COMPSCI 235 – Assignment 2 Design Journal

Alexander Khouri – 6402238 – akho225

Changes to Existing Domain Model

`Movie` Class:

Added the following attributes:

`movie_reviews`	List that stores reviews. Initialised by `MovieFileCSVReader` when data is loaded from CSV file. Property/setter = `reviews`.
`movie_rating`:	Float that stores the overall rating. Initialised by `MovieFileCSVReader` when data is loaded from CSV file. Property/setter = `rating`.
`movie_votes`:	Integer that stores the number of ratings. Initialised by `MovieFileCSVReader` when data is loaded from CSV file. Property/setter = `votes`.
`movie_year`	Attribute already existed, but added `year` property/setter.
`movie_ID`	String consisting of the title and year concatenated (with all whitespace removed), used to provide a unique identifier for each movie that can be easily interpolated inside Jinja/HTML. Initialised by constructor. Property/setter = `ID`.
`movie_review_count`	Integer recording the number of reviews added to a movie. This is used inside Jinja templates as a substitute for the `len()` function (which isn't supported by Jinja). Property/setter = `review_count`.

Added the following functions:

`add_review`	Adds a new review for a movie and updates `movie_rating` using a simple moving average. This ensures that all reviews are weighted equally regardless of their age.
`remove_review`	Removes a review from a movie and updates `movie_rating` using a simple moving average.

`Review` Class:

Added the following attributes:

`review_user`	User object that refers to the comment's author. Initialised by initialiser function. Property/setter = `user`.
`review_date`	String of the date on which the review was created (used for display within the website). Initialised by constructor. Property/setter = `date`.

`User` Class:

Added the following functions:

`add_review`	Adds a `Review` object to the user's `Watchlist` object.
`remove_review`	Removes a `Review` object from the user's `Watchlist` object.

`Actor` Class:

Added the following attributes:

`actor_movies`	A list of all movies that the actor has been in. Property/setter = `movies`.

Added the following functions:

`add_movie`	Adds a movie to the list of associated movies.
`remove_movie`	Removes a movie from the list of associated movies.

`Director` Class:

Added the following attributes:

`director_movies`	A list of all movies that the actor has directed. Property/setter = `movies`.

Added the following functions:

`add_movie`	Adds a movie to the list of associated movies.
`remove_movie`	Removes a movie from the list of associated movies.

`Genre` Class:

Added the following attributes:

`genre_movies`	A list of all movies that fall under this genre. Property/setter = `movies`.

Added the following functions:

`add_movie`	Adds a movie to the list of associated movies.
`remove_movie`	Removes a movie from the list of associated movies.

Miscellaneous:

- Moved import statements inside domain model files into lowered __name__ == “__main__” checks to prevent circular import errors. This enabled each class file to import all necessary modules when run individually (e.g. for testing purposes), without conflicting with the cumulative module importing that occurs when running the application from its root directory.
- Safety checks were added into each function to prevent users from modifying object variables that should only be modified by class functions (e.g. `reviews_count` from the `Movie` class, which should only be updated by the `add_review` function).

New Design Feature: Watchlist

I implemented this feature using the `Watchlist` class, which includes `__iter__` and `__next__` functions to make it iterable. Watchlists are associated with `User` objects via one-way 1-to-1 relationships (i.e. users have references to watchlists, but not vice versa). This design decision was made because there's no functional need to obtain user references from watchlists within any layer of the application.

A section of the main page is dedicated to displaying watchlists, and simply remains empty if there's no active watchlist (i.e. no user is logged in, or the current user's watchlist is empty). This decision was made to provide consistent formatting of the main page, whose sections would otherwise fluctuate in size if visibility of the watchlist column was toggled.

Users can add movies to their watchlists by simply pressing the 'Add to Watchlist' button underneath each movie in the main page. Conversely, they can remove movies from their watchlist by clicking the 'Remove' button under each movie in the list. The movies displayed in this list initially have their details hidden, but they can be revealed using a 'Show Details' button; this reduces clutter within the watchlist column (which is important given its reduced area).

Website (View Layer):

In the interest of personal education, I elected to write my entire website from scratch without using any components from the example Covid-19 application. While this meant my assignment took much longer to complete, it also provided me with a more comprehensive learning experience.

The entire website consists of one page, which is broken up into different sections for each feature. The two main sections are the banner (which contains the logo, search function, and authentication buttons, and the main window (which contains the browsing column, the main movie list, and the user's watchlist). This creates a clear distinction between simple button interfaces and those containing lists of data (e.g. the browsing window). The banner also has a 'sticky' position style, which ensures that users always have access to basic website functions.

By default, all movies are displayed in the main movie list (which utilises pagination to reduce the need for scrolling), and this list is then refined using either the browsing feature or the search features. The browsing section contains a list of the attributes that can be used to filter movies, and clicking on each attribute opens a list of the corresponding values that be chosen for filtering movies. Two design features were implemented in order to reduce clutter in this area:

1.) Only one attribute can be open at a time (i.e. opening a second attribute results in the previous one being closed)

2.) The list of attribute values resides inside a fixed-size box that uses scrolling to handle overflow

Each movie in the main list has an 'Add to Watchlist' button (as mentioned in the previous chapter of this report), followed by 'Show Reviews' and 'Add Reviews' buttons. The former of those displays all user-generated reviews for the website in left-aligned format (or a notification message if there are no reviews), while the 'Add Review' button opens a centre-aligned form where users can enter their reviews. This variation in alignment creates a distinction between reviews that are being written and reviews that are being read.

Flask Application (Service Layer):

HTTP requests from the application's website are handled by a Flask application, uses a variety of request handlers for different URL endpoints. Because the authentication pages are the only ones displayed separately from the main movie list, I decided to include all the website's sections on one page and adjust their visibility using hidden variables. These variables are passed between the client and server using a Flask `session` object, which ensures secure transmission of authentication status. However, complex data types can't be serialised into these Flask objects (e.g. lists of movie objects), so these are processed separately inside each request handler using a dictionary called `clientData` (which is passed via Flask's `render_template` function).

In addition to this, common server data is contained inside another dictionary called 'servData', which is also passed to clients using the `render_template` function. This significantly reduces the number of variables that are passed into each function call, which streamlines the code. Lastly, the Flask application contains a function that ensures newly-registered passwords are at least 8 characters long and contain both upper and lower-case letters.

Server (Memory Repository):

The basis of my memory repository is the `MemoryRepo` class, which is a simple abstraction layer that resides between `MovieFileCSVReader` class and the Flask service layer in my application. The class involves a relatively featureless implementation of a repository, but is designed to provide an easily swappable abstraction layer that can later be replaced by a database repository if necessary.

When the `MemoryRepo` object is initialised by Flask, it creates a `MovieFileCSVReader` object, reads the movie database CSV file, then stores each of the relevant datasets into its attributes. It also initialises a list that contains references to all registered users.

The class definition contains the following functions:

`add_user`	Adds newly-registered users to the repository (unless the username is already registered). This function accepts `User` objects.
`remove_user`	Removes a user from the repository. This function accepts `User` objects.
`get_user`	Returns the `User` object that corresponds to a username string (returns `None` if the user doesn't exist).
`get_movie`	Returns the `Movie` object that corresponds to a movie title string (returns `None` if the movie doesn't exist).
`get_watchlist`	Returns the `Watchlist` object that corresponds to a username string (returns `None` if the user doesn't exist). This function was added as an alternative to retrieving watchlists directly from `User` objects.