Faculty of Engineering

Ain Shams University

Project Final Delivery

Software Testing CSE337s

Team 7

Youssef Medhat Mahmoud	2200626
Mostafa Al Hassan Salah	2200747
Mario Milad Helmy	2200540
Ahmed Sayed Abdullah	2200737
Ibrahim Mahmoud Ibrahim	2200182
Omar Ahmed Mohamed	2200267
Seif Eldin Mustafa Abdel Fattah	2200794
Ahmed Saeed Abd Elhamid	2200689

1. Project Overview :

1.1 Project Description : A Java-based movie recommendation system that processes user preferences and movie data from text files to generate personalized movie recommendations based on genre matching.

1.2 Requirements Summary

Input Files:

movies.txt - Contains movie information (title, ID, genres)

users.txt - Contains user information (name, ID, liked movie IDs)

Output File:

recommendations.txt - Contains recommended movies for each user

Core Functionality:

Genre-based recommendation: If a user likes a movie from genre X, recommend all other movies in that genre

1.3 Validation Rules

Field	Validation Rules
Movie Title	Every word must start with a capital letter
Movie ID	Capital letters from title + 3 unique numbers
Movie Genre	Any valid genre string
User Name	Alphabetic characters and spaces only, cannot start with space
User ID	Exactly 9 alphanumeric characters, starts with numbers, may end with one alphabetic character

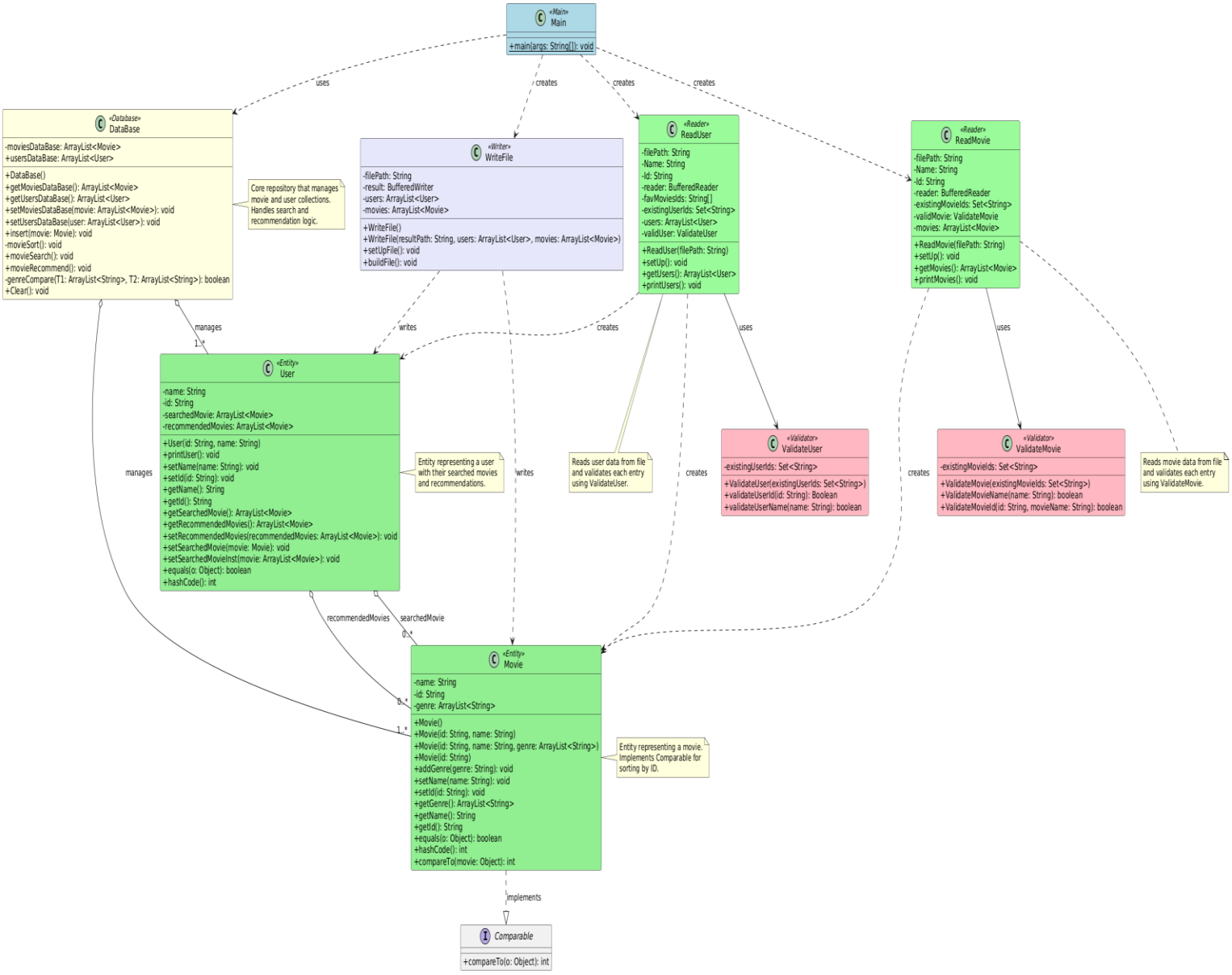
2. System Architecture :

2.1 Project Structure

```
src/
└── main/
    └── java/
        ├── org/
        └── ProjectTm/
            ├── DataBase.java
            ├── desktop.ini
            ├── Main.java
            ├── Movie.java
            ├── ReadMovie.java
            ├── ReadUser.java
            ├── User.java
            ├── ValidateMovie.java
            ├── ValidateUser.java
            └── WriteFile.java
```

```
src/test/java/ProjectTm/
├── BlackBox/
│   ├── BlackBoxDataBaseTest.java
│   ├── BlackBoxValidateMovieIdTest.java
│   ├── BlackBoxValidateMovieNameTest.java
│   ├── BlackBoxValidateMovieTest.java
│   ├── BlackBoxValidateMovie_IdTest.java
│   ├── BlackBoxValidateMovie_NameTest.java
│   ├── BlackBoxValidateUserIdTest.java
│   ├── BlackBoxValidateUserNameTest.java
│   ├── BlackBoxValidateUserTest.java
│   ├── BlackBoxValidateUser_IdTest.java
│   ├── BlackBoxValidateUser_NameTest.java
│   ├── ReadMovieBlackBox.java
│   ├── ReadUserBlackBox.java
│   └── WriteFileBlackBox.java
├── WhiteBox/
│   ├── DataBaseBranchCoverageTest.java
│   ├── DataBaseConditionCoverageTest.java
│   ├── DataBasePathCoverageTest.java
│   ├── DataBaseTestStatmentCoverageMovieRecommend.java
│   ├── DataBaseTestStatmentCoverageMovieSearchMethod.java
│   ├── ReadMovieWhiteBoxTest.java
│   ├── ReadUserWhiteBoxTest.java
│   ├── ValidateMovieBranchCoverageTest.java
│   ├── ValidateMovieConditionCoverageTest.java
│   ├── ValidateMoviePathCoverageTest.java
│   ├── ValidateMovieStatementCoverageTest.java
│   ├── ValidateUserBranchCoverageTest.java
│   ├── ValidateUserConditionCoverageTest.java
│   ├── ValidateUserPathCoverageTest.java
│   ├── ValidateUserStatementCoverageTest.java
│   └── WriteFileWhiteBox.java
├── ControllerLevelTest.java
├── EndToEndIntegrationTest.java
├── IOLayerIntegration.java
├── ReadIntegrationTest.java
├── ReadMovieTest.java
├── ReadUserTest.java
├── ValidateMovieTest.java
├── ValidateUserTest.java
└── WriteFileTest.java
```

2.2Classes UML :



3.Code Review

Main Class

Overview

The Main class serves as the entry point and orchestrator for the entire Movie Recommendation System. It coordinates the initialization, data loading, processing, and output generation workflows, acting as the central controller that manages the interaction between all other components.

Purpose and Responsibilities

The Main class is responsible for:

- System Initialization: Creating and configuring the DataBase object that will manage all movie and user data
- File Path Configuration: Defining the locations of input files (movies.txt, users.txt) and output file (recommendation.txt)
- Data Loading: Instantiating ReadMovie and ReadUser objects to parse and load data from external files
- Error Handling: Managing IOExceptions that may occur during file operations and converting them to RuntimeExceptions
- Workflow Orchestration: Executing the recommendation pipeline in the correct sequence (search, then recommend)
- Output Generation: Creating and executing WriteFile to produce the final recommendation results

Key Method

`main(String[] args)`

The static main method is the single point of entry for the application. It follows a linear execution flow: initialize database → load movies → load users → populate database → search for movies → generate recommendations → write results. This method ensures proper sequencing of operations and handles exceptions at the highest level.

DataBase Class

Overview

The DataBase class is the core repository and business logic engine of the system. It manages collections of movies and users, implements the movie search functionality, and executes the recommendation algorithm. This class represents the heart of the application's data management and processing capabilities.

Attributes

moviesDataBase (private ArrayList<Movie>)

A private collection storing all movie objects in the system. This list is maintained in sorted order by movie ID to enable efficient binary search operations during the movie matching phase.

usersDataBase (public ArrayList<User>)

A public collection storing all user objects. Each user contains their searched movies (as IDs initially, then full objects) and will receive recommended movies based on genre matching.

Core Methods

movieSearch()

This method transforms user movie ID references into full movie objects. It iterates through all users, retrieves their searched movie IDs, performs binary search on the sorted movie database to find matching movies, and replaces ID-only movie objects with complete movie data including genres. If a movie ID is not found, it creates a placeholder movie with "Not found" values.

movieRecommend()

This method implements the recommendation engine. For each user, it analyzes the genres of their searched movies and scans the entire movie database to find movies with matching genres. The algorithm ensures that already-searched movies are excluded from recommendations and uses case-insensitive genre comparison. All matching movies are added to the user's recommended movies list.

movieSort()

A private utility method that sorts the movie database by ID using Collections.sort with a lambda comparator. This sorting is essential for enabling efficient binary search during the movie matching phase.

genreCompare(ArrayList<String>, ArrayList<String>)

A private helper method that determines if two genre lists share any common genres. It performs case-insensitive comparison and returns true if at least one genre matches between the two lists, enabling flexible genre-based recommendations.

Movie Class

Overview

The Movie class is an entity class representing a movie in the system. It encapsulates all movie-related data including identification, naming, and genre categorization. By implementing the Comparable interface, it enables efficient sorting and searching operations within collections.

Attributes

name (private String)

The display name of the movie. According to validation rules, each word must start with a capital letter, ensuring consistent title formatting throughout the system.

id (private String)

A unique identifier for the movie, composed of capital letters from the movie name followed by three digits. This ID serves as the primary key for searching and sorting operations.

genre (private ArrayList<String>)

A collection of genre classifications for the movie. Movies can belong to multiple genres, enabling more flexible and accurate recommendations based on genre matching.

Constructors

The Movie class provides four constructors for different initialization scenarios: a default constructor creating a "Not found" placeholder, a basic constructor with ID and name, a full constructor including genres, and a minimal constructor with only ID (used when creating movie references that will be populated later).

Key Methods

compareTo(Object)

Implements the Comparable interface by comparing movie IDs lexicographically. This enables automatic sorting of movie collections and efficient binary search operations in the database.

addGenre(String)

Adds a genre to the movie's genre list with validation to prevent null or empty genres. This method enforces data integrity by throwing IllegalArgumentException for invalid inputs.

equals(Object) and hashCode()

Properly overridden methods ensuring correct behavior in collections. Two movies are considered equal if they have identical IDs, names, and genre lists, supporting proper Set and Map operations.

User Class

Overview

The User class represents a system user and manages their movie interactions. It maintains the user's identity information and tracks both the movies they have searched for and the movies recommended to them based on their preferences. This class serves as the central entity for personalized recommendation functionality.

Attributes

name (private String)

The user's display name, consisting of alphabetic characters and spaces only. The name must start with a non-space character according to validation rules.

id (private String)

A unique 9-character identifier for the user. The ID must start with a digit and contain only alphanumeric characters, ensuring unique identification across the system.

searchedMovie (private ArrayList<Movie>)

A collection of movies that the user has searched for or expressed interest in. Initially populated with Movie objects containing only IDs, these are later enriched with full movie data during the search phase.

recommendedMovies (private ArrayList<Movie>)

A collection of movies recommended to the user based on genre matching with their searched movies. This list is populated by the recommendation algorithm and represents personalized suggestions.

Key Methods

setSearchedMovie(Movie)

Adds a single movie to the user's searched movies list. This method is used during the initial data loading phase to build the user's movie preference profile incrementally.

setSearchedMovieInst(ArrayList<Movie>)

Replaces the entire searched movies collection with a new list. This method is used during the movie search phase to update ID-only movie objects with complete movie data including all attributes and genres.

printUser()

Outputs the user's information including name, ID, and all recommended movies with their details. This method provides a formatted view of the user's profile and recommendations.

ReadMovie Class

Overview

The ReadMovie class is responsible for reading and parsing movie data from external text files. It implements the Data Access Object (DAO) pattern, handling file I/O operations, data parsing, validation, and conversion of raw text into Movie objects. This class serves as the bridge between external data storage and the application's object model.

Attributes

filePath (private String)

The path to the input file containing movie data. This is configured during construction and remains constant throughout the object's lifetime.

reader (BufferedReader)

The BufferedReader instance used for efficient line-by-line reading of the movie data file. This provides buffered I/O for better performance when processing large files.

existingMovieIds (Set<String>)

A HashSet tracking movie IDs to ensure uniqueness across the dataset. This is shared with the ValidateMovie validator to prevent duplicate movie entries.

validMovie (ValidateMovie)

The validator instance responsible for enforcing business rules on movie data. It validates movie names, IDs, and ensures data integrity during the parsing process.

File Format

The expected file format is structured with movie name and ID on one line (comma-separated), followed by genres on the next line (comma-separated). Empty lines are skipped. Each movie entry spans exactly two lines: the identification line and the genre line.

Key Methods

setUp()

Initializes the BufferedReader for the specified file path. Throws FileNotFoundException if the file doesn't exist, allowing the caller to handle missing files appropriately.

getMovies()

The main parsing method that reads the entire file, validates each entry, and constructs Movie objects. It handles multi-line parsing, validates each component using ValidateMovie, manages errors with descriptive exceptions, and returns a complete list of validated Movie objects. The method implements robust error handling with clear error messages for debugging.

Error Handling

ReadMovie implements comprehensive error handling including FileNotFoundException for missing files, IllegalArgumentException for malformed data or validation failures, and IOException for general I/O errors. Each error includes contextual information about what went wrong and where in the file it occurred.

ReadUser Class

Overview

The ReadUser class handles the reading and parsing of user data from external text files. Similar to ReadMovie, it implements the DAO pattern for user entities, managing file I/O, data validation, and object construction. This class creates User objects populated with their searched movie references.

Attributes

filePath (private String)

The path to the input file containing user data, configured at construction and used throughout the reading process.

reader (BufferedReader)

BufferedReader instance for efficient line-by-line file reading, providing buffered I/O capabilities for better performance.

favMoviesIds (private String[])

Temporary array holding the parsed movie IDs from the file before they are converted into Movie objects and added to the User.

existingUserIds (Set<String>)

A HashSet tracking user IDs to enforce uniqueness constraints across the user dataset, shared with ValidateUser for duplicate detection.

validUser (ValidateUser)

The validator instance that enforces business rules on user data, including name format, ID structure, and uniqueness validation.

File Format

The user file format consists of user name and ID on the first line (comma-separated), followed by movie IDs on the second line (comma-separated). Each user entry spans exactly two lines, with empty lines being skipped during parsing.

Key Methods

getUsers()

The main method that parses the user file and constructs User objects. It reads user identification information, validates it using ValidateUser, parses the list of movie IDs, creates Movie objects with IDs only (to be populated later), and builds complete User objects with their searched movies. The method includes comprehensive error handling and debugging output.

printUsers()

A utility method that outputs all loaded users and their searched movie IDs to the console, useful for debugging and verification during development.

ValidateMovie Class

Overview

The ValidateMovie class encapsulates all validation logic for movie data. It enforces business rules regarding movie naming conventions, ID format requirements, and uniqueness constraints. By separating validation logic into its own class, the system achieves better maintainability and testability of validation rules.

Attributes

existingMovieIds (private final Set<String>)

A set tracking the numeric portions of movie IDs to enforce uniqueness. This is shared with ReadMovie and updated as movies are validated, ensuring no duplicate IDs exist in the system.

Validation Rules

Movie Name Rules:

- Cannot be null or empty
- Every word must start with a capital letter (title case)
- Spaces between words are allowed

Movie ID Rules:

- Cannot be null or empty
- Must start with capital letters extracted from the movie name (first letter of each word)
- Must end with exactly three numeric digits
- The numeric portion must be unique across all movies

Key Methods

ValidateMovieName(String)

Validates that the movie name adheres to title case formatting. It splits the name into words and verifies that each non-empty word starts with a capital letter, throwing IllegalArgumentException with descriptive messages for any violations.

ValidateMovieId(String, String)

Validates the movie ID format and uniqueness. It extracts capital letters from the movie name to form the expected prefix, verifies the ID starts with this prefix, checks that exactly three digits follow, validates that these digits are unique using the tracking set, and adds the new ID to the set if valid. This method ensures both structural correctness and uniqueness of movie identifiers.

ValidateUser Class

Overview

The ValidateUser class implements validation logic specifically for user data. It enforces business rules for user identification, name formatting, and uniqueness constraints. Like ValidateMovie, it separates validation concerns from data access logic, improving code organization and maintainability.

Attributes

existingUserIds (private final Set<String>)

A set maintaining all validated user IDs to enforce uniqueness constraints. This is shared with ReadUser and updated as each user is validated.

Validation Rules

User ID Rules:

- Cannot be null
- Must be exactly 9 characters long
- Must start with a digit (0-9)
- Can only contain alphanumeric characters (no special characters)

- Must be unique across all users

User Name Rules:

- Cannot be null or empty
- Cannot start with a space character
- Must contain only alphabetic characters and spaces

Key Methods

validateUserId(String)

Performs comprehensive validation of user IDs. It checks for null values and exact length requirement, verifies the first character is a digit, ensures only alphanumeric characters are present using regex pattern matching, checks uniqueness against the tracking set, and adds valid IDs to the set. Each validation failure throws a descriptive IllegalArgumentException.

validateUserName(String)

Validates user names according to formatting rules. It checks for null or empty values, ensures the name doesn't start with a space, and verifies that only alphabetic characters and spaces are present using regex validation. This ensures consistent name formatting across the system.

WriteFile Class

Overview

The WriteFile class handles the generation of output files containing recommendation results. It formats and writes user information along with their searched movies and recommendations to a text file. This class serves as the output layer of the system, converting processed data back into a human-readable format.

Attributes

filePath (private String)

The destination path where the recommendation results will be written. This is configured during construction.

result (BufferedWriter)

The BufferedWriter instance used for efficient writing of formatted output. Buffering improves performance when writing large amounts of data.

users (ArrayList<User>)

The collection of users whose recommendation data will be written to the output file.

movies (ArrayList<Movie>)

The complete movie database, maintained for reference though primarily the user objects contain the necessary data for output.

Output Format

The output file contains user information on the first line (name and ID, comma-separated), followed by a second line listing searched movie names and recommended movie names (comma-separated). Each user's data spans two lines with a clear, readable format.

Key Methods

setUpFile()

Initializes the BufferedWriter for the output file. It wraps potential IOExceptions in RuntimeExceptions with descriptive messages, ensuring any file creation or access issues are properly reported.

buildFile()

The main method that generates the complete output file. It iterates through all users, writes their identification information, formats and writes their searched movies, formats and writes their recommended movies with appropriate comma separation, and ensures proper line breaks between user entries. The method includes error handling and properly closes the writer when finished.

Unit Testing:

The following test casesreveal significant technical issues currently present in the codebase:

Logic and Algorithm Failures:

The search engine fails to detect movies if they are in the first position of the database.
The recommendation module incorrectly suggests movies the user has already watched.

Technical Crashes (NullPointerExceptions):

The program crashes during file writing if the file path is null.
A NullPointerException occurs in the genreCompare method when a movie has a null genre list.

Type Mismatch Errors:

Multiple tests fail because the code attempts to return ArrayList<Movie> when an ArrayList<String> is expected, or mixes up String and int data types.

Validation Successes:

The system successfully identifies and blocks invalid user data, such as names starting with digits, null IDs, or duplicate IDs.
Basic movie object creation with valid parameters is functioning correctly.

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
1	Method declared to return ArrayList< String> but returns ArrayList< Movie> Causing compile-error	RQ-01	CND-01	Two input files Users.txt Movies.txt	Open ReadMovie Class Locate method getMovies() Check return type in method signature Attempt to compile the project	Compile should accept the method return type. No type mismatch should occur.	Code should compile successfully without errors	Compiler throws error:ArrayList cannot be converted to ArrayList<String>	N
2	Method declared to return ArrayList< String> but returns ArrayList< Movie> Causing compile-error	RQ-02	CND-02	Two input files Users.txt Movies.txt	Open ReadUser Class Locate method getUser() Check return type in method signature Attempt to compile the project	Compile should accept the method return type. No type mismatch should occur.	Code should compile successfully without errors	Compiler throws error:ArrayList cannot be converted to ArrayList<String>	N
3	WriteFile crashes with NullPointerException when filePath is null	RQ-03	CND-03	Two input files Users.txt Movies.txt But the filePath of result is null.	Initialize WriteFile object with null filePath Call setUpFile() Call buildFile() Run the program	Program should handle null file path gracefully show error message and should not crash.	Program crashing.	Program throws NullPointerExceptionat FileWriter.	N
4	Verify that reader.getMovies() correctly reads movies records from file and returns Movie Objects with correct name, Id and categories.	RQ-04	CND-04	Movies.txt file containing: Lord Of The Ring, LOTR123 Action, Fantasia Dark Knight, DK102 Action Elsanafer, E264 Carton	Instantiate file reader Call reader.getMovies() Convert result to movieInfo array of Strings (name, id, categories) Compare arrays using assertsarrayEquals().	Movie list returned with exact values: [“Lord Of The Ring”, “LOTR123”, “Action”, “Fantasia”] [“Dark Knight”, “DK102”, “Action”] [“Elsanafer”, ”E264”, “Carton”]	It read the correct Data from the file and pass the test.	Compile without any error and pass this test case.	Y
5	Invalid movie file format: Missing genres line Missing Id in movie First line.	RQ-05	CND-05	Case 1: Lord Of The Ring, LOTR123 Without genres line after it. Case 2: Lord Of The Ring (missing Id)	Prepare a movies.txt file with the invalid format (case 1 or case 2) Call ReadMovies.getMovies() in the program. Run the program.	Case 1: Expected throw IllegalArgumentException: Missing genres for movie: Lord Of The Ring Case 2: Expected show message : There is a missing part in this file then throw IllegalArgumentException: Error in movie line.	Program stops and reports the specific formatting error.	Case 1: Program crashes with “Missing genres exception”. Case 2: program crashes at “Error in movie line exception”	N
6	Validate that users are	RQ-06	CND-06	3 user records in testUserFile	Initialize ReadUser with testUserFile.txt Call getUsers()	System should correctly read all user records with	It read the correct Data from the	Compile without any error and pass	Y

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
	correctly read from file with searched movie IDs			.txt	Convert result into string arrays Validate values using assertArrayEquals	exact name, ID, and searched movie IDs	file and pass the test	this test case	
7	Verify buildFile() writes correct file output using mocked user/movie data	RQ-07	CND-07	Mocked user file: User: Mario, 253644747 Movies: Lord Of The Ring, Avatar	Mock Movie and User objects Prepare list of movies and users Create temporary file Override setUpFile() to write to temp file Call buildFile() Read file content Compare with expected text	buildFile() should write user data followed by recommended movie names in correct format	File successfully created and closed	Output file exactly matches expected text	Y
8	the target movie exists in the first position of the database, it is never detected by the search..	RQ-08	CND-08	User 1: ID = 123456789, Name = Ahmed Mohamed, Searched Movie = A123 User 2: ID = 234567890, Name = Mona Ali, Searched Movie = HA456	Create object from the database Create object from the MockData Call the getMockMovies Call Ther getMockUsers 1. Push the return in the database 2. Call The MovieSearch 3. Call The MovieRecommend Sheck the return UserDataBase	User 1 User ID: 123456789 User Name: Ahmed Mohamed Searched Movie: ID: A123 Name: Avengers Genres: Action, Adventure Recommended Movies: ID: I789 – Name: Inception – Genres: Action, Thriller User 2 User ID: 234567890 User Name: Mona Ali Searched Movie: ID: HA456 Name: Home Alone Genres: Comedy, Family Recommended Movies: None (empty list)	UserDataBase Now hold the users and there searched and recommended movies Movie DataBase Holds the movies unchanged	Compiler throws java.lang.NullPointerException: Cannot invoke "java.util.ArrayList.size()" because "T1" is null	N
9	The application crashes with a java.lang.NullPointerException at line 88 in the genreCompare method.	RQ-09	CND-09	User 1: ID = 345678901, Name = Salma Hassan, Searched Movie = AF321 User 2: ID = 456789012, Name = Ziad Mahmoud, Searched Movie = HAT654	Create object from the database Create object from the MockData Call the getMockMovies2 Call Ther getMockUsers2 Push the return in the database Call The MovieSearch Call The MovieRecommed Sheck the return UserDataBase	User 1 — Salma Hassan User ID: 345678901 User Name: Salma Hassan Searched Movie: Movie object is empty (new Movie()) *No movie details provided* Recommended Movies: None (empty list) User 2 — Ziad Mahmoud User ID: 456789012	UserDataBase Now hold the users and there searched and recommended movies Movie DataBase Holds the movies unchanged	Compiler throws java.lang.NullPointerException: Cannot invoke "java.util.ArrayList.size()" because "T1" is null	N

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
						User Name: Ziad Mahmoud Searched Movie: ID: HAT654 Name: Home Alone Two Genres: Comedy, Family Recommended Movies (sorted by Movie ID): GOTG486 — Guardians Of The Galaxy Genres: Action, Comedy TSF267 — Toy Story Four Genres: Animation, Family			
10	This bug occurs because the MockData.java file is assigning or passing a **String value to a variable or method that expects an int**, causing the compiler to throw "java.lang.String cannot be converted to int."	RQ-10	CND-10	User 1 — Omar Said User ID: 567890123 User Name: Omar Said Searched Movie ID: I741 User 2 — Laila Nour User ID: 678901234 User Name: Laila Nour Searched Movie ID: TI852 User 3 — Youssef Amir User ID: 789012345 User Name: Youssef Amir Searched Movie ID: SI963 User 4 — Nada Samir User ID: 890123456 User Name: Nada Samir	Create object from the database Create object from the MockData Call the getMockMovies3 Call Ther getMockUsers3 Push the return in the database Call The MovieSearch Call The MovieRecommed Sheck the return UserDataBase	Expected Output User 1 — Omar Said User ID: 567890123 User Name: Omar Said Searched Movie: ID: I741 Name: Interstellar Genres: Action, Adventure Recommended Movies (sorted by ID): SI963 — Shutter Island (Thriller, Action) TI852 — The Intern (Comedy, Drama, Adventure) User 2 — Laila Nour User ID: 678901234 User Name: Laila Nour Searched Movie: ID: TI852 Name: The Intern Genres: Comedy, Drama, Adventure Recommended Movies (sorted by ID): C357 — Coco (Animation, Family, Comedy) I741 — Interstellar (Action, Adventure) User 3 — Youssef Amir	UserDataBase Now hold the users and there searched and recommended movies Movie DataBase Holds the movies unchanged	compiler throw "java.lang.String cannot be converted to int.".	N

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
				Searched Movie ID: C357		User ID: 789012345 User Name: Youssef Amir Searched Movie: ID: SI963 Name: Shutter Island Genres: Thriller, Action Recommended Movies: I741 — Interstellar (Action, Adventure) User 4 — Nada Samir User ID: 890123456 User Name: Nada Samir Searched Movie: ID: C357 Name: Coco Genres: Animation, Family, Comedy Recommended Movies: TI852 — The Intern (Comedy, Drama, Adventure)			
11	The recommendation module exhibits a logic error where it fails to filter out movies the user has already watched from the final output.	RQ-11	CND-11	User 1 — Sara Kamal User ID: 901234567 User Name: Sara Kamal Searched Movie ID: J147 User 2 — Ali Rashed User ID: 012345678 User Name: Ali Rashed Searched Movie ID: BP258 User 3 — Hana Tarek User ID: 112233445 User Name: Hana Tarek Searched Movie ID: T369 User 4 —	Create object from the database Create object from the MockData Call the getMockMovies3 Call Ther getMockUsers3 Push the return in the database Call The MovieSearch Call The MovieRecommed 8.Sheck the return UserDataBase	User 2 ID: 012345678 Name: Ali Rashed Searched Movie: Black Panther (ID: BP258, Genres: Action, Adventure) Recommended Movies: Jumanji (ID: J147, Genres: Comedy, Adventure) Tenet (ID: T369, Genres: Thriller, Action) User 3 ID: 112233445 Name: Hana Tarek Searched Movie: Tenet (ID: T369, Genres: Thriller, Action) Recommended Movies: Black Panther (ID: BP258, Genres: Action, Adventure) User 4 ID: 223344556 Name: Karim Adel	UserDataBase Now hold the users and there searched and recommended movies Movie DataBase Holds the movies unchanged	the system incorrectly includes that exact same movie in the generated recommendations. This results in redundant suggestions,	N

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
				Karim Adel User ID: 223344556 User Name: Karim Adel Searched Movie ID: J147		Searched Movie: Jumanji (ID: J147, Genres: Comedy, Adventure) Recommended Movies: Black Panther (ID: BP258, Genres: Action, Adventure) Ratatouille (ID: R654, Genres: Animation, Comedy)			
12	the code attempt to assign or return values where the data types do not match—specifically, ints are being used where Strings are expected, and Strings are being used where ints are required.	RQ-12	CND-12	Movies List (Input) Movie ID: BP258 Title: Black Panther Genres: Action, Adventure Movie ID: J147 Title: Jumanji Genres: Comedy, Adventure Movie ID: T369 Title: Tenet Genres: Thriller, Action Movie ID: R654 Title: Ratatouille Genres: Animation, Comedy	Create object from the database Create object from the MockData Call the getMockMovies3 Call Ther getMockUsers3 Push the return in the database Call The MovieSearch Call The MovieRecommed Sheck the returnMovieDataBase	**Movie ID:** BP258 **Title:** Black Panther **Genres:** Action, Adventure **Movie ID:** J147 **Title:** Jumanji **Genres:** Comedy, Adventure **Movie ID:** R654 **Title:** Ratatouille **Genres:** Animation, Comedy **Movie ID:** T369 **Title:** Tenet **Genres:** Thriller, Action	UserDataBase Now hold the users and there searched and recommended movies Movie DataBase Holds the movies Sorted Ascending	the compiler throws multiple "incompatible types" errors.	N
13	Verifying Correct User Id	RQ-13	CND-13	validator.validateUserId("12345678A")	Validating a user id before creating an object	Id should pass as valid	Returns True	Returned true	Y
14	Verifing Null Id	RQ-14	CND-14	validator.validateUserId(null)	Validating a null id before creating an object	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
15	Verifying Id with wrong length	RQ-15	CND-15	validator.validateUserId("123")	Create a user with a name that starts with a space	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
16	Adding a user with an id that is repeated id that was added before	RQ-16	CND-16	ids.add("12345678A"); validator.validateUserId("12345678A")	Validate an id that was added before	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
17	Validating an id that starts with letters	RQ-17	CND-17	validator.validateUserId("A23456789")	Validate an id that starts with a letter	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
18	Validating an id that contains a special character	RQ-19	CND-19	validator.validateUserId("12345@78A")	Validate an id that has special character	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
19	Validating a user name that is null	RQ-20	CND-20	validator.validateUserName(null)	Validating a User name that is null	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
20	Validating a valid user name	RQ-21	CND-21	validator.validateUserName("John Doe")	Validating a valid user name	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
21	Valiadating a user name that starts with a space	RQ-22	CND-22	validator.validateUserName(" John")	Validating a user name that starts with a space	Illegal Argument Exception should be thrown	The program terminates	Illegal Argument Eception is thrown	Y

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
22	Validating a user name with digits	RQ-23	CND-23	validator.validateUserName("John Doe1234")	Validating a user name that has digits in it	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
23	Validating a username that has special characters	RQ-24	CND-24	validator.validateUserName("John@ Doe")	Validate a user name with special characters	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
24	Validate Movie Id that is null	RQ-25	CND-25	validator.ValidateMovieId(null, "Harry Potter")	Validating a null id	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
25	Validate a movie id that is empty	RQ-26	CND-26	validator.ValidateMovieId("", "Harry Potter")	Validate a movie id that is empty	Illegal Argument Exception should be thrown	the program terminates	Illegal Argument Exception is thrown	Y
26	Validate a valid movie id	RQ-27	CND-27	validator.ValidateMovieId("HP123", "Harry Potter")	Validate a movie id that is valid	Should return true	Returns true	Returned true	Y
27	Validate a movie id that starts with wrong prefix	RQ-27	CND-27	validator.ValidateMovieId("WR123", "Harry Potter")	Validate a movie id that starts with the wrong prefix	An Illegal Argument Exception should be thrown	the program terminates	An illegal argument exception was thrown	Y
28	Validate a movie Id that has a numeric part that already exists	RQ-28	CND-28	ids.add("123"); validator.ValidateMovieId("HP123", "Harry Potter")	Validate a movie id that has a numeric part that already exists	An Illegal Argument Exception should be thrown	The program terminates	An illegal Argument exception was thrown	Y

scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass Fail Y/N
29	Validate a movie Id that has less than 3 digits	RQ-29	CND-28	validator.ValidateMovieId("HP12", "Harry Potter")	Validate a movie id that has a numeric part that is less than 3 digits long	An illegal argument exception should be thrown	The program terminates	An illegal Argument exception was thrown	Y
30	Validate a movie id that has non digit characters in the numeric part	RQ-30	CND-30	validator.ValidateMovieId("HP12A", "Harry Potter")	Validate a movie id that has non digit characters in the numeric part	An illegal argument exception should be thrown	The program terminates	An illegal Argument exception was thrown	Y
31	Validate a valid movie name	RQ-31	CND-31	validator.ValidateMovieName("Harry Potter")	Validate a valid movie name	Should return true	Returns true	Returns true	Y
32	Validate a Movie Name that is all lower case	RQ-32	CND-32	validator.ValidateMovieName("harry Potter")	Validate a movie name that is all lowercase	Should throw an illegal argument exception	The program terminates	Illegal argument exception was thrown	Y
33	Validate a movie name that is null	RQ-33	CND-33	Validator.ValidateMovieName(null)	Validate a null movie name	Should throw an illegal argument exception	The program terminates	Illegal argument exception was thrown	Y
34	Validate an empty movie name	RQ-34	CND-34	Validator.ValidateMovieName(“”)	Validate an empty Movie Name	Should throw an illegal argument exception	The program terminates	Illegal argument exception was thrown	Y

Req	Description	Cond	Description
RQ-01	The system must read movies correctly	CND-01	Movies.txt must be exist and initialized with valid parameters
RQ-02	The system must read users correctly	CND-02	user.txt must be exist and initialized with valid parameters
RQ-03	.The system must create and write output files successfully when a valid file path is provided	CND-03	.file Path must not be null and Write File object must be initialized with valid parameters
RQ-04	The system must correctly read movie records from the input file and return Movie objects with valid name, ID, and categories.	CND-04	the movies.txt file must contain well-formatted movie entries each movie line followed by valid genres lines
RQ-05	The system must detect invalid movie file formats and throw the correct exceptions when required data is missing.	CND-05	The movies.txt file must intentionally contain formatting errors (missing genres or missing ID) to trigger error handling.
RQ-06	The system must correctly read user records from the file, including user name, ID, and searched movie IDs.	CND-06	The user input file (testUserFile.txt) must contain exactly three valid user records.
RQ-07	The system must correctly generate an output file using user data and movie recommendations in the expected format.	CND-07	Mocked Movie and User objects must be prepared before executing buildFile()
RQ-08	The system search the movie data base correctly	CND-08	The movie database shall be holding data pushed and the user insert a valid Id
RQ-09	The system Shall work probably if the user searched non existing movie	CND-09	The movie database shall be holding data pushed and the user insert a no valid Id
RQ-10	The user Id and the Movie ID shall be string	CND-10	The movie database shall be holding data pushed with valid string type ID and the user insert a valid Id with string type
RQ-11	The System shall not repeat the Searched Movie in the Recommend ed Movie Arraylist	CND-11	The movie database shall be holding data pushed type ID and the user insert a valid Id with string type for a Movie of existing genere
RQ-12	The system shall organize the data and sort the String ID correctly	CND-12	The user Should Push different valid Movie IDs
RQ-13	The system shall validate a user ID before creating a user object.	CND-13	The user ID is exactly 9 characters long, begins with digits, and ends with a letter.
RQ-14	The system shall throw an exception when validating a null user ID.	CND-14	The user ID is null.
RQ-15	The system shall throw an exception when validating a user ID with an invalid length.	CND-15	The user ID is less than 9 characters long.
RQ-16	The system shall throw an exception when validating a user ID that already exists.	CND-16	The user ID matches an existing ID in the system.
RQ-17	The system shall throw an exception when validating a user ID that starts with a letter.	CND-17	The user ID begins with a non-digit character.
RQ-18	The system shall throw an exception when validating a user ID containing special characters.	CND-18	The user ID includes a special character.
RQ-19	The system shall throw an exception when validating a null user name.	CND-19	The user name is null.
RQ-20	The system shall validate a correctly formatted user name.	CND-20	The user name contains only letters and spaces.
RQ-21	The system shall throw an exception when validating a user name starting with a space.	CND-21	The user name begins with a whitespace character.
RQ-22	The system shall throw an exception when validating a user name containing digits.	CND-22	The user name includes numeric characters.
RQ-23	The system shall throw an exception when validating a user name containing special characters.	CND-23	The user name includes special characters.
RQ-24	The system shall throw an exception when validating a	CND-24	The movie ID is null.

	null movie ID.		
RQ-25	The system shall throw an exception when validating an empty movie ID.	CND-25	The movie ID is an empty string.
RQ-26	The system shall validate a correctly formatted movie ID.	CND-26	The movie ID starts with a valid prefix and ends with exactly 3 digits.
RQ-27	The system shall throw an exception when validating a movie ID with an invalid prefix.	CND-27	The movie ID does not start with the expected prefix
RQ-28	The system shall throw an exception when validating a movie ID with a duplicate numeric part.	CND-28	The numeric part of the movie ID already exists in the system.
RQ-29	The system shall throw an exception when validating a movie ID with fewer than 3 digits in the numeric part.	CND-29	The numeric part of the movie ID has less than 3 digits.
RQ-30	The system shall throw an exception when validating a movie ID with non-digit characters in the numeric part.	CND-30	The numeric part of the movie ID contains non-digit characters.
RQ-31	The system shall validate a correctly formatted movie name.	CND-31	The movie name starts with a capital letter and contains only valid characters.
RQ-32	The system shall throw an exception when validating a movie name starting with a lowercase letter.	CND-32	The movie name does not start with a capital letter.
RQ-33	The system shall throw an exception when validating a null movie name.	CND-33	The movie name is null.
RQ-34	The system shall throw an exception when validating an empty movie name.	CND-34	The movie name is an empty string.

Integration Testing

1.Introduction

This four integration test classes, each targeting different integration levels: component-level integration (ReadIntegrationTest), I/O layer integration (IOLayerIntegration), controller-level integration with stubs (ControllerLevelTest), and complete end-to-end integration (EndToEndIntegrationTest). Together, these tests provide comprehensive coverage of system integration points and validate the seamless operation of the entire application.

2. Integration Testing Approach

Bottom-Up Integration: Starting with lower-level components (ReadMovie, ReadUser) and progressively integrating higher-level components (Database, WriteFile)

Top-Down with Stubs: Using stub data to test controller logic independently of file I/O

End-to-End Validation: Testing the complete application flow from input to output

3. ReadIntegrationTest Class

3.1 Overview

The ReadIntegrationTest class validates the integration between file reading components (ReadMovie and ReadUser) and their respective validation mechanisms. It also:

- Verify ReadMovie can successfully parse movie data files
- Verify ReadUser can successfully parse user data files
- Ensure validation logic is invoked during parsing
- Validate proper exception handling for file and validation errors

• Confirm printed output displays correct parsed data

Components Integrated: ReadMovie → ValidateMovie, ReadUser → ValidateUser

Integration Type: Component-level integration (File I/O + Validation)

3.2 Test Cases

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
RI-1	ReadMovie Integration	1. Create ReadMovie with movies.txt 2. Call getMovies() 3. Call printMovies()	Successfully parses all movies with validation, displays movie count and details	ReadMovie ↔ File I/O ↔ ValidateMovie	**Pass**
RI-2	ReadUser Integration	1. Create ReadUser with users.txt 2. Call getUsers() 3. Call printUsers()	Successfully parses all users with validation, displays user count and details	ReadUser ↔ File I/O ↔ ValidateUser	**Pass**
RI-3	IOException Handling	Provide invalid file paths and verify exception handling	IOException caught with appropriate error message	Error propagation from I/O to calling code	**Pass**
RI-4	Validation Error Handling	Provide files with invalid data format	IllegalArgumentException caught with validation error details	Validation layer integration with parsers	**Pass**

4. IOLayerIntegration Class

4.1 Overview

The IOLayerIntegration class tests the complete I/O layer integration by connecting file reading components with the database layer and file writing components. It also:

• Validate end-to-end I/O layer integration
• Ensure data flows correctly from input files to output file
• Test Database class integration with read and write components
• Verify movieSearch() and movieRecommend() process real data correctly
• Confirm WriteFile generates valid output from processed data

Components Integrated: ReadMovie → Database → WriteFile, ReadUser → Database → WriteFile

Integration Type: I/O Layer integration (Complete data pipeline)

4.2 Test Cases

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
IO-1	Read-to-Database Integration	1. Read movies and users 2. Set data in database 3. Verify data loaded correctly	Database populated with correct movie and user data	ReadMovie/ReadUser → Database	**Pass**

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
IO-2	Database Processing Integration	1. Execute movieSearch() 2. Execute movieRecommend() 3. Verify processed data	Search and recommendation logic processes real data successfully	Database methods with real data	**Pass**
IO-3	Database-to-Write Integration	1. Create WriteFile with database data 2. Call buildFile() 3. Verify output file created	integration_result.txt created with processed recommendations	Database → WriteFile	**Pass**
IO-4	Complete I/O Pipeline	Execute entire flow: Read → Process → Write in single test	Complete integration successful message displayed	Full I/O layer pipeline	**Pass**
IO-5	Error Handling Across Layers	Test with invalid file paths or corrupted data	IOException or IllegalArgumentException caught with descriptive message	Exception propagation across I/O layers	**Pass**

5. ControllerLevelTest Class

5.1 Overview

The ControllerLevelTest class validates the controller logic (Main class flow) using stub data instead of real files. It also:

- Test controller logic with controlled stub data
- Verify Main class flow without file I/O dependencies
- Validate Database integration with stub movies and users
- Test search and recommendation logic with known data
- Confirm WriteFile handles stub data correctly

Components Integrated: Stub Data → Database → WriteFile (Main flow simulation)

Integration Type: Controller-level integration with stubs (Top-down testing)

5.2 Test Cases

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
CL-1	Stub Data Creation	1. Create stub movies (Inception, Dark Knight) 2. Create stub user (John Doe) 3. Verify stub data valid	Stub data created with proper IDs, names, and genres	Stub objects creation and initialization	**Pass**
CL-2	Database Integration with Stubs	1. Create Database instance 2. Feed stub data into database 3. Verify data set correctly	Database populated with stub movies and users	Stub Data → Database	**Pass**

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
CL-3	Search Logic with Stubs	1. Execute movieSearch() with stub data 2. Verify search completes 3. Check console output	"Executing Search Logic..." displayed, search completes successfully	Database.movieSearch() with stub data	**Pass**
CL-4	Recommendation Logic with Stubs	1. Execute movieRecommend() with stub data 2. Verify recommendation completes 3. Check console output	"Executing Recommendation Logic..." displayed, recommendations generated	Database.movieRecommend() with stub data	**Pass**
CL-5	WriteFile with Stub Results	1. Create WriteFile with stub data 2. Call buildFile() 3. Verify output file created	stub_recommendation.txt created successfully, completion message displayed	Database → WriteFile with stub data	**Pass**
CL-6	Complete Controller Flow	Execute entire controller flow from stub creation to file output	"Controller flow test complete" message with file path	Full Main class flow simulation	**Pass**

6. EndToEndIntegrationTest Class

6.1 Overview

The EndToEndIntegrationTest class validates the complete application flow by executing the actual Main class and verifying the final output. It also:

- Test complete application from start to finish
- Verify Main class executes without crashes
- Validate final output file creation and content
- Ensure all integration points work in production flow

Components Integrated: All system components (ReadMovie → ReadUser → Database → WriteFile → Main)

Integration Type: End-to-End System Integration (Full application flow)

6.4 Test Cases

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
E2E-1	Pre-Test Cleanup	1. Check for existing recommendation.txt 2. Delete if exists 3. Verify clean state	Old output file removed, test starts with clean state	Test environment preparation	**Pass**
E2E-2	Main Class Execution	1. Execute Main.main(new String[]{}) 2. Monitor for exceptions 3. Wait for completion	Main class executes without throwing exceptions	Complete Main class flow execution	**Pass**
E2E-3	Output File Existence	1. Check if recommendation.txt exists 2. Verify file is accessible	recommendation.txt file created successfully	File system output validation	**Pass**

Test ID	Test Scenario	Test Steps	Expected Result	Integration Validated	Status
E2E-4	Output File Content	1. Check file length > 0 2. Verify file not empty	File contains data (length > 0 bytes)	Data written successfully to output	**Pass**
E2E-5	Success Confirmation	Verify both file existence and non-empty content together	"SUCCESS" and "End-to-end flow is seamless" messages displayed	Complete system validation	**Pass**
E2E-6	System Crash Detection	Catch any exceptions during Main execution	If crash occurs: "SYSTEM CRASH: Integration failed" with stack trace	Critical failure detection mechanism	**Pass**

7. Integration Testing Summary

Test Class	Integration Level	Test Cases	Pass Rate	Status
ReadIntegrationTest	Component	4	100%	**All Passed**
IOLayerIntegration	I/O Layer	5	100%	**All Passed**
ControllerLevelTest	Controller	6	100%	**All Passed**
EndToEndIntegrationTest	End-to-End	6	100%	**All Passed**
TOTAL	**All Levels**	**21**	**100%**	**All Passed**

Source Code :

Github :

https://github.com/eng-ahmedsaeed/Movie-Recomendation-System.git

Jira :

https://the-bug-assassins.atlassian.net/jira/software/c/projects/TBA/list?atlOrigin=eyJpIjoiZWI1ODUwNDQzNGZjNGFjYTkxMThlNjI5OTg3NGRlMzUiLCJwIjoiaiJ9

Conclusion

After going through a series of tests, we've reached a point where we are confident in its stability and accuracy. Our goal wasn't just to make sure the code "ran," but to ensure it could handle real-world scenarios and messy data without failing.

Through our White-Box testing, we cleared out the "blind spots" in our logic by tracking every possible branch and path the code could take. Meanwhile, our Black-Box testing acted as a safety net, confirming that even when users enter invalid IDs or unexpected names, the system remains resilient and user-friendly. Finally, seeing everything come together during Integration testing proved that our data flows—from reading files to suggesting genres—work as one cohesive unit.