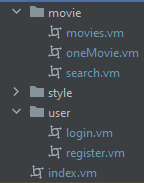
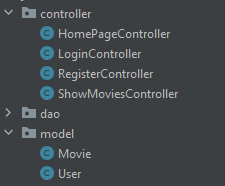
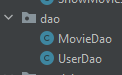
2.1B PART A

Which architectural patterns did you implement and how did you do it?



*Figure 1.1: Screenshots showing the implementation of the MVC pattern*

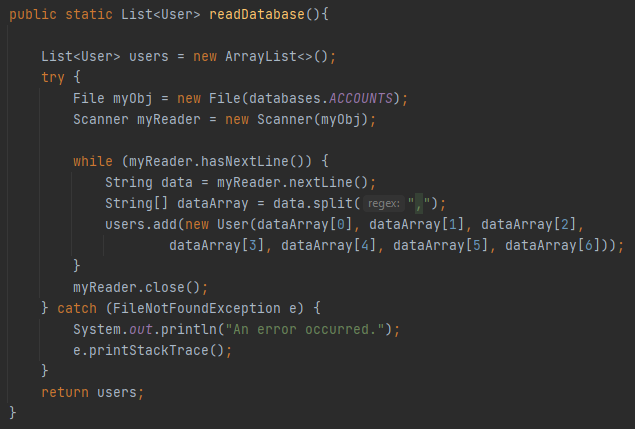
**

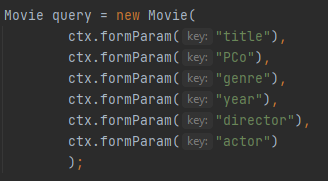
*Figure 1.2: Screenshot showing the DAO packages*

One of the fundamental patterns we implemented was the MVC architecture pattern, where the models were implemented as classes/Java Objects. In our implementation the two models were Movie and User. The movie class holds 6 fields, the title of the movie, the Production Company responsible for creating the movie, the genre. The release year, the director and the star actors as well as the appropriate getters and an additional helper method to place the actors into an array. The second model was a users class, which holds the username/email, password, first and last names, gender, date of birth as well as country, like the Movie model, this also contains the appropriate getters. As for the views, these were implemented in HTML/java velocity. There are five views in total, one for each of the four features we implemented (logging in/logging out, registering for an account, displaying every movie and searching using different parameters) as well as a fifth central homepage. As for the controllers, we had four in total, one for the rendering of the central page, as well as one controller for handling logging in/logging out, one for registration, and one for displaying a collection of movies. Since displaying all the movies and displaying the search results for a list of movies are similar functions, we grouped them into a singular controller. In order to accentuate the use of the MVC pattern we placed the models and the controllers in separate java packages, and the views were placed in folders found in the “resources/public” directory, and are denoted by the “.vm” file extension. Our implementation was also complimented by the DAO pattern, which we used to deal with the collection of movies and accounts. Regarding the movie collection we used the dao pattern to read, store and return an iterable movie collection, as well as search by a specific attribute, ie Director, Year .. etc, and thus the DAO pattern directly used to implement our search function as well as the display all movies function. The UserDao is very similar since it stores, updates and returns a collection of users, but contains additional functionality in the form of helper methods that validate whether a user’s credentials are correct or not, and so the login function in the controller is dependent on this method. Furthermore this class also adds new users to the accounts database, which is how the registration is completed, so the DAO pattern is also used for account creation and the process of logging in/out.

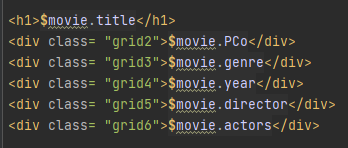
2.1B PART B

Which architectural patterns you did not implement and why?

*Figure 2.1: Snippets of code that would have been in the data tier,  had we not chosen csv.*

**

*Figure 2.2: An example of passing data from HTML to java*

**

*Figure 2.3: An example of passing data from java to HTML*

One of the patterns that we chose not to implement was the n-tiers architectural pattern,  this is because we chose to store the Accounts and Movies data in csv files, and java’s inbuilt string methods such as “split” were sufficient for reading the database. Thus, regarding the 3-tier architecture we did not need to have a separate Data Tier to access the data, since this is simply achieved through the java code, which served as somewhat of an application layer, the “.vm” files making up the front end could also be regarded as a presentation layer, ultimately the approach we decided on taking was functionally correct, although due to the lack of a data layer it is not valid to say the approach we have taken properly follows the 3-tier architecture. Since the 2-tier architecture is also composed of a data tier and presentation tier, and we’ve chosen to deviate away from using a data tier, it means that we also haven’t followed a 2-tier architecture either. However we did end up taking inspiration from the n-tiers patterns, as we have taken what would have normally been in the data tier, and incorporated it into each java file in the dao package, (see Figure 2.1 for an example of this).  Likewise we also didn’t explicitly use the DTO pattern, this is because our program consists of a presentation layer and a combined data/application layer which both maintains the database, and carries the logic, since we were using Javalin, it allowed us to quite easily pass data between the two layers, that is the .vm files and the java code, and so we didn’t see the need to complicate the transferring if objects between layers. Figure 2.2 depicts how for the search form we were able to create a new movie by collecting the user input. This seems to be the most efficient way to handle user input, likewise Figure 2.3 demonstrates the opposite of this situation, that is passing data from java to html. This snippet is taken from the displaying movies html code, and it shows how we were able to output a movie object that was created in Java through html. Since it seemed effective to use Javalin to pass data between the presentation layer and application layer, so we elected to use Javalin’s inbuilt functionality to achieve a similar outcome to implementing a DTO pattern.