Module Code & Module Title

CS5003NI Data Structures and Specialist Programming

30% Individual Coursework 1

Submission: Final Milestone.

Academic Semester: AY 2025/2026

Credit: 30 credit year long module

Student Name: Utsarga Chaulagain

Project Title: Movie Vault: Java Desktop Movie Management System

London Met ID: 24046662

College ID: NP01AI4A240070

Assignment Due Date: 16/01/2026

Assignment Submission Date: 16/01/2026

Submitted To: Subarna Sapkota

GitHub Link	**https://github.com/25usy/JAVACOURSEWORK.git**

I confirm that I understand my coursework needs to be submitted online via MST Classroom under the relevant module page before the deadline for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded.

Final CourseWork 1 Submission

Islington College,Nepal

Document Details

Submission ID
trn:oid:::3618:126309908

Submission Date
Jan 16, 2026, 3:40 PM GMT+5:45

Download Date
Jan 16, 2026, 3:43 PM GMT+5:45

File Name
24046662 Utsarga Chaulagain.docx

File Size
10.1 MB

73 Pages

4,685 Words

24,848 Characters

turnitin Page 2 of 79 - Integrity Overview Submission ID trn:oid:::3618:126309908

13% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Small Matches (less than 8 words)

Exclusions

- 4 Excluded Sources
- 7 Excluded Matches

Match Groups

- 56 Not Cited or Quoted 11%
  Matches with neither in-text citation nor quotation marks
- 6 Missing Quotations 2%
  Matches that are still very similar to source material
- 0 Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

5% Internet sources
1% Publications
12% Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

1.1. Project Title

Movie Vault – Java Desktop Movie Management System

1.2. Introduction

Movie Vault is a java based application that is used for managing and tracking movies that the users have watched in an effective manner. The system allows the admins to perform crucial operation such as Creating, Updating, Deleting, and Reading the ArrayList which contains Movie objects.

The application is designed in a MVC (Model-View-Controller) architectural pattern to ensure good management of code is in place and is understandable for developers for maintainability. The project also demonstrates use of ArrayList, Queue, and Stack.

1.3. Purpose of the System

The main purpose of movie vault is so that the users can manage their watchlists in a desktop friendly interface instead of doing it manually. This system ensures accurate storage, easy retrieval and controlled modification of movie which enforces basic input validation.

1.4. Target Audience

The primary users of the system are:

- Movie Enthusiasts
- Students and Young Professionals
- Casual Streamers
- Content Creators and Movie Reviewers
- Collectors and Archivists

1.5. Aims and Objectives

The main aims and objectives of this project are:

- Develop a java application using SWING
- To implement CRUD operations
- To apply the MVC architectural pattern correctly

- To utilize java data structures such as queue, array lists and stack
- Perform basic input validations
- Create a clean and user friendly graphical interface

2. Problem Statement

Managing movie manually is through unstructured systems can lead to many anomalies while updating the records and leads to data duplication, inconsistency. Without a proper management system the administrators may struggle to track recently added movies, modify existing information or remove some outdated entries.

The MovieVault system addresses this problem by providing a structured desktop based solution that will enable the administrator to manage the data reliably by implementing CRUD functionality and enforcing the validation rules, the system reduces errors and improves data handling efficiency.

3. Wireframe Design

3.1. Admin dashboard:

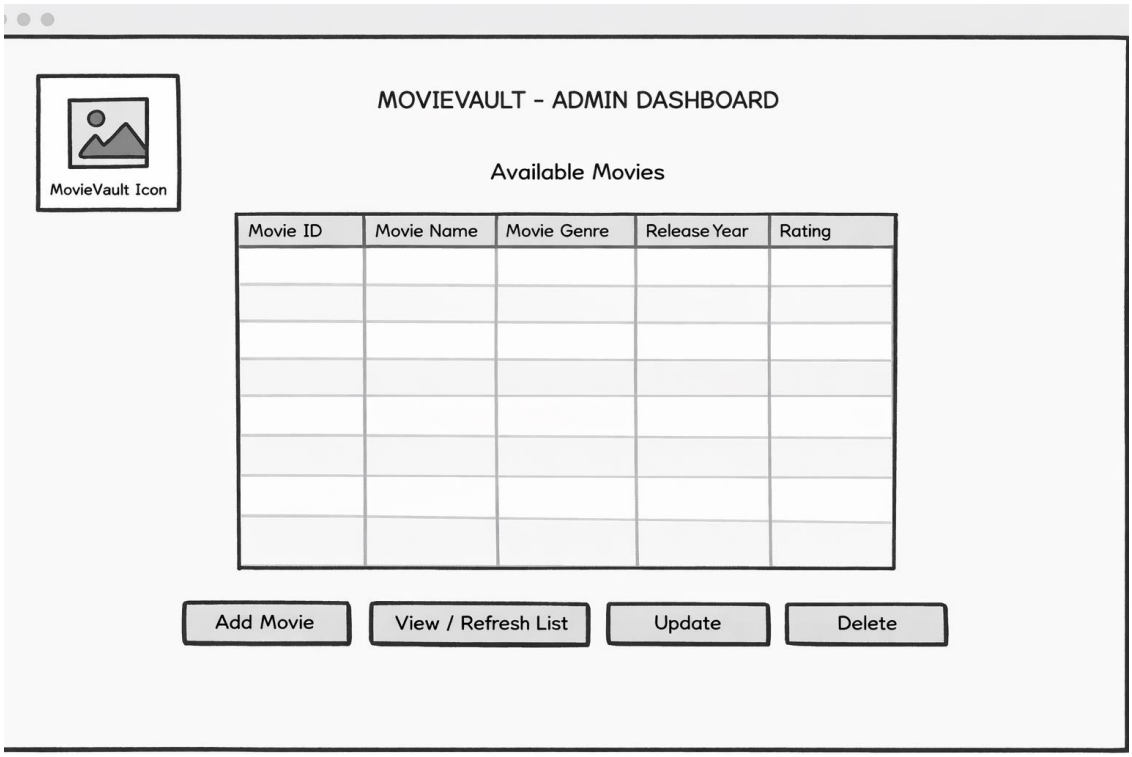


Figure 1: WireFrame of AdminDashboard

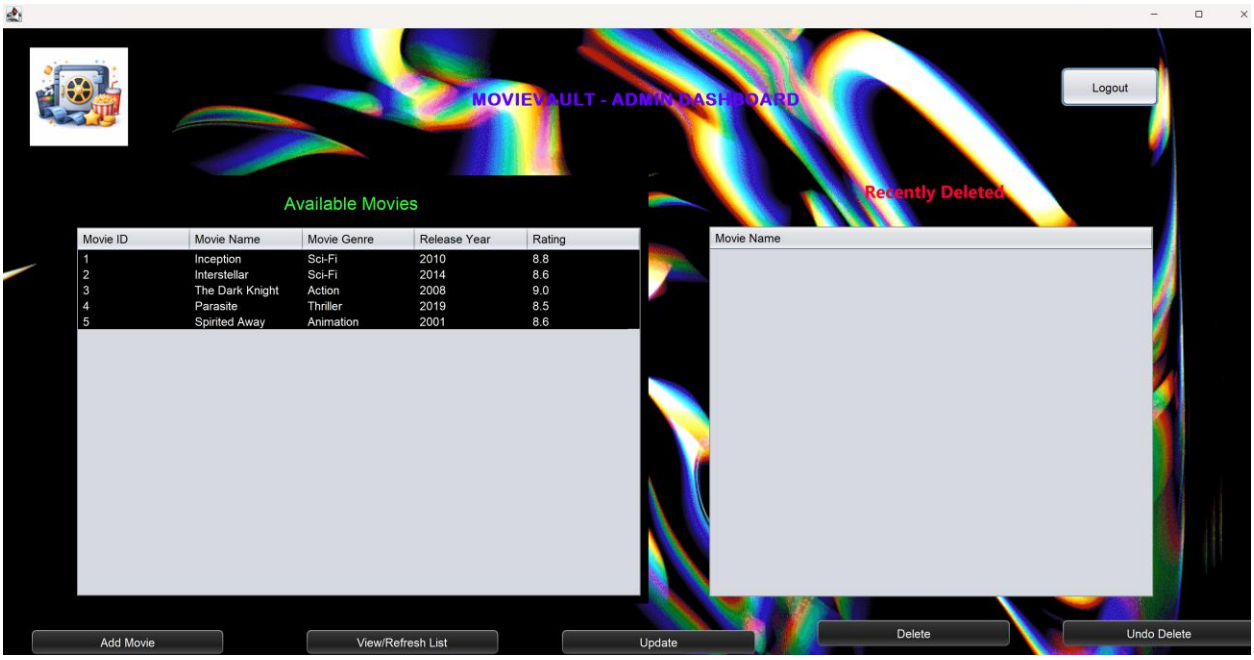


Figure 2: Admin Dashboard UI

3.2. Add Movie Form:

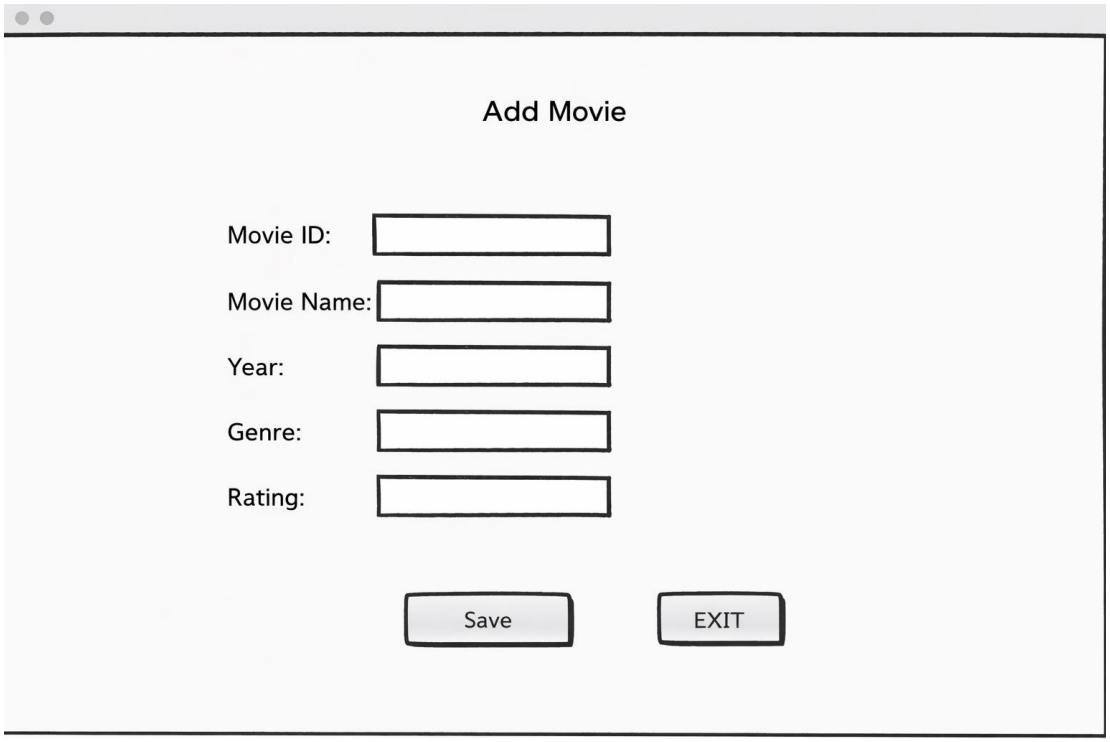


Figure 3: WireFrame of AddMovie

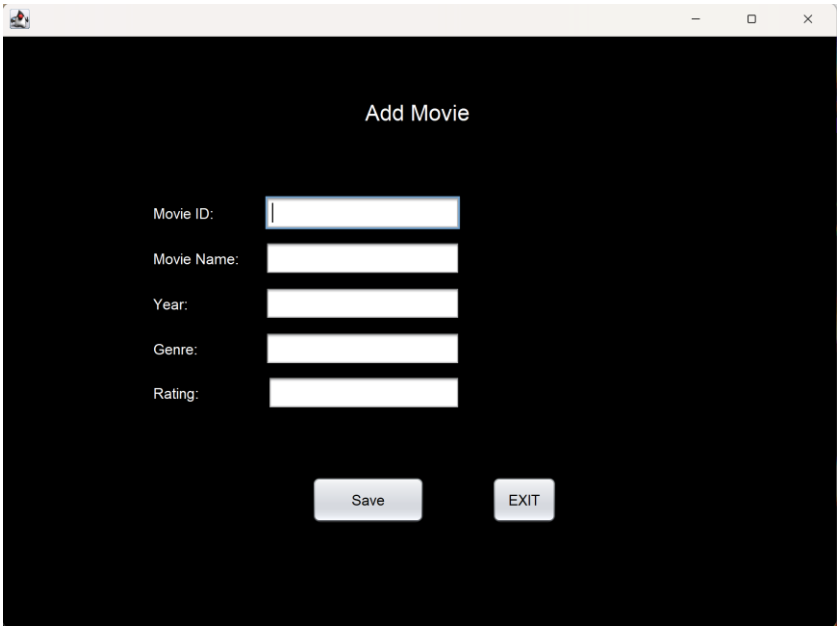


Figure 4: Add Movie UI

3.3. Update Movie Form:

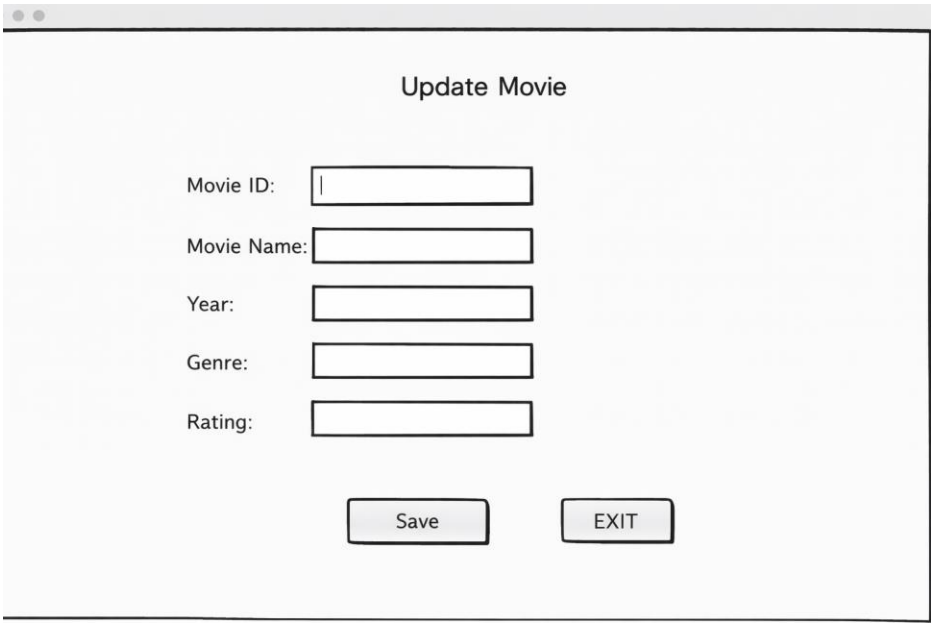


Figure 5: WireFrame UpdateMovie

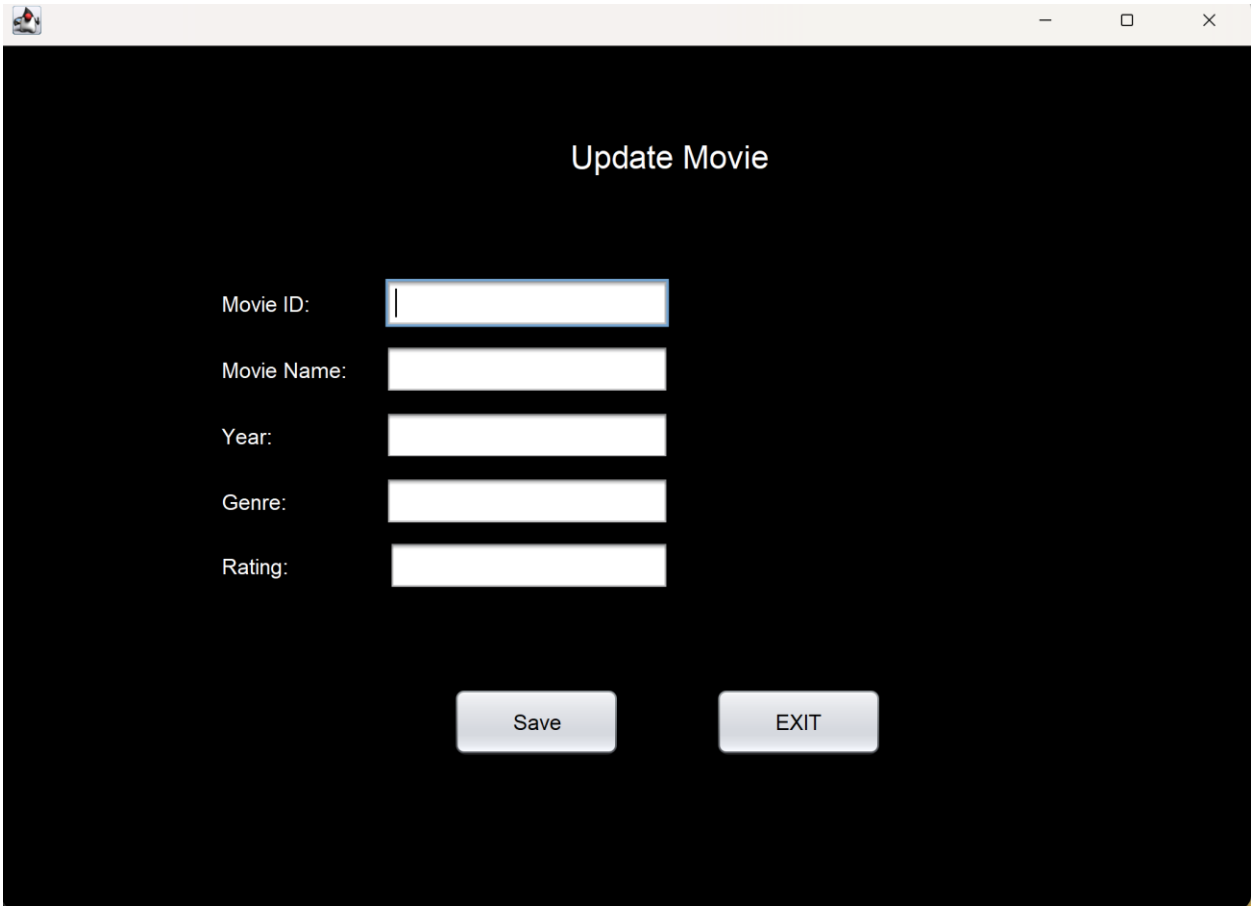


Figure 6: Update Movie UI

3.4. Login/Register Form:

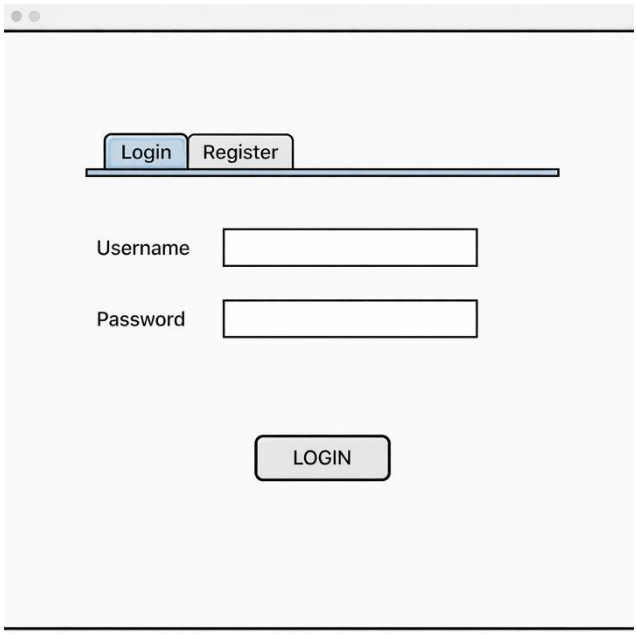


Figure 7: WireFrame of Login/Register

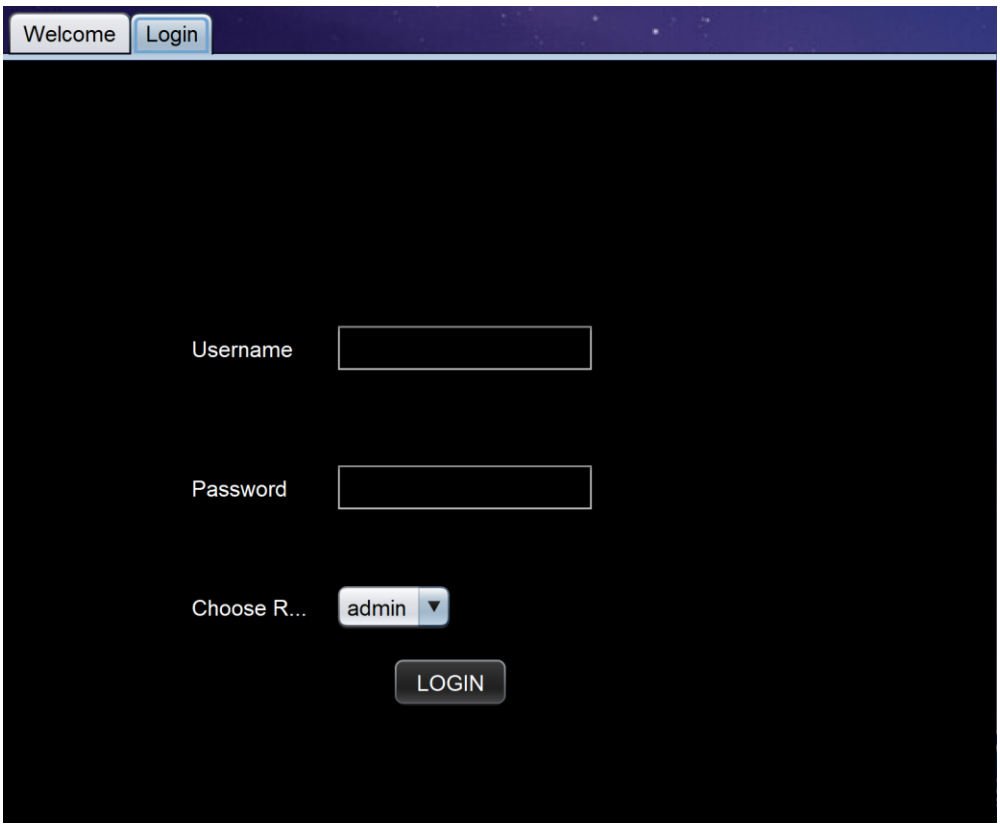


Figure 8: :ogin UI

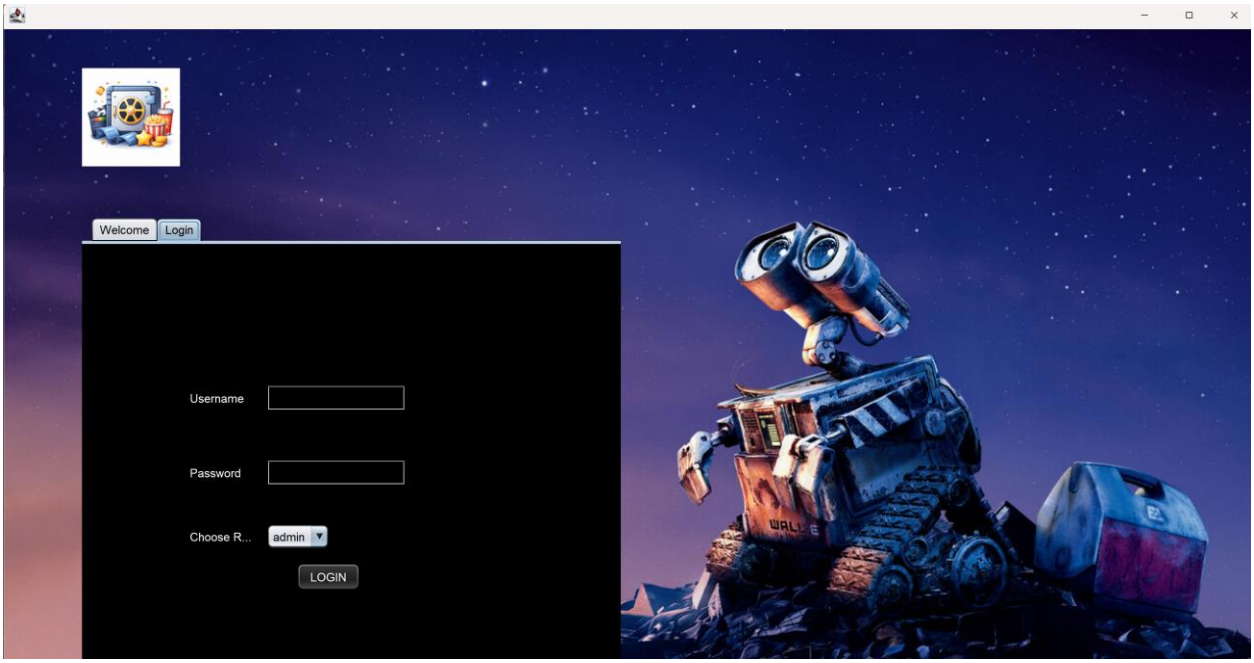


Figure 9: Login/Welcome Page UI

3.5. User Dashboard

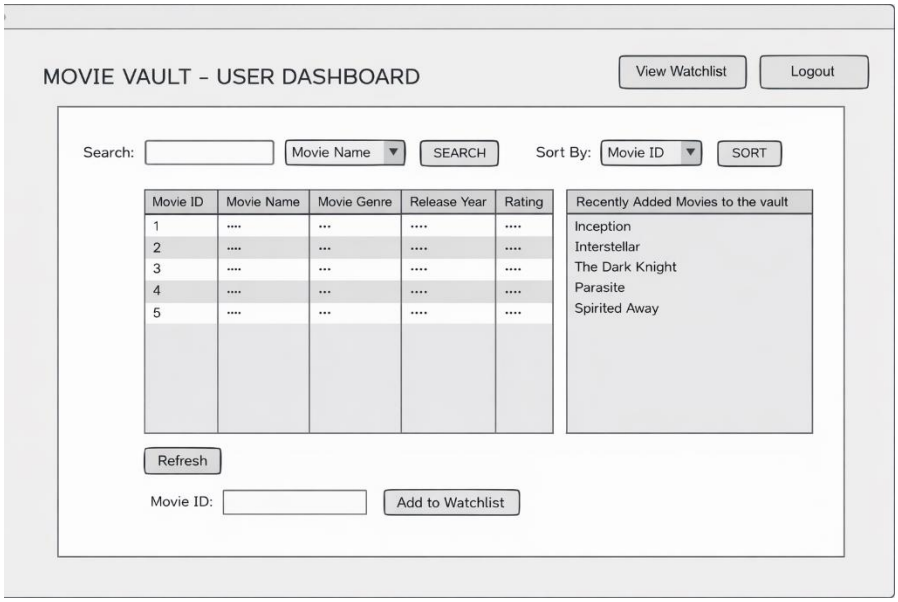


Figure 10: User Dashboard wireframe

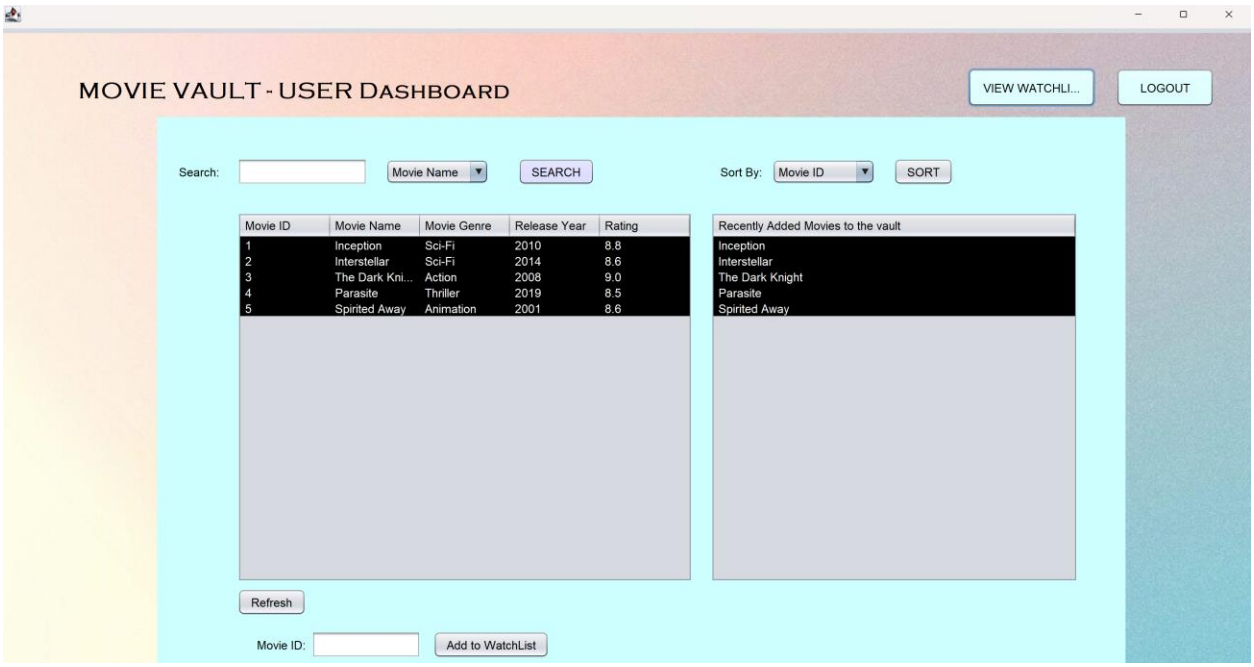


Figure 11: User Dashboard UI

3.6. Watchlist Page / UserHome

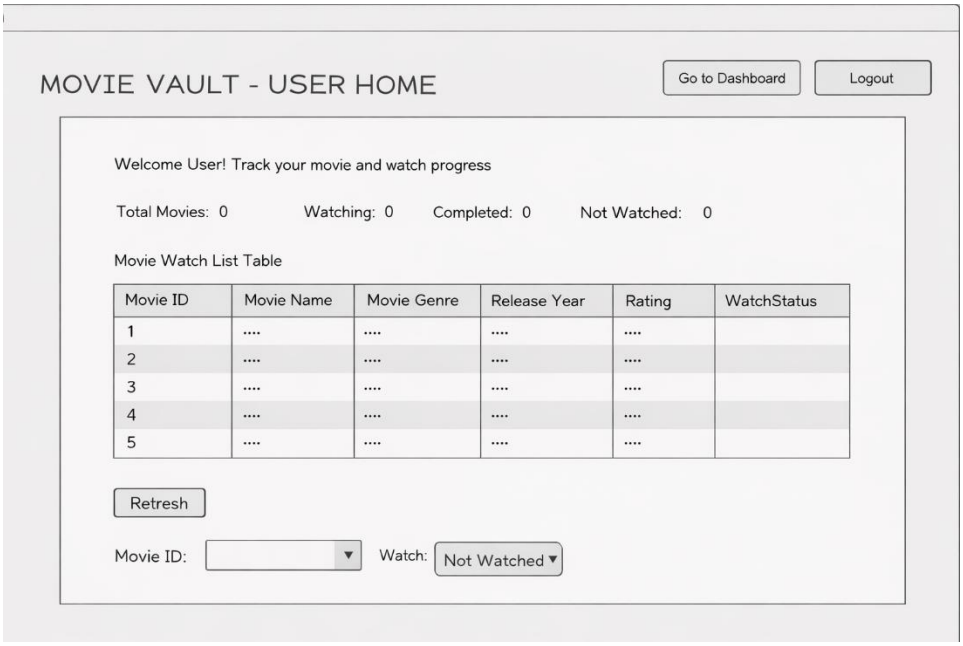


Figure 12: User home wireframe

Figure 13: User home UI

4. MVC architecture and ANT setup

The movie vault system follows MVC pattern. The design approach sperates the application intro three interconnected parts, which improves code readability, understandability, maintainability and scalability.

4.1. MVC explaination

Model: Model represents the data and business logic of the system. It is responsible for storing movie records and performing some operations on the data.

What this component does:

- Stores movie attributes such as ID, name genre and rating
- Manage the movie collection using Java data structures
- Perform CRUD operations (all the methods defined here)

Classes present:

- **Movie**
- **MovieManager**

Views: The view component holds the GUI of the app. It displays information to the user and collects user input.

What this component does:

- Display forms for adding movie
- Shows movie list in table format
- Provides buttons for CRUD operations
- Display validation and confirmation message

Classes present:

- AdminDashboard
- AddMovie
- UpdateMovie
- LoginRegister

Controller: The controller acts as a bridge between Model and the Views component. It receives user actions from the View and processes them by interacting with the Model.

What it does:

- Validates User Requests
- Call appropriate methods in the Model
- Return results back to the View

KeyClass:

- MovieController

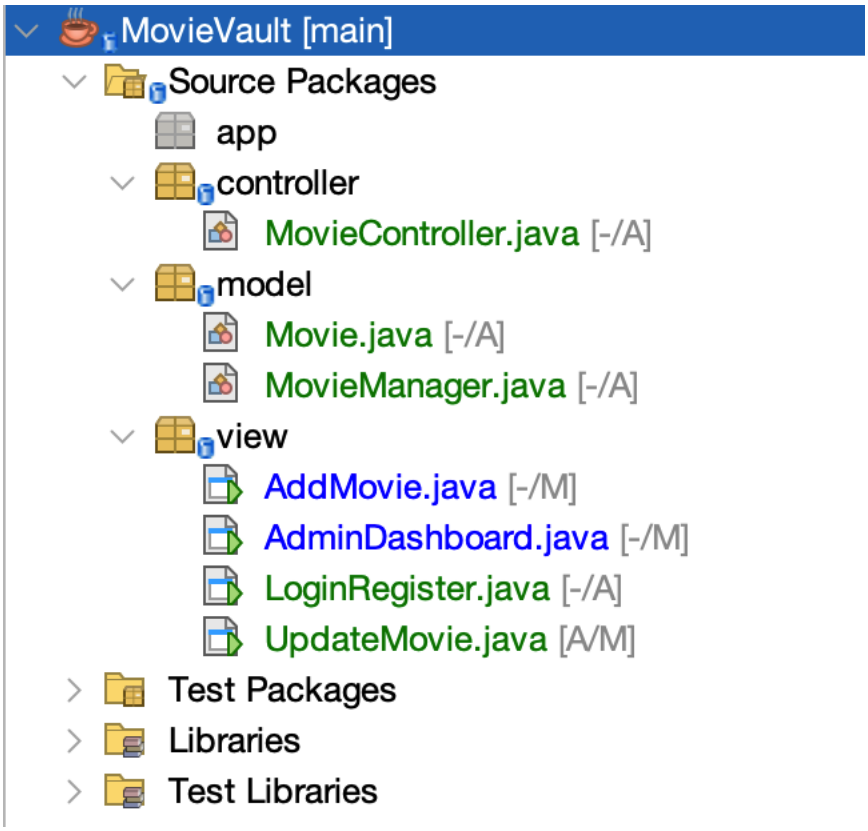


Figure 14: MovieVault Directory Setup

4.2. MVC ANT Project Setup

4.2.3. Creating a Java Project

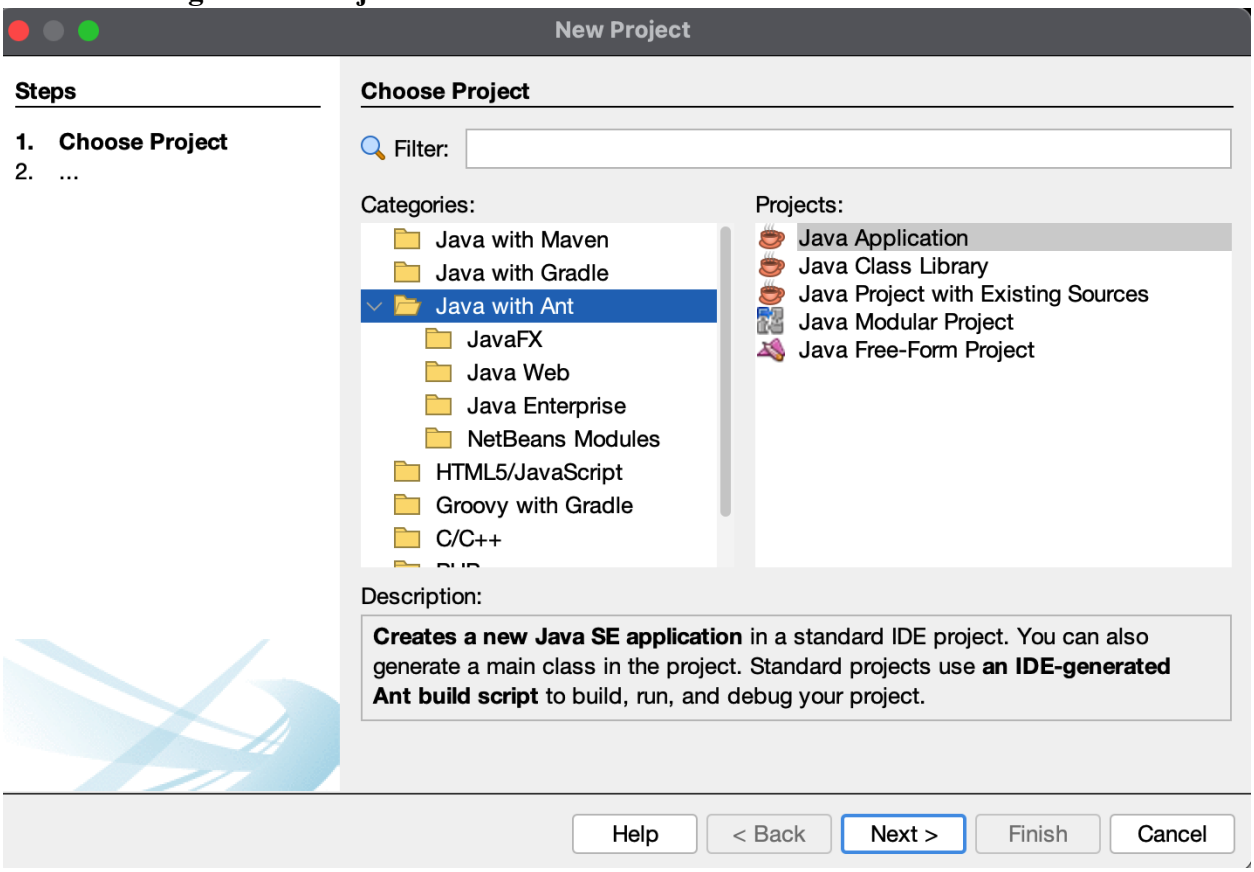


Figure 15: ANT setup for Creating a Java Project

4.2.2. Assigning Name and Directory of the Project

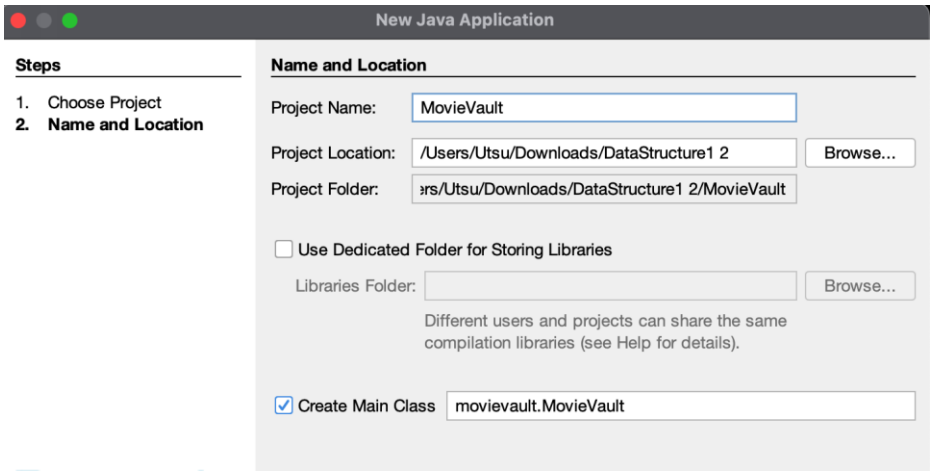


Figure 16: ANT setup for Assigning Name and Driectory of the Project

4.2.3. Assigning Name and Directory of the Project

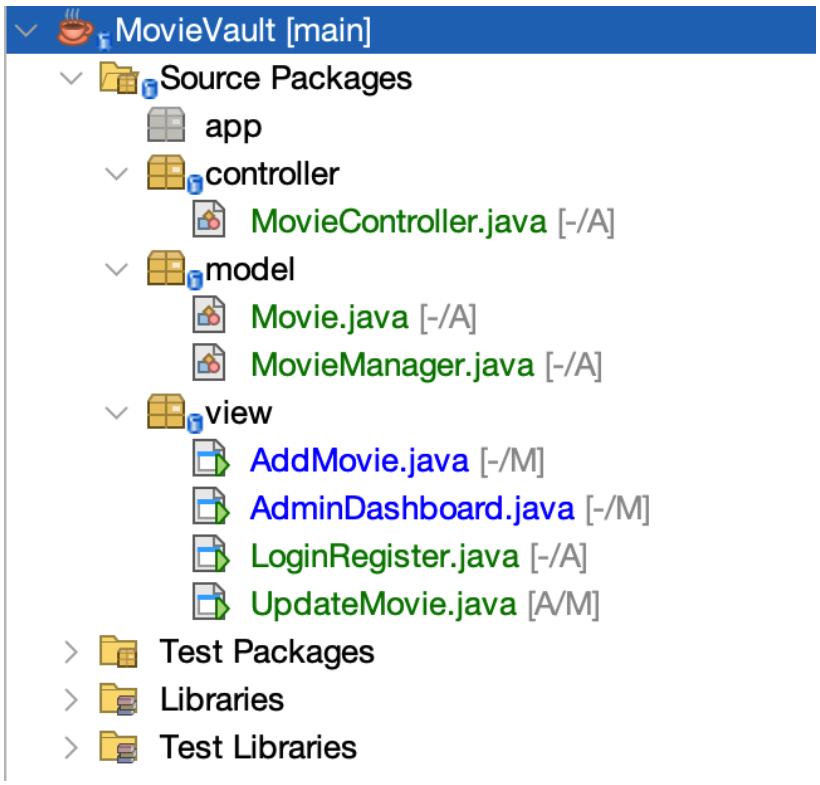


Figure 17: ANT setup for MVC architecture

5. Java Classes and Method Explanation

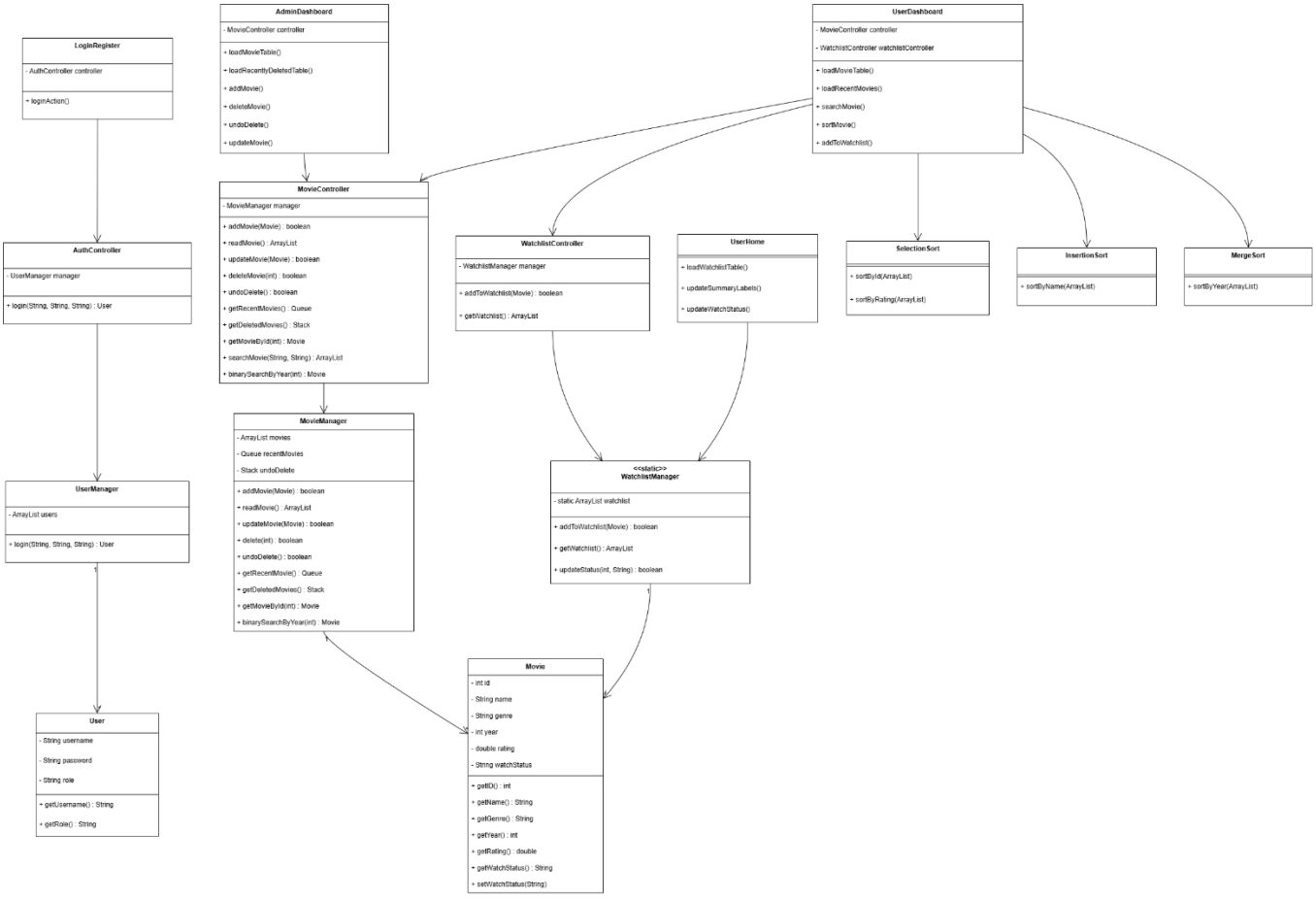


Figure 18: Class Diagram

Class Description

1. **Name of Class: Movie Class**

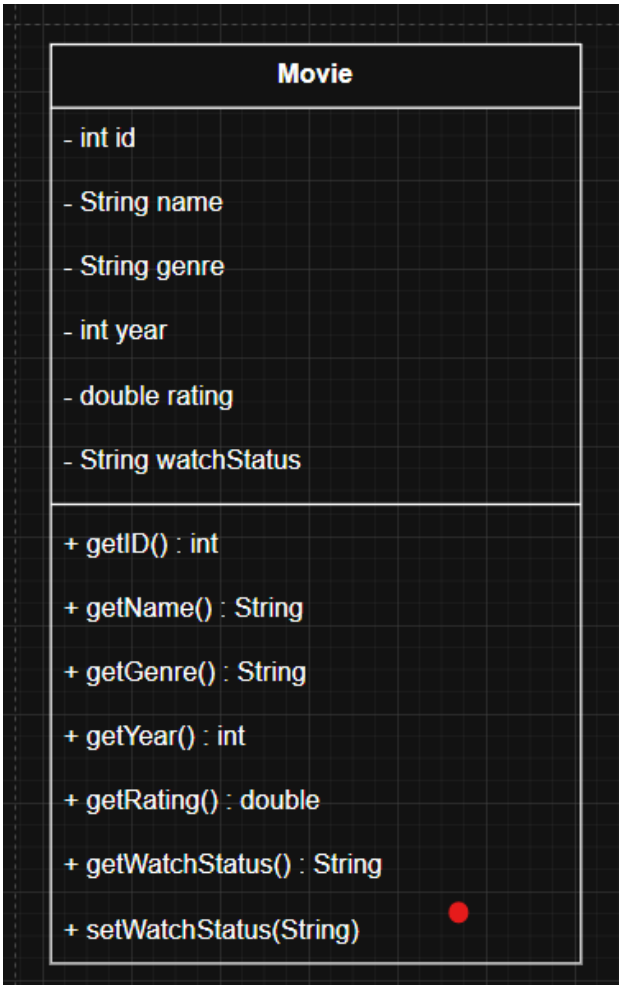


Figure 19: Movie Class

Purpose: Represents a single movie entity in our system. It is the core data model that will store all our records for a movie. This class follows the encapsulation principle by keeping attributes private and accessing them through getter and setter methods.

Key Attributes and Importance:

- **ID(int):** Uniquely identifies each movie. This attribute is important because it is used to prevent duplicate entries and perform update and delete operations.
- **name(String):** Stores the movie name. This is essential for identifying and displaying movies to users.
- **genre(String):** Represents the category of the movie. Useful for classification and filtering.

- **year(Int):** Stores the release year of the movie. Helps provide additional contextual information.
- **rating(Int):** Stores the movie rating. This allows users or admins to evaluate the quality of movie.

Method Logic:

- Getter method return the corresponding attribute values. Example getId(), getName().
- Setter methods update attribute values safely without exposing variables directly. Example setId(), setName()

Code Snippet of Methods:

```
//all the getter functions
public int getID()
{
    return ID;
}
public String getName()
{
    return name;
}
  public String getGenre()
{
    return genre;
}

public int getYear()
{
    return releaseYear;
}

public double getRating()
{
    return rating;
}
```

```
// all the setter functions
public void setName(String nameInput)
{
    this.name = nameInput;
}

public void setGenre(String genreInput)
{
    this.genre = genreInput;
}

public void setYear(int yearInput)
{
    this.releaseYear = yearInput;
}

public void setRating(double ratingInput)
{
    this.rating = ratingInput;
}
```

2. Name of Class: MovieManager

Purpose: Perform CRUD operations.

Figure 20: Movie Manager Class

Key Attributes and Importance:

- **movies (ArrayList<Movie>)**
  Stores all movies in the system. ArrayList allows dynamic resizing and indexed access.
- **recentMovies (Queue<Movie>)**
  Stores the five most recently added movies using FIFO logic.
- **undoDelete (Stack<Movie>)**
  Stores deleted movies temporarily using LIFO logic, enabling undo functionality in future milestones.

Short Explanation of Logic Methods:

addMovie(): Checks for duplicates before adding a movie. Adds the movie to both the main list and the recent movie queue.

```
public boolean addMovie(Movie movieInput) //here movieInput is a user input which contains (id,name,genre,yearReleased,rating)

{
    for(int i=0; i<= movies.size()-1; i++) //looping through array list here.

    {
        /*checking if the ids are same for movies(storage) and movieInput
        also the getID is a getter function present in Movie class
         */
        if(movies.get(i).getID() == movieInput.getID())
        {
            return false;
        }
    }
    movies.add(movieInput);

    recentMovies.add(movieInput); //usage of Queue FIFO for storing 5 recently added movies
    if (recentMovies.size() > 5)
    {
      recentMovies.remove();
    }
    return true;
}
```

Figure 21: addmovie()

readMovie(): Returns list of all movies.

```
//Used to Display the Movie in our storage
public ArrayList<Movie> readMovie()
{
    return movies;
}
```

Figure 22: readMovie()

updateMovie(): Searches for a movie by ID and replaces it with updated details.

```
//Checks user input and updates the movie also sends a confirmation
public boolean updateMovie(Movie movieInputU)
{
    for(int i=0; i<= movies.size()-1; i++)
    {
        if(movies.get(i).getID() == movieInputU.getID())
        {
            movies.set(i, movieInputU );
            return true;
        }

    }
    return false;
}
```

Figure 23:updateMovie()

delete(): Removes a movie by ID and pushes it onto stack for undo which is implemented in the future milestones to come.

```
//delete method for removing the movie from the array list
public boolean delete(int id)
{
    for(int i=0; i<= movies.size()-1; i++)
    {
        if(movies.get(i).getID()== id)
        {
            undoDelete.push(movies.get(i)); //usage of stack LIFO
            movies.remove(i);
            return true;
        }

    }
    return false;
}
```

Figure 24: deleteMovie()

getRecentMovie(): It returns the queue of recently added movies., this method is used to show the latest movies in FIFO order.

```
public Queue<Movie> getRecentMovie()
{
    return recentMovies;
}
```

Figure 25: getRecentMovie()

undoDelete(): Restores the recently deleted movie from the stack back into the movie list. If the stack is empty no action is performed.

```
// Restores the most recently deleted movie using Stack (LIFO)
public boolean undoDelete()
 {
    if (undoDelete.isEmpty())
     {
       return false; // nothing to undo
     }

Movie restoredMovie = undoDelete.pop();
movies.add(restoredMovie);
return true;
 }
```

Figure 26: undoDelete()

getDeletedMovies(): Returns the stack containing deleted movies. Used by the admin dashboard to show the recently deleted movies in LIFO order.

```
/*
Method to return deleted movies stack
Used to display recently deleted movies in admin dashboard
*/
public Stack<Movie> getDeletedMovies() {
return undoDelete;
}
```

Figure 27: getDeletedMovies()

binarySearchByYear(): searches for a movie based on its release year using binary search. Before searching the movie is sorted by year to meet the binary search requirements. It improves search efficiency compared to linear search.

```
    public Movie binarySearchByYear(int year) {

    // Creating a copy so original order is not affected
    ArrayList<Movie> list = new ArrayList<>(movies);

    // Sorting the list by year before using binary search
    for (int i = 0; i < list.size() - 1; i++) {
        for (int j = i + 1; j < list.size(); j++) {
            if (list.get(i).getYear() > list.get(j).getYear()) {
                Movie temp = list.get(i);
                list.set(i, list.get(j));
                list.set(j, temp);
            }
        }
    }

    int low = 0;
    int high = list.size() - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midYear = list.get(mid).getYear();

        if (midYear == year) {
            return list.get(mid);
        }
        else if (midYear < year) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }

    return null;
}
```

Figure 28: binarySearchByYear()

3. Name of Class: MovieController

Purpose: Acts as a bridge between the GUI and MovieManager. It follows the MVC architecture.

Figure 29: MovieController class

Key Attributes and Importance:

- **movieManager (MovieManager)**
  Handles all business logic and data operations. The controller relies on this object to perform all movie-related actions.

Short Explanation of Logic Methods:

- Each method in this class calls the corresponding method in MovieManager.
- This ensures **separation of concerns** and cleaner code.

Code Snippet of few Methods and object creation:

```
private static MovieManager manager = new MovieManager();

public boolean addMovie(Movie m) {
    return manager.addMovie(m);
}

public ArrayList<Movie> readMovie() {
    return manager.readMovie();
}

public boolean deleteMovie(int id) {
    return manager.delete(id);
}

public boolean updateMovie(Movie movie) {
    return manager.updateMovie(movie);
}

public Movie getMovieById(int id) {
    for (Movie m : manager.readMovie()) {
        if (m.getID() == id) {
            return m;
        }
    }
    return null;
}

public Queue<Movie> getRecentMovies() {
    return manager.getRecentMovie();
}
```

Figure 30: getter and setter methods

4. Name of Class: User

Purpose: Responsible for storing and managing user related data such as login credentials and roles. This class supports the authentication process of the system by differentiating between the user and admin.

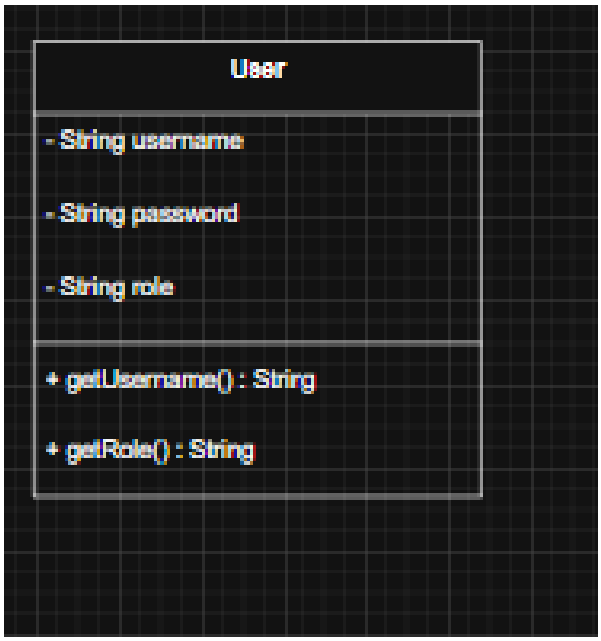


Figure 31: User Class

Key Attributes and Importance:

- String Username: Stores the unique username.
- String Password: Holds the password associated with the user account.
- String role: Represents the role. Either user or admin.

Short Explanation of Logic Methods:

1. getUsername(): Returns the username of the user.

```
public String getUsername() {
    return username;
}
```

Figure 32: getUsername()

2. getPassword(): Returns to password of the user.

```
public String getPassword() {
    return password;
}
```

Figure 33: getPassword()

3. getRole(): Returns the role of the user.

```
public String getRole() {
    return role;
}
```

Figure 34:getRole()

5. Name of Class: UserManager

Purpose: Acts as a central storage for user data. Handles tasks such as user login, authentication and validation.

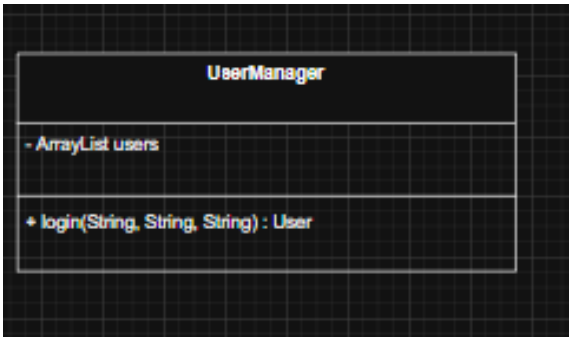


Figure 35: UserManager Class

Key Attributes and Importance:

- ArrayList<User> users: It allows storing multiple users and supports operations such as searching and auth. Arraylist data structure is chosen for simplicity and efficiency when validating the login credentials.

Short Explanation of Logic Methods:

1. UserManager(): This constructor initializes user list and includes our hard coded predefined values which is used for system demonstration process.

```
public UserManager() {
    users = new ArrayList<>();

    // Hardcoded users (ACCEPTED for coursework)
    users.add(new User("admin", "admin123", "admin"));
    users.add(new User("user", "user123", "user"));
}
```

Figure 36: UserManager() constructor

2. UserLogin(): Validated the login credentials which is provided by the user. The method will check the user list to find a matching username and password, also check the role type of the user. If valid credentials are found, the corresponding user object will be returned else if no match is found then the method will return null which indicates a failed login attempt.

```
public User validateLogin(String username, String password, String role) {

    for (User u : users) {
        if (u.getUsername().equals(username)
            && u.getPassword().equals(password)
            && u.getRole().equals(role)) {

            return u;
        }
    }
    return null;
}
```

Figure 37: validateLogin()

6. Name of Class: WatchlistManager

Purpose: Responsible for managing the user's watchlist in the movie vault system. It handles operation such as adding the movies to the watchlist, updating the status of the movie in the watchlist and so on.

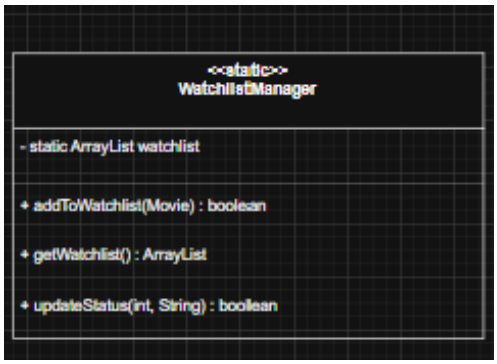


Figure 38: WatchlistManager

Key Attributes and Importance:

- ArrayList<Movie> watchlist: Stores the list of movies added to the watchlist. Each movie object represents a movie added to watchlist by the user along with the chosen watch status.

Short Explanation of Logic Methods:

1. addMovie() Adds a movie to the watch list. The method also checks for duplicates to prevent addition of the same movie multiple times.

```
public static boolean addMovie(Movie movie) {

    // prevent duplicates
    for (int i = 0; i < watchlist.size(); i++) {
        if (watchlist.get(i).getID() == movie.getID()) {
            return false;
        }
    }

    watchlist.add(movie);
    return true;
}
```

Figure 39: addMovie()

2. getWatchList(): Returns the complete list of movies in the watchlist.

```
public static ArrayList<Mcvie> getWatchlist() {
    return watchlist;
}
```

Figure 40: getWatchlist()

3. updateStatus(): Updates the watch status of specific movie in the watchlist. The method will search for the movies using their corresponding ID and will update its status to values such as Not Watched, Watching or Completed. Returns true if the status is updated and false if the movie is not found in the watchlist.

```
public static boolean updateStatus(int movieId, String status) {

    for (int i = 0; i < watchlist.size(); i++) {
        if (watchlist.get(i).getID() == movieId) {
            watchlist.get(i).setWatchStatus(status);
            return true;
        }
    }
    return false;
}
```

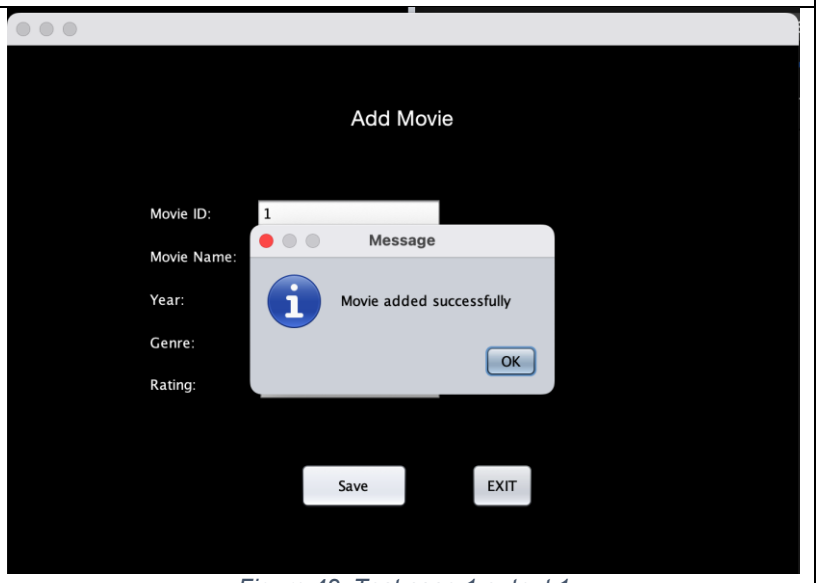
Figure 41: updateStatus()

6. Testing Evidence

6.1. Functional Testing

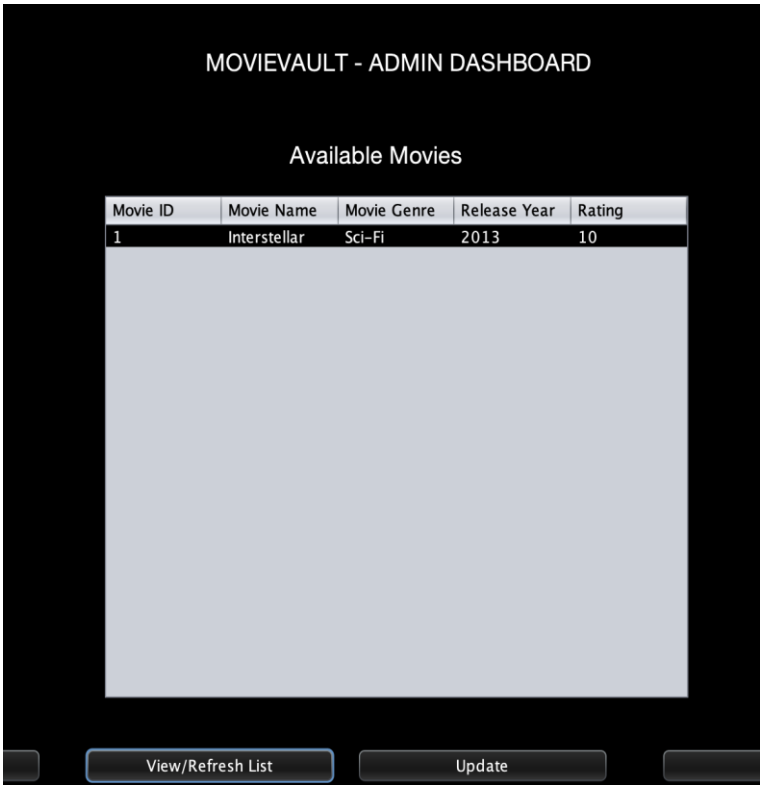
Test Case 1: Add movie

Table 1: Test Case 1

Test Case ID	1
Test Description	Allows the admin to add a new movie with valid details into the system.
Action	Admin enters valid Movie ID, Movie Name, Year, Genre, and Rating, then clicks the **Save** button.
Expected Output	Movie is successfully added and a confirmation message is displayed.
Actual Output	 *Figure 42: Test case 1 output 1* Movie is added successfully and message is displayed.
Result	Test Passed

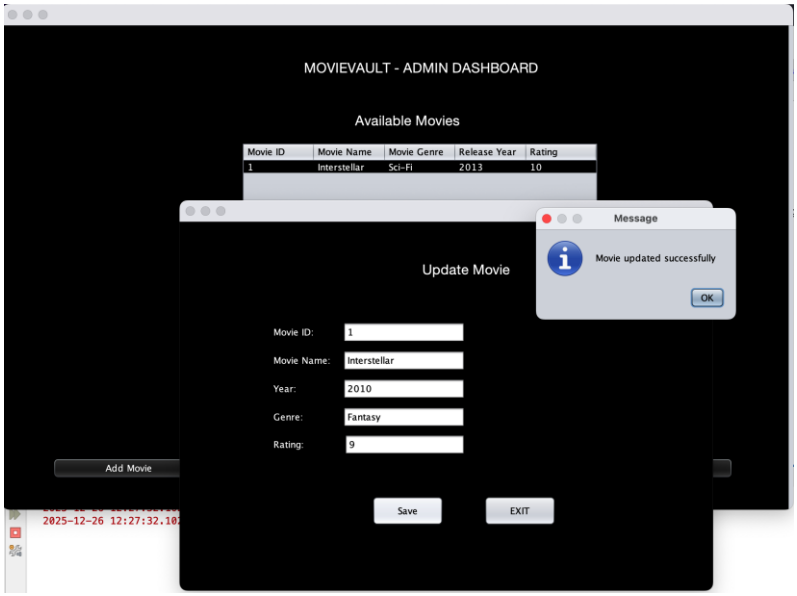
Test Case 2: View / Refresh Movie List

Table 2: Test case 2

Test Case ID	2
Test Description	Displays all stored movies in a table format.
Action	Admin clicks the **View/Refresh List** button.
Expected Output	Table updates and shows all movies with ID, Name, Genre, Year, and Rating.
Actual Output	Movie table refreshes correctly and displays updated data.  *Figure 43: Test case 2*
Result	Test Passed

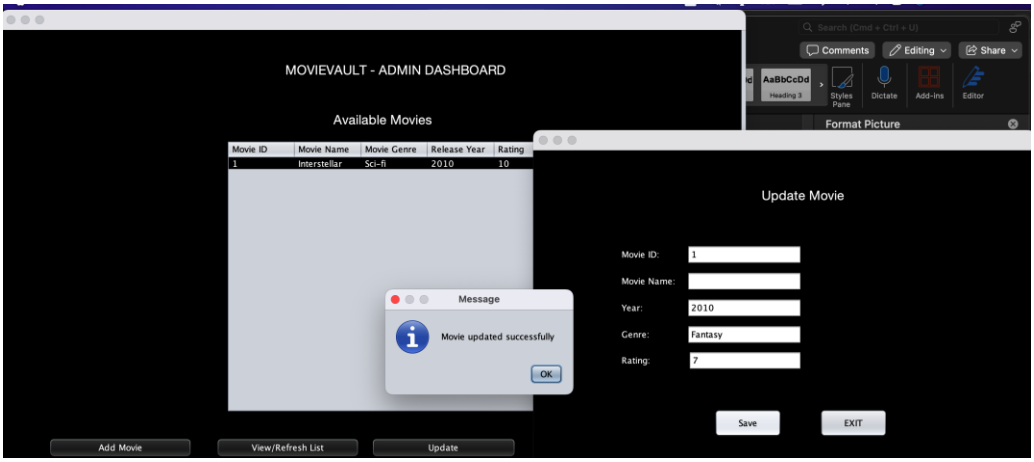
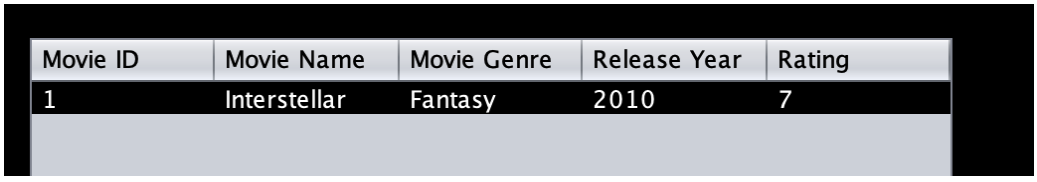
Test Case 3: Update Movie with Valid Data

Table 3: : Test case 3

Test Case ID	3
Test Description	Updates movie details when valid data is provided
Action	Enter valid Movie ID and updated values, click Save
Expected Output	Movie updated successfully
Actual Output	Movie updated successfully.  *Figure 44: Test case 3 output*
Result	Test Passed

Test Case 4: Update Movie with Empty Optional Fields

Table 4: Test case 4

Test Case ID	4
Test Description	Ensures empty fields retain previous values
Action	Enter Movie ID, leave other fields empty
Expected Output	Movie updated without changing empty fields
Actual Output	Existing values retained  *Figure 45: Test Case 4 output 1*  *Figure 46: Test Case 4 output 2*
Result	Test Passed

Test Case 5: Delete Movie with Valid Movie ID

Table 5: Test case 5

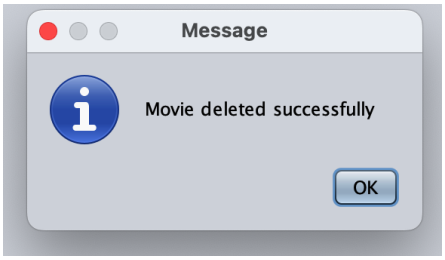
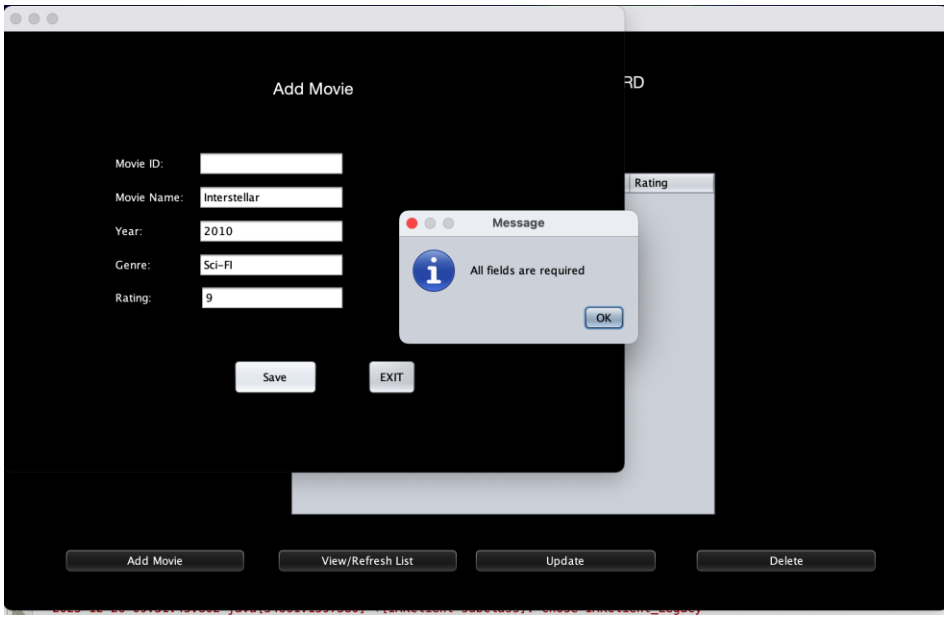
Test Case ID	5
Test Description	Deletes an existing movie using its Movie ID.
Action	Admin enters a valid Movie ID in the delete dialog and confirms.
Expected Output	Movie is removed from the system and table refreshes.
Actual Output	Movie is removed from the system and table refreshes.  *Figure 47: Test Case 5 output 1*  *Figure 48: Test case 5 output 2*

	 *Figure 49: Test case 6 output 3*
Result	Test Passed

6.2. Validation Testing

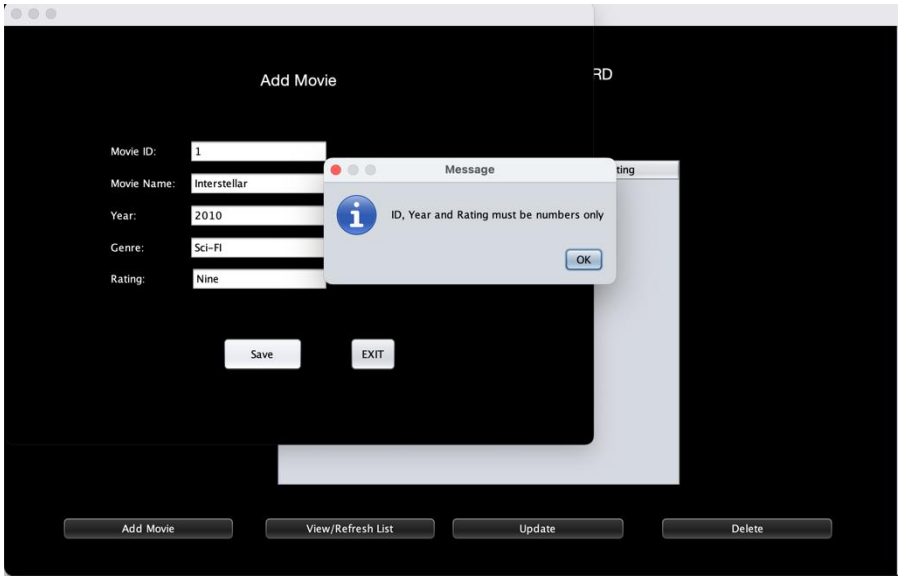
Test Case 6: Add Movie (EMPTY FIELDS)

Table 6: Test case 6

Test Case ID	6
Test Description	Prevents adding a movie when required fields are left empty.
Action	Admin clicks save leaving Movie ID empty (This works if one or more than one field is empty)
Expected Output	System will display a validation message that indicates that all the fields are required.
Actual Output	 *Figure 50: Test case 6 output 1* Validation message is shown
Result	Test Passed

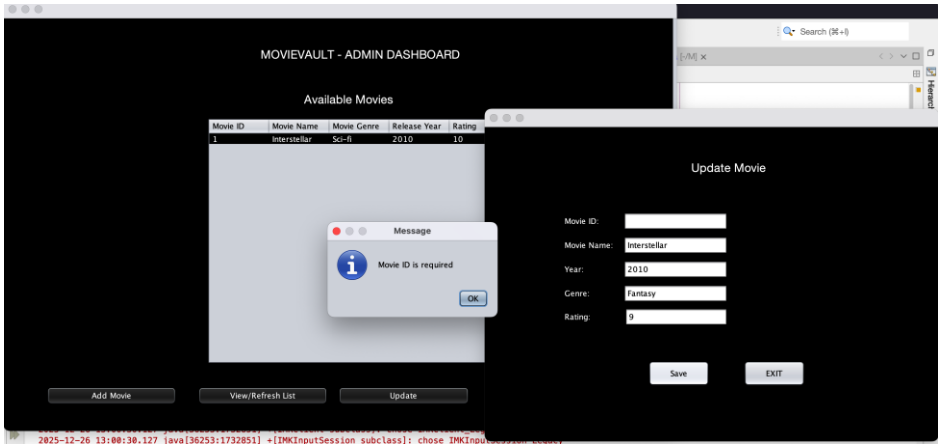
Test Case 7: Add Movie (INVALID NUMERIC INPUT)

Table 7: Test case 7

Test Case ID	7
Test Description	Ensures numeric fields accept numbers only.
Action	Admin enters alphabetic characters in Movie ID, Year, or Rating fields and clicks **Save**.
Expected Output	System displays an error message stating numeric values are required.
Actual Output	System prevents input and displays validation error.  *Figure 51: Test case 7 output 1*
Result	Test Passed

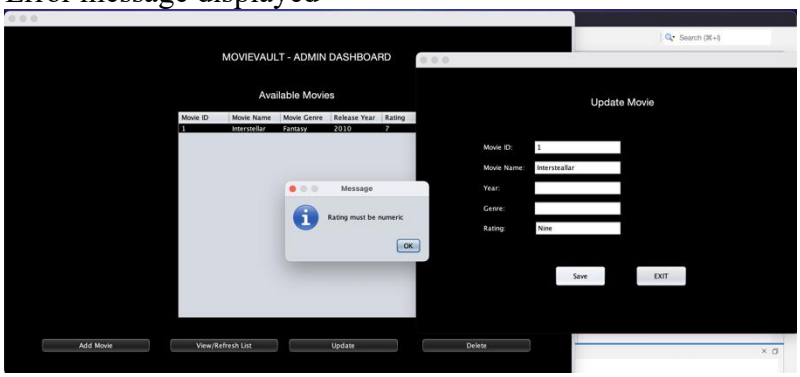
Test Case 8: Update Movie with Empty Movie ID

Table 8: Test case 8

Test Case ID	8
Test Description	Checks validation when Movie ID is empty
Action	Leave Movie ID empty and click Save
Expected Output	Error message “Movie ID is required”
Actual Output	Error message displayed  *Figure 52: Test case 8 output 1*
Result	Test Passed

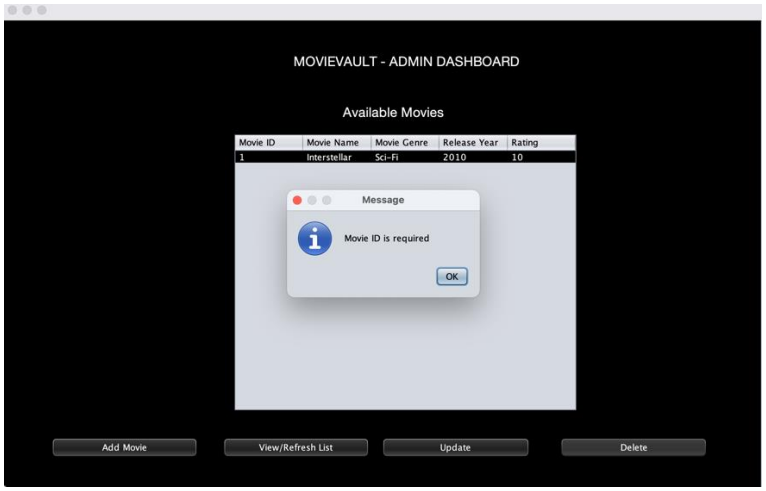
Test Case 9: Update Movie with Invalid Year or Rating

Table 9: Test Case 9

Test Case ID	9
Test Description	Validates numeric input for Year and Rating
Action	Enter non-numeric Year or Rating
Expected Output	Error message shown
Actual Output	Error message displayed  *Figure 53: Test case 9 output 1*
Result	Test Passed

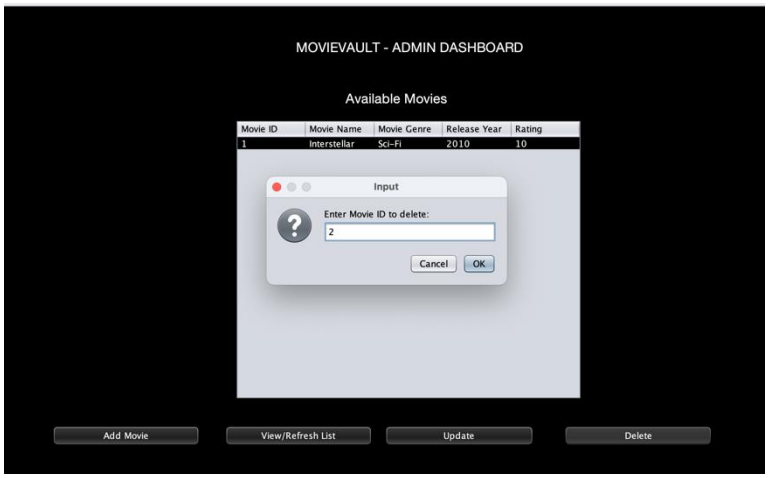
Test Case 10: Delete Movie with Valid Movie ID

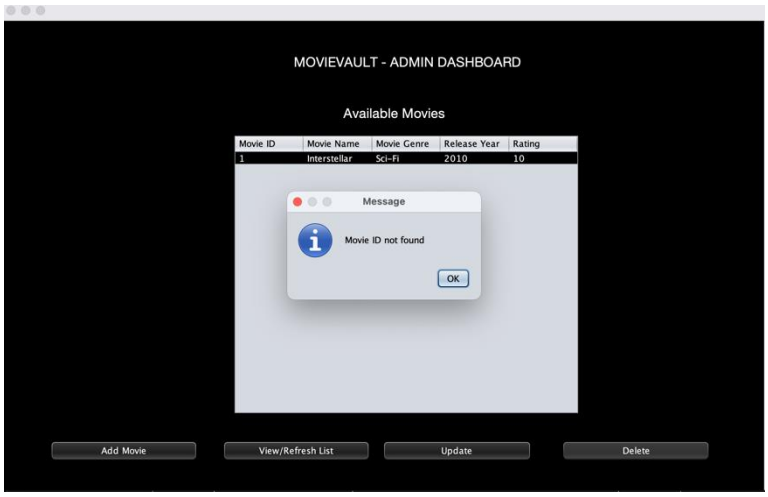
Table 10: Test case 10

Test Case ID	10
Test Description	Checks validation when Movie ID is empty
Action	Leave Movie ID empty
Expected Output	Error message shown
Actual Output	Error message displayed  *Figure 54: Test case 10 output 1*
Result	Test Passed

Test Case 11: Delete Movie with Invalid Movie ID

Table 11: Test case 11

Test Case ID	11
Test Description	Checks validation for non-numeric Movie ID
Action	Enter alphabets as Movie ID
Expected Output	Error message shown
Actual Output	Error message displayed  *Figure 55: Test case 11 output 1*

	 *Figure 56: Test case 11 output 2*
Result	Test Passed

6.3. Exception Handling Testing

Test Case 12: Update Movie with Non-Existing Movie ID

Table 12: Test case 12

Test Case ID	12
Test Description	Updates movie details when non-existing id is provided
Action	Enter invalid Movie ID and updated values, click Save
Expected Output	Movie id not found
Actual Output	Movie id not found.

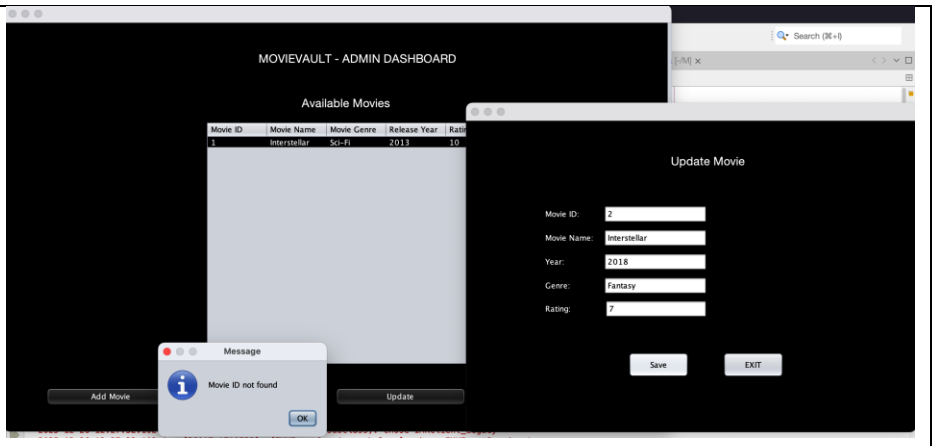
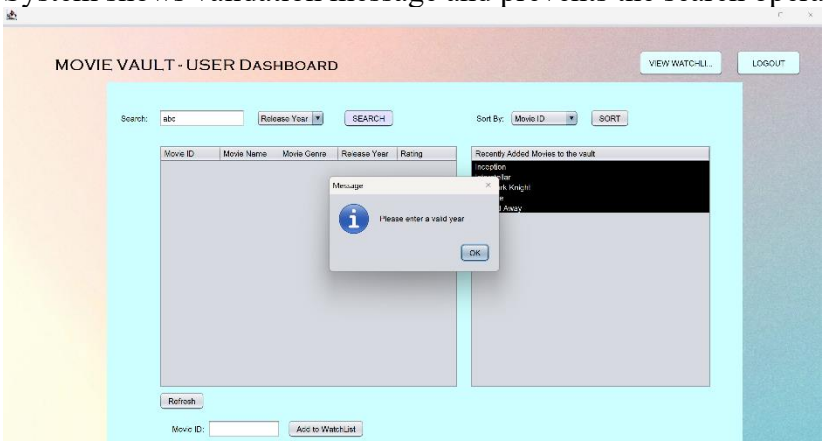
	 *Figure 57: Test case 12 output 1*
Result	Test Passed

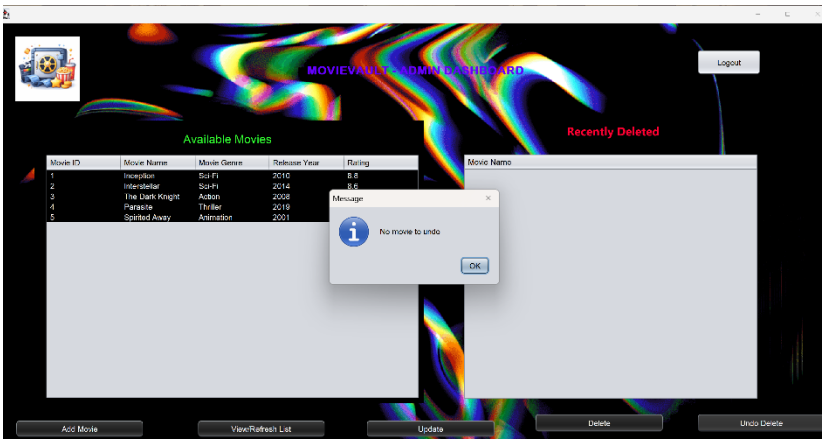
Table 13: Test case 13

Test Case 13: **Ensures the system handles invalid year input during binary search without crashing.**

Test Case ID	13
Test Description	Ensures the system handles invalid year input during binary search without crashing.
Action	User selects Release Year from the search criteria and enters alphabetic characters (e.g., abcd), then clicks the Search button.
Expected Output	System displays an error message prompting the user to enter a valid numeric year.
Actual Output	System shows validation message and prevents the search operation.  *Figure 58: Test case 13 output 1* This prevents a NumberFormatException when converting year input to integer.
Result	Test Passed

Test Case 14: **Stack Underflow (Undo delete when stack is empty)**

Table 14: Test case 14

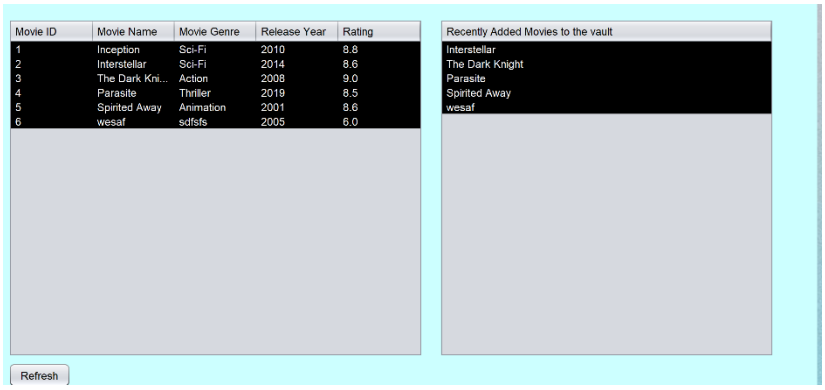
Test Case ID	14
Test Description	Ensures the system handles stack underflow when attempting to undo a delete operation with no deleted movies available.
Action	User selects Release Year from the search criteria and enters alphabetic characters (e.g., abcd), then clicks the Search button.
Expected Output	Admin clicks the Undo Delete button when no movies have been deleted.
Actual Output	System displays a message indicating that there are no movies available to restore.  *Figure 59:Test case 14 output 1*
Result	Test Passed

Test Case 15: **Queue Overflow (Recent Movies)**

Table 15: Test case 15

Test Case ID	15
Test Description	Ensures the system handles queue overflow by maintaining a fixed size for recently added movies.
Action	Admin adds more than five movies consecutively to the system.

Expected Output	Only the most recent five movies are stored in the Recently Added Movies queue, and older entries are removed automatically. Because in our code we made recently added movies to handle max 5 movies but if exceed than we delete the oldest one. As in below code.
	`recentMovies.add(movieInput); //usage of Queue FIFO for storing 5 recently added movies`
	`if (recentMovies.size() > 5)`
	`{`
	`recentMovies.remove();`
	`}`
	`return true;`

Actual Output	Queue removes the oldest movie once the size exceeds five and displays only the latest five movies.  *Figure 60: Test case 15 output 1*
Result	Test Passed

7. Coursework Development

7.1. Tools Used

7.1.1. NetBeans IDE

Apache Netbeans IDE is an Opensource IDE which is used for developing Java Based applications. In this coursework netbeans was selected as the main development tool to build the system using the Java programming language while applying the MVC design approach.

Usage:

- Application setup and organization: The coursework system was created using Java progeam within netbeans. It was used to organize and structure the source code into logical packages for models, views and controllers which helped maintain a clean structured codebase.
- Used to design GUI using swing and GUI builder
- Used for debugging runtime and logical errors during development.

Evidence

MVC structure:

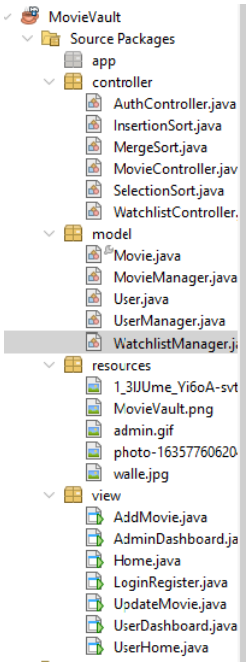


Figure 61: MVC evidence

Using the GUI editor interface in NetBeans:

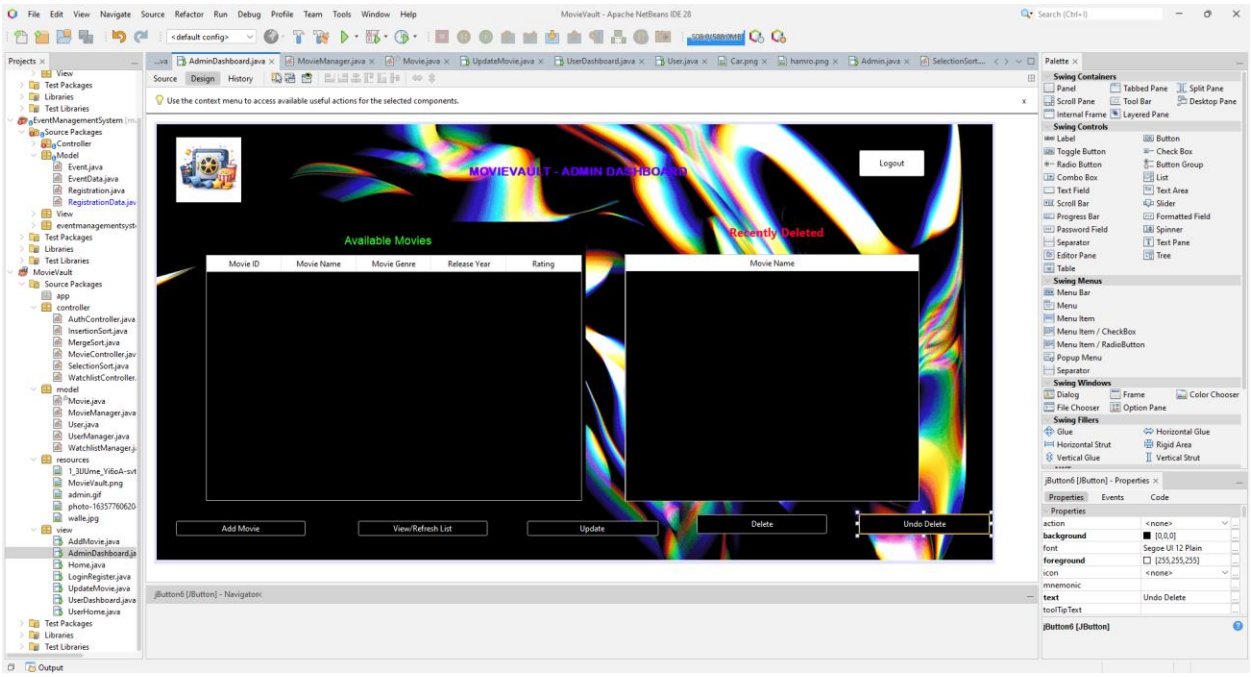


Figure 62: Netbeans evidence

Code editor:

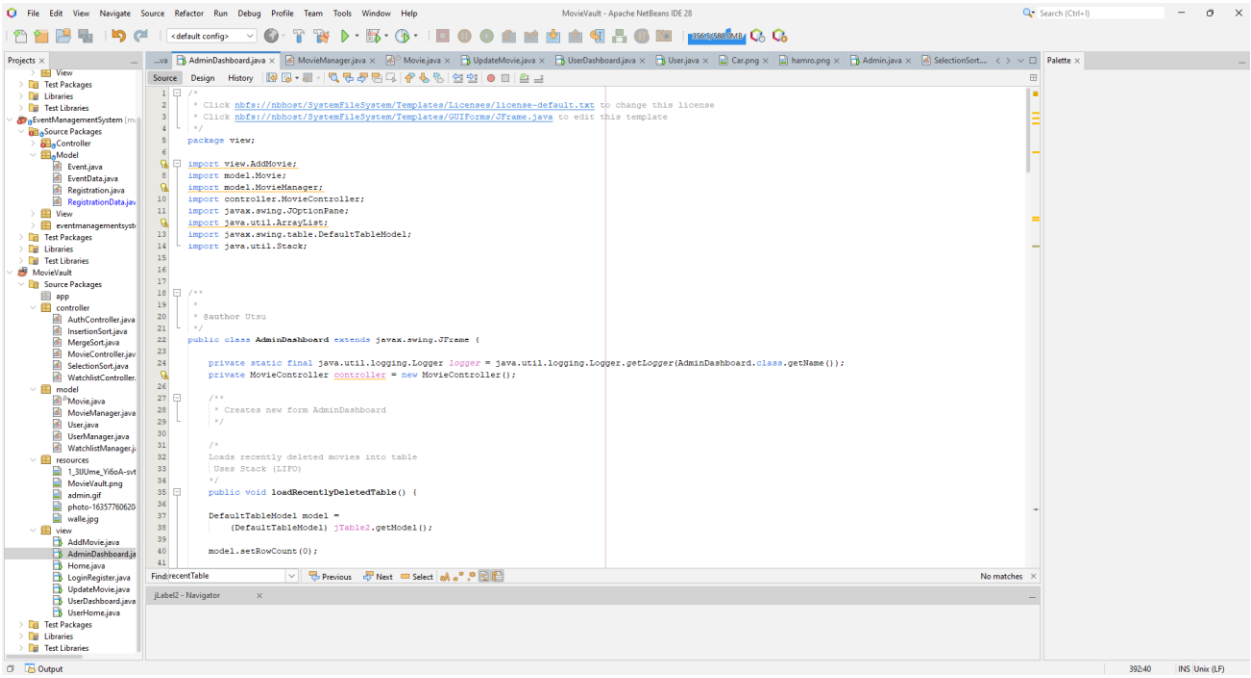


Figure 63: Code editor evidence

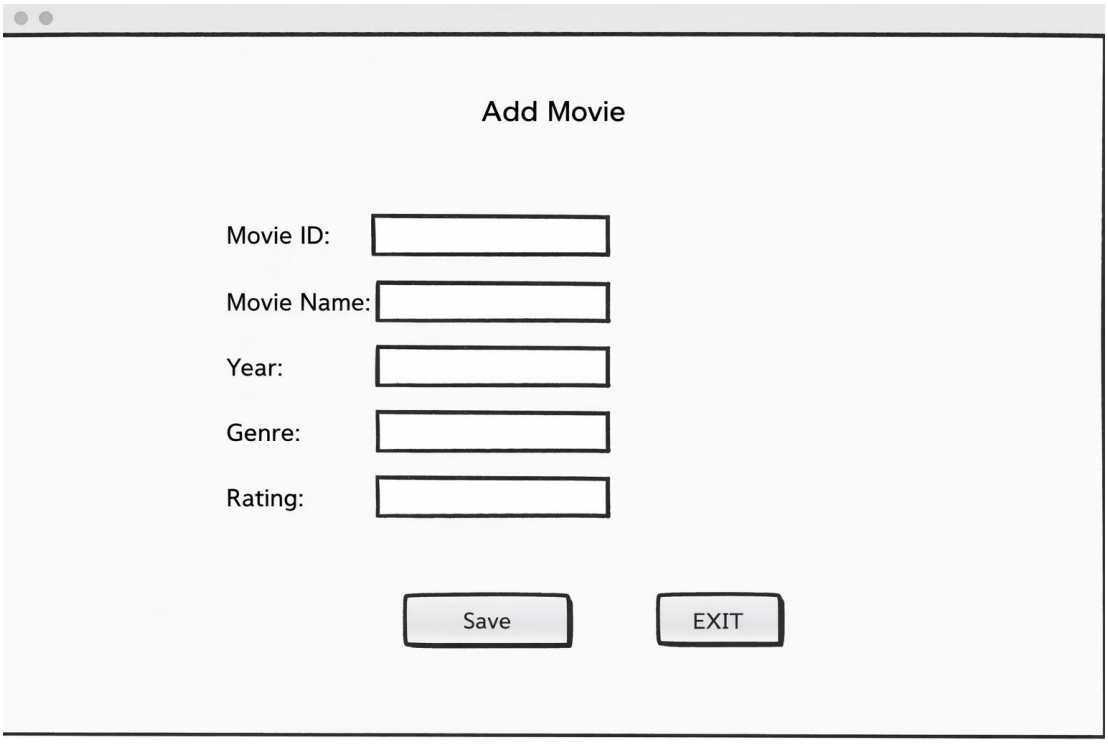
7.1.2. Balsamiq

It is a wireframe designing tool which is used to create interface design screens. In this coursework balsamiq was used to plan the layout of the system before development

Usage:

- Used to design wireframe for views such as forms, dashboards
- Helped planning and visualize the structure, navigation flow and user interaction

Evidence



7.1.3. Draw.io

It is a diagramming tool used to create system diagrams. It was used in this coursework to visually represent the system design using class diagrams.

Usage:

- To make class diagrams to represent the MVC structure
- Class diagram helps visualize the code properly, helps visualize the relationships between classes and shows the methods and attributes present in the system.

Evidence

Creating a class diagram in draw.io

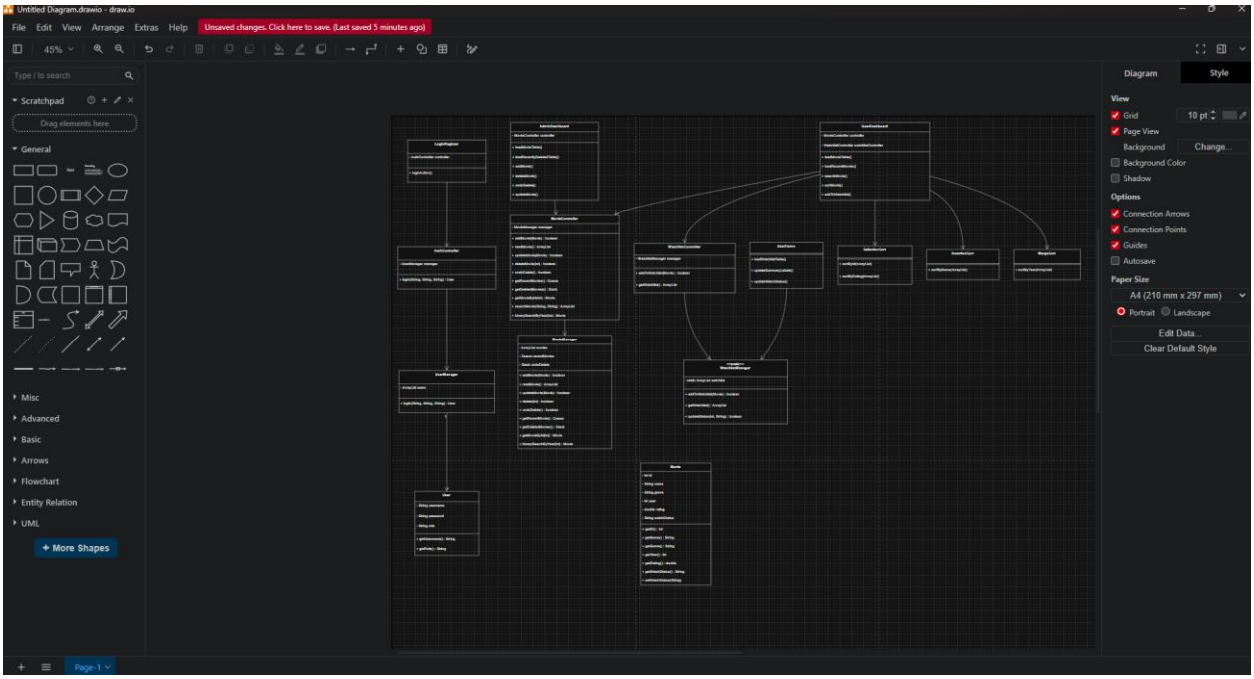


Figure 64: Draw.io evidence

7.2. Algorithms and Implementations

7.2.1. Crud Functionlaity

1. **Create**: The create operation will allow the admin to add new movies by entering the details such as MovieID, Title, Genre, Release Year and Rating. The system validates the input before saving the data to the collection. Upon clicking the add button the system will show add movie panel then after entering the values and clicking the add button in the add panel a new movie is added to the table row.

```
/*Method to add movie
to the list and also checking for duplicate id
and sending confirmation.
*/
public boolean addMovie(Movie movieInput) //here movieInput is a user input which contains (id,name,genre,yearReleased,rating)

{
    for(int i=0; i<= movies.size()-1; i++) //looping through array list here.

    {
        /*checking if the ids are same for movies(storage) and movieInput
        also the getID is a getter function present in Movie class
         */
        if(movies.get(i).getID() == movieInput.getID())
        {
            return false;
        }
    }
    movies.add(movieInput);

    recentMovies.add(movieInput); //usage of Queue FIFO for storing 5 recently added movies
    if (recentMovies.size() > 5)
    {
      recentMovies.remove();
    }
    return true;
}
```

Figure 65: addMovie() implementation

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

    //add movie controller
    AddMovie addMovie = new AddMovie(controller);
    addMovie.setVisible(true);
}
```

Figure 66: Add movie button

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

// Get text values first
String idText = txtMovieId.getText();
String name = txtMovieName.getText();
String genre = txtGenre.getText();
String yearText = txtYear.getText();
String ratingText = txtRating.getText();

// Check empty fields FIRST
if (idText.isEmpty() || name.isEmpty() || genre.isEmpty()
        || yearText.isEmpty() || ratingText.isEmpty()) {

    JOptionPane.showMessageDialog(this, "All fields are required");
    return;
}

// Number check
if (!idText.matches("\\d+") || !yearText.matches("\\d+") || !ratingText.matches("\\d+")) {
    JOptionPane.showMessageDialog(this, "ID, Year and Rating must be numbers only");
    return;
}

// Safe conversion
int id = Integer.parseInt(idText);
int year = Integer.parseInt(yearText);
int rating = Integer.parseInt(ratingText);

Movie movie = new Movie(id, name, genre, year, rating);

boolean added = controller.addMovie(movie);

if (added) {
    JOptionPane.showMessageDialog(this, "Movie added successfully");
    this.dispose();
} else {
    JOptionPane.showMessageDialog(this, "Duplicate Movie ID");
}
```

Figure 67: Save button

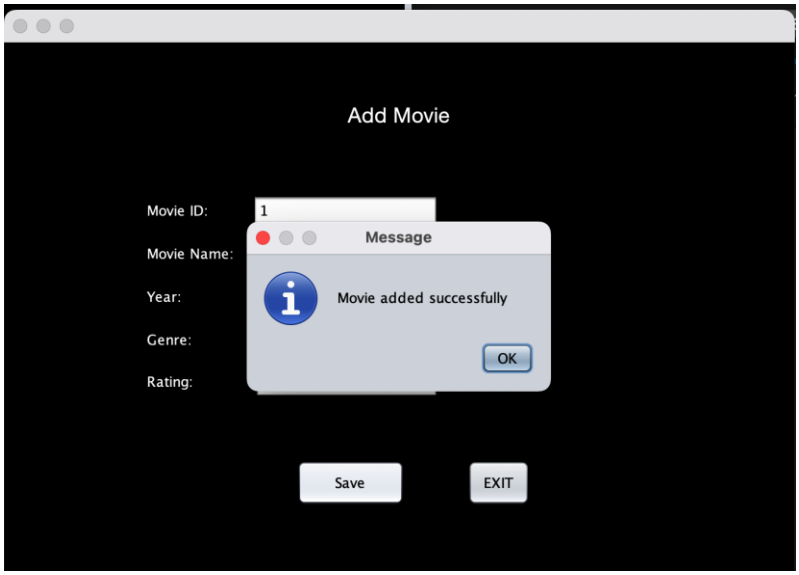


Figure 68: UI response after adding movie

2. **Read:** The read operation retrieves movie records from the collection and displays them in the JTable. This allows users to view all movies and their details clearly. Both admin and users are able to view table.

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    //loads current movies to table
    loadMovieTable();


}
```

Figure 69: Read movie implementation

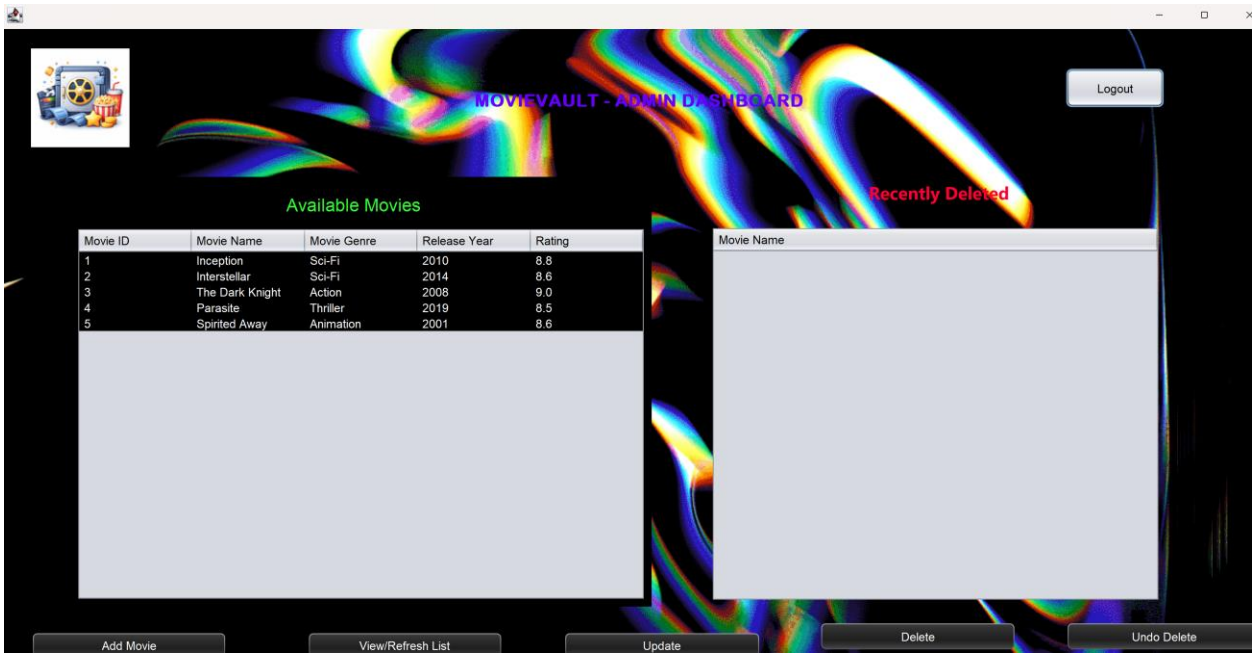


Figure 70: AdminDashboard Displays the read movies

3. **Update**: Updates the attributed in the movie, same as add movie a new panel opens up after pressing the button, in this panel we need to enter an ID and any field/fields that require change, if a field does not require change we do keep it's text field empty.

```
private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
UpdateMovie update = new UpdateMovie(controller);
update.setVisible(true);
}
```

Figure 71: Update Button

```java
//Checks user input and updates the movie also sends a confirmation
public boolean updateMovie(Movie movieInputU)
{
    for(int i=0; i<= movies.size()-1; i++)
    {
        if(movies.get(i).getID() == movieInputU.getID())
        {
            movies.set(i, movieInputU );
            return true;
        }

    }
    return false;
}
```

Figure 72: Update Implementation

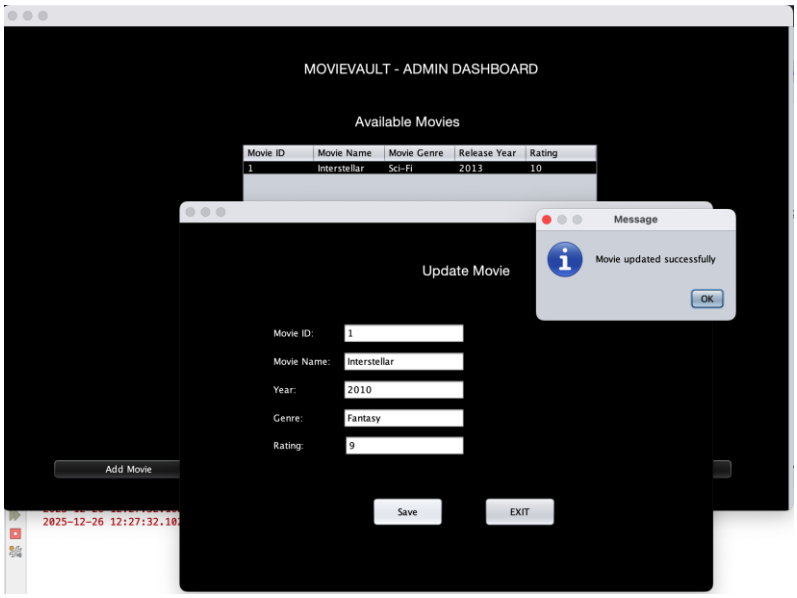


Figure 73: UI response

4. **Delete:** Deleted the entire movie object from the arraylist. UI flow is similar to add and update where a input dialog box asks for the id of the movie. After entering the id the movie is deleted from the collection.

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    String input = JOptionPane.showInputDialog(this, "Enter Movie ID to delete:");

 // 1. User pressed Cancel
 if (input == null) {
return;
}

// 2. Empty input
if (input.isEmpty()) {
JOptionPane.showMessageDialog(this, "Movie ID is required");
return;
}

// 3. Numeric check
if (!input.matches("\\d+")) {
JOptionPane.showMessageDialog(this, "Movie ID must be a number");
return;
}

// 4. Safe to parse
int id = Integer.parseInt(input);

// 5. Delete movie
boolean deleted = controller.deleteMovie(id);

if (deleted) {
JOptionPane.showMessageDialog(this, "Movie deleted successfully");

// refresh available movies table
loadMovieTable();

// refresh recently deleted table
loadRecentlyDeletedTable();

}
else
{
JOptionPane.showMessageDialog(this, "Movie ID not found");
}
```

Figure 74: Delete Button

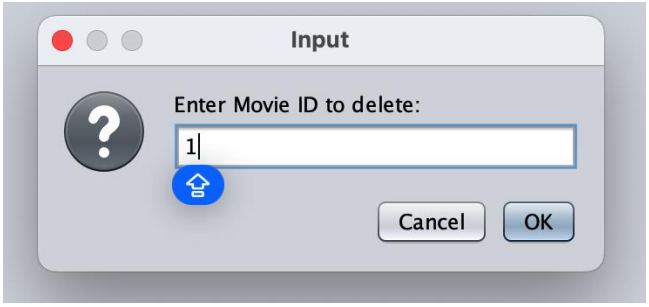


Figure 75: Input Dialog Box for delete

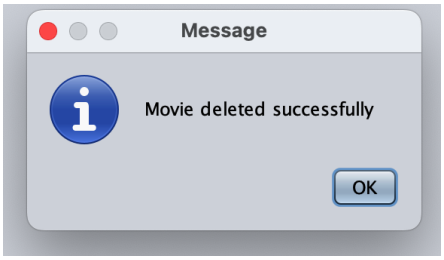


Figure 76: UI response

7.2.2. Searching and Sorting Algorithms Used:

7.2.2.1. Linear Search

Linear search is a searching algorithm where we start from one end and check every element of the list till the desired element is found. It is the simplest searching algorithm. (Programiz, 2025)

In our coursework we use linear search to search for titles in the movie list where partial matching is required.

```
public ArrayList<Movie> searchMovie(String keyword, String criteria) {

    ArrayList<Movie> results = new ArrayList<>();

    for (Movie m : manager.readMovie()) {

        if (criteria.equals("Movie Name")) {
            if (m.getName().toLowerCase().contains(keyword.toLowerCase())) {
                results.add(m);
            }
        }
        else if (criteria.equals("Release Year")) {
            if (String.valueOf(m.getYear()).equals(keyword)) {
                results.add(m);
            }
        }
    }

    return results;
}
```

Figure 77: Linear Search Implementation

7.2.2.2. Binary Search

It is a searching algorithm for finding element's position in a sorted array. In this way, the element is always searched in the middle of a portion of an array. (Programiz, 2025)

```
public Movie binarySearchByYear(int year) {

// Creating a copy so original order is not affected
ArrayList<Movie> list = new ArrayList<>(movies);

// Sorting the list by year before using binary search
for (int i = 0; i < list.size() - 1; i++) {
    for (int j = i + 1; j < list.size(); j++) {
        if (list.get(i).getYear() > list.get(j).getYear()) {
            Movie temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }
}

int low = 0;
int high = list.size() - 1;

while (low <= high) {
    int mid = (low + high) / 2;
    int midYear = list.get(mid).getYear();

    if (midYear == year) {
        return list.get(mid);
    }
    else if (midYear < year) {
        low = mid + 1;
    }
    else {
        high = mid - 1;
    }
}

return null;
```

Figure 78: Binary Search Implementation

7.2.2.3. Selection Sort

Selection sort is a sorting algorithm which smallest elements from an unsorted list in each iteration and places that element at the beginning of the unsorted list. (Programiz, 2026)

```
// Sort by Movie ID
public static void sortById(ArrayList<Movie> list) {
    for (int i = 0; i < list.size() - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < list.size(); j++) {
            if (list.get(j).getID() < list.get(minIndex).getID()) {
                minIndex = j;
            }
        }

        Movie temp = list.get(minIndex);
        list.set(minIndex, list.get(i));
        list.set(i, temp);
    }
}
```

Figure 79: Selection Sort Implementation

7.2.2.4. Insertion Sort

Insertion sort is a sorting algorithm which places an unsorted element at its suitable place in each iteration. (Programiz, 2026)

```
/*
 Insertion sort algorithm
 Used to sort movies based on movie name
 This works by taking one element at a time
 and placing it in the correct position
*/
public static void sortByName(ArrayList<Movie> movies) {

    // Loop starts from second element
    for (int i = 1; i < movies.size(); i++) {

        // Current movie to be placed correctly
        Movie current = movies.get(i);

        int j = i - 1;

        /*
         Moving movies that are greater than current
         one position ahead to make space
        */
        while (j >= 0 &&
               movies.get(j).getName()
               .compareToIgnoreCase(current.getName()) > 0) {

            movies.set(j + 1, movies.get(j));
            j--;
        }

        // Placing current movie at correct position
        movies.set(j + 1, current);
    }
}
```

Figure 80: Insertion Sort Implementation

7.2.2.5. Merge Sort

Merge sort is a sorting algorithm where it follows a divide and conquer algorithm that sorts an array by breaking it into small small arrays, and the building the array back together the correct way meaning it is sorted. (w3schools, 2026)

```
/*
 Merge sort algorithm
 Used to sort movies based on release year
 This uses divide and conquer technique
*/
public static void sortByYear(ArrayList<Movie> movies) {

    // Base condition
    if (movies.size() <= 1) {
        return;
    }

    // Finding middle index
    int mid = movies.size() / 2;

    // Splitting array list into two parts
    ArrayList<Movie> left = new ArrayList<>(movies.subList(0, mid));
    ArrayList<Movie> right = new ArrayList<>(movies.subList(mid, movies.size()));

    // Recursively sorting both halves
    sortByYear(left);
    sortByYear(right);

    // Merging the sorted halves
    merge(movies, left, right);
}
```

Figure 81: Merge Sort Implementation

7.2.3. Queue and Stack

7.2.3.1. Queue

Queue is an interface in java. To use the functionalities of a queue, we need classes that implement it such as LinkedList. In queues the elements are stored and accessed in First In, First Out manner meaning that elements are added from behind and removed from the front. (Prpgramiz, 2026)

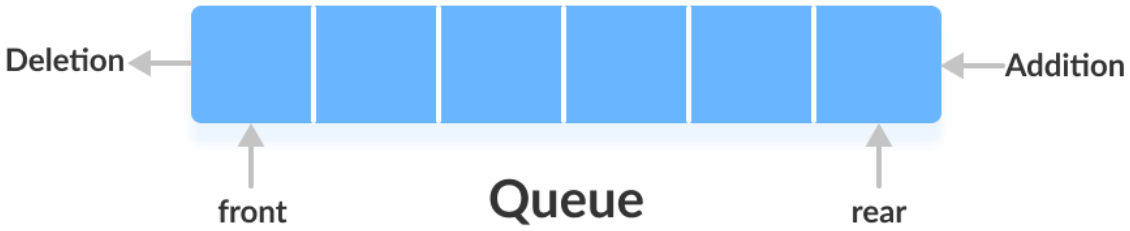


Figure 82: Queue Concept

(Programiz, 2025)

```
/*Declaring an Arraylist called movies
of datatype Movie.
*/
private ArrayList<Movie> movies;
private Queue<Movie> recentMovies;
private Stack<Movie> undoDelete;

/*Initializing the
arraylist :)
*/
public MovieManager()
{
    movies = new ArrayList<>();
    recentMovies = new LinkedList<>(); //queue is an interface but LinkedList implements Queue
    undoDelete = new Stack<>();
```

Figure 83: Creating a queue

```
recentMovies.add(movieInput); //usage of Queue FIFO for storing 5 recently added movies
if (recentMovies.size() > 5)
{
    recentMovies.remove();
}
return true;
```

Figure 84: Usage of queue

7.2.3.2. Stack:

A stack is a linear data structure that follows the principle of Last In First Out (LIFO) meaning the last element inserted inside the stack is removed first. (Programiz, 2026)

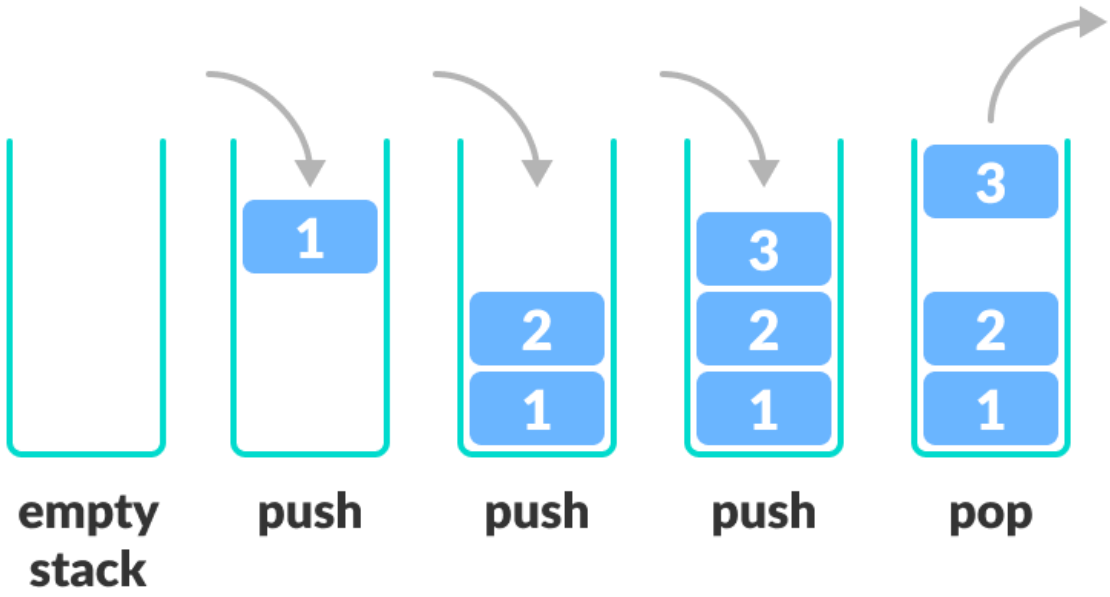


Figure 85: Concept of Stack

(Programiz, 2025)

```java
import java.util.Stack;

public class MovieManager {

    /*Declaring an Arraylist called movies
    of datatype Movie.
    */
    private ArrayList<Movie> movies;
    private Queue<Movie> recentMovies;
    private Stack<Movie> undoDelete;

    /*Initializing the
    arraylist :)
    */
    public MovieManager()
    {
        movies = new ArrayList<>();
        recentMovies = new LinkedList<>(); //queue is an interface but LinkedList implements Queue
        undoDelete = new Stack<>();
```

Figure 86: Creation of stack

```java
if(movies.get(i).getID()== id)
{
    undoDelete.push(movies.get(i)); //usage of stack LIFO
    movies.remove(i);
    return true;
}
```

Figure 87: Usage of stack

8. Critical Analysis

This part of the report looks at the Movie Vault system. It talks about the problems that came up when it was being developed. It also looks at how the Movie Vault system works and compares it to other systems that already exist. The Movie Vault system is also checked to see what is good and bad, about it which is called a SWOT analysis of the Movie Vault system.

8.1. System Breakdown (Issues and Challenges Faced)

When we were making the Movie Vault system we had to deal with a lot of problems. The Movie Vault system had some issues and some design flaws. We fixed these problems by checking for errors, changing the design and making the Movie Vault system work better.

1. Data Persistence Limitation

The system is using some ways to store information right now. It uses things like Array List and Stack and Queue to hold the data in the computer's memory. The system stores data in these in-memory data structures like Array List. Also, things, like Stack and Queue.

So when you restart the application, the movie data and the watchlist data will go back, to how it was. This means you will lose the movie and watchlist data. The movie and watchlist data will reset when the application is restarted.

Impact:

- Data is not persistent across sessions.

- Limits real-world scalability.

The solution that was put into action is the following:

For the coursework scope this was okay because it focused on how to put the data structure into action than making it work with a database. The main thing was the data structure implementation

so that is what was concentrated on, not the database integration. This made sense for the coursework scope and the data structure implementation.

2. UI Synchronization Issues

When we performed undo, deletion or change the watch status the UI tables did not update on their own. The UI tables just stayed the same until we did something. This was a problem because the UI tables are supposed to show you what is going on.

Impact:

- Confusing the users
- Although the system updated in the backend, the front end was not updated

The solution that was put into action is the following:

Explicit refresh methods such as:

- loadMovieTable()
- loadRecentlyDeletedTable()
- loadWatchlistTable()

were called after each operation to ensure UI consistency.

3. Sorting and Searching Conflicts

Binary search needs the information to be, in order. The list of movies we had was all mixed up. The list of movies was not sorted.

Impact:

The binary search did not work right. It gave me the wrong results. The binary search was supposed to find what I was looking for but it returned wrong value because binary search only worked on sorted values.

To find a movie we first made a copy of the movie list. We sorted this copy of the movie list. This way the original movie list stayed the same. We did this so that when we used search, on the copy of the movie list it would work correctly and find the movie we were looking for. We sorted the year before performing the binary search.

4. Watch Status Update Errors

The watch status was getting input error while trying to change the status of the movie watched.

Impact:

- Run time errors crashing the program
- Not updating the status of the movie in the watchlist

The solution that was put into action is the following:

We added input validation to the system

8.2. Evaluation

Strengths

- Clear MVC architecture separating Model, View, and Controller.

- Proper use of data structures (ArrayList, Stack, Queue).

- I have implemented different algorithms for sorting and searching things. Included Binary Search, Linear Search, Selection Sort, Insertion Sort and Merge Sort.

- User-friendly UI with real-time feedback.

Weaknesses

- No database integration (data loss on restart).
- When we use binary search only one value is returned.
- No role-based access control beyond basic login.

Opportunities

- Can be extended with database support (MySQL / SQLite).
- Can support multi-user accounts and profiles.
- Recommendation system based on watch history.
- Advanced filters (genre, rating range).

Threats

- Scalability issues with large datasets.
- Security risks due to plain-text password handling.

- If the list gets really big it does not work well as it should. The bigger the list size is, the more the performance of the list degrades. When the list size grows significantly the performance degradation of the list becomes a problem.

8.3. Comparison with Existing Systems

Existing Systems Issues

Most existing movie tracking systems:

- Are complex and cluttered with ads
- Require constant internet access
- Do not demonstrate internal algorithm usage

How movie vault solves the above problems:

- Movie Vault makes it easy to find the movies you want to watch.
- Movie Vault also helps you keep track of the movies you have seen.
- Simple and clean desktop-based UI
- Offline functionality
- Transparent use of algorithms (sorting, searching)
- Educational focus on data structures

8.4. Performance Metrics

The program works fast when we do not have a lot of data to deal with. It is very responsive. The dataset size is limited so the performance of the dataset is fast and responsive, with little delay. When we are working with a dataset size the performance of the dataset size is what matters.

Search Performance

1. **Linear Search:**

Average time complexity**:** O(n)

It is suitable for small to medium datasets.

2. **Binary Search:**

Time complexity: O (log n) after sorting

Faster and more efficient for numeric data like release year.

3. **Sorting Performance**

Selection Sort: $O(n^2)$

Insertion Sort: $O(n^2)$ (efficient for small datasets)

Merge Sort: O (n log n)

9. Conclusion

In this coursework the foundational structure of our Java Application Movie Vault has been designed successfully. The system demonstrates very clear MVC architecture, which ensures proper separation between data management, business logic and UI components.

Core CRUD functionality has been implemented for managing movies which includes adding, viewing, updating and deleting movie records. We applied binary searching, linear searching, merge sort, insertion sort and selection sort. We also implemented stack and queue and learnt about their working mechanism.

Basic Input Validation has been applied to prevent invalid or incomplete data entry which improves the system stability and usability. Overall, this coursework will establish a solid base for further feature expansion and enhancement of the system into maybe a giant movie database or a recommendation system.

Future Recommendation:

- Use MySQL or SQLite for persistent storage.
- Advanced Search and Filters
- Filter by genre, rating range, or watch status.
- User Profile System
- Separate watchlists for different users.
- Recommendation Engine
- Suggest movies based on viewing history.
- Security Enhancements

- Password encryption and authentication improvements.

10. References

Programiz, 2025. *Programiz.* [Online]
Available at: https://www.programiz.com/dsa/linear-search
[Accessed 11 1 2026].

Programiz, 2026. Programiz. [Online]
Available at: https://www.programiz.com/dsa/binary-search

Programiz, 2026. Programiz. [Online]
Available at: https://www.programiz.com/dsa/selection-sort

Programiz, 2026. Programiz. [Online]
Available at: https://www.programiz.com/dsa/insertion-sort

Programiz, 2026. Programiz. [Online]
Available at: https://www.programiz.com/dsa/stack

Prpgramiz, 2026. Programiz. [Online]
Available at: https://www.programiz.com/java-programming/queue

w3schools, 2026. https://www.w3schools.com/dsa/dsa_algo_mergesort.php. [Online]
Available at: https://www.w3schools.com/dsa/dsa_algo_mergesort.php
[Accessed 15 1 2026].

11.Appendix

```
// Restores the most recently deleted movie using Stack (LIFO)
public boolean undoDelete()
 {
    if (undoDelete.isEmpty())
     {
       return false; // nothing to undo
     }

Movie restoredMovie = undoDelete.pop();
movies.add(restoredMovie);
return true;
 }
```

```
recentMovies.add(movieInput); //usage of Queue FIFO for storing 5 recently added movies
if (recentMovies.size() > 5)
{
  recentMovies.remove();
}
return true;
```

```java
package model;

import java.util.ArrayList;

public class WatchlistManager {

    private static ArrayList<Movie> watchlist = new ArrayList<>();

    public static boolean addMovie(Movie movie) {

        // prevent duplicates
        for (int i = 0; i < watchlist.size(); i++) {
            if (watchlist.get(i).getID() == movie.getID()) {
                return false;
            }
        }

        watchlist.add(movie);
        return true;
    }

    public static ArrayList<Movie> getWatchlist() {
        return watchlist;
    }

    public static boolean updateStatus(int movieId, String status) {

        for (int i = 0; i < watchlist.size(); i++) {
            if (watchlist.get(i).getID() == movieId) {
                watchlist.get(i).setWatchStatus(status);
                return true;
            }
        }
        return false;
    }
}
```

```
private void watchlistbtnActionPerformed(java.awt.event.ActionEvent evt) {
    String idText = jTextField2.getText();

if (idText.equals("")) {
    JOptionPane.showMessageDialog(this, "Enter Movie ID");
    return;
}

if (!idText.matches("\\d+")) {
    JOptionPane.showMessageDialog(this, "Movie ID must be a number");
    return;
}

int id = Integer.parseInt(idText);

Movie movie = controller.getMovieById(id);

if (movie == null) {
    JOptionPane.showMessageDialog(this, "Movie not found");
    return;
}

boolean added = watchlistController.addToWatchlist(movie);

if (added) {
    JOptionPane.showMessageDialog(this, "Movie added to watchlist");
} else {
    JOptionPane.showMessageDialog(this, "Movie already in watchlist");
}
}
```