URECHE ANDREEA MARIA

322CC

Difficulty: Hard

The application can operate in 2 ways , depending on how the user wants to use the application.

1) Terminal

2) Graphical Interface

1) Terminal mode

IMDB class:

The application uses the singleton design pattern for the IMDB class, ensuring a single instance is created and used throughout the application.It supports different user types including Regular Users, Contributors, and Admins. It implements a login system where users can authenticate with their email and password.

Upon running the application, users are prompted to choose between the graphical or terminal interface. After successful authentication, users are presented with options specific to their role, allowing them to interact with the application's features.

It initializes data from JSON files using the parsing methods , creating a structured database of actors, movies, and user information and allows users to execute actions like viewing, adding, or deleting information based on the selected option.

Next I will briefly describe some of the methods used in the IMDB class :

-> selectMode() : Prompts the user to choose between graphical interface or terminal mode .

-> initializeData() : calls the methods for parsing and extracting the data from the json files

-> run() : initialize the data from json files and implements the logic for calling the methods used for authentification and displaying options based on the user's role

->isValidUser : checks if the credentials entered by the user are correct

-> authenticateUser() : return the user type based on the credentials used

->displayOptions : display the options based on the user's role

->execute : executes the option by calling the appropriate method

-> viewProductionsDetails() : method used to display the productions details

->viewActorsDetails() : method used to display the actors details

->searchForActorMovieSeries() : method used to search for an actor/movie/series

->addDeleteToFromFavorites() : method used to delete or add a actor/movie/series from favorites

->createDeleteRequest() : method for creating or deleting a request

->addDeleteFromSystem() : method for adding/deleting a production/actor from the system

->addDeleteReview() : method for adding/deleting a review

->addDeleteUser(): method for adding/deleting a user

->generateUniqueUsername : method for generating an unique username

->generateStrongPassword() : method for generating a strong password

-> solveRequest() : method used for solving a request

-> updateActor() : method for updating actor's details

->updateProduction() : method for updating production's details

Actor class:

The Actor class represents an actor with various attributes and functionalities.

The class includes an inner class named Performance. This inner class is designed to represent a single performance, with attributes for the title and type of the performance.

Admin class:

The Admin class representing an Admin user, extending Staff, is designed to manage users and their contributions . It inherits attributes such as user information, account type, and experience from Staff, and includes specialized methods for adding or removing users (Regular, Contributor, Admin) to their respective lists within the system.

Additionally, the class maintains a list of users (addedUsers) added by an admin.

For content management, admins can add, update, or remove productions and actors. They also inherit and can override methods for handling requests and contributions.The class contains a method to transfer contributions from Contributor to admins .

Contributor class:

The `Contributor` class , extending `Staff` and implementing `RequestsManager`, represents the Contributor role with the ability to manage content and requests. Contributors can create and remove requests, which are then added or removed from their list of requests.The class inherits methods for managing contributions, including adding, updating, and removing productions and actors. Additionally, contributors have the ability to resolve requests, presumably by implementing specific logic tailored to the type of request.

Credentials class:

The `Credentials` class is designed to securely handle user credentials, specifically email and password. It employs the Builder pattern to facilitate the construction of `Credentials` objects. The Builder pattern is implemented through a nested `Builder` class, providing a fluent interface for setting the email and password. The class includes getter methods for both email and password, allowing read access to these private fields.

Episode class:

The `Episode` class represent an episode of a series or show. The class provides two constructors: one that initializes `episodeName` and `duration`, and a default constructor that allows for an empty episode instance. Additionally, the class includes getter methods (`getEpisodeName`, `getDuration`, and `getAdditionalDetails`) for accessing these private attributes.

Movie class:

The `Movie` class, extending the `Production` class, represents a movie. It includes unique attributes for a movie, namely `duration` (the length of the movie) and `releaseYear`.

Production class:

The `Production` class is an abstract class in this code, represents a production (movie/series) . It encapsulates attributes common to all types of productions, such as title, type, directors, actors, genres, ratings, average rating, plot, and a flag to indicate if it was previously rated. It includes functionality to add and remove ratings, check if a user has rated the production, and evaluate the production (notifying users who have rated it). The class also implements the `Comparable` interface, providing a method to compare productions based on titles and, if necessary, class names for differentiation.

ProductionComparator:

The `ProductionComparator` class implements the `Comparator` interface and is designed to compare two objects.

The `compare` method determines the order of the objects based on their names or titles. It uses the `getNameOrTitle` private method to extract the relevant string for comparison: if the object is an instance of `Actor`, it retrieves the actor's name; if it's an instance of `Production`, it gets the production's title. If the object is neither an `Actor` nor a `Production`, it falls back to using the object's `toString` method for a default string representation.

Rating class:

The `Rating` class in this code represents a user's rating of a media production and incorporates the Observer design pattern. It maintains a list of `Observer` objects, allowing it to notify these observers whenever a relevant event occurs. The class includes methods to add and remove observers, as well as a method to notify all observers with a given message. In addition to the observer-related functionality, the class holds specific details about a rating, including the `username` of the user who made the rating, the `rating` value itself and an optional `comment`.

Regular class:

The `Regular` class, extending `User` and implementing the `RequestsManager` interface, represents a regular user .This class maintains a list of `Request` objects, allowing regular users to create and remove requests. The user's experience is managed through an `ExperienceStrategy`, which is used when adding reviews to adjust the user's experience points. Regular users can add and delete reviews for productions, with logic to prevent duplication of reviews and to ensure experience points are only awarded for first-time reviews. The class also includes a private method to check if the user has already rated a specific production. When a review is added or deleted, the production is updated accordingly, and other users who have rated the same production are notified.

Request class:

The `Request` class in this code represents a request and implements the `Subject` interface from the Observer pattern. This class maintains a list of `Observer` objects and provides methods to add, remove, and notify these observers. The class encapsulates details about a request, such as its type (`RequestType`), creation date, associated actor name and movie title, a description, the username of the requester, and the intended resolver (`to`). The `Request` class also includes a method to resolve the request, either affirmatively or negatively, and notifies the request creator of the resolution.

RequestsHolder:

The `RequestsHolder` class is a utility class designed to manage and process user requests, particularly for administrators. It uses a static list, `allRequests`, to store and share requests among administrators. The class provides static methods to add and remove requests from this list. The `displayAllRequests` method prints all requests that have not been previously loaded (i.e., resolved or rejected), providing a summary of each request's information. It presents each unresolved request to the admin, who can choose to resolve or reject it. Upon resolution or rejection, the corresponding user is notified, and their experience points are updated if they are not an admin.

Series Class:

The `Series` class, extending the `Production` class, is representing a series. It includes series-specific attributes such as `releaseYear`, `numSeasons`, and a map `seasons` containing lists of `Episode` objects organized by season.

Staff class:

The `Staff` class, extending `User` and implementing both `StaffInterface` and `Observer` interfaces, represents staff members .

It manages a list of `Request` objects and a sorted set of contributions (using a `ProductionComparator` for ordering), where each staff member can add, modify, or remove productions and actors from their contributions. The class also observes and manages user requests, providing functionality to resolve them interactively and notify request creators of the resolution or rejection. Staff members can update details of productions and actors if they have the authority . The class supports the Observer pattern by registering staff members as observers for requests, enabling them to receive updates. The `updateExperience` method allows modifying a staff member's experience points, and the class maintains an `ExperienceStrategy` for managing experience-related operations.

User class:

The `User` class, an abstract class implementing both `Observer` and `Subject` interfaces, represents the general User type (Admin,Contributor,Regular) . It contains user-specific details encapsulated in the nested `Information` class (using the Builder pattern for instantiation), and includes attributes like account type, username, experience points, notifications, and favorites (sorted using a `ProductionComparator`). The class also manages observers for the Observer pattern, allowing users to receive notifications and update their notification list. The `User` class is equipped with methods to manage user actions and experience points using an `ExperienceStrategy`. It provides functionalities to add, remove, and print favorite items, and to update user experience based on performed actions. Additional features include methods for user authentication (login/logout) and account management.

UserFactory class:

The `UserFactory` class in the provided code serves as a factory for creating `User` objects of different types in a media content management system. It employs a generic factory method, `createUser`, which takes parameters common to all user types, such as user information, account type, username, experience, notifications, and favorites. The method uses the `accountType` parameter to determine the specific type of user to create – `Regular`, `Contributor`, or `Admin`. Depending on this type, it returns a new instance of the corresponding class (`Regular`, `Contributor`, or `Admin`), each of which extends the `User` class.

Interfaces:

-Subject:

Contains methods for:

-adding an Observer

-notifying the Observers

-StaffInterface:

Contains methods for :

-adding a production in the system

-adding an actor in the system

-deleting a production that has been added by the user from the system

-deleting an actor that has been added by the user from the system

-updating the details for a production that has been added by the user from the system

-updating the details for an actor that has been added by the user from the system

-solving the requests given by the users

-RequestsManager:

Contains methods for creating and deleting a request.

-Observer

Contains a method for updating the notification of a user

-ExperienceStrategy:

Contains a method for calculating the experience the user gets.

Enums:

-AccountType

Contains the type of users.

-Genre

Contains all the movie/series genres

-RequestType

Contains all types of requests a user can make

2) Graphical Interface

The IMDBGUI class is a graphical user interface (GUI) component built using Java Swing.

The class initializes with a main JFrame (mainFrame) which acts as the window for the application. It contains various JPanel components such as loginPanel, homePanel, and actorsPanel, each serving a specific purpose in the UI. The loginPanel is used for user authentication, homePanel serves as the main dashboard after login, and actorsPanel is dedicated to displaying actor-related information.

User Authentication: The initializeLoginPanel method sets up the user authentication interface. It includes text fields for entering username and password and a login button. This panel is the first interaction point for the user, guiding them to input their credentials to access the application's features.

Home Panel and Content Browsing: Once authenticated, users are navigated to the homePanel, which acts as the central hub for content browsing. It includes a search bar, genre filter (using a JComboBox), and a recommendations list. Users can search for movies, series, or actors, filter content based on genres, and view recommended productions.

The actorsPanel provides a focused view on actors, listing them and allowing users to view detailed information about each actor. This panel enhances the application's functionality by dedicating a section to actor-related content.

The GUI dynamically renders content based on user actions, such as displaying actor details or updating the recommendations list. This dynamic rendering is achieved through methods like viewProductionDetails and updateRecommendations.

It handles images and graphics, as seen in initializeHomePanel, where it sets a background image and resizes it to fit the panel.

The class includes various event handlers to respond to user actions like button clicks and list selections. Actions like adding to favorites, searching, and logging out are all handled.