings. Users can add or update reviews, and the recommendation engine checks the user's previous highly rated movies to generate personalized recommendations. In the case where a new user has not submitted any ratings, the system provides recommendations based on globally highly rated movies in the dataset. The system is intended to be lightweight, file-based, and executable on any local machine that supports Java and CSV text files.

1.3 Design Objectives

The primary design objective of the Movie Recommendation System is to implement an architecture that fulfills the functional and non-functional requirements outlined in this document and in the software requirements specification document. The design supports all essential system capabilities, including file parsing, user rating functionality, and a movie recommendation engine that compares movie genres and user ratings. The system must read and parse movie data from movies.txt, load existing user ratings from my_ratings.txt, and make sure that users can add or update reviews. In addition to functional goals, the design must emphasise modularity, separation of concerns, and handle missing files and malformed data. Each class has a dedicated responsibility. For example, FileHandler exclusively manages file operations while MovieDB stores movie objects. Performance and usability requirements are also reflected in the design. Since all processing occurs locally and in memory, the system produces instant responses to all user actions. The CLI remains simple and intuitive, minimizing user error and ensuring that malformed input does not lead to a system crash.

1.4 References

Requirements Specification Document for the Movie Recommendation System

Provided assignment instructions

COMPSCI 2ME3 course slides and design templates

1.5 Definitions, Acronyms, and Abbreviations

MovieDB - A class responsible for storing all Movie objects loaded from the movies.txt dataset

CLI - Command-Line Interface used by the user to interact with the system.

CSV - Comma-separated values. Format used for system data files.

GUI – Graphical user interface.

2 Design Overview

2.1 Introduction

This system follows an object-oriented design approach. Each major responsibility is encapsulated within a class: file management, movie storage, review handling, user operations, and the recommendation engine. The system architecture is modular. UML diagrams and structured class descriptions are used to illustrate class design, data flows, component interactions, and architecture.

2.2 Environment Overview

The Movie Recommendation System is designed to run on a local machine with the following environment characteristics:

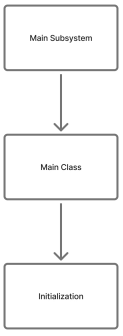
- *Execution Environment: Local Java Runtime. No network connectivity is required.*
- *User interaction: A command-line interface where the user types commands such as "review movie" or “get recommendations”.*
- *Data Environment: Two external CSV files that are stored in the same directory as the rest of the program.*
- *Hardware Requirements: Minimal. Only sufficient memory is required to load approximately 100 movie objects and a small set of user ratings.*

The system does not depend on APIs, databases, or servers. All processing occurs in memory once the files are loaded.

2.3 System Architecture

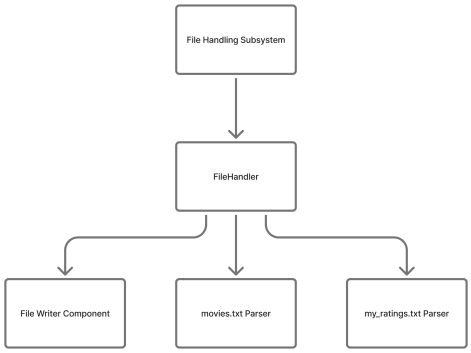
The architecture consists of seven major components:

2.3.1 Main Subsystem



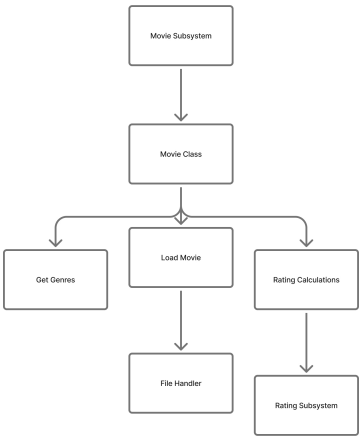
Initializes the system components.

2.3.2 File Handler Subsystem



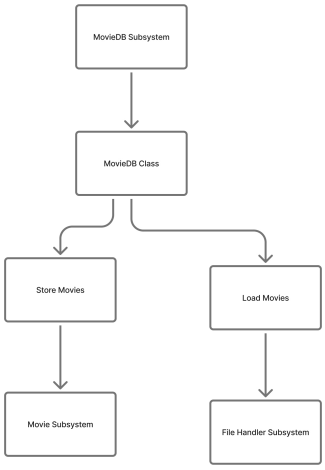
Manages reading and writing the CSV files containing movies and user ratings. Interacts with the review class to update rating files.

2.3.3 Movie Subsystem



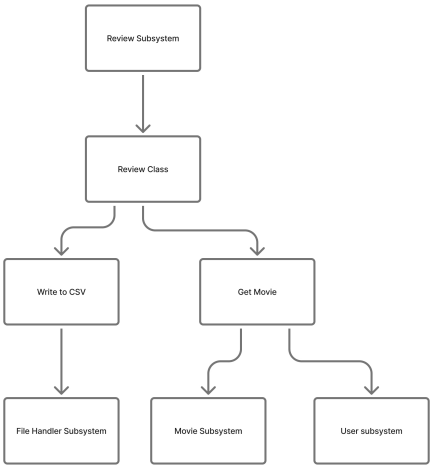
Encapsulates relevant data for each movie place from movies.txt.

2.3.4 MovieDB Subsystem



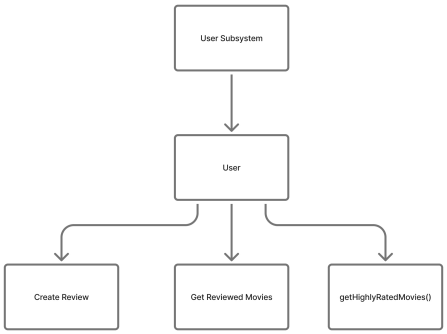
Stores all Movie objects that are pulled from the movies.txt CSV file.

2.3.5 Review Subsystem



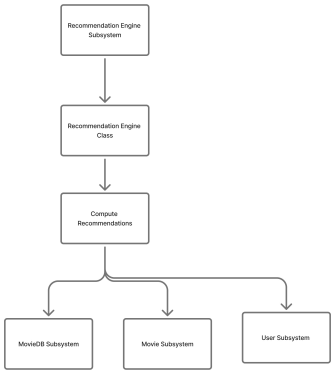
Encapsulates and updates review data pulled using the FileHandler class and the my_ratings.txt CSV file. Stores the User and Movie object with every review.

2.3.6 User Subsystem



Encapsulates all relevant user data such as their reviews, and their associated operations. Calls Movie, Review, and MovieDB classes to perform functions that get highly rated movies and create reviews. Creates a map of the Movie objects to the Review objects to pull reviews for each movie.

2.3.7 Recommendation Engine Subsystem



Compares genres and ratings to compute recommendations based on similarity. MovieDB provides all movies. User provides rating history. Movies provide metadata such as genre and score.

2.4 Constraints and Assumptions

This project must be implemented in Java using standard libraries for file operations such as BufferedReader, FileReader, BufferedWriter, and FileWriter.

The project must read from two CSV files which are movies.txt and my_ratings.txt. The movies.txt file must follow the format movie_id,title,director,year,genres,avg_rating,num_ratings, where genres may contain multiple values separated by semicolons. The my_ratings.txt file must follow the format movie_id,rating,timestamp.

The project has to handle missing or malformed files without crashing. If movies.txt is missing, then the program cannot run and should exit safely. If my_ratings.txt is missing, then the program creates the file automatically when the first rating is added. Any malformed lines in either text file should be skipped and errors should be logged or printed without stopping the program.

The project follows the Waterfall development process which requires that the design is fully documented before being implemented. All requirements must trace back to the SRS and SDD.

The assignment allows the use of either a GUI or CLI interface. Our group chose to implement a CLI interface because it is simpler to implement, avoids the complexities of JavaFX or Swing, allows us to focus on core functionality such as file parsing and the recommendation algorithm, and is easier to debug and test.

3 Interfaces and Data Stores

3.1 System Interfaces

User interface will be a simple and intuitive CLI that the user can get access to the systems features with commands such as:

- *Review movie: Start movie reviewing process*
- *Get recommendations: System generates recommendations and return results*
- *Cancel: Can cancel any operation or command they have executed*

3.2 Data Stores

3.2.1 Data: Files

System processes two files: movies.txt and my_ratings.txt and both must be in CSV format.

movies.txt format: movie_id,title,director,year,genres,avg_rating,num_ratings

- *Genres are separated by semicolons like so “Action;Adventure;..”*

my_ratings.txt format: movie_id,rating,timestamp

3.2.2 Data: User Ratings

User ratings will be encapsulated inside a Review Class to store all of its relevant data (movie: Movie, rating: float, and date: LocalDate)

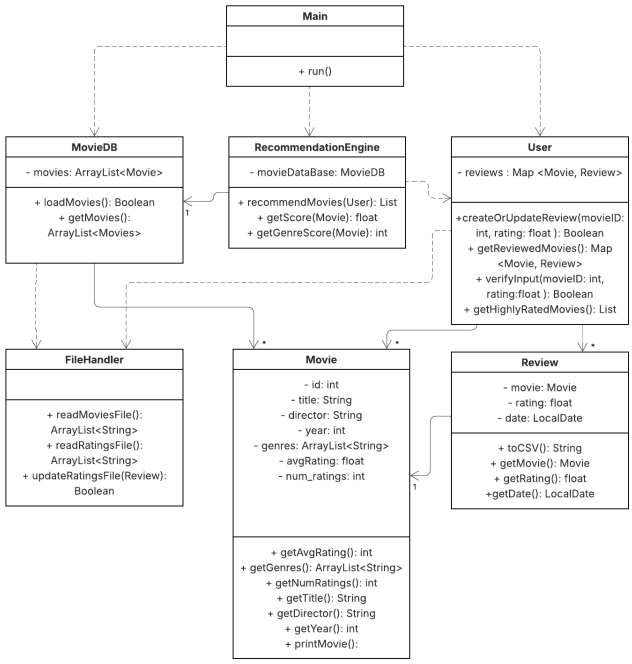
These reviews will then be stored as an attribute in the User object as a Map <Movie, Review> which is an implementation of a Hashmap to allow for quick lookups of specific reviews.

3.2.3 Data: Movies

Movies will be read from movies.txt and stored inside a MovieDB Class as an ArrayList<Movie>.
Each element in the ArrayList will be of type Movie class that has attributes for its relevant data (id: Integer, title: String, director: String, year: Integer, genres: ArrayList<String>, average rating: float, number of ratings: Integer)

4 Structural Design

4.1 Class Diagram



4.2 Classes in the Movie Recommendation System

4.2.1 Class: Main

- *Purpose: To consolidate system running. Handles instantiating all objects.*
- *Constraints:None*
- *Persistent: No (created at system initialization from other available data)*

4.2.1.1 Attribute Descriptions

- *No Attributes*

4.2.1.2 Method Descriptions

1. *Method: run()*
   Return Type: void
   Parameters: none
   Return value: none

Pre-condition: Access to movie and review text files
Post-condition: Movie file unchanged, review file potentially updated
Attributes read/used: User object, MovieDB object, RecommendationEngine object
Methods called: User(), MovieDB(), RecommendationEngine()

Processing logic:
Create instances of MovieDB, RecommendationEngine, and User Classes to then be manipulated to allow the user to update and create reviews, and get recommendations.

4.2.2 Class: FileHandler

- *Purpose: To handle file processing logic of system, read and write.*
- *Constraints: movies.txt and my_ratings.txt must be accessible and be in CSV format*
- *Persistent: No (created at system initialization from other available data)*

4.2.2.1 Attribute Descriptions

- *No attributes*

4.2.2.2 Method Descriptions

1. *Method: readMoviesFile*
   Return Type: ArrayList<String>
   Parameters: none
   Return value: List where each entry represents a line of movies.txt file
   Pre-condition: movies.txt should be accessible and in CSV format
   Post-condition: movies.txt should be unmodified
   Attributes read/used: none
   Methods called: none

   Processing logic:
   Read movies.txt and handle any malformed files properly. Ensure no crashes happen. Return the ArrayList<String> housing every line from movies.txt when valid.

   Test case 1: Call readMoviesFile with valid movies.txt and print results.
   Expected output is: List where each entry represents a line of the file, length should be equal to number of lines in file

   Test case 2: Call readMoviesFile with malformed movies.txt and print results.
   Expected output is: Empty list

2. *Method: readRatingsFile*
   Return Type: ArrayList<String>
   Parameters: none
   Return value: List where each entry represents a line of my_ratings.txt file
   Pre-condition: my_ratings.txt needs to be accessible and valid
   Post-condition: my_ratings.txt should be unmodified
   Attributes read/used: none
   Methods called: none

   Processing logic:
   Read my_ratings.txt and handle any malformed files properly. Ensure no crashes happen. Return the ArrayList<String> housing every line from my_ratings.txt when valid.

   Test case 1: Call readRatingsFile with valid my_ratings.txt and print results.

Expected output is: List where each entry represents a line of the file, length should be equal to number of lines in file

Test case 2: Call readRatingsFile with a malformed my_ratings.txt and print results.
Expected output is: Empty list

3. *Method: updateRatingsFile*
   Return Type: boolean
   Parameters: Review
   Return value: true or false
   Pre-condition: my_ratings.txt has to be accessible and valid, and Review parameter must not be null
   Post-condition: my_ratings.txt will have a new line for the new Review or update an already existing line (movie already reviewed previously)
   Attributes read/used: movie, rating, date (from Review object)
   Methods called: none

   Processing logic:
   Either append new line to end of my_ratings.txt if review doesn't already exist, or update if it does exist.

   Test case 1: Call updateRatingsFile with valid reviews.txt and Review for new movie then observe my_ratings.txt.
   Expected output is: true and a new line in my_ratings.txt representing the new review

   Test case 1: Call updateRatingsFile with valid reviews.txt and Review for a movie that was already reviewed then check my_ratings.txt
   Expected output is: true and the same amount of lines except the line corresponding to the already reviewed movie will be updated.

   Test case 3: Call updateRatingsFile with malformed reviews.txt and a new Review or an existing Review then check my_ratings.txt
   Expected output is: false and no change to my_ratings.txt

4.2.3 Class: Movie

- *Purpose: Encapsulate relevant data for each movie place from movies.txt*
- *Constraints: Movies must have unique ID, movies file must be in a valid format and readable.*
- *Persistent: No (created at system initialization from other available data)*

4.2.3.1 Attribute Descriptions

1. *Attribute: id*
   Type: Integer
   Description: Stores the unique Identifier for the movie
   Constraints: should be a unique integer that doesn't clash with any other movie object. Between 0 and 100 inclusive

2. *Attribute: title*
   Type: String
   Description: Store title of the movie
   Constraints: none

3. *Attribute: director*
   Type: String

Description: Store name of the director of the movie
Constraints: none

4. *Attribute: year*
   Type: Integer
   Description: Store the year the movie was released
   Constraints: non-negative

5. *Attribute: genres*
   Type: ArrayList<String>
   Description: Store the genres of the movie
   Constraints: none

6. *Attribute: avgRating*
   Type: Integer
   Description: Stores the average rating of the movie
   Constraints: Between 0 and 5

7. *Attribute: num_ratings*
   Type: Integer
   Description: Store the number of ratings the movie has
   Constraints: non-negative

4.2.3.2 Method Descriptions

1. *Method: getAvgRating*
   Return Type: float
   Parameters: none
   Return value: float between 0 and 5
   Pre-condition: Movie object must exist and have a non-null avgRating field
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: avgRating
   Methods called: none

   Processing logic:
   Getter method to return the avgRating of a specific movie. Method simply returns the number stored in the avgRating field.

   Test case 1: Call getAvgRating for pressure Inception (movieID = 6) and print results. Expected output is: 4.7

2. *Method: getGenres()*
   Return Type: ArrayList<String>
   Parameters: none
   Return value: genres attribute of specific Movie object
   Pre-condition: Movie object must exist and have a non-null genres field
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: genres
   Methods called: none

   Processing logic:
   Getter method to return the genres field of a specific movie. Method simply returns the ArrayList<String> stored in the genres field.

Test case 1: Call getGenres() with Star Wars: Episode V (movieID = 13) and print results.
Expected output is: ["Action", "Adventure", "Fantasy"]

3. *Method: getNumRatings*
   Return Type: Integer
   Parameters: none
   Return value: num ratings attribute of specific Movie object
   Pre-condition: Movie object must exist and have a non-null num ratings field
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: numRatings
   Methods called: none

   Processing logic:
   Getter method to return the numRatings field of a specific movie. Method simply returns the Integer stored in the numRatings field.

   Test case 1: Call getNumRatings with Toy Story (movieID = 39).
   Expected output is: 1876

4. *Method: getTitle*
   Return Type: String
   Parameters: none
   Return value: title attribute of specific Movie object
   Pre-condition: Movie object must exist and have a non-null num title field
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: title
   Methods called: none

   Processing logic:
   Getter method to return the title field of a specific movie. Method simply returns the String stored in the title field.

   Test case 1: Call getTitle with Toy Story (movieID = 39).
   Expected output is: ToyStory

5. *Method: printMovie*
   Return Type: void
   Parameters: none
   Return value: none
   Pre-condition: Movie object must exist and have non-null fields
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: title, director, years, genres, avgRating, num_ratings
   Methods called: getAvgRating(), getGenres(), getNumRatings(), getTitle(), getTitle(), getDirector()

   Processing logic:
   Method to print specific movie object in a readable way to the user.

   Test case 1: TBD (Exact design TBD)

6. *Method: getDirector*
   Return Type: String
   Parameters: none
   Return value: director field of movie object
   Pre-condition: Movie object must exist and have non-null director field

Post-condition: Exact same state as pre-condition, method does not modify state
Attributes read/used: director
Methods called:

Processing logic:
Getter method to return the director field of a specific movie. Method simply returns the String stored in the director field.

Test case 1: Call getNumRatings with Toy Story (movieID = 39).
Expected output is: John Lasseter

7. *Method: getYear*
Return Type: Integer
Parameters: none
Return value: Integer stored in year field
Pre-condition: Movie object must exist and have non-null year field
Post-condition: Exact same state as pre-condition, method does not modify state
Attributes read/used: year
Methods called: none

Processing logic:
Getter method to return the year field of a specific movie. Method simply returns the Integer stored in the year field.

Test case 1: Call getNumRatings with Toy Story (movieID = 39).
Expected output is: 1995

4.2.4 Class: MovieDB

- *Purpose: To store all Movies read from movies.txt along with their relevant information*
- *Constraints: movies.txt must be accessible and in the proper format, only one instance of MovieDB should exist*
- *Persistent: No (created at system initialization from other available data)*

4.2.4.1 Attribute Descriptions

1. *Attribute: movies*
Type: ArrayList<Movies>
Description: Stores all movies read from movies.txt inside an array
Constraints: Should not be null, and should be of type Movies

4.2.4.2 Method Descriptions

1. *Method: loadMovies*
Return Type: Boolean
Parameters: none
Return value: true or false
Pre-condition: MovieDB must be instantiated and movies.txt must be readable and valid
Post-condition: movies attribute will now store all movies from movies.txt, length of movies should be same as amount of lines in movies.txt
Attributes read/used: movies
Methods called: readMoviesFile()

Processing logic:

loadMovies method calls the FileHandler method readMoviesFile() to retrieve the data from movies.txt. It will return true if everything is done properly.

Test case 1: Call loadMovies with valid movies.txt and print results of movies field along with return value of loadMovies.
Expected output is: true and data from movies file

Test case 2: Call loadMovies with invalid movies.txt (empty, invalid format) and print results of movies field along with return value of loadMovies.
Expected output is: false and null

2. *Method: getMovies*
   Return Type: ArrayList<Movies>
   Parameters: none
   Return value: movies attribute of MovieDB object
   Pre-condition: movies attribute should be non-null and of type ArrayList<Movies>
   Post-condition: Exact same state as precondition, this method does not alter state
   Attributes read/used: movies
   Methods called: none

   Processing logic:
   Getter method that returns the movies field of the MovieDB object. Simply returns the movies attribute.

   Test case 1: Call getMovies with proper movies.txt file.
   Expected output is: ArrayList storing all Movie objects, with length equal to amount of lines in file.

   Test case 2: Call getMovies with malformed movies.txt file
   Expected output is: null ArrayList

4.2.5 Class: Review

- *Purpose: To encapsulate Review data read from my_ratings.txt inside an object*
- *Constraints: Only one review per movie (per user), my_ratings.txt must be accessible and valid*
- *Persistent: No (created at system initialization from other available data)*

4.2.5.1 Attribute Descriptions

1. *Attribute: movie*
   Type: Movie
   Description: Stores the movie object the review is about
   Constraints: Must be non-null and of type Movie with all associated fields instantiated properly

2. *Attribute: rating*
   Type: Float
   Description: Store the rating the User assigned to the movie
   Constraints: Between 0 and 5

3. *Attribute: date*
   Type: LocalDate
   Description: Stores the date the User created the review
   Constraints: Should be of form YYYY-MM-DD

4.2.5.2 Method Descriptions

1. *Method: toCSV*
   Return Type: String
   Parameters: none
   Return value: CSV representation of Review
   Pre-condition: Review instantiated with all attributes non-null
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: movie, rating, date
   Methods called: getMovie(), getRating(), getDate()

   Processing logic:
   Convert the Review into CSV form so it can be stored inside my_ratings.txt

   Test case 1: Call toCSV() with movieID = 88 and unmodified my_ratings and print results.
   Expected output is: 88,5,2024-10-29

2. *Method: getMovie*
   Return Type: Movie
   Parameters: none
   Return value: Movie object corresponding to the Review
   Pre-condition: Review instantiated with movie attribute non-null
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: movie
   Methods called: none

   Processing logic:
   Getter method that returns the movie attribute of the specific Review object. This method simply returns the value stored inside the movie attribute.

   Test case 1: Call getMovie with unmodified and valid my_ratings.txt for review with movieID = 7 and print results.
   Expected output is: 7,The Matrix,The Wachowskis,1999,Action;Sci-Fi,4.6,2876

3. *Method: getRating*
   Return Type: float
   Parameters: none
   Return value: rating for the specific Review object
   Pre-condition: Review object instantiated, with a non-null rating attribute.
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: rating
   Methods called: none

   Processing logic:
   Getter method that returns the rating attribute of the specific Review object. This method simply returns the value stored inside the rating attribute.

   Test case 1: Call getRating with unmodified and valid my_ratings.txt for review with movieID = 35 and print results.
   Expected output is: 2

4. *Method: getDate*
   Return Type: LocalDate
   Parameters: none
   Return value: date review was created

Pre-condition: Review object instantiated, with a non-null rating attribute.
Post-condition: Exact same state as pre-condition, method does not modify state
Attributes read/used: date
Methods called: none

Processing logic:
Getter method that returns the date attribute of the specific Review object. This method simply returns the value stored inside the date attribute

Test case 1: Call getDate with unmodified and valid my_ratings.txt for review with movieID = 88 and print results.
Expected output is: 2024-10-29

4.2.6 Class: User

- *Purpose: To encapsulate all relevant user data such as their reviews, and their associated operations*
- *Constraints: my_ratings.txt is accessible and valid, only one User object should exist*
- *Persistent: No (created at system initialization from other available data)*

4.2.6.1 Attribute Descriptions

1. *Attribute: reviews*
   Type: Map<Movie, Review>
   Description: Hashmap that maps key (movie) to values (Review)
   Constraints: Should be of length equivalent to amount of lines in my_ratings.txt

4.2.6.2 Method Descriptions

1. *Method: createOrUpdateReview*
   Return Type: Boolean
   Parameters: movieID: int, rating: float
   Return value: true or false
   Pre-condition: my_ratings.txt and movies.txt file are accessible and valid, User is instantiated with well-defined reviews object and parameters are non-null.
   Post-condition: Either appends new review to my_ratings.txt or adds a new line when my_ratings.txt is valid
   Attributes read/used: reviews
   Methods called: verifyInput(), getReviewedMovies(), toCSV(), updateRatingsFile(Review)

   Processing logic:
   Takes a movieID and a rating which is verified by the verifyInput() method to ensure it is valid. Method getReviewedMovies() is called to check if a review exists already. If a review exists the my_ratings.txt file is updated otherwise it is appended to the end

   Test case 1: Call createOrUpdateRview with valid and unmodified my_ratings.txt and Review: movieID = 23, rating = 5 and observe my_ratings.txt.
   Expected output is: new line corresponding to - 23, 5, current date of format YYYY-MM-DD

   Test case 2: Call createOrUpdateRview with valid and unmodified my_ratings.txt and existing Review: movieID = 15, rating = 5 and observe my_ratings.txt.
   Expected output is: Updated line for movieID = 15 to be – 15, 5, current date of format YYYY-MM-DD

2. *Method: getReviewedMovies*
   Return Type: Map<Movie, Review>
   Parameters: none
   Return value: reviews attribute for specific User object
   Pre-condition: my_ratings.txt and movies.txt are accessible and valid, and User object instantiated with ratings non-null
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: reviews
   Methods called: none

   Processing logic:
   Getter method that returns the reviews attribute of the specific User object. This method simply returns the value stored inside the review attribute

   Test case 1: Call getReviewedMovies with valid and unmodified my_reviews.txt and print results.
   Expected output is: Mapping of movie object (use id to represent movie) to review like so –
   3 : 3,5,2024-10-15
   6 : 6,5,2024-10-18
   7 : 7,4,2024-10-20
   17 : 17,5,2024-10-22
   20 : 20,4,2024-10-25
   15 : 15,3,2024-10-26
   35 : 35,2,2024-10-28
   88 : 88,5,2024-10-29

3. *Method: verifyInput*
   Return Type: boolean
   Parameters: movieID: Integer, rating: Float
   Return value: true or false
   Pre-condition: my_ratings.txt and movies.txt are accessible and valid, and User object instantiated
   Post-condition: Exact same state as pre-condition, method does not modify state
   Attributes read/used: none
   Methods called: none

   Processing logic:
   Method verifies if user input is valid, helps with review creation to ensure further methods aren't called with invalid input.

   Test case 1: Call verifyInput with movieID = 101, rating = 5 .
   Expected output is: false since movie does not exist

   Test case 2: Call verifyInput with movieID = 9, rating = 6 .
   Expected output is: false since rating is out of bounds

   Test case 3: Call verifyInput with movieID = 43, rating = 2.1 .
   Expected output is: true, both are proper type and within bounds

4. *Method: getHighlyRatedMovies*
   Return Type: List
   Parameters: reviews
   Return value: List housing the Users highly rated movies (Score of 4 or greater)

Pre-condition: my_ratings.txt and movies.txt are accessible and valid, and User object instantiated
Post-condition: Exact same state as pre-condition, method does not modify state
Attributes read/used: none
Methods called: getReviewdMovies()

Processing logic:
Method goes through the Users reviewed movies and creates a List housing all the Users top rated movies.

Test case 1: TBD

4.2.7 Class: RecommendationEngine

- *Purpose: To house main recommendation algorithm that processes user ratings to produce recommendations*
- *Constraints: my_ratings.txt and movies.txt must be valid and accessible, User object must be instantiated with non-null attribute*
  - *Special case cold-start (no reviews): Recommend popular movies*
- *Persistent: No (created at system initialization from other available data)*

4.2.7.1 Attribute Descriptions

1. *Attribute: movieDataBase*
   Type: MovieDB
   Description: Stores the movie database, i.e all movies loaded from movies.txt
   Constraints: Should be of length 100 with each entry of type Movie

4.2.7.2 Method Descriptions

1. *Method: recommendMovies*
   Return Type: List
   Parameters: User
   Return value: List of recommended movies
   Pre-condition: movies.txt and my_ratings.txt valid and accessible, User object instantiated and movieDataBase attribute non-null
   Post-condition: Exact same state as pre-condition, this method does not alter state
   Attributes read/used: movieDataBase,
   Methods called: User – getReviewedMovies() , Review – getRating() , MovieDB – getMovies() , Movie – getAvgRating() and getNumRatings() and getGenres()

   Processing logic:
   Calculate recommendations based on Users top rated movies (4-5 stars) and compare it to the movieDB. Do genre matching and consider the movies average score

   Test case 1: TBD

2. *Method: getScore*
   Return Type: Float
   Parameters: Movie
   Return value: Similarity score of given movie, to be used in recommendation algorithm
   Pre-condition: movies.txt and my_ratings.txt valid and accessible, User object instantiated and movieDataBase attribute non-null
   Post-condition: Exact same state as pre-condition, this method does not alter state
   Attributes read/used: none

Methods called: none

Processing logic:
Calculate similarity score of input Movie based on Users top rated movies (4-5 stars) and. Does genre matching and considers the movies average score.

Test case 1: TBD

3. *Method: getHighlyRatedScore*
Return Type: Integer
Parameters: Movie
Return value: Integer value representing amount of genres the input movie matches to the users highly rated movies
Pre-condition: movies.txt and my_ratings.txt valid and accessible, User object instantiated and movieDataBase attribute non-null
Post-condition: Exact same state as pre-condition, this method does not alter state
Attributes read/used: none
Methods called: none

Processing logic:
Calculates amount of genres this movie matches to Users highly rated movies (Score of 4 to 5).

Test case 1: TBD

---

5 Dynamic Model

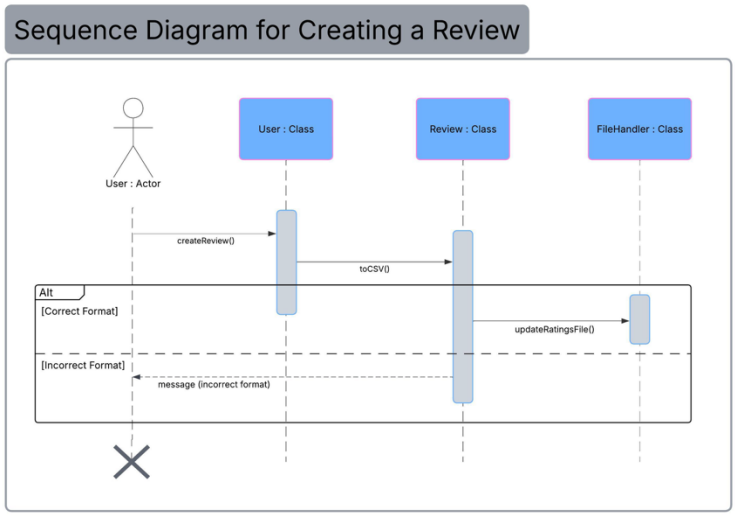
5.1 Scenarios

5.1.1 Scenario: User Making a Review

5.1.1.1 Scenario Description

When the user wants to make a review the createReview() method in the User class will be called. This will take the user's inputs for the review then call toCSV() in the Review class which will receive the users inputs and if they are valid put it into the CSV format and call updateRatingsFile() in the FileHandler class which will then insert the review into the reviews file. However if the user did not put in a valid input for one of the review sections the review class will send a message back to the user saying that it is not properly formatted and the user will have to try again.

5.1.1.2 Sequence Diagram



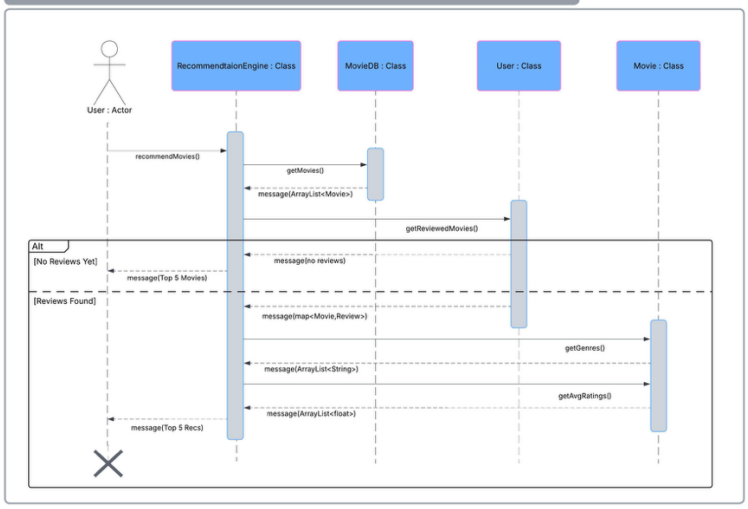
5.1.2 Scenario: User Requesting Recommendations

5.1.2.1 Scenario Description

When the user would like to receive movie recommendations the recommendMovies() method will be called in RecommendationEngine. This will get the list of movies from MovieDB using the getMovies() method, it will then getReviewedMovies() from User class and then do one of two things. If the user has no reviews then it will just return the top 5 highest rated movies in the movie list. If the user has reviews RecommendationEngine will getGenres() and getAvgRating() from the Movie class and calculate the score of movies using their genres and average rating based on the users reviews and return a customized top 5 movie recommendations.Sequence Diagram

5.1.2.2 Sequence Diagram

Sequence Diagram for Getting Movie Recommendations



6 Non-functional requirements

NF-1: File handling is processed within an isolated FileHandler Class so any errors that occur are contained within that class and can be handled properly.

NF-2: This is partially out of the control of the system being designed as performance is related to what kind of machine the user runs it on. However, the system should implement efficient algorithms to ensure it is as optimized as possible.

NF-3: Whenever User input is expected it will be contained with in a try and catch block with meaningful exceptions that way the User won't be penalized for inputting incorrect data.

NF-4: The system follows separation of concerns as much as possible, where each class has its own designated role and doesn't stray from it. For example, FileHandler (purely handles file processing), RecommendationEngine (purely handle constructing recommendations for the User), and so on.

NF-5: This will be done upon implementation. During implementation, meaningful comments will be produced and Java best practices will be followed.

Future developments: The system could be expanded to have a GUI instead of the current CLI approach, and can be added to include multiple User functionality. The classes are already in place to handle this except an additional attribute like a uniqueID will be required for the User class to differentiate between Users.

7 Recommendation Algorithm Pseudocode

```
// this returns a score for how many genres match between highly rated reviewed movies and the movie given
Private int getGenreScore(Movie movie){
        Reviews = user.getHighlyRatedMovies(); //arraylist of movies for only good reviews
        int score;
        For(review : reviews){
                for(i = 0; i < movie.getGenres().length(); i++){
                        for(j = 0; j < review.getGenres().length(); j++){
                                if(movie.getGenres().get(i) == review.getGenres().get(j)){
                                        Score++; //increments score if there is a matching genre
                                }
                        }
```

```
                }
        }
        Return score;
}

//checks if user has reviews, if no reviews exist returns a score based on average rating and
number of rankings, if reviews exist returns a score based on average ratings, number of ratings
and a genre matching score
Private float getScore(Movie movie){
        if( user.getReviewedMovies() == null){
                return movie.getAvgRating() * movie.getNumRatings();
        else{
                return movie.getAvgRating() *
                movie.getNumRatings() *
                getGenreScore(movie);
        }
}



//creates a temporary list of movies
ArrayList<movies> tempMovies = new ArrayList<>();
ArrayList<Movie> reviews = User.getReviewedMovies();

for(Movie movie : movies){
        if(!reviews.contains(movie)){
        tempMovies.add(movie);
       }
}

//sorts movies based on their score from greatest to least and prints them for the user
tempMovies.Sort((m1, m2) -> Float.compare(getScore(m2), getScore(m1)));
for(i = 0: i < 5 : i++){
        tempMovies.get(i).printMovie();
}
return;
```