CSCI222 Assignment 2 – Group 5

e-Movies – Elaboration and Construction

Glenn Harper, Ian Mckay, Michael Boge, Tiaki Rice

Contents

Introduction

This report details the development process used by us, group 5, to produce the eMovies application.

Overview

The eMovies project declared was to develop an application that stores and provides appropriate uses for movie information (such as title, year of release, directors, genres, etc).

Our application was developed to fulfil all of the functional requirements needed to perform this task, namely it:

- Allows this information to be easily and quickly accessed in an ordered fashion, by storing and accessing the information using a SQLite database.
- Provides access only to relevant individuals, who must login with their respective username and password. This information is also stored in the database in a User table, where the password is stored in a Caesar cypher encrypted form for security.
- Allows administrators to add, edit and delete the users who can access this application, by modifying the User table
- Allows users to add, edit and delete the movie information stored if they have the proper privileges
- Allows users to search through this information by any part of it (i.e. one can search for titles, or one could search for a genre and director, or other combinations)
- Can import movie information from external files (from IMDB, or another place if in the same format), also allowing these imported records to update the previously stored ones
- Is very simple to use, even for people lacking training, due to its console-based interface

Our development process was executed with full regard to fulfill all these requirements, focusing on fulfilling a few of these functionalities in each iteration.

The following sections further describe these requirements of the specification, and how we aim to fulfill them.

eMovies Vision Document

Introduction

eMovies is a console based application which uses a database to allow users to organise and store large amounts of movie information. It is intended to allow for users to search through this data, and to allow administrators to create, change or delete the records.

Business Needs/Requirements

Currently staff working in movie related businesses (such as rental stores) do not have suitable methods of storing and maintaining large amounts of movie information.

In order for this information to be managed easily a computer application is to be developed to store the information on these movies (such as title, release year, genre, etc). This system must specifically be able to:

- Store all this information (with reasonable size/time constraints)
- Store relevant information appropriately (such as genres and directors, and ensuring it is of the correct type and not a duplicate)
- Allow users to create, edit and remove movie records
- Allow users to search through this information
- Import movie information from internet repositories of movies (like IMDB)
- Only be accessible by suitable users (staff/users and administrators), blocking others by using a password
- Allow administrators to define these users

All of this must be wrapped in an easy to use application.

Product / Solution Overview

eMovies is a console-based application which manages and provides access to large amounts of movie information by using a SQL database (SQLITE specifically). It shall run from a number of simple and easy to use menu screens/forms, in order to allow users to create, edit, remove and search for movie records. These forms will construct SQL queries which perform the required actions.

The application will be able able to import movie records from specific files (from IMDB), namely movies.list and genres.list, and add them to the database.

The database itself is only accessible after logging into the application (based on defined users in a database table, with encrypted passwords). These users will have specific permissions depending on if they are a standard user or an administrator (admins being able to create/edit/delete users).

Major Features – Interface Design

Due to the console-based nature of the application its specific features are less defined than a window based program (where each window is a feature), however it has a number of parts which represent these features:

- **Main Menu:** This is where the user is taken to after they logon. It presents a list of all the activities that use can perform and calls that activity's respective methods. The user makes a choice by entering the relative number for an activity (or Q to logout). The choices displayed is different depending on the user type (administrators or regular user) to represent their respective privileges.
- **User Details Menu:** This is only available to the administrators, to add or change user information. It again presents a menu with choices as to what the user wishes to add. The choices are selected through entering their respective number. The user is then prompted for that piece of information. Finally an option is provided to add/change the entered information.
- **User Details Menu:** This is only available to both types of user, to add or change movie information or to provide what a search is to be performed on. It again presents a menu with choices as to what the user wishes to add. The choices are selected through entering their respective number. The user is then prompted for that piece of information. Finally an option is provided to add/change/search for the entered information.
- **Search Results:** These are displayed in segments of data relevant to each movie after calling a function which displays.

Scope and Limitations

This product will cover all of the requirements mentioned above (and in the specification), its main focus will be on providing the user with an easy method of adding, changing and searching through movie data.

In terms of limitations, it has a few mainly due to this strict focus and the lack of ability to expand:

Movies, their related data and users must be of a specific format (for example movies must always have a title and release date).

Also, this means specific related data (such as Actors) that is to be held, cannot be easily added into the system

Other Needs

Backend

The user must have SQLite installed on their machine, as the application interfaces with this to access its information, which is kept in a SQL database. This also means the user must be able to cope with the SQL overhead.

Problem Statement

The problem of	Organising and managing movies to allow for easy searching and editing (as well as importing movie information from databases)
affects	the management staff of movie related companies (such as video stores). Specifically, it makes it difficult for them to manage large numbers of movies effectively;
the impact of which is	the staff must deal with a disorganised group of movies, which is difficult and time consuming to search through, and even more so to edit/change (which can result in incorrect data, duplicates and other issues).
A successful solution would be	a system which organises the movie information in a way that the staff can easily browse through it and change. Through this the staff would be able to keep more accurate movie information and be able to use it more effectively.

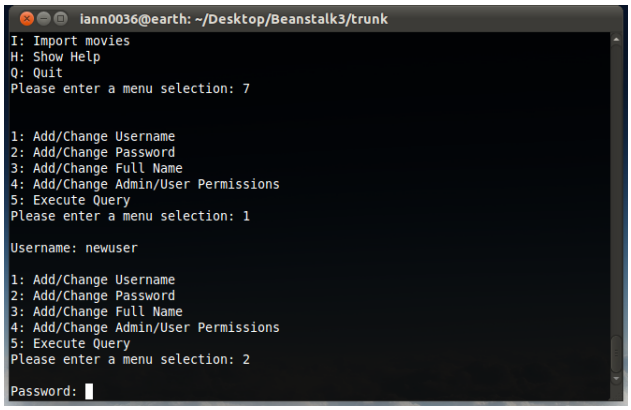
Position Statement

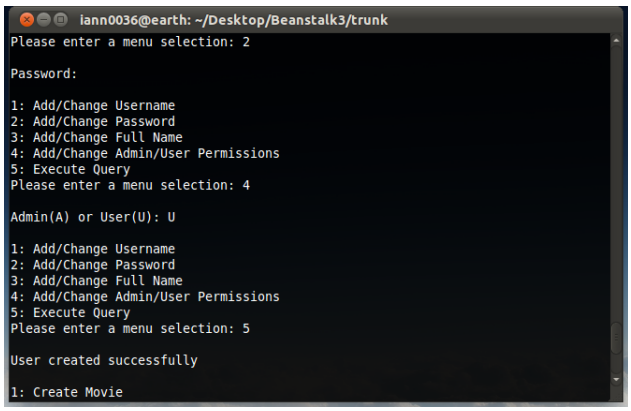
For	companies related to movies
who	need a method to manage large amounts of movie information, so it can be effectively searched and edited
eMovies is a	simple database (SQLite) application
that	stores information on movies and provides menu based methods for searching and editing records.
Unlike	the alternative of storing the movie information in a non-database format and trying to keep track of it manually,
our product	stores the data in a database of concise and formal format, and provides functionality for the users to search through the data and edit appropriate records. Which prevents invalid/out-dated/duplicate data from being stored.

Presentation of Functionality

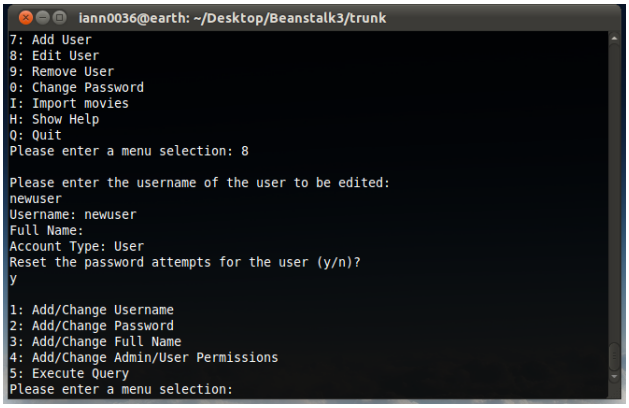
This section summarises and displays the full functionality of the program, using a series of screenshots.
This application implements the full functionality for the project, as it meets all the functional requirements laid down by inception/specification (from high to low functionality). This means the completed system:

Allows for the creation, editing and deletion of a user
All of these are provided as appropriate options for an administrator, by entering the appropriate data, the data to add for creation, which record to edit and how to edit it, or which user to delete. The user data (username, password, full-name and privileges) is stored in the User table of the database. The following screenshots show the creation, editing and deletion respectively.

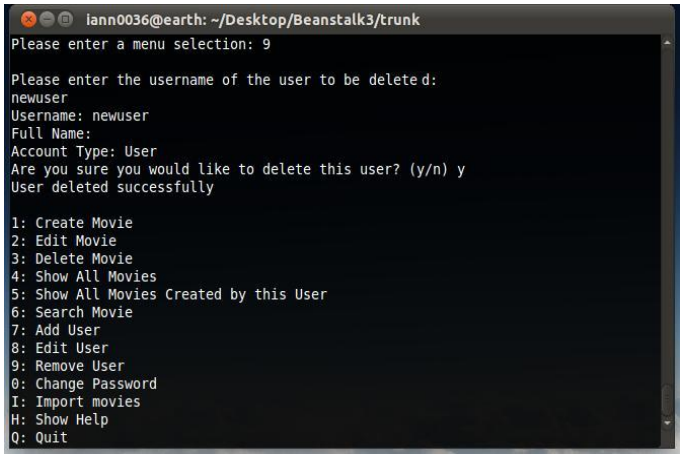




Here after prompting the admin's main menu to go to the add user method, it prompts the user for details on the user to create. The screenshots here showing the prompting for data, and the successful creation of a user, after executing query.



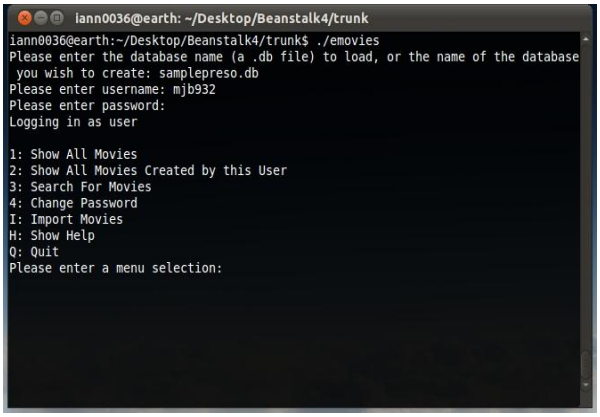
Editing a user works similarly, except the user is prompted for the user they wish to edit before getting to the form for entering details. Note this screenshot showing the prompt before the form, and the edit user function asking if it is to reset that users password.

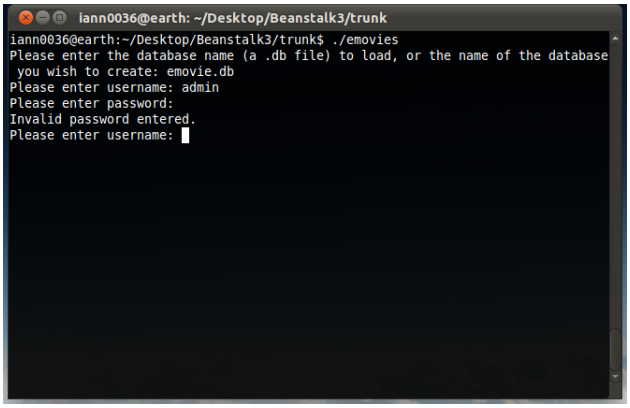


Delete user is different as it doesn't access the menu for input it simply prompts for the username of the user to be deleted, and then confirms this choice before deleting, as is shown in the screenshot.

Access provided to the system by logging in using a username and password

The entered data is compared against the data held in the User table, if a match is found the user is logged in, else an message is generated.

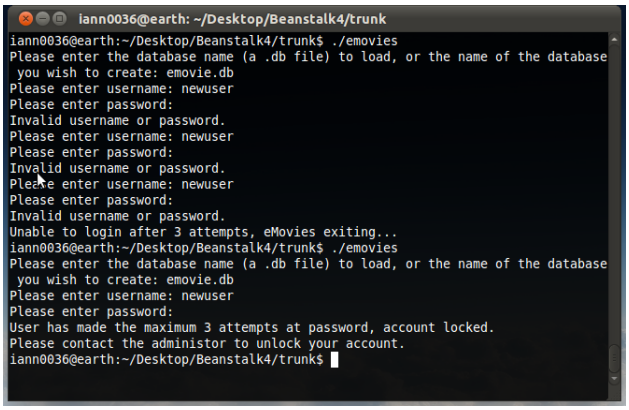




This is run before the menus are shown, and simply prompts the user for a username and password, then either logging in or return an error message. Above are the screenshots showing a successful and unsuccessful login respectively.

If a user is unable to enter the correct password after 3 tries that account is locked

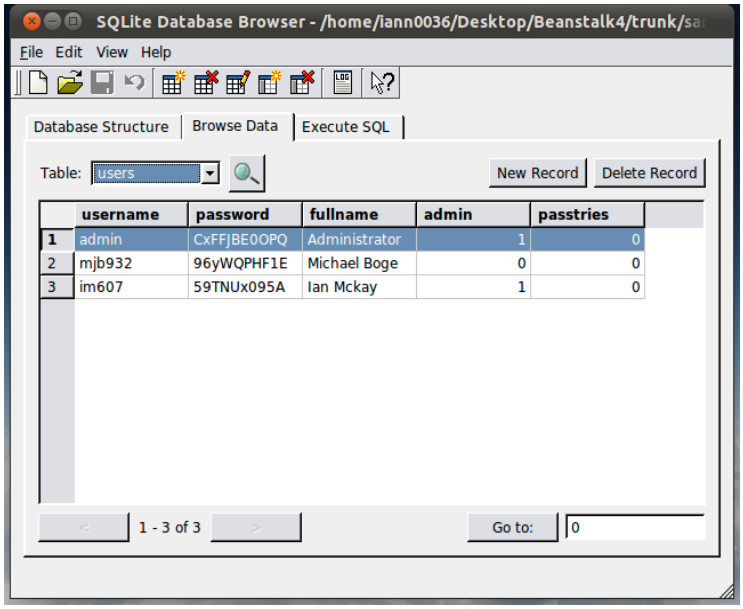
The number of password attempts for that account is stored in the database and is modified on an attempted login (either successful which resets it, or unsuccessful which increments it). Any more attempts on that account will return the "Locked" error message.



This automatically happens if the user fails to enter their password after 3 attempts. As the screenshot shows it will report your account is locked, that you should contact an administrator and then quit the program.

Encrypts the passwords Caesar cypher

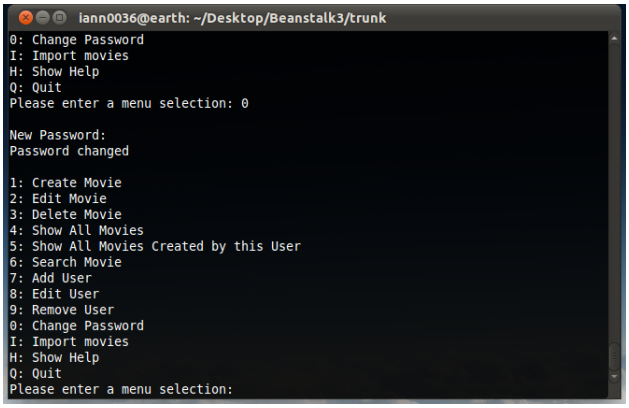
These passwords are entered by the user and immediately encrypted, this means they are passed and stored in this encrypted state for more security



This is done automatically every time the user enters a password. As can be seen in the screenshot of a view of the actual database, the passwords are encrypted (using the Caesar cypher in Interface), with admin's real password being password123, which is used to login (as it can't decrypt the stored password).

Allows the user to change their password

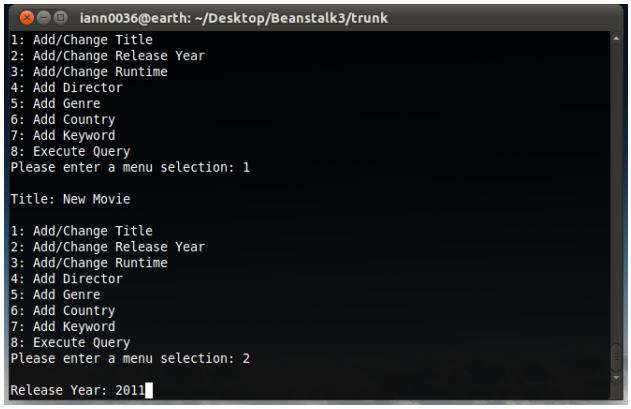
This is an option on the main menu, it will check the new password for validity and encrypt it as appropriate before updating and storing.

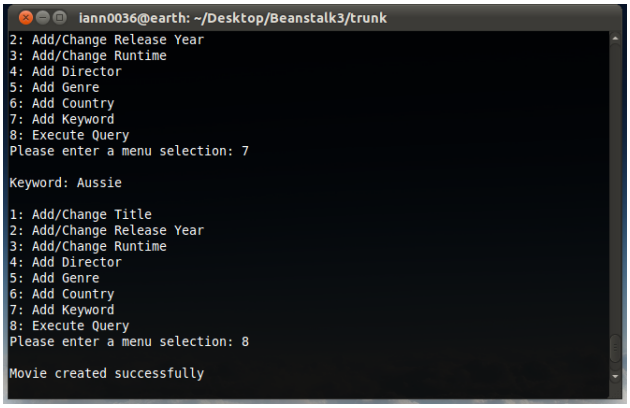


Simply done by asking to change password in the main menu, then entering the new password (which must be valid). The screenshot shows a successful change.

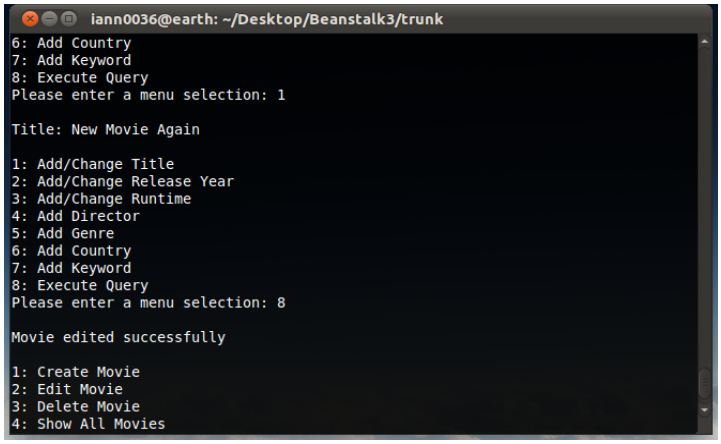
Allows the creation, editing and deletion of a movie

The information on the movie is stored in the database in multiple tables with appropriate links (to allow for example multiple directors for one movie). The user can access these functions from the main menu and a SQL query will be generated from the movie data entered to perform the requested action.

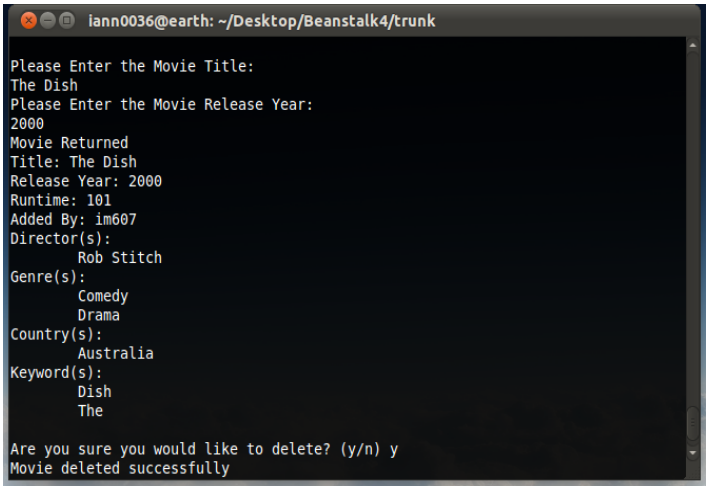




Similar to the add, edit and delete user functions except that the user enters movie data. Here the user selects the appropriate operation from the main menu, where they are then redirected to the movie data form menu, where they can enter movie information. The above screenshots show the add functionality, with the user entering information and then executing the successful creation query.



The edit works similarly, except it prompts the user to enter the movie to be edited first, then allows data to be changed/added through the other form. The screenshot shows a successful

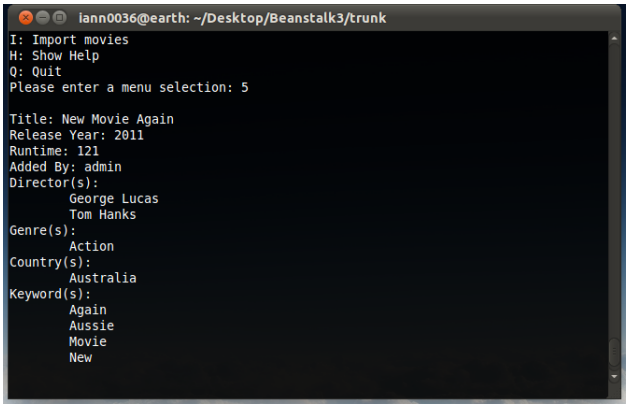


operation updating the title of the previously created movie.

The delete only prompts for the movie to be deleted (asking for title and release year). It then asks the user to confirm their delete, as is shown in the screenshot.

Keeps track of who creates a movie

The username of this user is stored with the movie when it is created, it can then be searched for to see all movies created by that user (this type of search is again accessible from the main menu)



This is done in the create method, the created by value is set to the current user. This can be seen in the example where the user 'admin' is searching for movies they've added, namely the film shown (there are others which are not as they have been added by other users). Also note it is showing the 'Added By: admin', which shows this is stored in the movie information.

Can search through and display movies created by all users or movies only created by the current user

The application provides full access to the movies table in the database, so any user can access all the movies, regardless of who they are created by, simply by calling show all or a search. For screenshots of this see the searches displayed in this section.

Alternatively, it can also only show movies created by the current user, as it simply searches for movies which match this created by username, which is performed by selecting the "Show all movies created by this user" function from the menu. See the screenshot from the above section for a presentation of this.

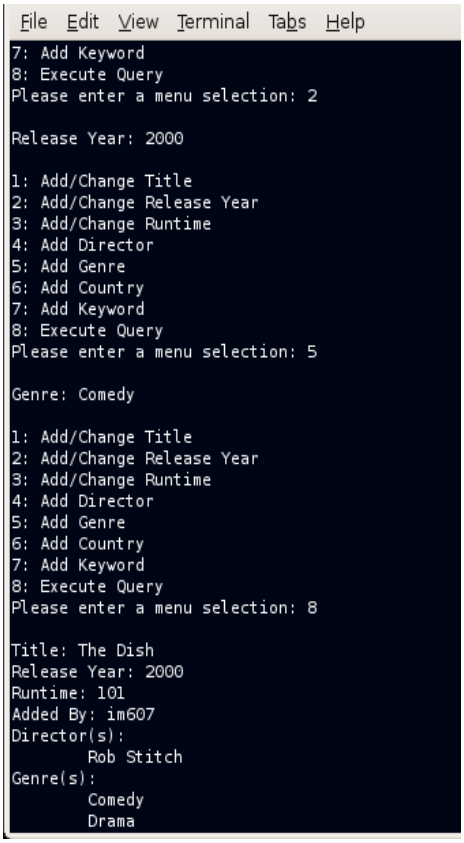
Displays movie information in alphabetical order

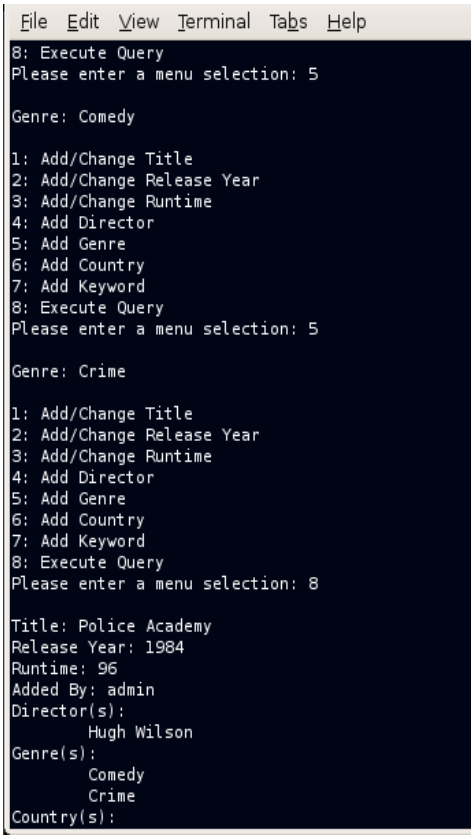
The search information is displayed in alphabetical order based on the title

This is done automatically by the search function, it sorts it into order, using the SQL Order By function, the screenshot here shows a sample, with its movies in alphabetical order.

Allow for searching

The movie data can be searched based on any combination of criteria, such as title, keywords, etc.

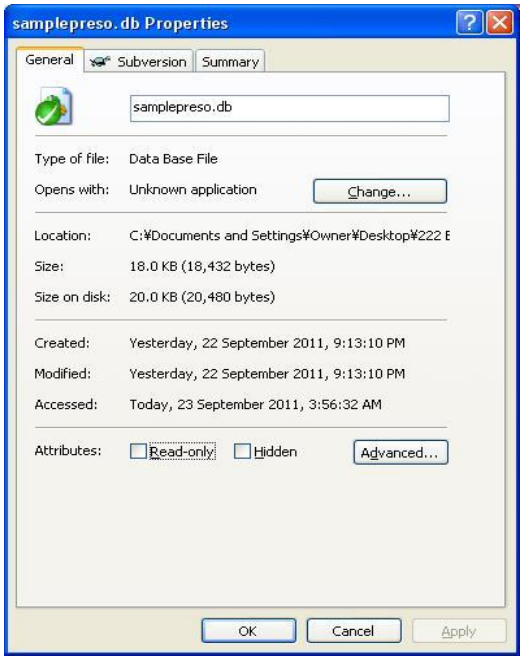




The user searches by selecting search from the main menu, they are then directed to the movie data form menu, where they enter the criteria for the search, and then run it. The screens above show examples of this searching, one for both year and genre, the other for two different genres in one movie.

Store the information in a file or multiple files

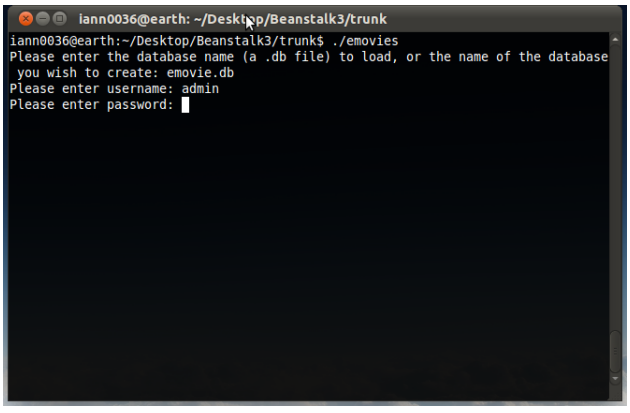
The application's database holds all the information and is stored as a .db file for loading. Both the user data and the movie data is stored in the same file, but the information for each is in separate database tables.



The eMovies profiles are all stored in .db files, this screenshot shows an example of one, namely the one that was used for our presentation.

Allow for the loading of previously saved eMovies files

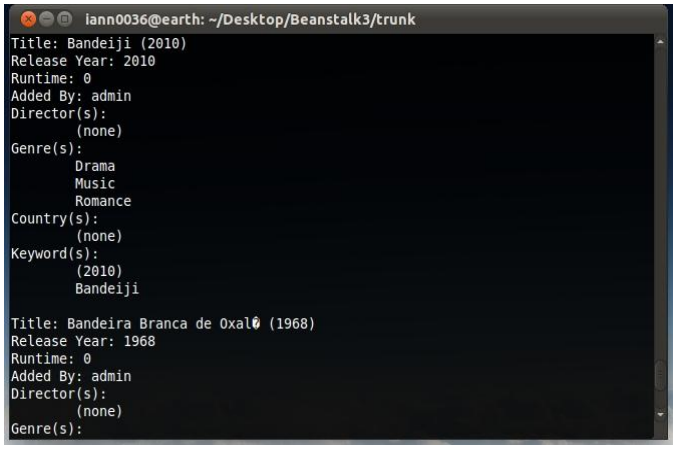
The application loads a user specified database (i.e. the saved eMovie profile) at runtime



Here, is a screenshot showing after the user has been prompted for which database to load from, in this case 'emovie.db', which has loaded and is now prompting for a username and password

Allow the importing of data from 2 files from IMDB

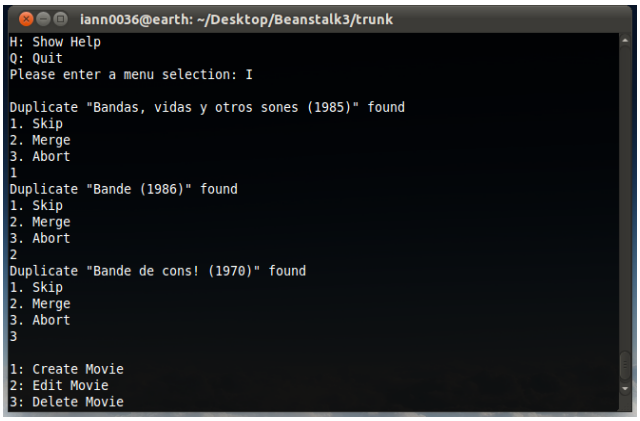
The application can read the two files if requested and load the read data into the database as it reads (this includes updating the read movies to be linked to all the read genres).



The import is performed by the user selecting 'I' in the main menu, which then executes until all movies and genres have been read in and added to the database. This screenshot shows the result of the import, with some of the files that were imported now being shown in the database.

Detect duplicates on import and act accordingly

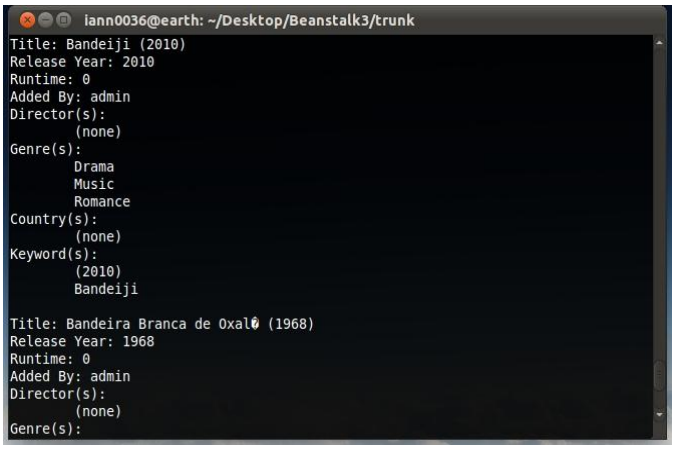
If a duplicate record is detected on import, the user is given options to merge (updating the old data), skip that record, or abort the import altogether.



When a duplicate record is discovered the user is then prompted for a choice to either skip, merge or abort import. The screenshot above shows the user being prompted when these duplicates are hit. The second choice, for 'Bande (1986)' where merge was chosen, will result in the movie being shown as updated to to include the imported information.

Extract keywords on import

When a title is read in (both regularly and on import), it is automatically broken up into keywords



This screenshot shows the automatically generated keywords for the film, 'Bandeiji (2010)', with the keywords being 'Bandeiji' and '(2010)' respectively, for longer titles there will be more keywords generated. The keywords will also be checked against common words like 'the' and only stored if they are not one of these.

For Non-Functional Requirements

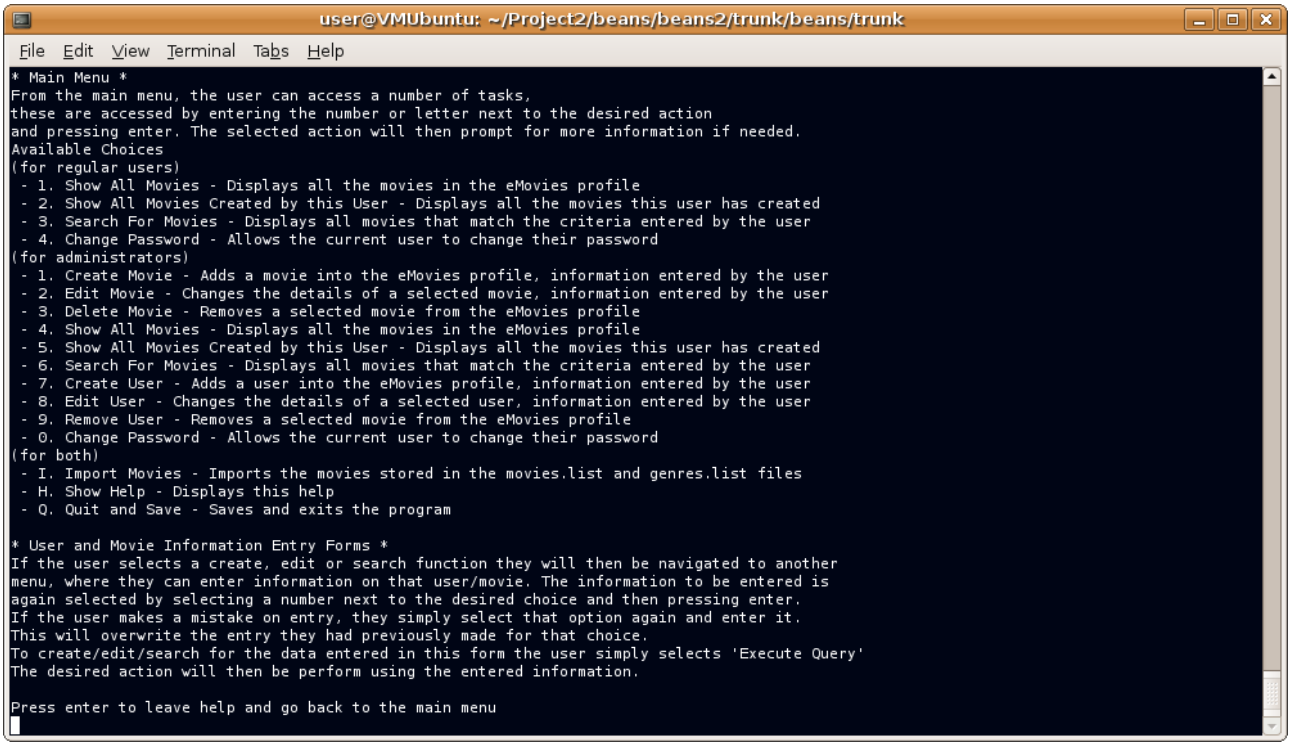
Application is less than 5MB

Since our application is text and console-based, it is very small, being only a few hundred kilobytes.

Screenshot showing final application file of size 142KB

Provide a help function

The application provides a help function which describes the operations that can be performed, and how to perform them.



Screenshot showing the help function, after calling it from the main menu using 'H'. Note it is displayed until the user presses enter to return to the menu.

Pre-planning

This stage was not specifically an iteration, as it did not involve constructing and actually implementing components/functionality, but it was important as it determined how all the work would be carried out in the following iterations.

In this stage we:

- Made note of the functional requirements, so as to see what the application needed to do
- Decided upon major design considerations, such as what type of interface to use (we chose a text-based console)
- Tested out sub-version, so we could use it effectively in the future
- Tested basic commands and methods used in the SQLite library, so we knew how to use them in the next iteration.

As this stage had no actual implementation, there was no real need for testing. The tools designed in this stage were created during meetings, when the whole group could discuss them, and were saved on everyone's computers/books (as we created them together), so they were not published yet on SVN (this was also as we did not all have the same software for creating/reading the diagrams)

Design

Use Cases

The design task we undertook during the completion of the eMovies project was to analyse the functional requirements in order to come up with a range of use cases that sufficiently covered all of the requirements. Initially we devised 12 use major use cases and two actors which we believed covered all of the major functionality required in the system. The use cases associated with the user actor were:

- Login
- Change password
- Display user's movies
- Display all movies
- Search movies

The use cases associated with the administrator actor, which is a specialisation of the user actor, were:

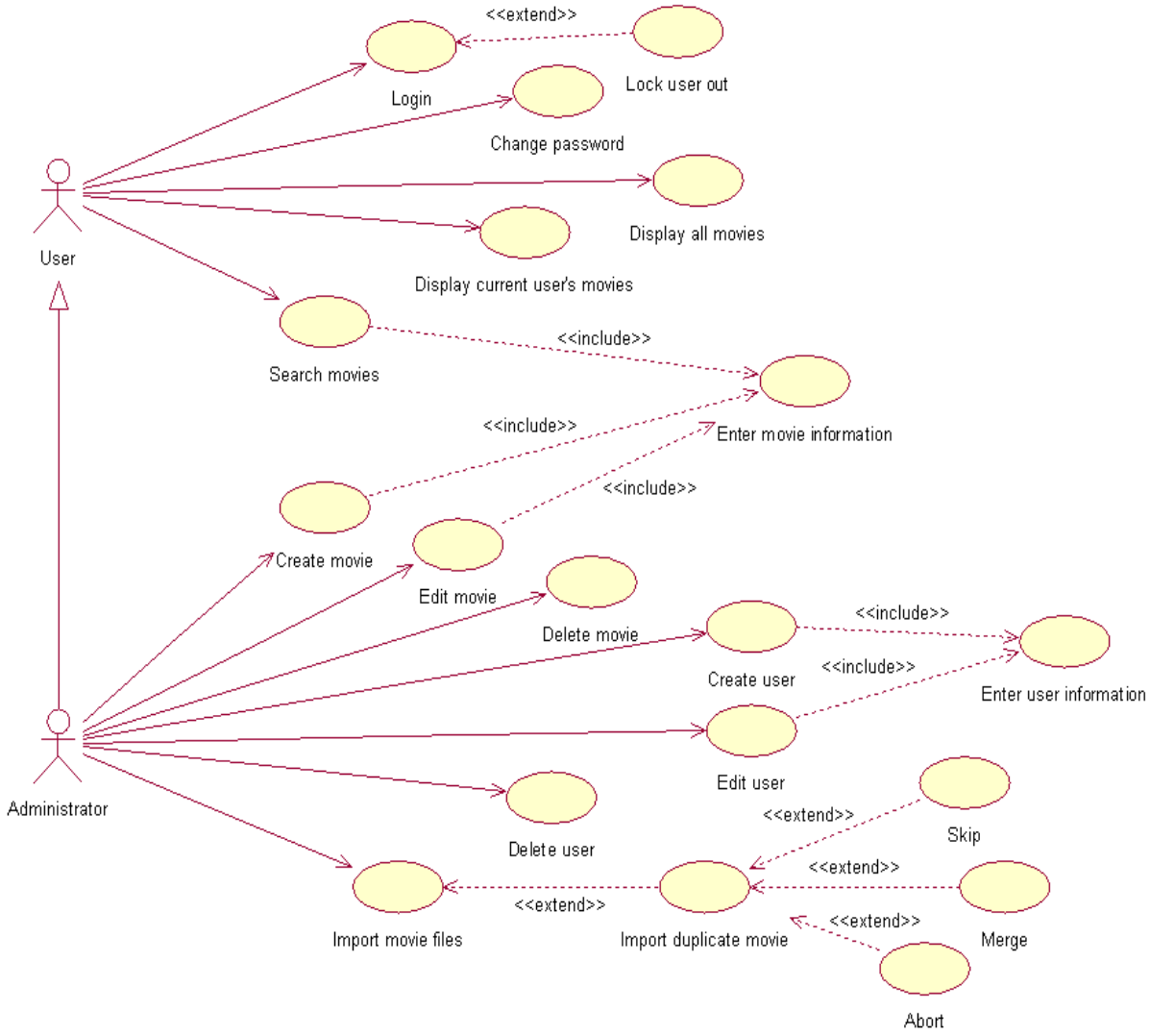
- Create movie
- Edit movie
- Delete movie
- Create user
- Edit user
- Delete user
- Import movie files

We then refined our use case model by finding a number of use cases which shared some similarities, which we were able to describe using the inclusion relationship with a number of new supporting use cases. These were "Enter user information" which is included in the "Create user" and "Edit user" use cases and "Enter movie information" which is included in the "Create movie", "Edit movie" and "Search movies" use cases.

Finally, we defined a number of extension use cases to further define the functionality of the system these were:

- Lock user out which extends Login
- Import duplicate movie which extends Import movie files
- Skip which extends Import duplicate movie
- Merge which extends Import duplicate movie
- Abort which extends Import duplicate movie

The following use case diagram shows the uses cases that we ended up with:



Note: Full use case descriptions can be found in Appendix 1.

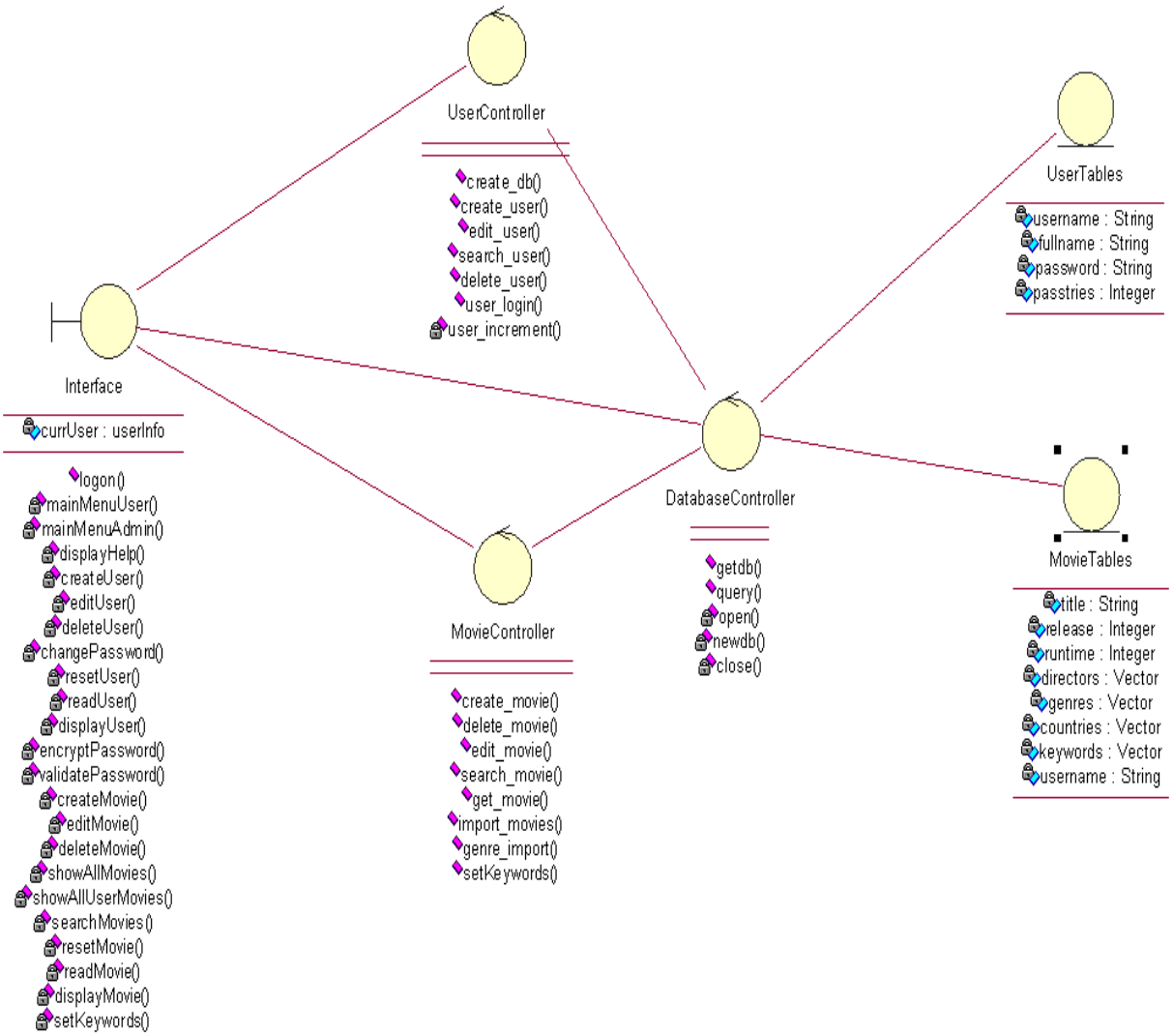
Classes

The second design task we undertook was to determine the classes that we felt would be necessary to provide the required functionality. To do this we used the Boundary-Control-Entity model. Initially we decided on having a boundary class called Interface, two control classes; DatabaseController and Wrappers, and two entity classes; UserTables and MoviesTables.

We planned that an instance of Interface class would handle all interaction between the system and users, including displaying menus and other prompts to the user and gathering any required input from the user. An instance of the Wrappers class would be created by the Interface class which in turn would create an instance of the DatabaseController class. Our plan here was that the DataBaseController object would control access to the database, including opening and closing the database and running any database queries. The Wrappers object would provide functions for producing appropriate SQL statements from the input taken from the Interface to be run by the DatabaseController and interpreting the results return from the DatabaseController. We planned that the entity classes, MovieTables and UserTables, would both be implemented as a number of database tables, in order to store movie and user data respectively.

Part way through the construction phase of the project we modified this model to split the Wrappers class into two smaller control classes, MovieController and UserController. We felt this provided a better abstraction of the real world domain of the system. It also provided that benefit of making the code more manageable. This change meant that the Interface object would now create instances of all of the three control classes.

The following is our final class diagram:



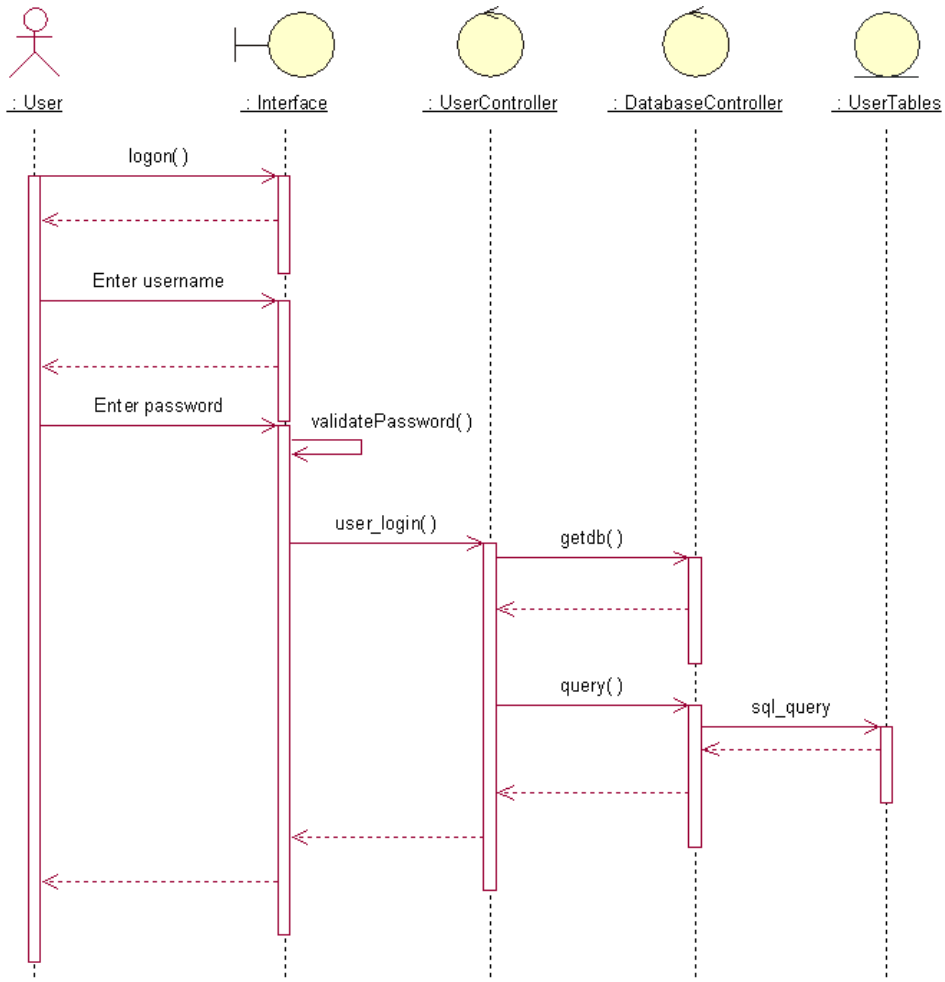
Note: The majority of the operations featured in the diagram were determined during the next design task (determining the interactions between objects) or during construction.

Interaction

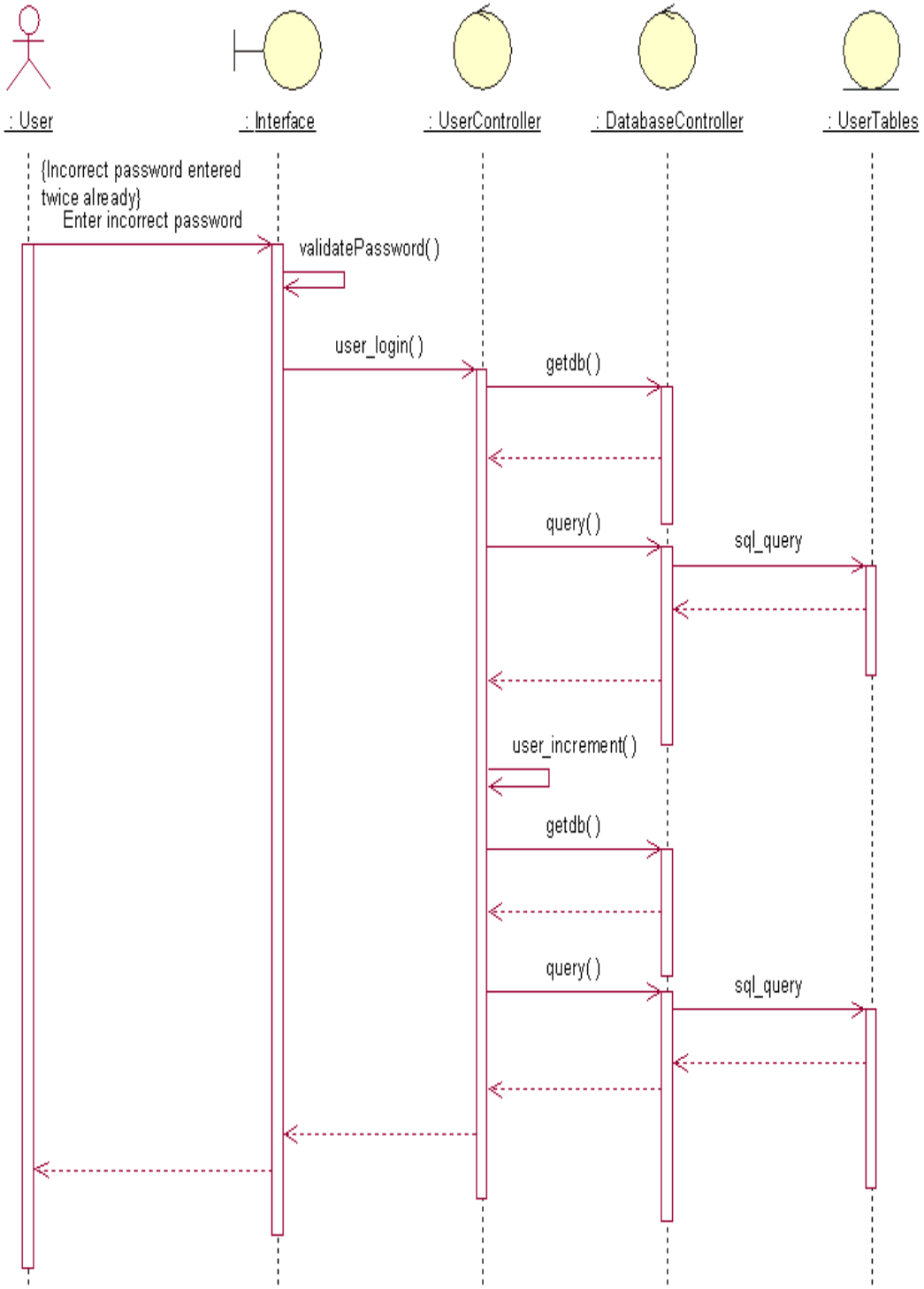
The next task we completed during our design work was to determine how all of our classes would interact with each other and the users of our system. This involved looking at each of the major use cases in turn and determining what 'messages' would need to be passed between our classes and also the user to provide the functionality specified in the use case. Much of this process was guided by the principles of the Boundary-Control-Entity model and considerations of possible implementations. Once we determined the basic messages and the order they would occur we were able to come up with operations for our classes which would handle these messages. This process resulted in a number of sequence diagrams, which, although they needed various revisions throughout the remainder of the project, made the construction phase simpler.

The following are our final sequence diagrams:

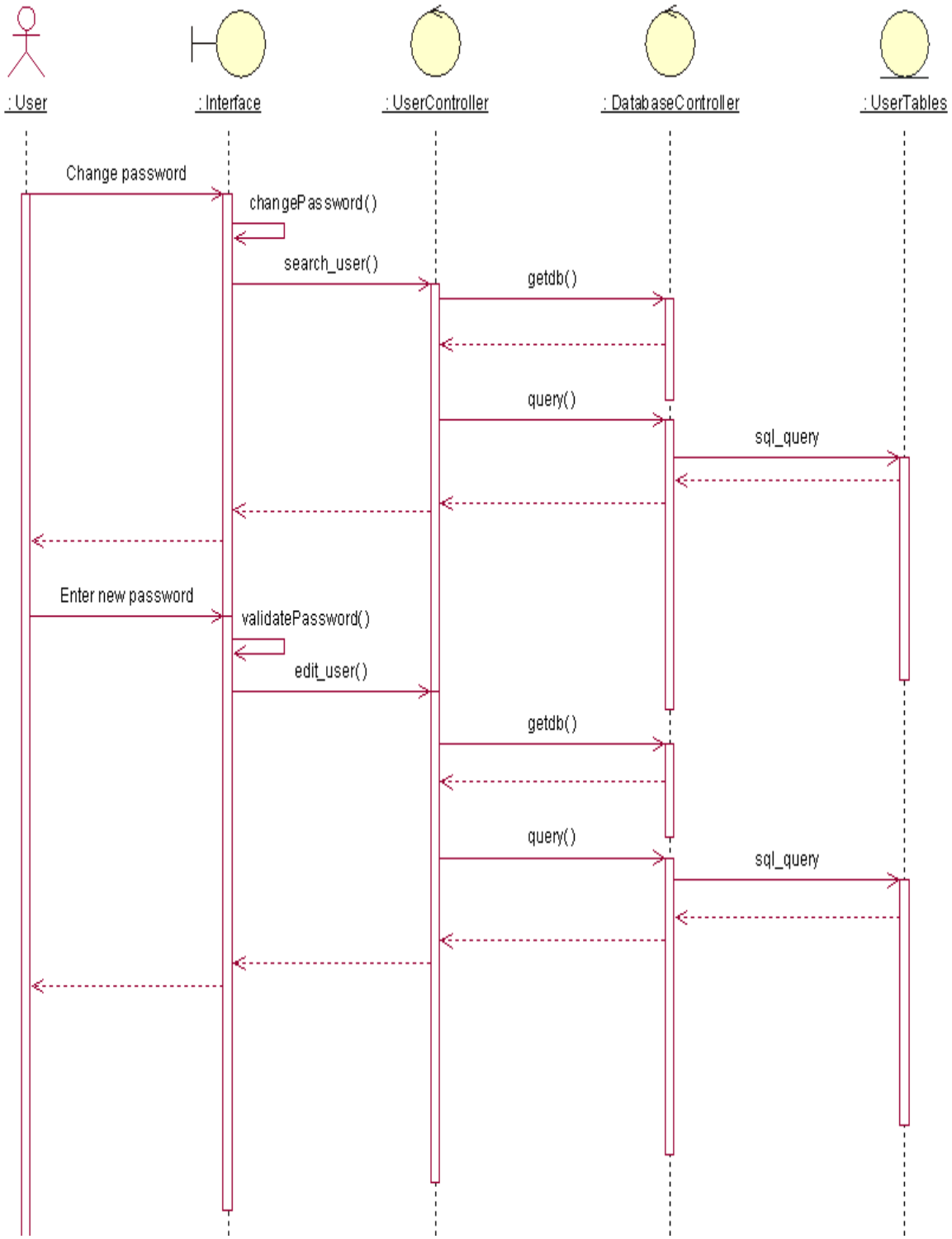
Login:



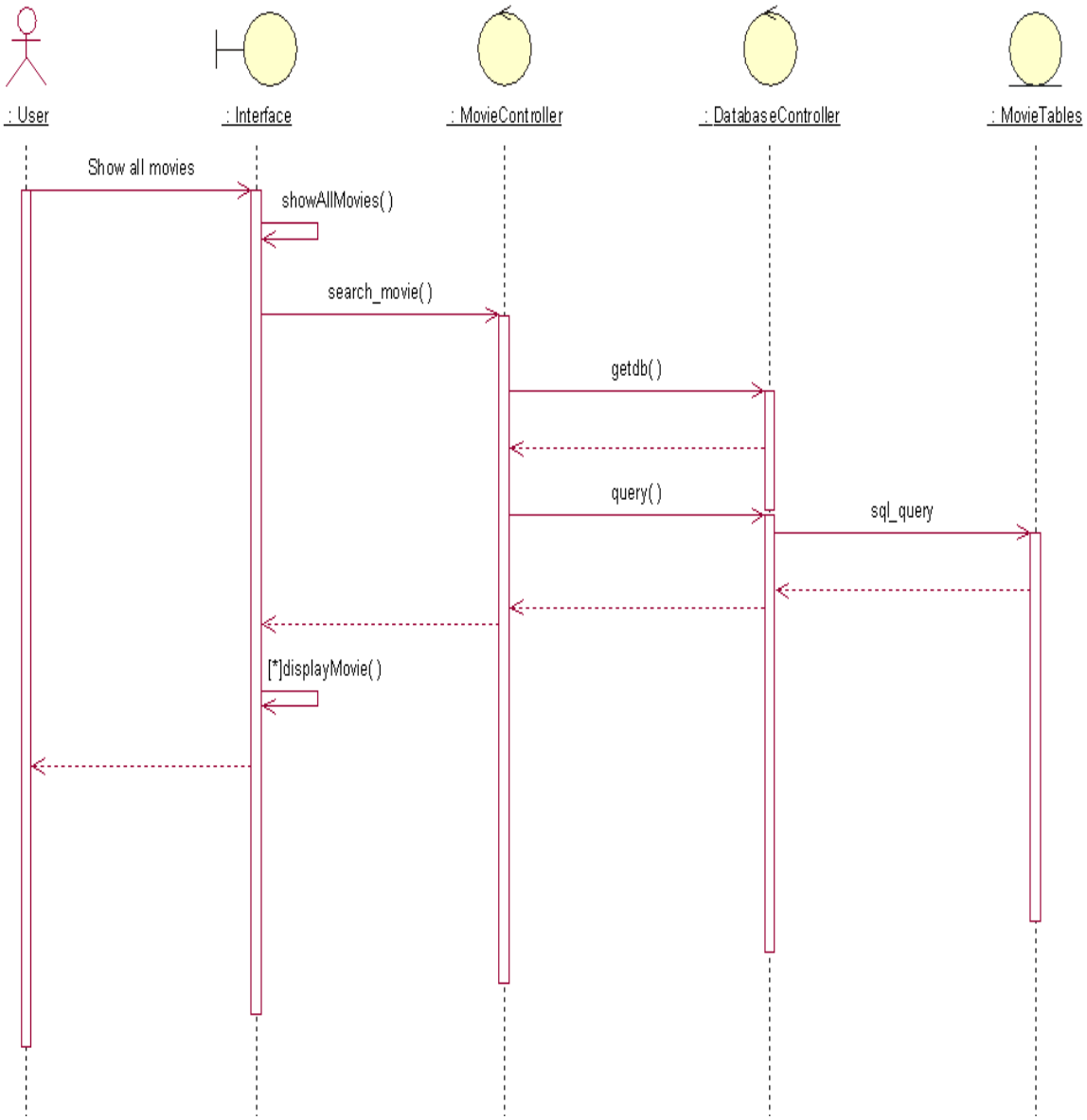
Lock user out



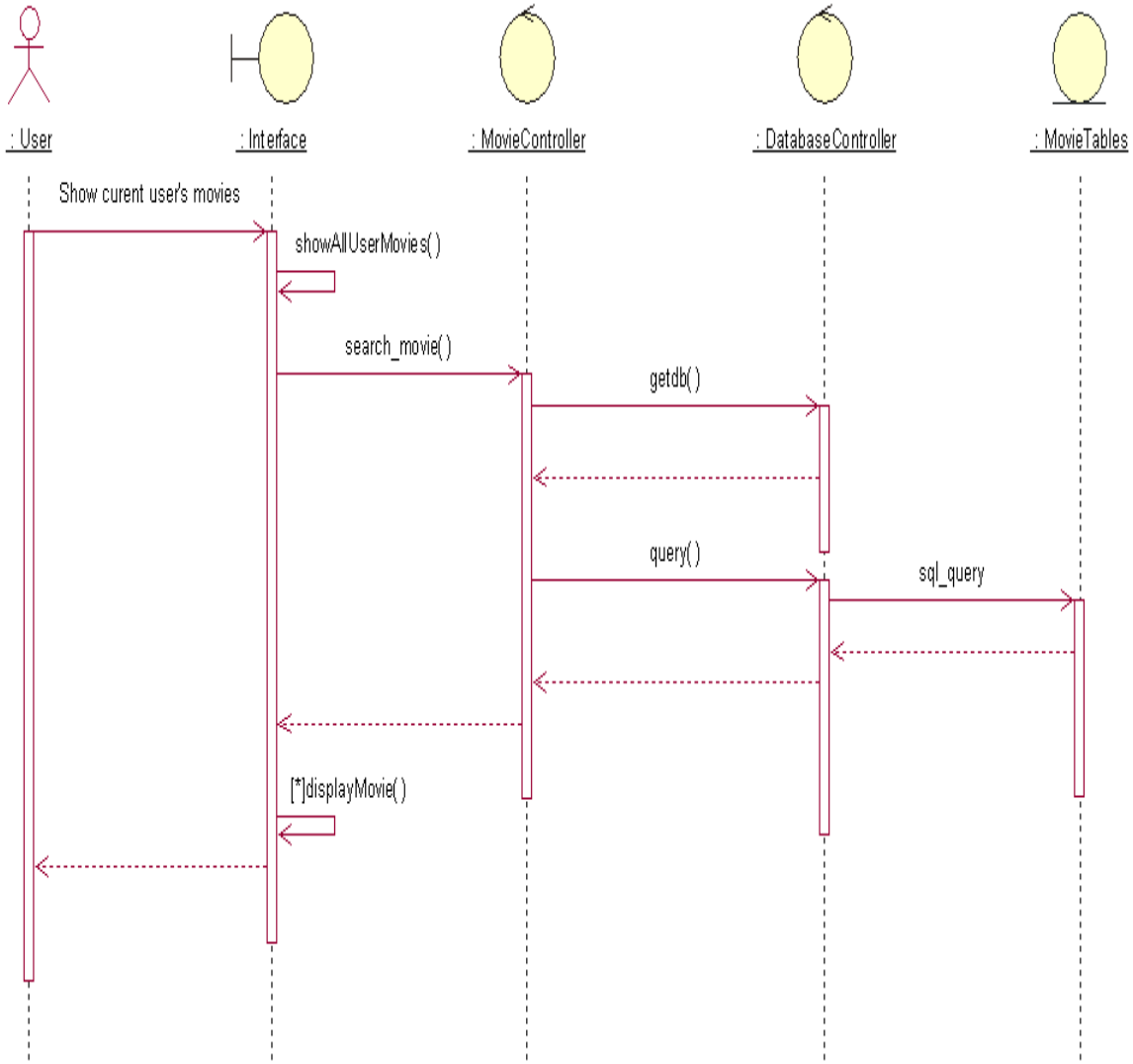
Change Password:



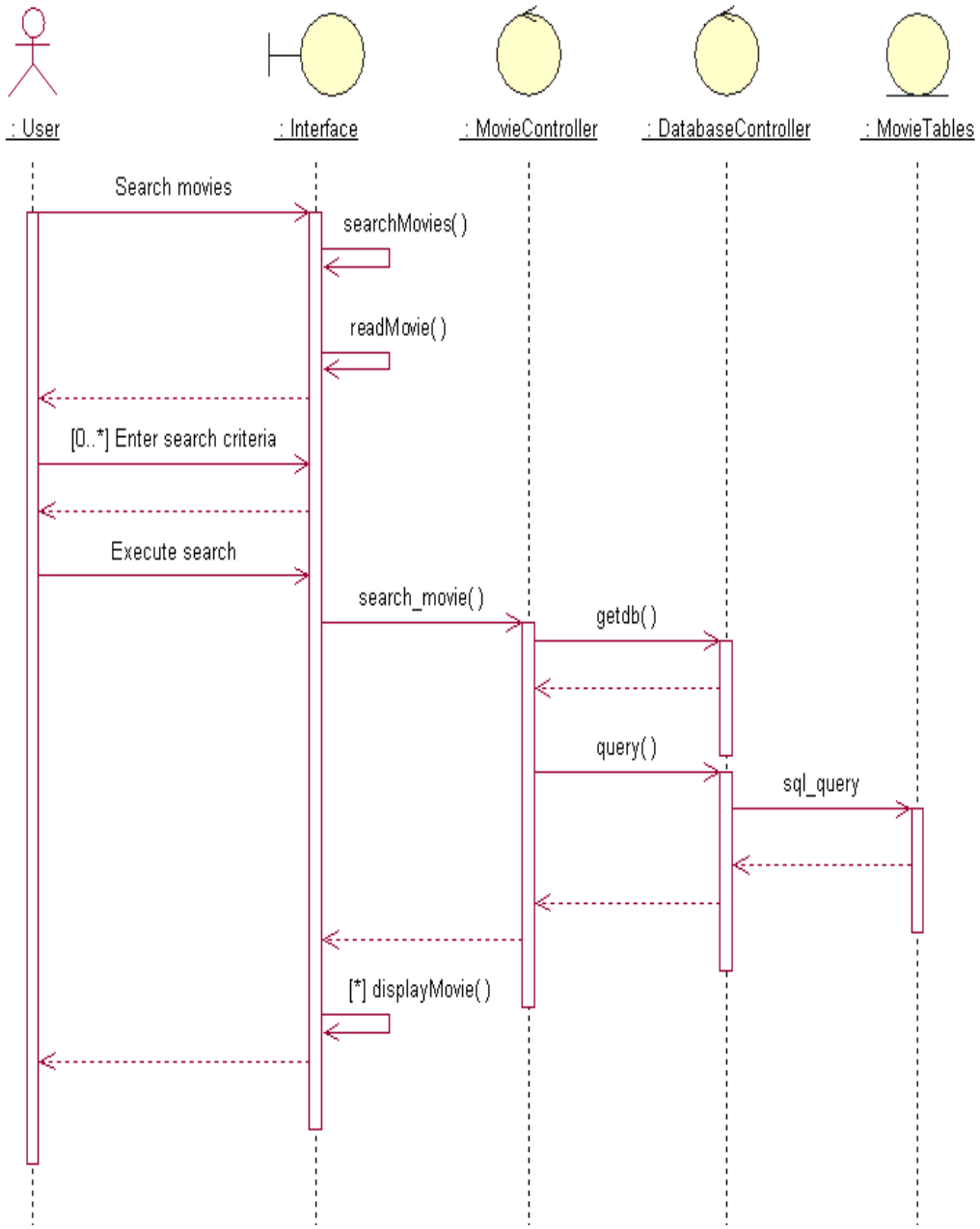
Display all movies



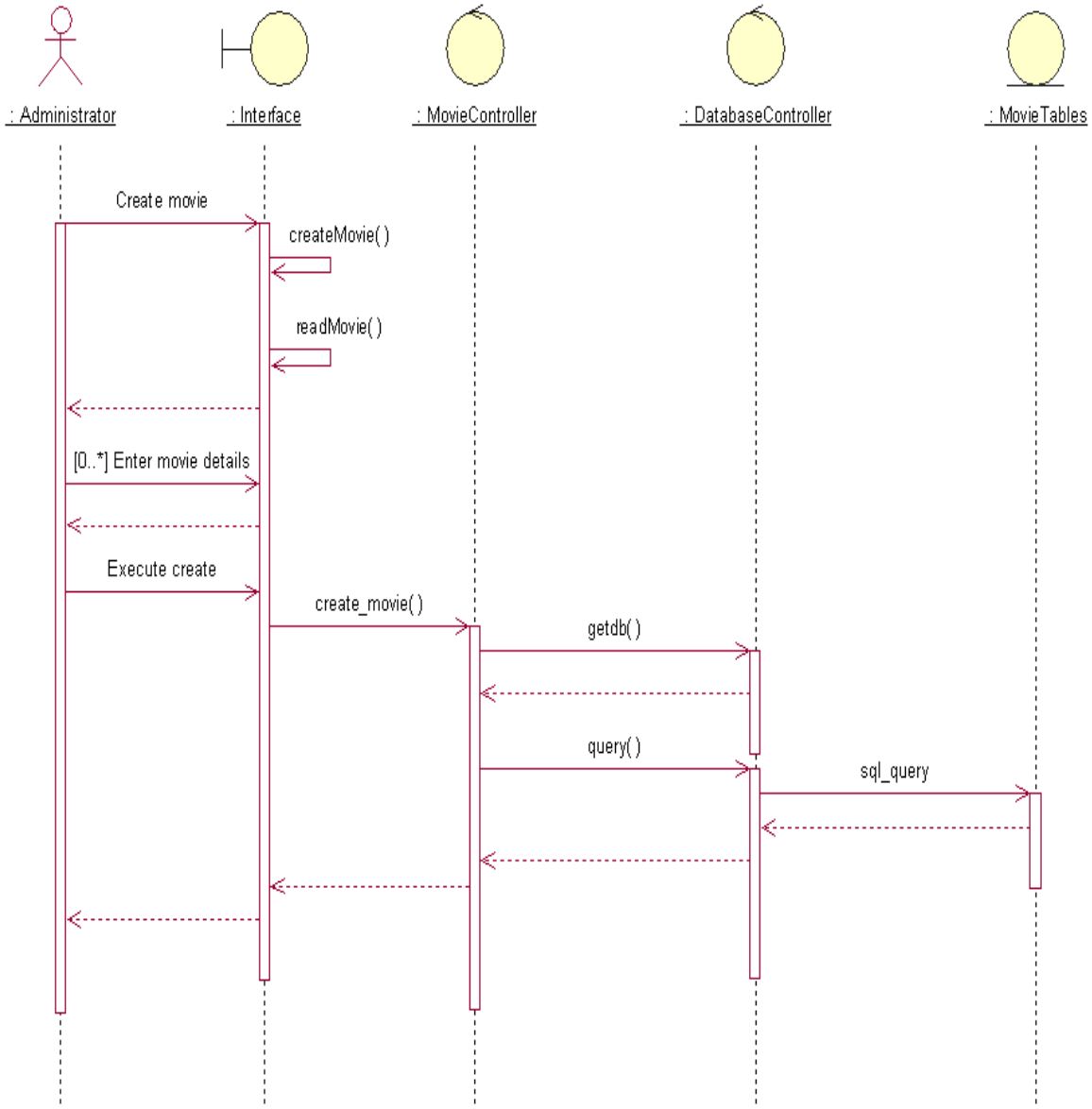
Display current user's movies



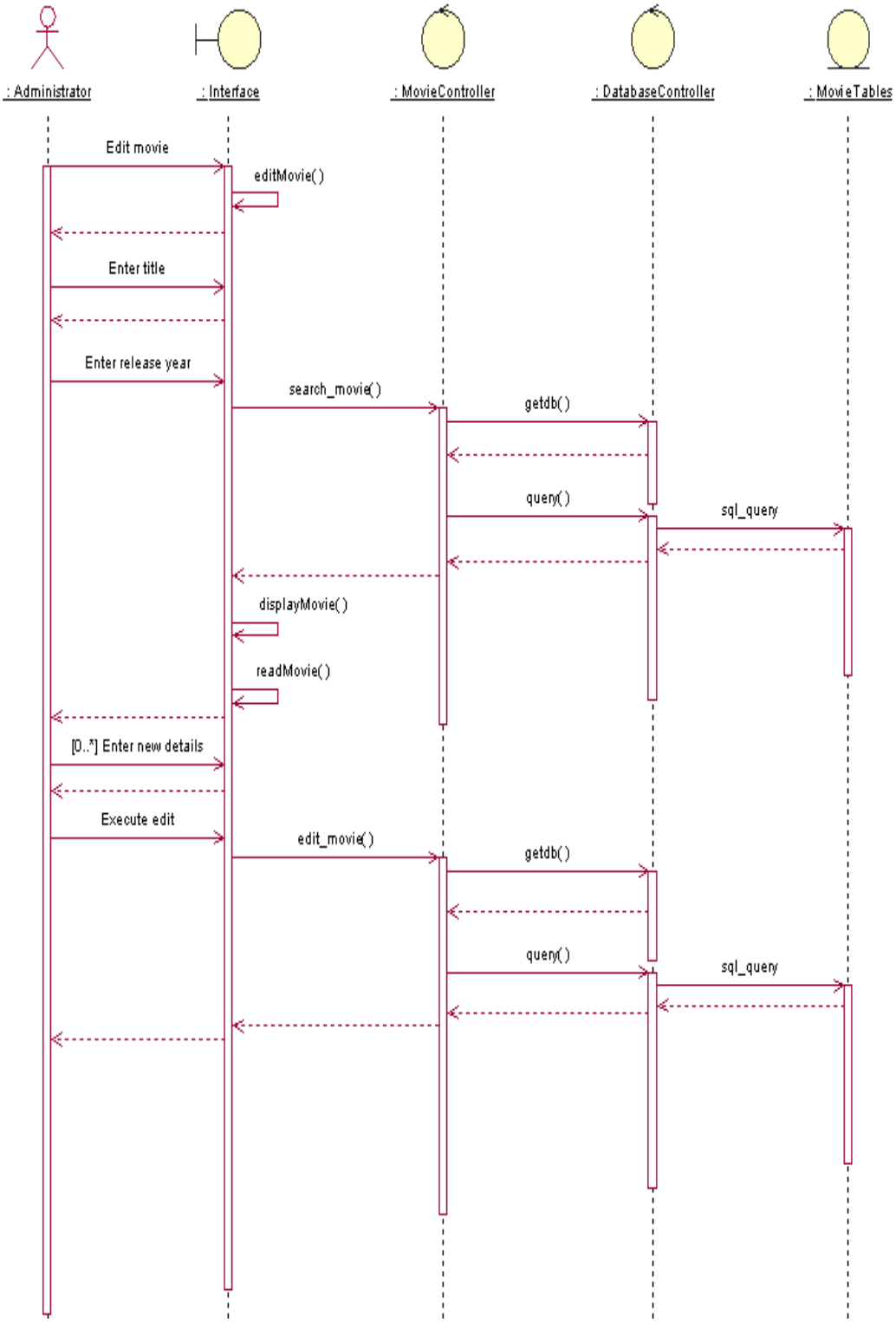
Search movies



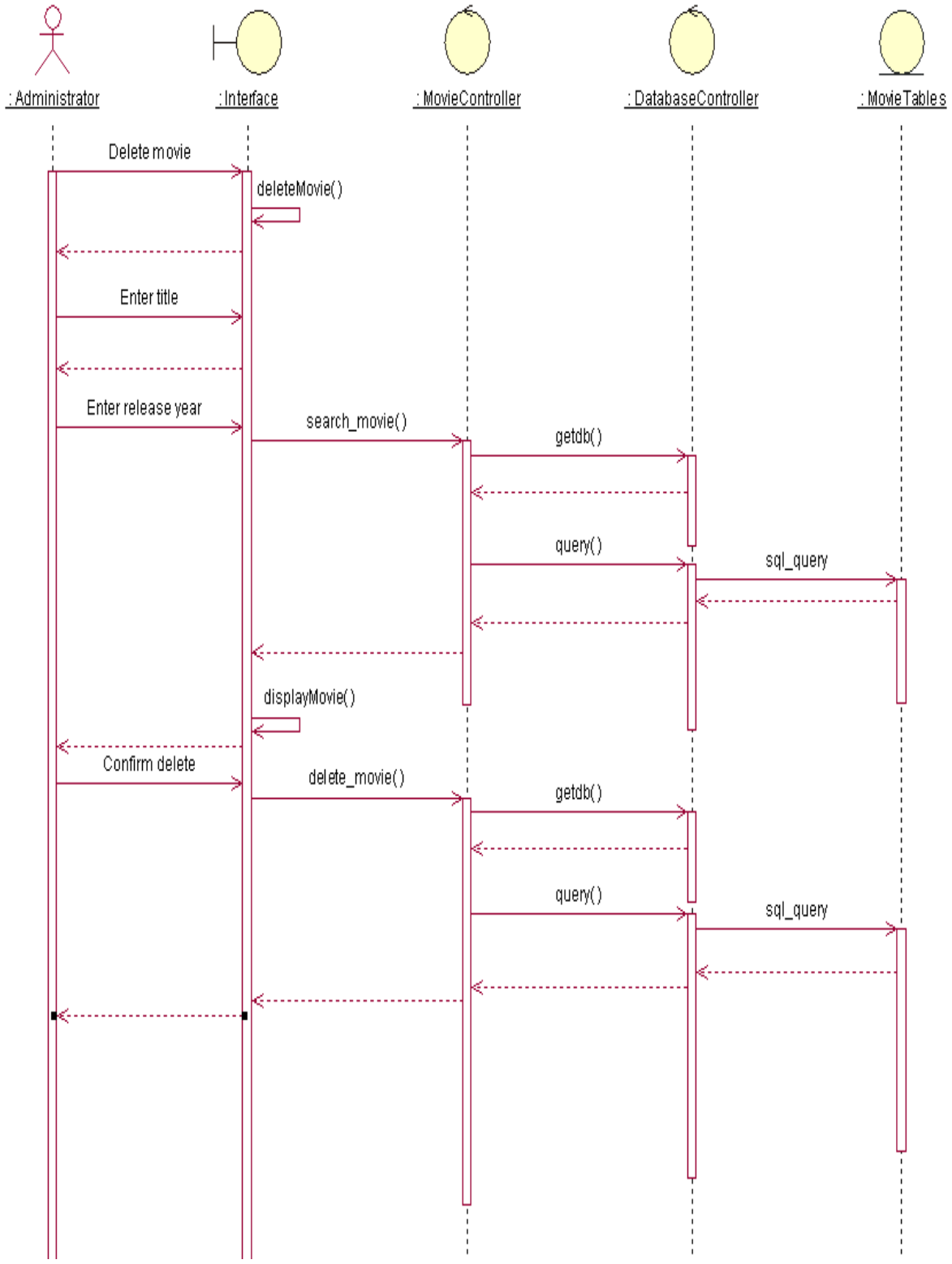
Create movie



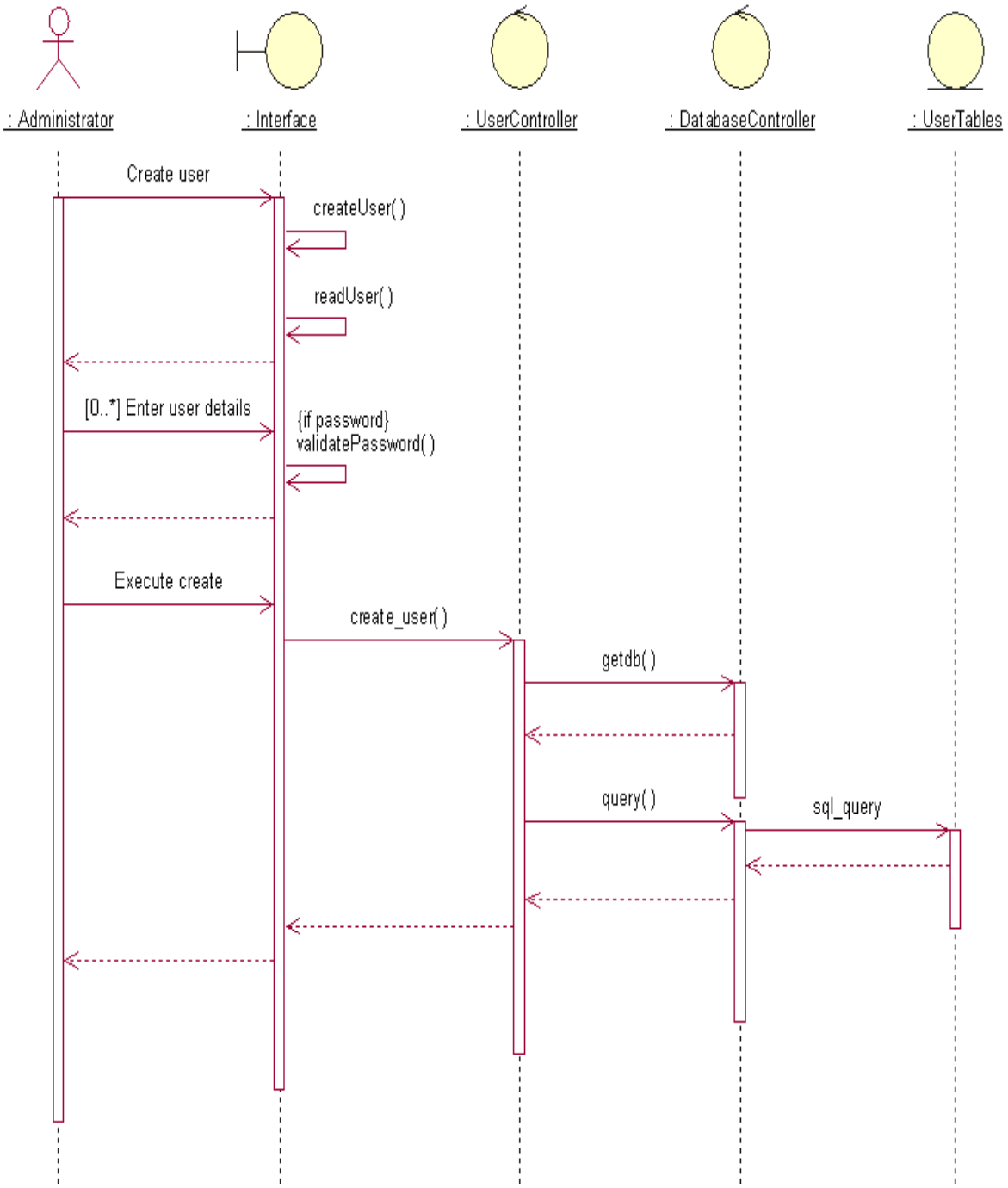
Edit movie


: Administrator
: Interface
: MovieController
: DatabaseController
: MovieTables
Edit movie
editMovie( )
Enter title
Enter release year
search_movie( )
getdb( )
query( )
sql_query
displayMovie( )
readMovie( )
[0..*] Enter new details
Execute edit
edit_movie( )
getdb( )
query( )
sql_query

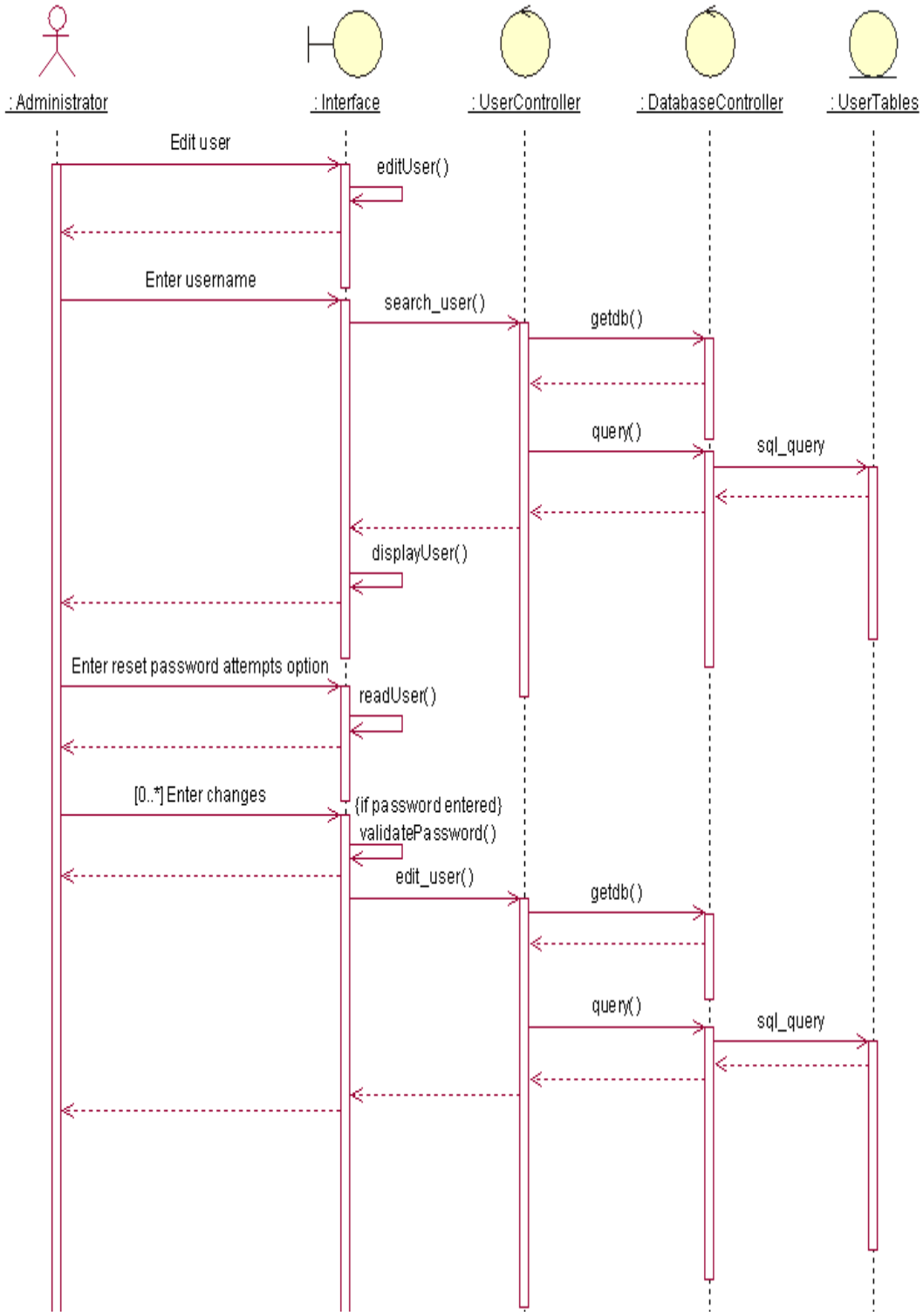
Delete movie



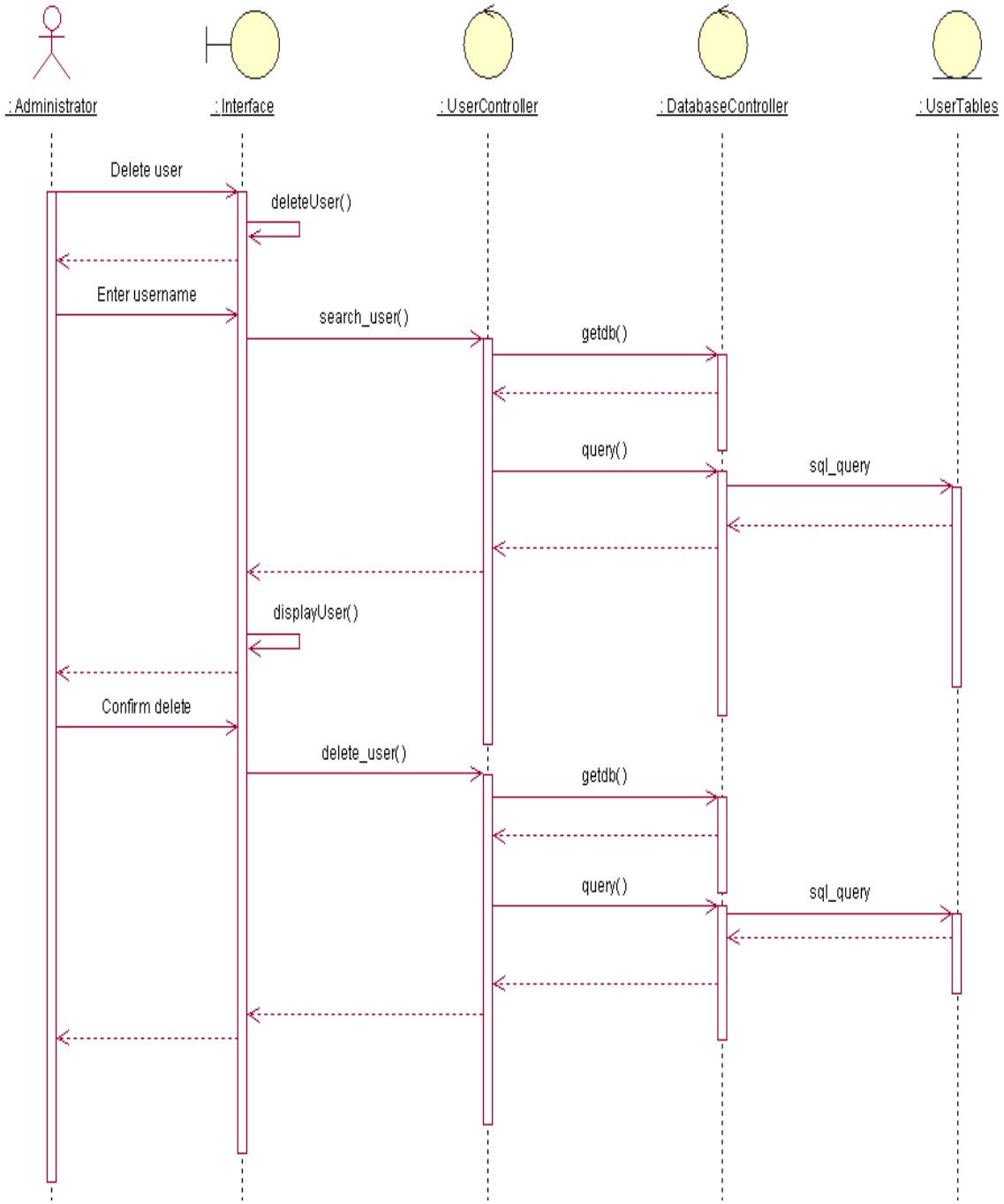
Create user


: Administrator
: Interface
: UserController
: DatabaseController
: UserTables
Create user
createUser( )
readUser( )
[0..*] Enter user details
{if password}
validatePassword( )
Execute create
create_user( )
getdb( )
query( )
sql_query

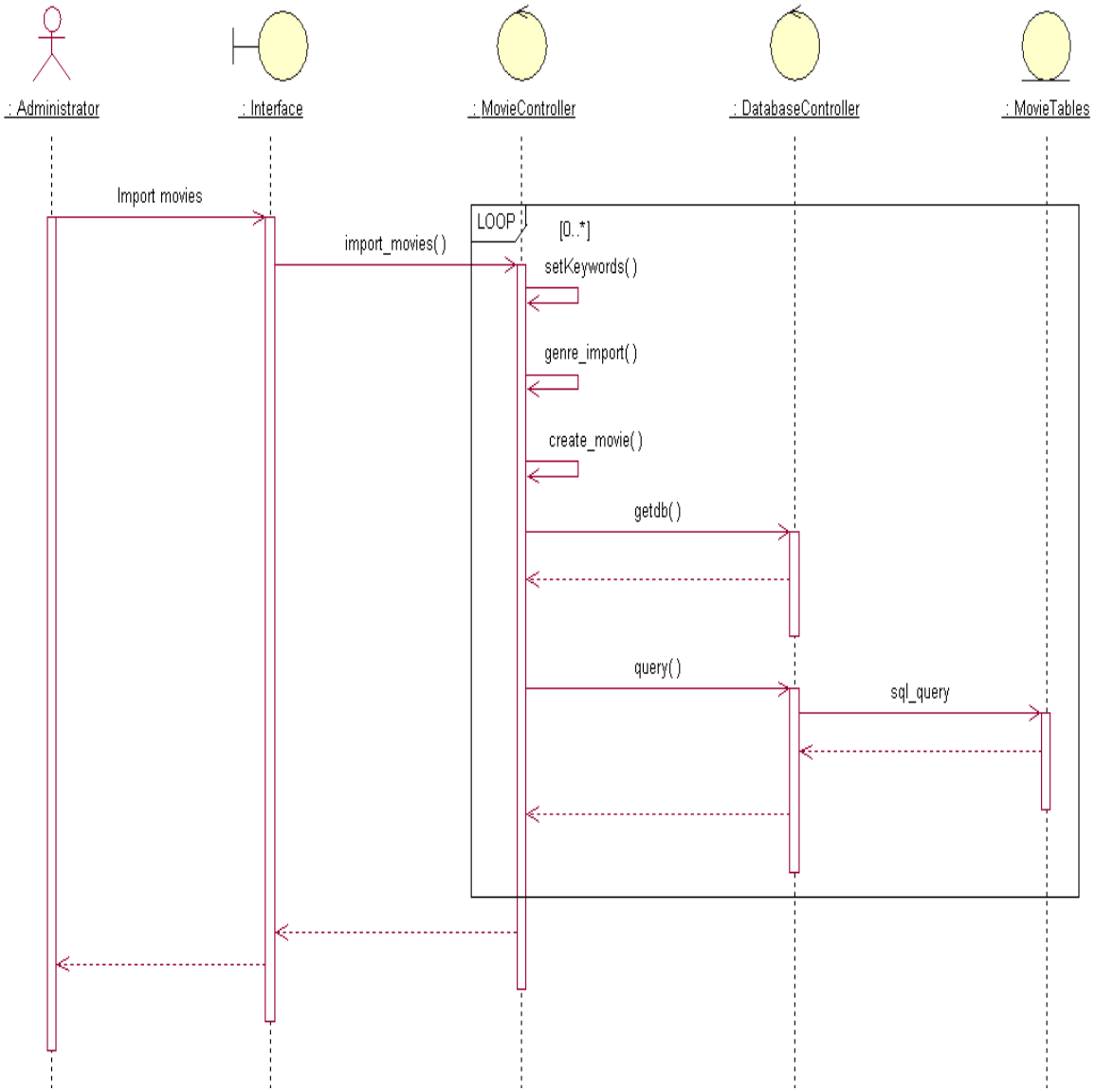
Edit user


: Administrator
: Interface
: UserController
: DatabaseController
: UserTables
Edit user
editUser()
Enter username
search_user()
getdb()
query()
sql_query
displayUser()
Enter reset password attempts option
readUser()
[0..*] Enter changes
{if password entered}
validatePassword()
edit_user()
getdb()
query()
sql_query

Delete user


:Administrator
:Interface
:UserController
:DatabaseController
:UserTables
Delete user
deleteUser()
Enter username
search_user()
getdb()
query()
sql_query
displayUser()
Confirm delete
delete_user()
getdb()
query()
sql_query

Import movie files



User Interface Design

Our team's decision was to create a console based application, and as such, our user interface needed to be within the constraints of a standard console window. As a result, we chose to create a text-based menu UI that runs in a console, rather than a GUI. We made this decision, as our method:

- Can run on any system (including terminal based ones which would likely be used for searching)
- Is simple for users to learn to use (being given clearly defined choices)
- Is simple to implement, meaning time was not wasted creating a GUI instead of working upon the functional requirements.

The General Design

The general design of the application's UI is very basic and simple to use. It consists of a few text-based menus and forms, as well as a few text-based prompts for information (depending upon the action).

The general breakdown of the UI goes like so:

- **The Main Menu:** Displays the actions the user can perform. It then links to each of those respective action's functions which will either prompt for data or pass it onto the user/movie form menu.
- **The User Form Menu:** Used for the create and edit user functions, it displays the pieces of data that can be entered to describe a user.
- **The Movie Form Menu**: Used for the create and edit user functions, it displays the pieces of data that can be entered to describe a user.
- The basic prompts for all other actions (if needed), such as change password simply prompting for the new password, these are used where the input is not sophisticated enough to warrant a menu. These may also be called before the other menus.

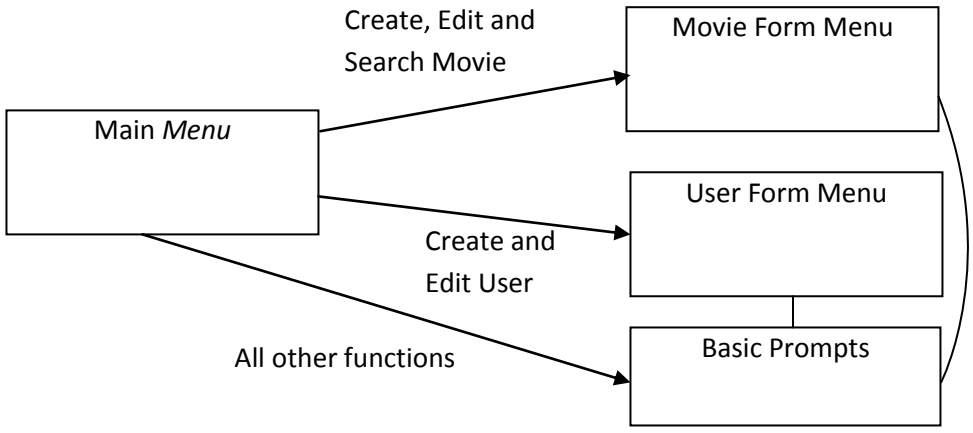
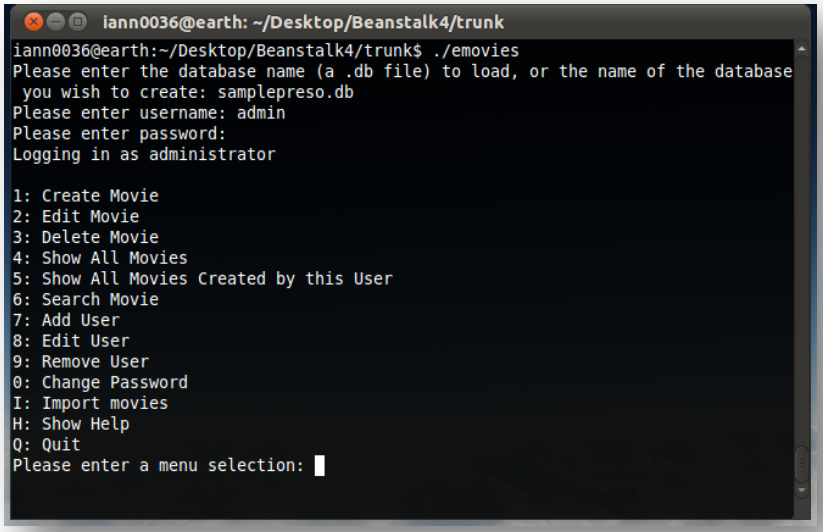


Diagram showing basic flow of interface. Note basic prompts may call the form menus or vice versa

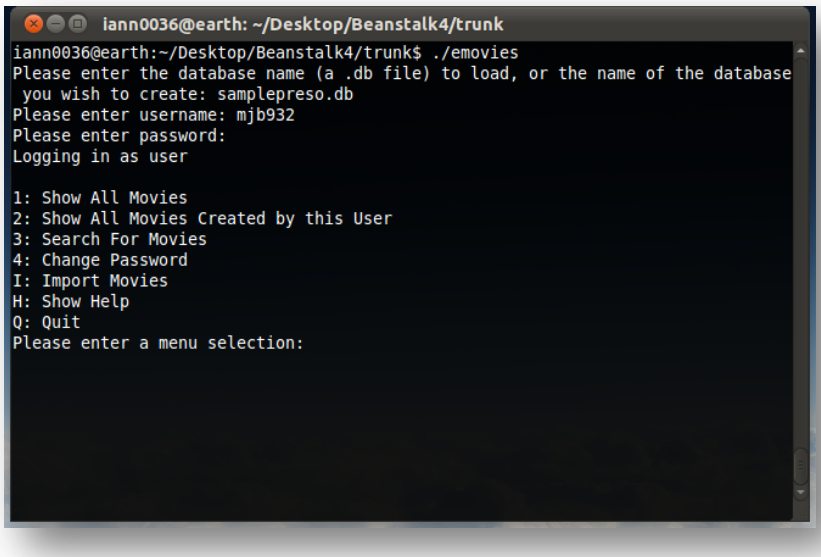
The Main Menu

After logging in the user is presented with the main menu. This menu supplies choices as to what the user can do. The options the user can choose from are different depending on the type of user (either administrator or standard user). The main menu looks like the following:

For the Administrator:



For the User:



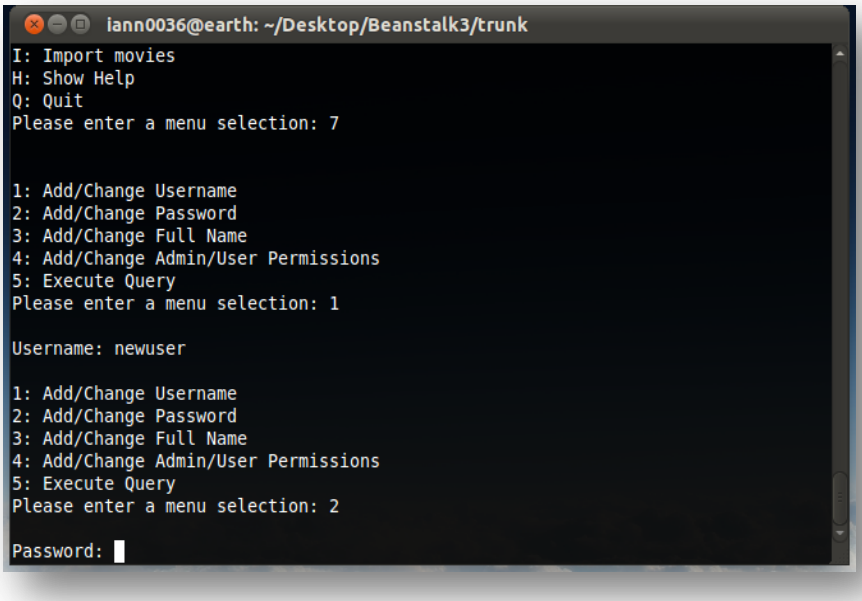
After presenting the menu the user is prompted to enter a choice corresponding to that action (either a number of a letter). An option for help is also presented on this menu, which if selected displays a better description of what that choice does. This menu continues to run until the user asks to quit the program, which saves all progress.

The User Form Menu

If the user chooses to Create or Edit a user they will be directed to this text-based form (in the case of edit, this happens after the user is prompted for the username of the user to be edited). This form is displayed as a menu (much like the former) where the user can select which element they would like to add/edit.

The user selects a choice, again by entering the respective number of that choice on the menu and is then prompted for that piece of data, after this they are redirected to the menu to continue editing/adding. Editing a field again overwrites the previous entry, allowing the user to correct mistakes.

This menu continues to loop until the user executes the current operation's query.

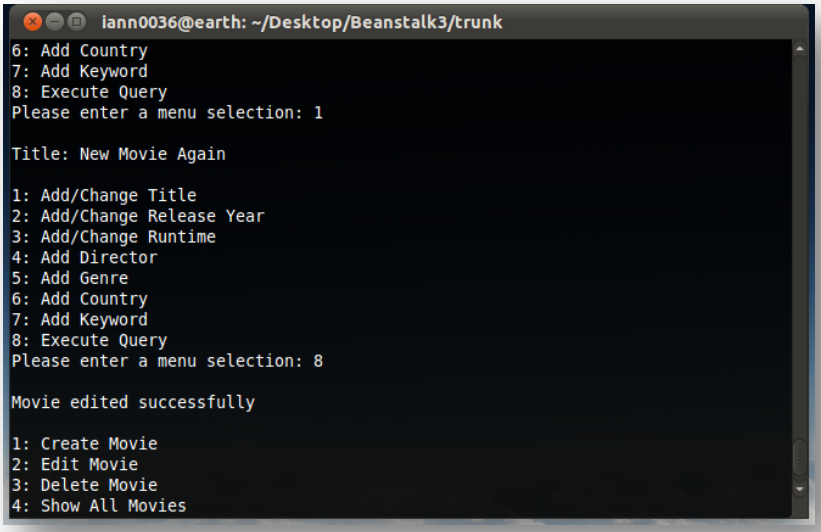


The Movie Form Menu

This menu is similar to the one used to enter user data, differing in that the choices given by this menu are related to movie data instead of user data. This menu is called by executing the Create, Edit or Search movie function (since they all involve entering movie data).

Again, this menu continues looping until the user executes the current operation's query.

This menu performs the same as the previous one, selecting a choice by entering the corresponding letter, and then being prompted for that piece of information. It also allows the correcting of errors by overwriting previous entries, and allows for the addition of multiple directors, genres, contries and keywords relating to that movie.



Search also uses this form, but in a slightly different way, as no fields have to be filled (as opposed to create and edit, where the title and release year are needed), and it searches based on which fields are filled.

The Basic Prompts

These are not specific interface elements; they are simply statements prompting the user for input. They can be used separately.

Construction Iterations

Our group's iterations were initially planned out similarly to the way recommended by the specification, in that they are centred around introducing new features at each iteration, and each iteration focused on adding requirements of successively lower importance (high first, then medium, finally low).
The iterations were as following:

1. High Level Functionality Requirements (Storing, accessing, adding, editing and deleting users and movies)
2. Medium Level Functionality Requirements (Searching)
3. Low Level Functionality Requirements (Importing and finalising)

Our actual development followed these planned iteration cycles closely, but not exactly, the most notable deviation being us delaying the editing functionality to the 2nd iteration for medium level, to allow more time to get the other high level requirements down. These iterations and the work done in them are better explained below:

Iteration 1 – High Level Requirements

This iteration involved designing and implementing the functional requirements defined as being of high level of importance.
The main elements (and functionality) we added to this iteration were:

- Storing the information, in our case as an SQLite database
- Accessing the database
- Creating and deleting user records within it
- Creating and deleting movie records within it
- Creating an interface that allowed the user to perform these actions

Iteration Construction Summaries
The construction of these elements can be further broken down into the following summaries:

In terms of storing the information, using the diagrams and other tools we had created in the pre-planning stage, we had determined the structure for the database in our schema, and simply needed to port this to C++. This involved:

- Creating structs to hold the data and pass it through the application (structs were chosen, as they were simple to implement, and to prevent the need for overly complex scoping/accessing)
- Creating/loading the database tables on execution. This was done by implementing the open_db and create_db options, which use SQLite to create tables

In terms of database access, we needed to implement methods for only allowing accepted users, this involved:

- Creating a table type inside the database for storing user information (again defined by our schema, the User table)
- Implementing login functions for both the interface (for accepting the users input for logging in and returning errors) and for actually accessing the database to compare user information.
- Implementing a function to encrypt/decrypt a password using a Caesar cypher
- Implementing the lock-out functionality in the login functions

The creation and deletion methods were simple, they involved:

- Constructing the create and delete methods for user data, which makes up the SQL queries based on data passed to it
- Constructing the create and delete methods for user data, which makes up the SQL queries based on data passed to it
- Implementing forms to enter movie and user data respectively for the interface, to then call these methods.

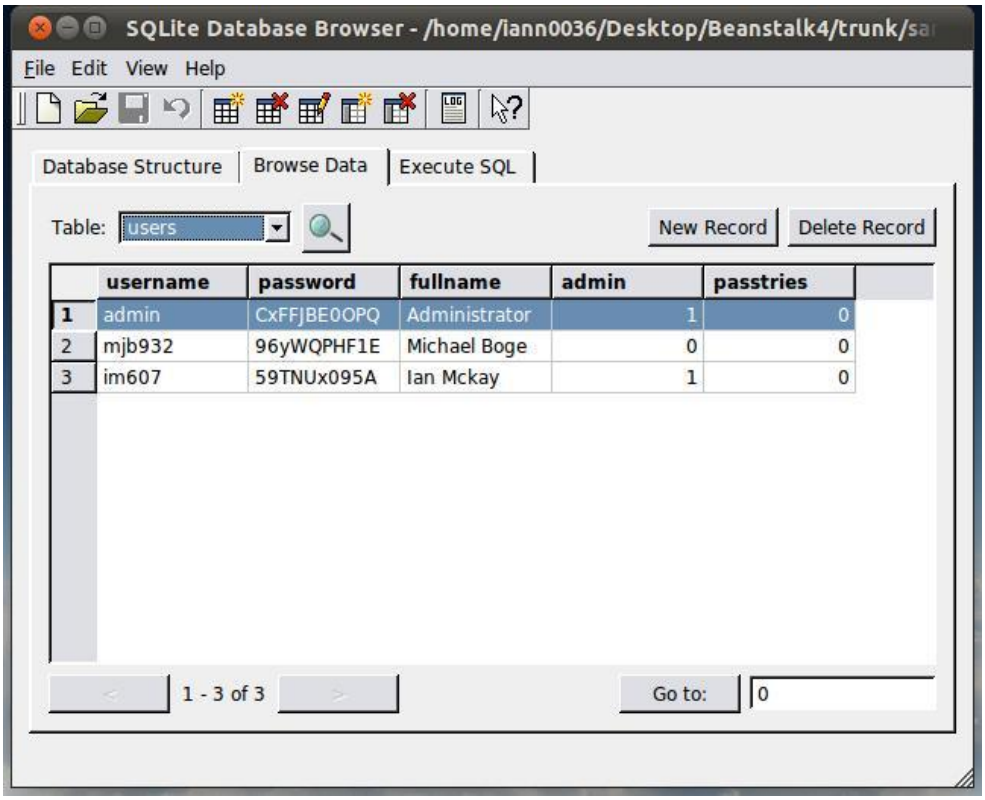
In terms of the interface, this simply meant constructing functions which:

- Prompt for login information
- Give a main menu
- Prompt for information to add, or record to delete

Testing

This iteration required a large amount of testing, as all the other iterations would be building upon it.

Blackbox testing was used thoroughly in this iteration, and was mainly performed using a shell script, as CppUnit tests weren't always valid test methods for the functionalities added.
This was particularly true in the initial section where the aim was to create a database and store information, this was not possible to test with CppUnit, as the actual creation of the database had no return values aside from the actual database creation.
The main method of testing Ian and Michael used here was to run the program and check the database file created in an SQLite database browser (to see if it was correct).
The record addition/deletion functions were similarly tested, during the first iteration, as there was no search function to check.



Screenshot showing the use of a SQLite database browser to check changes to the Users table. In this case we are checking the passtries was reset for mjb932.

However, black-box testing using the script was not the only method used. For the creation and deletion methods we created very basic CppUnit cases, that simply checked for binding and execution errors (not valid returns), to test them separately from the interface.
The methods for logging in and password encryption/validation were the only methods that were mainly tested through using CppUnit in this iteration.
Tiaki constructed the unit test cases for these functions, to test a number of both valid and invalid passwords (to test for success and failure as appropriate).

```
84  void eMoviesTest::importMovies()
85  {
86      //create a string for username to get credit and a movie
87      //that exists in the movie list file
88      string username = "bobdole";
89      movieInfo movie;
90      bool isEmpty;
91
92      //create existing movie
93      movie.title = "Banana Boy (2003)";
94      movie.release = 2003;
95
96      //Search for movie that doesnt exist
97      CPPUNIT_ASSERT((mc->search_movie(movie)).empty());
98
99      //Import the movie list
100     mc->importMovies(username);
101
102     //Search for the same movie that now should exist
103     CPPUNIT_ASSERT(!(mc->search_movie(movie)).empty());
104 }
105
106 void eMoviesTest::validatePassword()
107 {
108     //Check a valid password
109     CPPUNIT_ASSERT(interf->validatePassword("password1"));
110     //Check various invalid passwords
111     CPPUNIT_ASSERT(interf->validatePassword("invalidpass"));
112     CPPUNIT_ASSERT(interf->validatePassword("invalid1!"));
113     CPPUNIT_ASSERT(interf->validatePassword("pass word1"));
114     CPPUNIT_ASSERT(interf->validatePassword(""));
115     CPPUNIT_ASSERT(interf->validatePassword("short1"));
```

Screenshot of a section of CppUnit code used to test the validatePassword function. Note that this screenshot was taken in another iteration, hence it also tests the import function.

Summarising our additional tests created for this iteration, aside from the standard tests for what a function returns (i.e. did it succeed or did it error), were:

- Testing for linking between tables (done through SQLite browser again)
- Testing for input being accepted correctly, for the interface (such as not accepting nulls)
- Testing user lock-out correctly worked

Defects and Integration

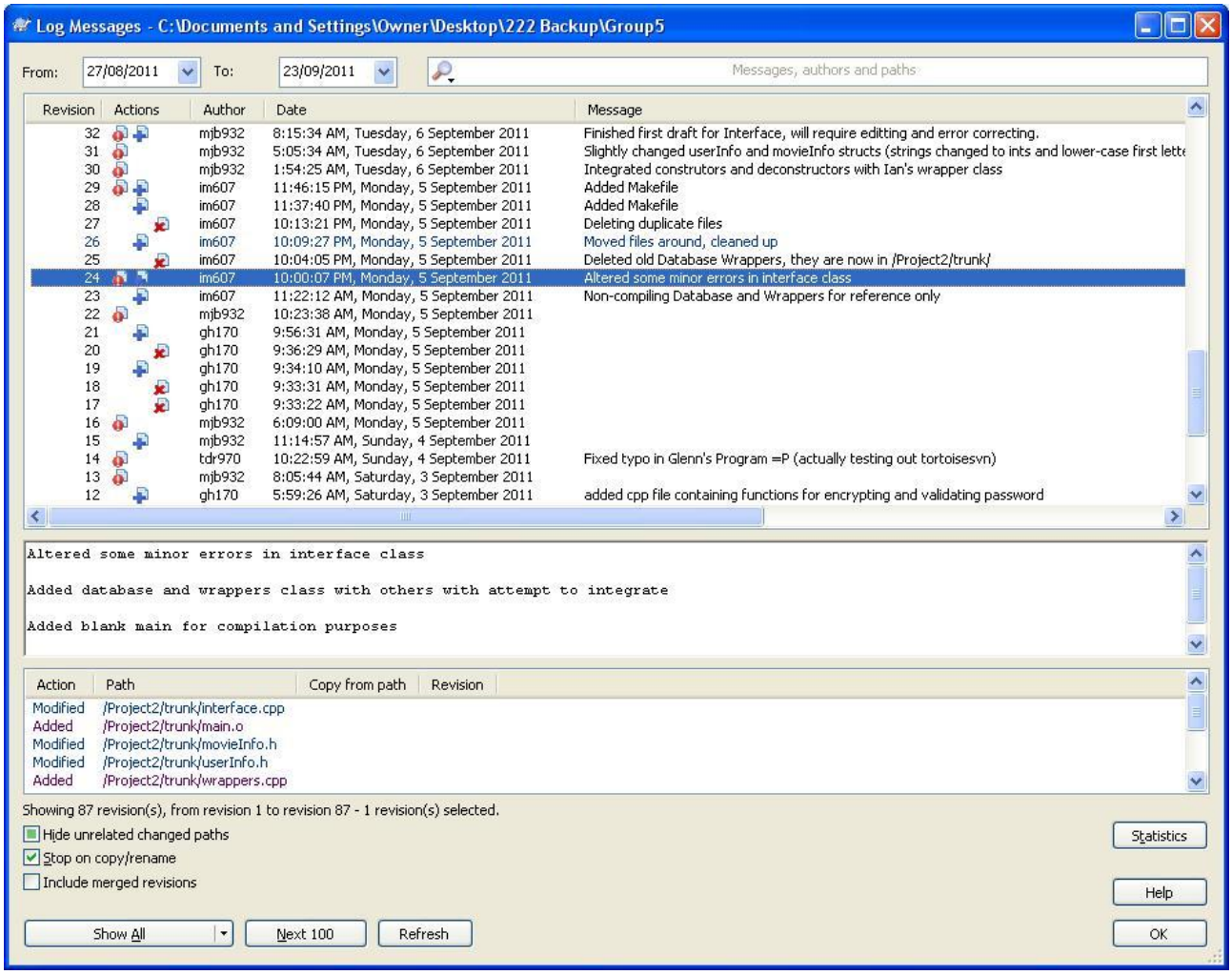
In terms of integration, the main focus was on integrating the interface with the controller classes. This required much communication between the developers of each section, as they needed to ensure they were calling the right functions correctly, and expecting the correct returns. One significant integration problem was when the controllers were accepting each piece of information separately instead of using the structs, which the interface was passing.
Notable defects for this section were, aside from the problem mentioned above:

- Incorrectly calling the database constructor, due to incorrect function calls from the interface, meaning the database was not always created successfully. Fixed by changing these calls in the interface
- ID's for movies, directors, genres, etc not incrementing. Fixed by adding a query to get

the maximum ID value, then adding 1

Version Control Information

The following screenshots of log files show some of the tasks performed in the first iteration:



Screenshot shows the general summary of the first iteration, showing Michael and Ian creating the products then integrating them together (as is shown in the selected message), with Glenn and Tiaki fixing problems and adding design documents.

Iteration 2 – Medium Level Requirements

This iteration was originally to involve the designing and implementing of functional requirements of medium level of importance, however it differed from this as it also introduced the edit functionality for the program (this was to spread out the workload and because the edit and search function shared much in common, as the both had to generate similar SQL queries).
This iteration involved :

- Designing a method of editing user and movie records in the database
- Designing the for searching for a movie given any combination of search criteria
- Modifying the creation and edit functions to be able to cope with duplicates
- Integrating this with the interface

Iteration Construction Summaries
The construction of these elements can be further broken down into the following summaries:

In terms of editing, we re-used the forms used for creation of users/movies from the first iteration. However, it also involved generating a suitable SQL query, which was the most difficult part of this construction.

The search function was much more difficult, as it had a number of parts to implement, namely:

- A method to generate the initial query for searching the movies table (as it could have any combination of search elements)
- Methods that searched their respective tables for matches if required (i.e. searching the directors table separately if a director is to be searched for)
- Finally a method to combine the results and parse them into a suitable form for reading by the interface.

The search still used the standard movie input form however, to ensure a standardisation of input.

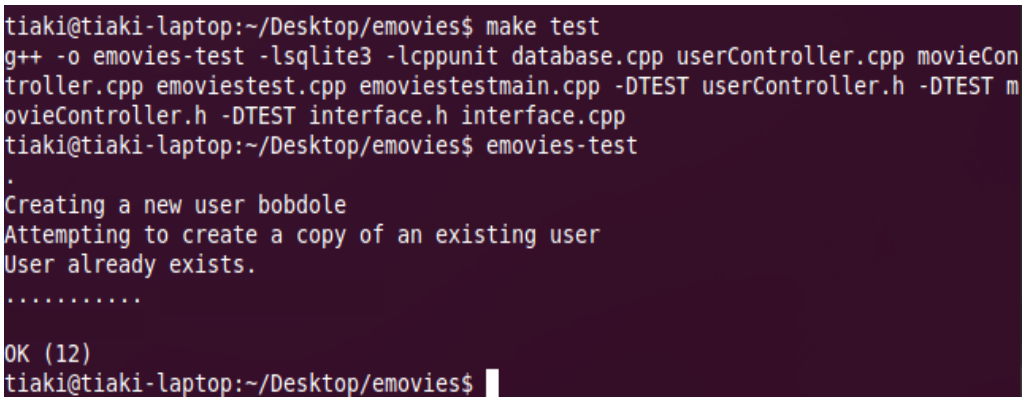
The modification to cover duplicates simply involved using the newly created search function within the edit and create functions to check before a record was edited/added.
The integration with the interface in this iteration again involved using the new search function as appropriate in its own functions.

Testing

CppUnit was used more prominently in this iteration. Most notably the search function used CppUnit extensively, in order to test the large number of possible inputs, and also to ensure user error (in inputting test-cases was not causing any errors). The search was easy to test this way as it returns vectors of results matching the inputs, which are easy to assert for correctness.

Editing, other than using the standard CppUnit tests for errors, again used the black-box testing method for testing its results, again running a script. However, the addition of the search

function meant instead of using a database browser, the script simply needed to call the search to confirm additions/deletions/edits.



Screenshot of CppUnit test run during this iteration. In this case it is trying to edit a user so that it has the same values as another in the database (which is not allowed, so it is erroring as appropriate)

Additional cases added for testing (both unit and otherwise) were:

- Cases testing correct response for duplicates
- Cases ensuring the edit did not overwrite other records
- Cases testing for a combination searches (such as genre, title and keywords)
- Cases testing for what happened with searches containing duplicates (such as 2 of the same director in one search)

Defects and Integration

In terms of integration, the main focus this time was on integrating the search function to be used in the other controller functions and by the interface. The other sections were using their own SELECT queries before this change (or were simply not checking), so the integration simply involved passing the variables previously being bound into the search function which then performed the query and returned the results to the respective function.

Another integration step was integrating the duplication checks into the create/edit functions, this involved creating new calls to the search function to check before it was added/edited.

The integration between the interface and the search and edit functions was relatively easy, as the previous iteration had already defined the forms used to input the data into these functions. The only notable problem with integrating was how the search was called and it returned results. In this case the interface was expecting only one movieInfo struct at time, instead of a vector, so it had to be changed.

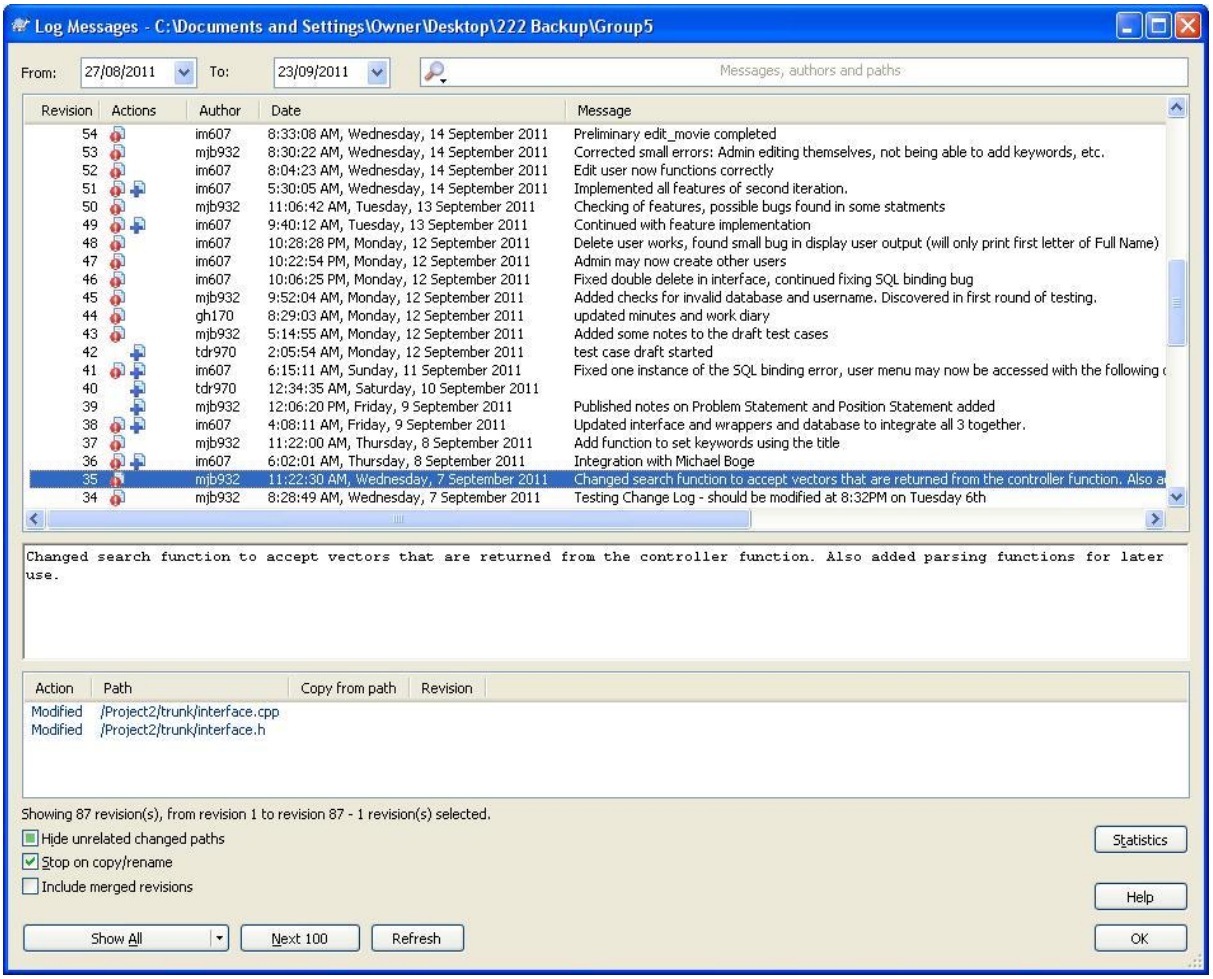
Notable defects for this iteration were:

- Incorrectly adding the multiple queries in the search function together, causing search to find movies which matched at least one of the queries instead of all of them. Fixed by changing the SQL from using UNION to IN

- Edits resulting in duplicated links between directors, etc. Fixed by adding the duplication checking to the edit functions
- Search for username not working. Fixed by adding a bind statement to add the username to all searches.

Version Control Information

The following screenshot shows an SVN log of the commits performed in the second iteration:



SVN log files from second iteration. Note the integration defect being fixed in the selected message. Also not the edit_movie function being completed at the top of the log.

Iteration 3 – Low Level Requirements and Finalising

This iteration involved implementing the low level functional requirements of the application, as well as correcting any problems overlooked from the previous stages. The steps in this iteration were:

- Create the importing function that reads the files given and adds them to the database
- Check for any possible problems that were overlooked
- Tweak user interface to finished level

Iteration Construction Summaries

In terms of the import function, this was quite simple, as it only needed to read in the files and then call the database controller to add them. The steps in this functions construction were:

- Creating a function to read the movies (movies.list)
- Creating a function to parse the input into a title an year
- Creating a function to read in the genres (genres.list)

This was the only real functionality added in this iteration, the rest simply involved tweaking and fixing other already implemented sections.

Testing

This iteration used a culmination of all the previous test cases, as it needed to ensure every part was working correctly. This meant it used both the script (black-box) and the unit testing (white-box).

In terms of the newly added import testing, it used the black-box method, as the import itself returned no values, and white-box testing for unexpected inputs was meaningless because of the fixed file format, it would simply import correctly or error (which could be checked through black-box easily).

Unit testing was also very important in this section, as we needed it to confirm all our sections worked. As such, we created unit tests for almost all functions, and used a number of invalid cases as well as valid (to test if it would continue to run and break other functions or not). An example of this was adding tests for deleting users with no username.

Additional cases were run to attempt to break the previous iterations work, such as:

- Entering newlines as input, which had to parsed out in the interface, or acted upon accordingly in the controller
- More search tests (to cover more possible combinations)
- A number of cases to test if the merge worked correctly (and it could change all parts of the movie being merged to)

Defects and Integration

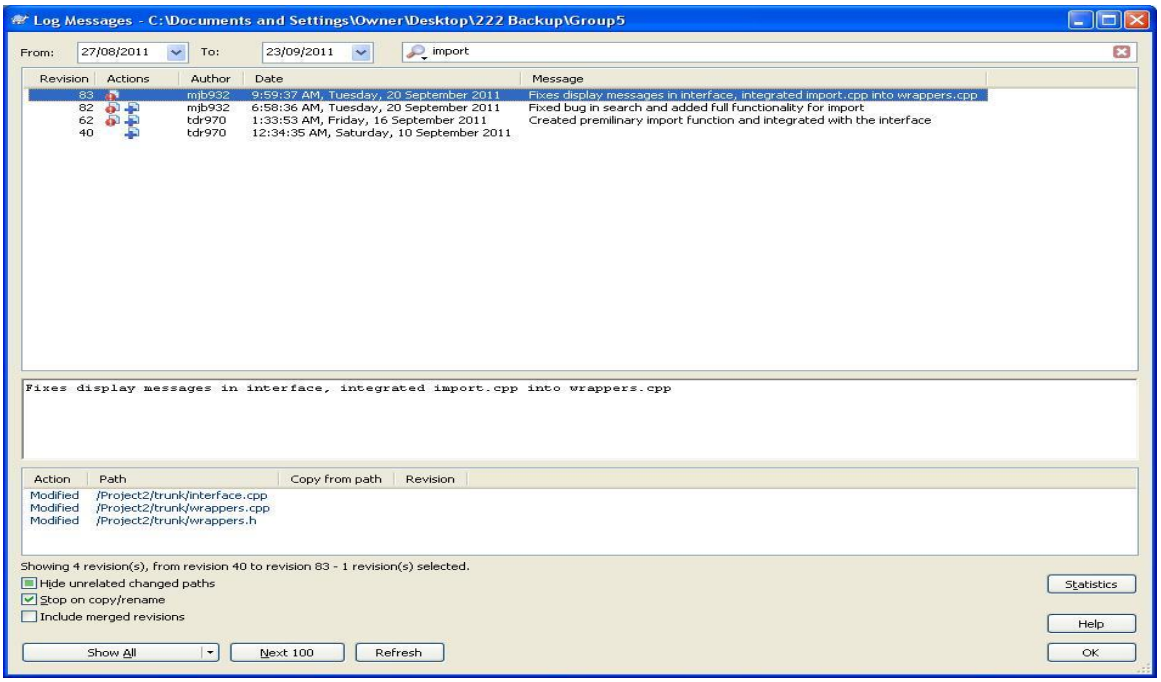
The only part that required integration in this section were the import functions. As the import function had been created separately from the objects, to allow it to be developed and tested separately, without worrying about integration at first. This meant that when the integration came time, a number of calls became irrelevant, so had to be stripped (such as it having its own keyword generation function, which was already defined in the movieController class). Other than these changes, there were no real problems integrating with the system.
The integration itself took slightly longer than expected though, as it involved checking over the integration from the previous iterations (and finding problems that were missed).

A large number of defects were found in this section, as it performed considerable testing on the other iterations as well. These defects were:

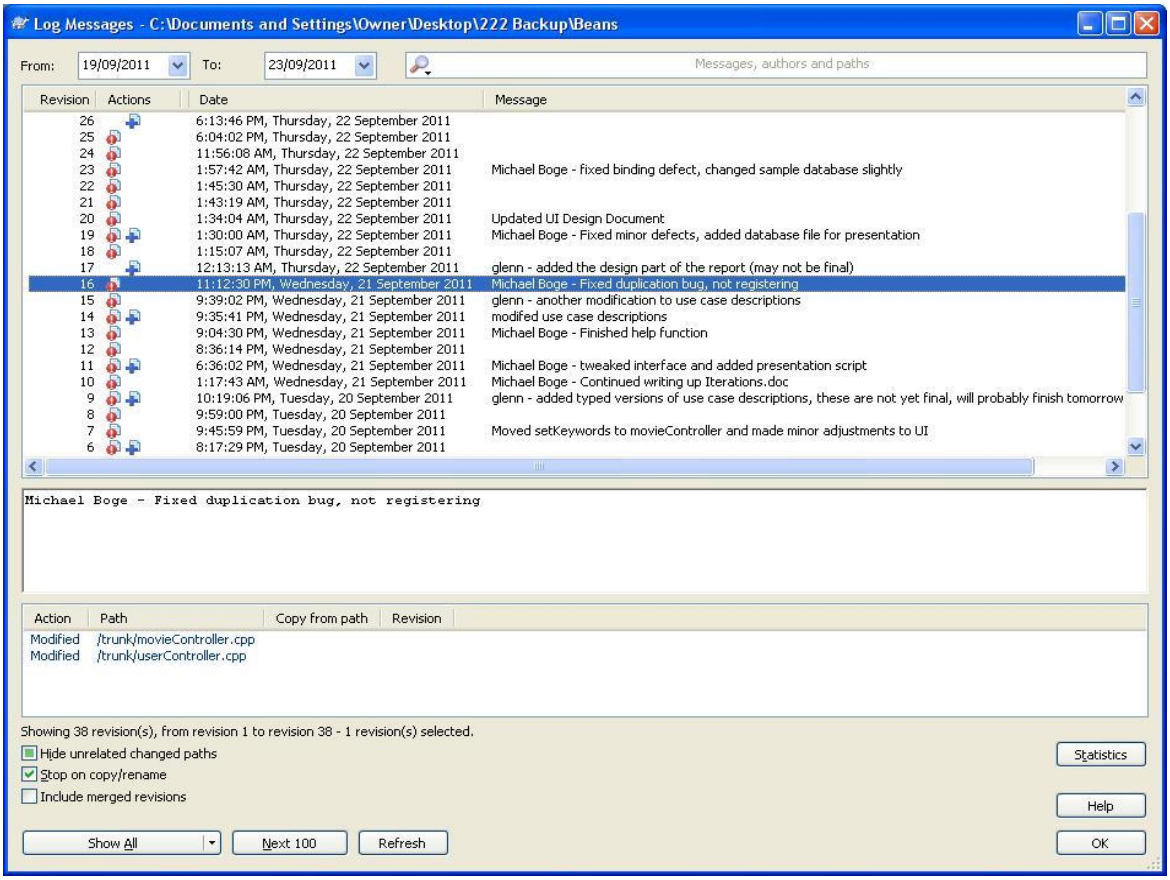
- Import incorrectly getting titles, by keeping the white space from the file format. Fixed by adding a trim function.
- Search function input being corrupted (such as if a string was entered as an int). Fixed by adding cin.clear()'s

Version Control Information

This iteration was much more difficult to track through SVN, as the Virindi SVN had problems at the beginning of this iteration, meaning we migrated onto another SVN (this time on Beanstalk), which only allowed one user access, meaning we had to share an account. We go around this by recording our names in our log messages, but this means we are unable to show the usage statistics easily.



Screenshot showing the import function being developed as per the log files (this displays only some of the logs, others were on the Beanstalk SVN)



Screenshot from the Beanstalk SVN. Note this screenshot of the final iteration focuses mainly on fixing problems (like the selected log) and documentation.

Testing

For testing we used black box testing in the forms of unit testing (cppUnit) for testing controller functions and a shell script for testing most of the interface. We quickly discovered that CppUnit was not very effective for most parts of this project as it is heavily dependent on user input. (See Appendices for sample code from CppUnit classes and the shell script)

Testing Report

The following shows our final testing report which was complete during and after the final construction iteration. It shows the feature that is being tested, various test cases for that feature, the expected response from each test case, the last date the test cases were run and the results of the last test.

------Program asks for database name------

Try:

1. Supply valid database name
2. Supply invalid database name
3. Supply valid database name with a variety of legal characters whitespace etc
4. Supply invalid database name with illegal characters
5. Supply extreme length database name
6. No input
7. Supply database without .db extension

Expect response:

1. Accept input and proceed
2. Error message
3. Accept input and proceed
4. Error message
5. Accept and proceed
6. Error message
7. Error message "Database must end with .db"

Last tested: 17/09/2011
Results: All passed

------Program asks for username------

Try:

1. Supply valid username
2. Supply valid username with a variety of legal characters whitespace etc
3. Supply extreme length username
4. No input

Expect response:

1. Accept and proceed
2. Accept and proceed
3. Accept and proceed
4. error message

Last tested: 17/09/2011
results: All passed

------Program asks for password------

Try:

1. Correct password for username
2. Incorrect password for correct username with < 3 attempts
3. Incorrect password for correct username with >= 3 attempts
4. Supply invalid password with illegal characters whitespace etc

Expect response:

1. Accept input and proceed
2. "incorrect password" error message
3. "incorrect password, account locked" error message
4. "incorrect password" error message

Last tested: 17/09/2011
Results: All passed

------Add Movie------

Try:

1. Fill in no fields
2. Fill in both required fields
3. Fill in only 1 of the required fields
4. Fill in only optional fields

5. Fill in both required and 1 optional
6. Fill in both required and 2 optional
7. Fill in both required and 3 optional
8. Fill in both required and 4 optional
9. Fill in both required and all optional fields
10. Fill in both required and varying numbers of optional fields (more than one of different fields)
11. Duplicate entry to existing movie

Expect response:
1. Error "Fields 1 and 2 required"
2. Success message
3. Error "Fields 1 and 2 required"
4. Error "Fields 1 and 2 required"
5. Success message
6. Success message
7. Success message
8. Success message
9. Success message
10. Success message
11. Error "Movie already exists"

Last tested: 20/09/2011
Results: All passed

------Edit Movie------

Try:
1. Empty or invalid movie title, empty or invalid year
2. Valid movie title, empty or invalid year
3. Invalid movie title, valid year
4. Valid title, year, same input as for add movie
5. Valid title, year and 1 optional
6. Valid title, year and 2 optional
7. Valid title, year and 3 optional
8. Valid title, year and 4 optional
9. Valid title, year and all optional fields
10. Valid title, year and varying numbers of optional fields (more than one of different fields)

Expect response:

1. Could not find movie
2. Could not find movie
3. Could not find movie
5. Success message
5. Success message
6. Success message
7. Success message
8. Success message
9. Success message
10. Success message

Last tested: 20/09/2011
Results: All passed

------Delete Movie------

Try:
1. Empty movie title, empty year
2. Valid movie title, invalid year
3. Invalid movie title, valid year
4. Valid movie title, valid year

When asked to confirm:
5. lower and upper case 'y'
6. lower and upper case 'n'
7. Any other input

Expect response:
1. Could not find movie
2. Could not find movie
3. Could not find movie
4. Movie deleted
5. Success message
6. Message stating that nothing was changed
7. Invalid input

Last tested: 20/09/2011
Results: All passed

------Show all movies------

Try:

1. using this option with an empty database
2. using this option with a database with movies in it

Expect response:
1. Message stating that there are no movies to be shown
2. Movies to be shown

Last tested: 18/09/2011
Results: All passed

------Show all movies by this user------

try:
1. using this option with an empty database
2. using this option with a database with movies in it submitted only by the user
3. using this option with a database with movies in it submitted by this user and others
4. using this option with a database with movies in it only by other users

Expect response:
1. Message stating that there are no movies to be shown
2. All movies to be shown
3. All movies to be shown only by this user
4. Message stating that there are no movies to be shown

Last tested: 18/09/2011
Results: All passed

------Search Movie------

Try:
1. Fill in no fields
2. Fill in both required fields
3. Fill in only 1 of the required fields
4. Fill in only optional fields
5. Fill in both required and 1 optional
6. Fill in both required and 2 optional
7. Fill in both required and 3 optional
8. Fill in both required and 4 optional
9. Fill in both required and all optional fields

10. Fill in both required and varying numbers of optional fields (more than one of different fields)

Expect response:
1. all movies shown
2. movies shown with given fields
3. movies shown with given fields
3. movies shown with given fields
3. movies shown with given fields
3. movies shown with given fields
3. movies shown with given fields
3. movies shown with given fields

Last tested: 20/09/2011
Results: All passed

------Add User------

Try:
1. Username only
2. Password only
3. Username and Password
4. Empty fields
5. Admin/user field random values
6. Correct admin/user values with correct user pass
7. Duplicate user

Expect response:
1. Error: user/pass fields required
2. Error: user/pass fields required
3. Accept input
4 Error: user/pass fields required
5. Error: invalid input
6. Accept input
7. Error "User exists"

Last tested: 19/09/2011
Results: All passed

------Edit User------

Try:

1. Valid username
2. Empty username
3. Invalid username

For reset password attempts:

4. lower and upper case 'y'
5. lower and upper case 'n'
6. Any other input

For editing user values

7. Username only
8. Password only
9. Username and Password
10. Empty fields
11. Admin/user field random values
12. Correct admin/user value
13. Duplicate user

Expect response:

1. Accept input and continue
2. Error "user doesnt exist"
3. Error "user doesnt exist"
4. Accept and continue
5. Accept and continue
6. Invalid input
7. Accept input
8. Accept input
9. Accept input
10 Accept input
11. Error: invalid input
12. Accept input
13. Error "User exists"

Last tested: 19/09/2011
Results: All passed

------Remove User------

Try:

1. Valid username
2. Invalid username

3. Empty username
4. for confirmation y lower and upercase
5. for confirmation n lower and upercase
6. for confirmation random input

Expect response:
1. accept input
2. error "user doesnt exist"
3. error "user doesnt exist"
4. success message
5. Message stating that no users were deleted
6. error "invalid input"

Last tested: 19/09/2011
Results: All passed

------Change Password------

Try:
1. Valid password
2. Invalid password
3. Empty password

Expect response:
1. Success message
2. Error "invalid input"
3. Error "invalid input"

Last tested: 19/09/2011
Results: All passed

------Import Movies------

Try:
1. To import into an empty database
2. To import after importing into the same database

Expect response:
1. Success message
2. Conflicts

Last tested: 20/09/2011

Results: All passed

------Show Help------

Try:
1. use this option

Expect response:
1. show help

Last tested: 16/09/2011
Results: All passed

------Quit------

Try:
1. Use this option

Expect response:
1. Quit program

Last tested: 15/09/2011
Results: All passed

Project Management

As this project was quite large project management and working well as a group was very important. As part of our project management we held meetings, kept work diaries, allocated tasks fairly, and used Subversion for version control.

Meetings

At the group's first informal meeting we all agreed to attend weekly formal meeting on Wednesdays at 12:30pm. During the project we held these meeting every week from week 5 (24/8) to week 9 (21/9). During the final week we held an additional meeting on the Monday to ensure that the final stages of the project would be done on time. The fact that every group member attended all of our formal meeting is a good indication of each member's dedication to the group and the project.

Our weekly meetings (with the exception of the first) generally followed fairly consistent pattern. First, each group member would give a progress report, detailing how they were going on any task they had been allocated in the previous meeting and any additional work they had managed to get done. Then we would discuss any issues that had arisen during the week. Finally, set goals for the next week and allocate task to be completed by the next meeting. These meeting generally lasted about an hour. (Our full meeting minutes can be found in Appendix 2)

In addition to our formal meetings, we had many informal "meetings" when required. These meeting were held when the opportunity arose for several group members to discuss the project. They were often short and not necessarily all members were there.

Work Diaries

Each group member was expected to maintain a work diary. In such a diary each member recorded the date, length of time and the activity they were working on any time that they did some work for the project. We decided that we would use these if any problems developed within the group due to members not pulling their weight. Fortunately, this never became an issue and each member's diary basically remained their own business. (Sampler work diaries can be found in Appendix 3)

Task Allocation

During the project we attempted to allocate task fairly and according to each member's strengths. The tables on the following pages show how we allocated role within the group and detail the artefacts that each member was responsible for and when they were delivered.

	Glenn	Ian	Michael	Tiaki
System analyst	X	X	X	X
User-Interface designer			X	
Data designer		X		
Designer	X	X	X	X
Lead integrator			X	
Lead implementer		X	X	
Test Designer				X
Tester	X	X	X	X
Technical writer	X			X
Project manager	X			

Milestone	**Artefact**	**Member Responsible**	**Completion Dates**	
			Planned	**Actual**
Pre-planning				
	Initial Project Plans (summarising specification)	Michael	31/8	31/8
	Iteration Plan	Michael	31/8	31/8
	Vision, Problem and Position Statements	Michael	31/8	31/8
	Initial Class Diagrams	Glenn/Tiaki	31/8	31/8
	Data Persistence (Database Schema)	Ian	31/8	31/8
	Initial Use Case Diagrams	Glenn/Tiaki	31/8	31/8
	Initial Sequence Diagrams	Glenn/Tiaki	31/8	7/9
	User Interface Design	Michael	7/9	7/9
	Learning SQLite and setting it up for use	Ian	31/8	31/8
Iteration 1 – High Level Requirements				
	Implementing Database Schema (create tables)	Ian	7/9	7/9
	Implementing high priority functionality	Ian/Michael	7/9	9/9
	Revised Use Case Diagrams	Glenn	7/9	7/9
	Revised Class Diagrams	Glenn	7/9	7/9
	High priority test cases	Tiaki	7/9	7/9

Iteration 2 – Medium Level Requirements				
	Implementing medium priority functionality	Ian/Michael	14/9	12/9
	Final Use Case Diagrams	Glenn	14/9	14/9
	Final Class Diagrams	Glenn	14/9	14/9
	Final Sequence Diagrams	Glenn	14/9	17/9
	Medium priority test cases	Tiaki	14/9	16/9
Iteration 3 – Low Level Requirements and Report				
	Implementing low priority functionality	Ian/Michael	19/9	21/9
	Low priority test cases	Tiaki	19/9	20/9
	Iteration report	Michael	21/9	22/9
	Functionality report	Ian	21/9	21/9
	Design report	Glenn	21/9	21/9
	Test report	Tiaki	21/9	22/9
	Project management report	All	23/9	23/9

Version Control

All of our group members used Subversion, through either the SVN client on Linux, the built in support in Netbeans or TortoiseSVN on Windows, during to project for version control. This provided two major benefits. Firstly, it allowed group members to make changes to implementation files or other documents without the fear of disrupting previously completed work. It also meant that each members work could be shared quickly and easily with all members of the group, allowing much easier collaboration between members.

We did experience a number of issues during the project with our use of Subversion. The biggest issue was that there were problems with the server in the last week of the project. To work around this we set up private SVN server on Beanstalk. Unfortunately, this only allowed for one account to be created making tracking the member responsible for each update more difficult.

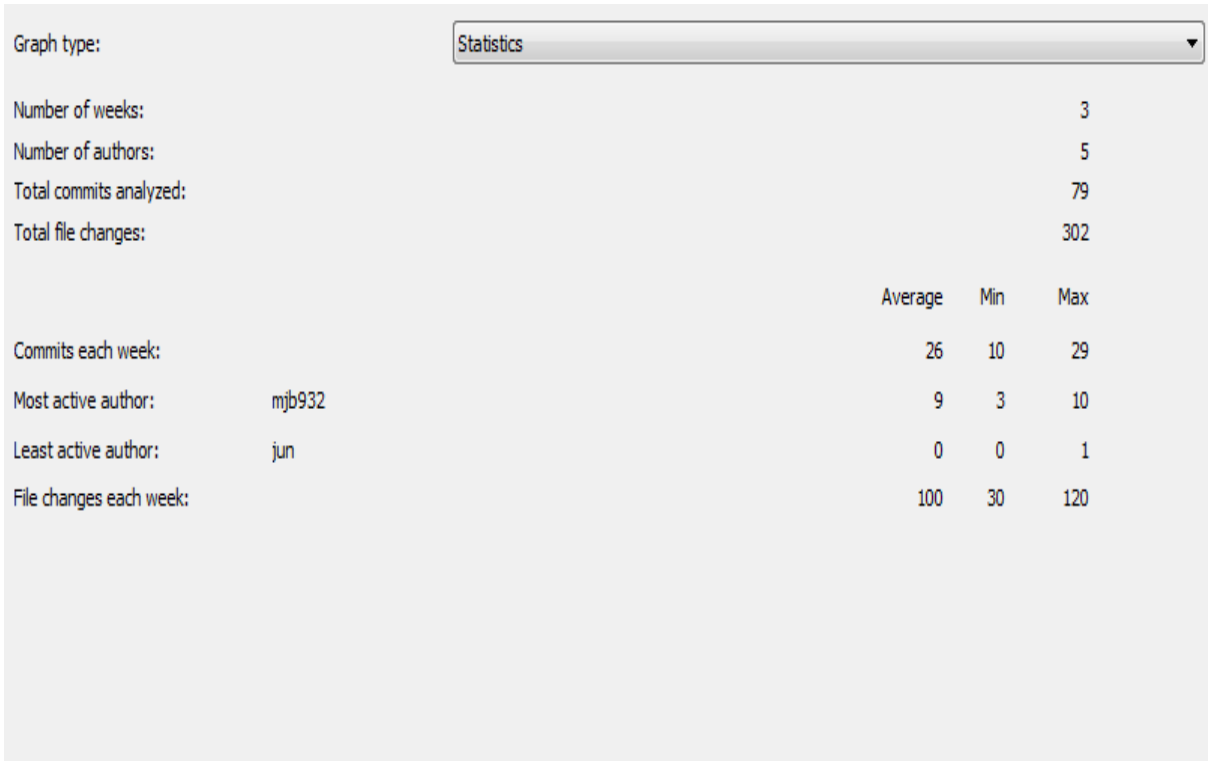
We also recognised the potential issue of having multiple people working on one file at once. To avoid this we made it a group policy to always lock any files w that we were working on so that other members would know not to work on them as well.

The following are screenshots of the SVN log on both the Virindi and Beanstalk servers respectively:

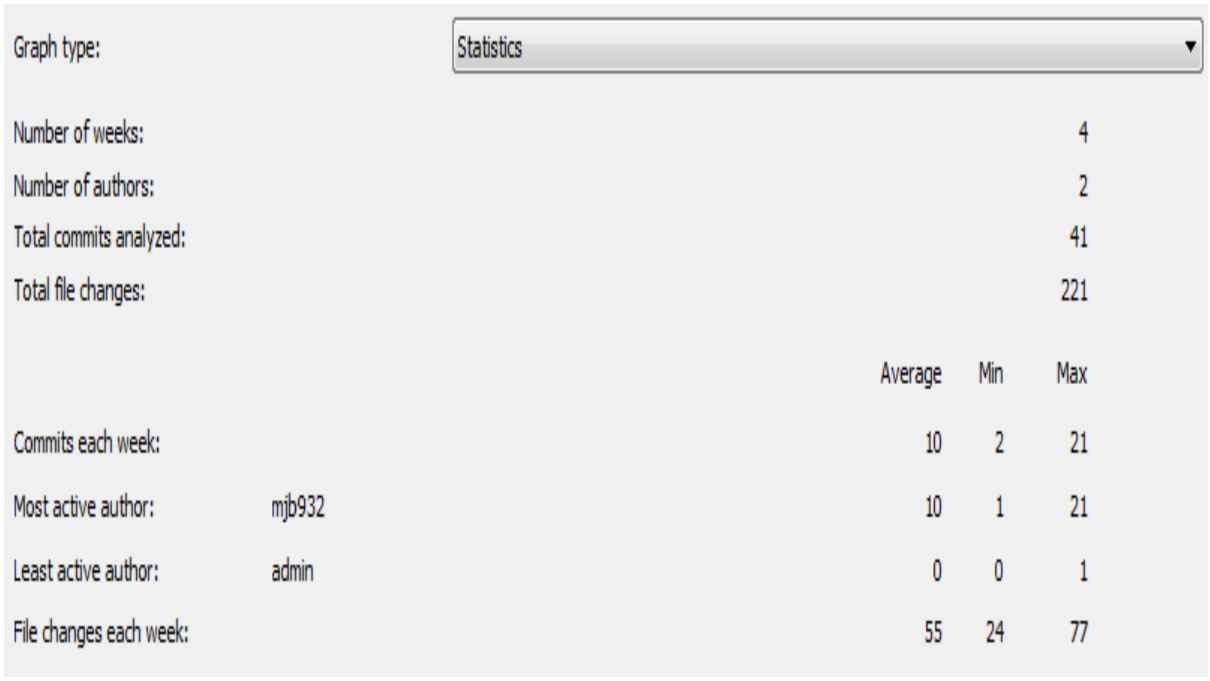
Revision	Actions	Author	Date	Message
87		mjb932	7:14:18 AM, Friday, 23 September 2011	Started compiling documentation, added screenshots
86		mjb932	10:09:08 PM, Thursday, 22 September 2011	Added backup of presentation files
85		tdr970	10:40:42 AM, Thursday, 22 September 2011	
84		mjb932	10:11:02 AM, Tuesday, 20 September 2011	Corrected compile errors in interface.cpp
83		mjb932	9:59:37 AM, Tuesday, 20 September 2011	Fixes display messages in interface, integrated import.cpp into wrappers.cpp
82		mjb932	6:58:36 AM, Tuesday, 20 September 2011	Fixed bug in search and added full functionality for import
81		tdr970	2:52:11 AM, Tuesday, 20 September 2011	
80		tdr970	1:52:28 AM, Tuesday, 20 September 2011	
79		tdr970	1:41:38 AM, Tuesday, 20 September 2011	
78		mjb932	1:51:28 PM, Monday, 19 September 2011	Added comments to methods
77		tdr970	11:14:41 AM, Monday, 19 September 2011	added test script (incomplete) modified the password input code for script purposes
76		mjb932	8:51:26 AM, Monday, 19 September 2011	Drafted introduction and iteration documents for final report
75		**gh170**	**7:58:12 AM, Monday, 19 September 2011**	**updated minutes and work diary**
74		gh170	7:12:42 AM, Monday, 19 September 2011	redid class and use case diagrams in rational rose and updated them, added most of t
73		tdr970	3:36:59 AM, Monday, 19 September 2011	removed ignore line causing menu problems
72		tdr970	3:19:45 AM, Monday, 19 September 2011	
71		tdr970	2:59:57 AM, Monday, 19 September 2011	updated test cases
70		mjb932	12:09:00 PM, Sunday, 18 September 2011	Added published drafts for: - Vision (published what we had written up earlier) - UI De
69		mjb932	7:37:09 AM, Sunday, 18 September 2011	Fixed menu reading error and possible double-free
68		tdr970	7:14:05 AM, Sunday, 18 September 2011	Added some more test cases, still going
67		tdr970	5:45:25 AM, Sunday, 18 September 2011	forgot the Makefile
66		tdr970	5:42:28 AM, Sunday, 18 September 2011	compiling this produces an executable...which does nothing
65		tdr970	2:21:24 AM, Sunday, 18 September 2011	added friend code to interface with compiler define TEST Started test cases, problem
64		tdr970	12:43:41 AM, Sunday, 18 September 2011	Fixed Boge's switch. Had two cases for '1'
63		mjb932	12:28:25 AM, Sunday, 18 September 2011	Small fixes to UI, corrected error with change password
62		tdr970	1:33:53 AM, Friday, 16 September 2011	Created premilinary import function and integrated with the interface
61		gh170	7:27:37 PM, Thursday, 15 September 2011	
60		im607	4:06:32 AM, Thursday, 15 September 2011	Completed features list including checks for duplicate identifiers when adding or editin
59		gh170	10:38:37 AM, Wednesday, 14 September 2011	added class diagram to visual studio stuff
58		im607	10:32:19 AM, Wednesday, 14 September 2011	Continued with edit_movie

Revision	Actions	Author	Date	Message
26		mjb932	6:13:46 PM, Thursday, 22 September 2011	
25		mjb932	6:04:02 PM, Thursday, 22 September 2011	
24		mjb932	11:56:08 AM, Thursday, 22 September 2011	
23		mjb932	1:57:42 AM, Thursday, 22 September 2011	Michael Boge - fixed binding defect, changed sample database slightly
22		mjb932	1:45:30 AM, Thursday, 22 September 2011	
21		mjb932	1:43:19 AM, Thursday, 22 September 2011	
20		mjb932	1:34:04 AM, Thursday, 22 September 2011	Updated UI Design Document
19		mjb932	1:30:00 AM, Thursday, 22 September 2011	Michael Boge - Fixed minor defects, added database file for presentation
18		mjb932	1:15:07 AM, Thursday, 22 September 2011	
17		mjb932	12:13:13 AM, Thursday, 22 September 2011	glenn - added the design part of the report (may not be final)
16		mjb932	11:12:30 PM, Wednesday, 21 September 2011	Michael Boge - Fixed duplication bug, not registering
15		mjb932	9:39:02 PM, Wednesday, 21 September 2011	glenn - another modification to use case descriptions
14		mjb932	9:35:41 PM, Wednesday, 21 September 2011	modifed use case descriptions
13		mjb932	9:04:30 PM, Wednesday, 21 September 2011	Michael Boge - Finished help function
12		mjb932	8:36:14 PM, Wednesday, 21 September 2011	
11		mjb932	6:36:02 PM, Wednesday, 21 September 2011	Michael Boge - tweaked interface and added presentation script
10		mjb932	1:17:43 AM, Wednesday, 21 September 2011	Michael Boge - Continued writing up Iterations.doc
9		mjb932	10:19:06 PM, Tuesday, 20 September 2011	glenn - added typed versions of use case descriptions, these are not yet final, will proba
8		mjb932	9:59:00 PM, Tuesday, 20 September 2011	
7		mjb932	9:45:59 PM, Tuesday, 20 September 2011	Moved setKeywords to movieController and made minor adjustments to UI
6		mjb932	8:17:29 PM, Tuesday, 20 September 2011	
5		mjb932	12:13:37 AM, Tuesday, 20 September 2011	Michael Boge - fixed compile error due to changes
4		mjb932	12:13:00 AM, Tuesday, 20 September 2011	
3		mjb932	12:00:40 AM, Tuesday, 20 September 2011	Michael Boge - More virindi transfer
2		mjb932	11:55:35 PM, Monday, 19 September 2011	Michael Boge - Loading from virindi
1		admin	11:35:28 PM, Monday, 19 September 2011	Creating initial repository structure

The following show the statistics from the Virindi and Beanstalk server respectively:

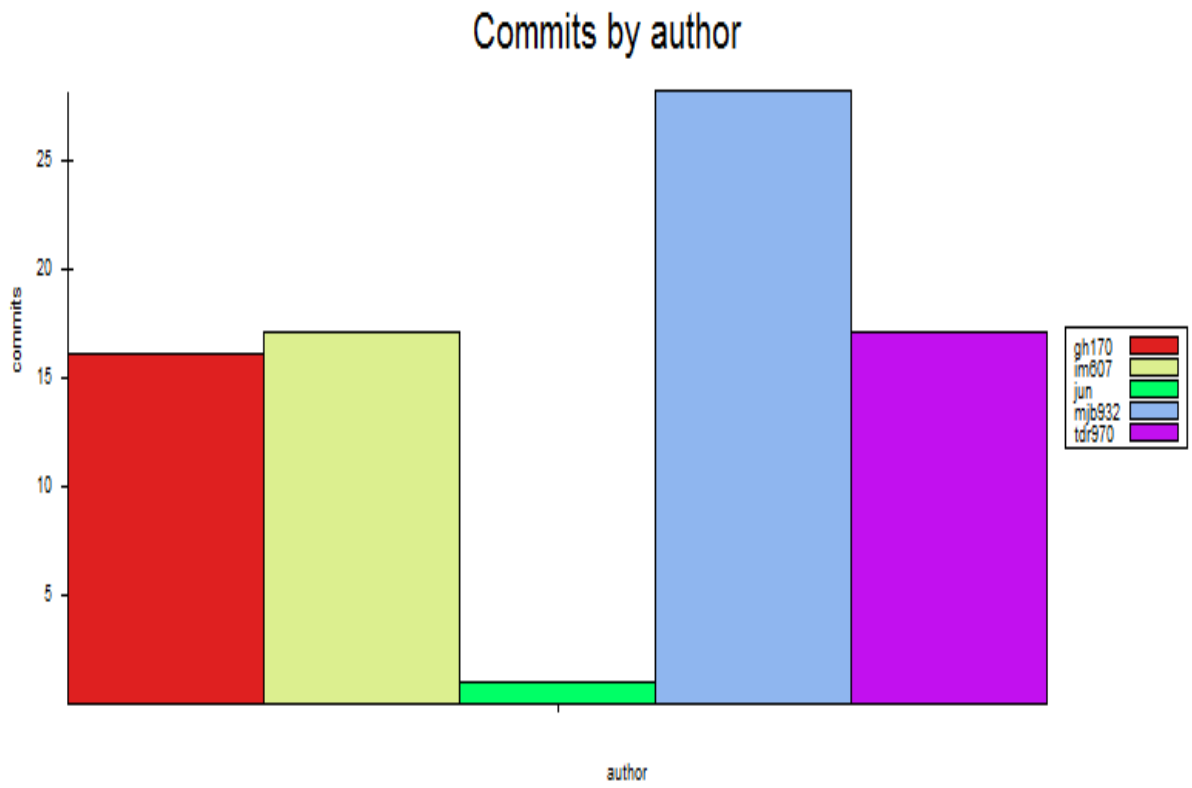


Graph type:	Statistics			
Number of weeks:				3
Number of authors:				5
Total commits analyzed:				79
Total file changes:				302
		Average	Min	Max
Commits each week:		26	10	29
Most active author:	mjb932	9	3	10
Least active author:	jun	0	0	1
File changes each week:		100	30	120



Graph type:	Statistics			
Number of weeks:				4
Number of authors:				2
Total commits analyzed:				41
Total file changes:				221
		Average	Min	Max
Commits each week:		10	2	21
Most active author:	mjb932	10	1	21
Least active author:	admin	0	0	1
File changes each week:		55	24	77

The following show the “commits by author” graph from the Virindi server:



Note: These are not a totally accurate representation of the entire project as a large number of commits were also performed on the Beanstalk server but individual author statistics were not available.

Appendices

Appendix 1: Use Case Descriptions

Use Case:	Login
Primary Actor:	User
Preconditions:	The user has been added to the system by an administrator
Success End Condition:	The user is logged in to the eMovies program
Failed End Condition:	The user is not logged in
Trigger:	The eMovies program is run

MAIN SUCCESS SCENARIO

1. The system prompts the user for their username
2. User enters username
3. The system prompts the user for their password
4. User enters their password
5. System checks that username and password are valid
6. System displays main menu for user

EXTENSIONS

5a. Username and password are not a valid combination
    5a1. User is prompted for username and password again
5b. Username is valid but incorrect password enter for third time
    5b1. System locks user out
    5b2. System displays message telling user they are locked out

Use Case:	Change password

Primary Actor:	User

Preconditions:	The user is logged in

Success End Condition:	The user's password is changed

Failed End Condition:	The user's password is unchanged

Trigger:	User selects change password in main menu

MAIN SUCCESS SCENARIO
1. User selects change password in main menu
2. System prompts user to enter new password
3. User enters new password
4. System validates new password
5. System set user's password to the new password
6. Control returns to main menu

EXTENSIONS
4a. Password is not valid
2a1. System displays message informing that the password was not valid and that their password has not been changed

Use Case:	Browse movies alphabetically

Primary Actor:	User

Preconditions:	The user is logged in

Success End Condition:	An alphabetical listing of all movies is displayed

Failed End Condition:	An error message is displayed

Trigger:	User selects show all movies in main menu

MAIN SUCCESS SCENARIO
1. User selects show all movies in main menu
2. System displays an alphabetical listing of all movies in database
3. Control returns to main menu

EXTENSIONS
2a. No movies are present in database
2a1. System displays message informing user that there are no movies in the database

Use Case:	Display current user's movies

Primary Actor:	User

Preconditions:	The user is logged in

Success End Condition:	A list of movies created by current user is displayed

Failed End Condition:	No movies are displayed

Trigger:	User selects search all movies in main menu

MAIN SUCCESS SCENARIO
1. User selects display all user movies in main menu
2. System displays a list of movies created by current user
3. Control returns to main menu

EXTENSIONS
2a. User has not created any movies
2a1. System displays message informing user that there are no movies created by user were found

Use Case:	Search movies

Primary Actor:	User

Preconditions:	The user is logged in

Success End Condition:	A list of movies matching search criteria is displayed

Failed End Condition:	No movies are displayed

Trigger:	User selects search all movies in main menu

MAIN SUCCESS SCENARIO
1. User selects search movies in main menu
2. Enter movies information (see separate use case description) for search criteria
3. System search database for movies matching search criteria
4. System displays a list of matching movies
5. Control returns to main menu

EXTENSIONS
4a. No matching movies are present in database
4a1. System displays message informing user that there are no matching movies were found

Use Case:	Create movie

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	A new movie record is created

Failed End Condition:	A new movie record is not created

Trigger:	Administrator selects create movie in main menu

MAIN SUCCESS SCENARIO

1. Administrator selects create movie in main menu
2. Enter movies information (see separate use case description)
3. System check that a title and release year have been supplied
4. System creates a new movie record
5. Control return to main menu

EXTENSIONS

4a. No title or release year have been supplied
    4a1. System displays message informing that a title and release year must be supplied
    4a2. Control returns to step 2

4b. Title and release date have been match another entry in database
    4b1. System displays a message informing that the title and release date are not unique and that the creation has not been completed
    4b2. Control returns to main menu

Use Case:	Edit movie

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	An existing movie record is modified

Failed End Condition:	Nothing is modified

Trigger:	Administrator selects edit movie in main menu

MAIN SUCCESS SCENARIO

1. Administrator selects edit movie in main menu
2. System prompts administrator to enter title and release date of movie to edit
3. Administrator enters movie title
4. System displays the existing details for the movie
5. Enter movies information (see separate use case description)
6. System check that a title and release year are not blank
7. System modifies the selected movie record
8. System display a message confirming the modification of the new record
9. Control returns to main menu

EXTENSIONS

4a. Matching movie could not be found
    4a1. System displays message telling user that the movie could not be found
    4a2. Control returns to main menu

7a. Title or release date have been changed to blank
    7a1. System displays message informing that a title and release year cannot be blank
    7a2. Control returns to step 5

7b. New title and release date have been match another entry in database
    7b1. System displays a message informing that the title and release date are not unique and that the modification has not been completed
    7b2. Control returns to main menu

Use Case:	Enter movie information

Primary Actor:	User/Administrator

Preconditions:	-A user is logged in and performing a movies search operation -An administrator is logged in and performing a Create or Edit movie operation

Success End Condition:	New movie information is obtained by the system

Trigger:	A user/administrator is performing an operation that required the entry of movie information

MAIN SUCCESS SCENARIO
1. The system prompts user to select a type of movie information to add
2. User selects an option
3. System prompts user to enter the selected information
4. User enters the information
5. Repeat steps 2 to 5
6. User selects execute creation

Use Case:	Delete movie

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	An existing movie record is deleted

Failed End Condition:	Nothing is modified

Trigger:	Administrator selects delete movie in main menu

MAIN SUCCESS SCENARIO
1. Administrator selects delete movie in main menu
2. System prompts administrator to enter title and release date of movie to delete
3. Administrator enters movie title and release date
4. System displays the existing details for the movie
5. The system prompts administrator to confirm deletion
6. Administrator selects yes
7. System deletes movie record
8. Control returns to main menu

EXTENSIONS
4a. Matching movie could not be found
    4a1. System displays message telling user that the movie could not be found
    4a2. Control returns to main menu

Use Case:	Create user

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	A new user record is created

Failed End Condition:	Nothing is modified

Trigger:	Administrator selects create user in main menu

MAIN SUCCESS SCENARIO

1. Administrator selects create user in main menu
2. Enter user information (see separate use case description)
3. System check that a username and password have been supplied
4. System creates a new user record
5. Control return to main menu

EXTENSIONS

4a Username or password have not been entered
- 4a1. System displays message informing administrator that a username and password must be supplied
- 4a2. Control returns to step 2

4b. Username already exists
- 4b1. User record is not created
- 4b2. System displays message informing administrator that the username already exists
- 4b3. Control returns to main menu

Use Case:	Edit user

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	An existing user record is modified

Failed End Condition:	Nothing is modified

Trigger:	Administrator selects edit user in main menu

MAIN SUCCESS SCENARIO

1. Administrator selects edit user in main menu
2. System prompts administrator to enter the username of the user to edit
3. Administrator enters username
4. System displays the existing details for the user
5. Enter user information (see separate use case description)
6. Administrator selects execute creation
7. System check that a username and password are not blank
8. System edits a new user record
9. Control return to main menu

EXTENSIONS

8a A blank username or password has been entered
- 8a1. System displays message informing administrator that a username and password must not be blank
- 8a2. Control returns to step 5

8b. Username already exists
- 8b1. User record is not edited
- 8b2. System displays message informing administrator that the username already exists
- 8b3. Control returns to main menu

Use Case:	Enter user information

Primary Actor:	Administrator

Preconditions:	Administrator is logged in and performing Create User or Edit User operation

Success End Condition:	New user information is obtained by the system

Trigger:	Administrator need to add new user information during Create or Edit User operation

MAIN SUCCESS SCENARIO
1. The system prompts administrator to select a type of user information to add
2. Administrator selects an option
3. System prompts administrator to enter the selected information
4. Administrator enters the information
5. Repeat steps 2 to 5 as required
6. Administrator selects execute query

EXTENSIONS
5a. A password is entered
5a1. System validates the password
5a2. Control returns to step 1

Use Case:	Delete user

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	An existing user record is deleted

Failed End Condition:	Nothing is modified

Trigger:	Administrator selects delete user in main menu

MAIN SUCCESS SCENARIO
1. Administrator selects delete user in main menu
2. System prompts administrator to enter username of user to delete
3. Administrator enters username
4. System displays the details for the user
5. The system prompts administrator to confirm deletion
6. Administrator selects yes
7. System deletes user record
8. Control returns to main menu

EXTENSIONS
4a. Matching user could not be found
4a1. System displays message telling administrator that the user could not be found
4a2. Control returns to main menu
4b Username entered is the current administrator's username
4b1. System displays message informing administrator that they cannot delete self
4b2. Control returns to main menu

Use Case: Import movies

Primary Actor:	Administrator

Preconditions:	An administrator is logged in

Success End Condition:	A new movie record is created for each movie in the a pair of title and genre files

Failed End Condition:	No record is modified

Trigger:	Administrator selects import movies in main menu

MAIN SUCCESS SCENARIO
1. Administrator selects import movie in main menu
2. The system creates a new record for each movie listed in the files
3. Control returns to main menu

EXTENSIONS
2a. Movie matching title and release date in file already exists
    2a1. Import duplicate movie (see separate use case description)
    2a2. Control resumes at step 2

Use Case:	Import duplicate movie

Primary Actor:	Administrator

Preconditions:	Import operation is in progress and duplicate is detected

Success End Condition:	Existing record is modified to include new information

Failed End Condition:	No record is modified

Trigger:	Duplicate movie is detected during import operation

MAIN SUCCESS SCENARIO
1. System prompts administrator to choose either merge skip or abort
2. Administrator selects merge
3. System modifies exiting movie record to include any new information in the title and genre files
4. Import operation resumes

EXTENSIONS
2a. Administrator select skip
    2a1. Existing record is not modified
    2a2. Import operation resumes
2b. Administrator select abort
    2b1. Existing record is not modified
    2b2. Import operation is terminated
    2b3. Control returns to main menu

Appendix 2: Meeting minutes

Week 4 – 18/8

- Informal meeting in lab time
- Discussed possible roles for each member and division of tasks
- Organised time for future meetings – Wednesdays at 12:30 each week

Week 5 – 24/8

- Assigned roles
    - Manager – Glenn
    - Implementation/integration – Michael
    - Database design/implementation – Ian
    - Integration and testing – Tiaki
    - Design - All
    - Other tasks will be shared as will implementation work when possible/appropriate
- Decided to use SQL lite database to store program data
- Decided to use text based user interface
- Came up with a list of starting use cases
- Discussed possible design of classes to be used an came up with initial (very) rough class diagram
- Decided on tasks to be completed by next week
    - initial vision - Michael
    - initial problem and position statements - Michael
    - initial use case diagram – Tiaki & Glenn
    - initial class diagram – Tiaki & Glenn
    - initial sequence diagrams for the high priority use cases – Tiaki & Glenn
    - look into how to use SQL lite to implement - Ian

Week 6 – 31/8

- Progress reports from all members
    - Michael has written initial vision and problem and position statements
    - Tiaki and Glenn have produced initial use case and class diagram and sequence diagrams for high priority use cases (not on computer yet)
    - Ian has been trying out SQL lite and thinks it won't be too hard
- Decided on tasks to be completed by next week
    - stubs for all required classes including functions for high priority functionality – Ian & Michael
    - start on high importance functionality - Ian & Michael
    - further sequence diagrams – Glenn

- start on test cases - Tiaki

Week 7 – 7/9

- Progress reports from all members
  - Everything going pretty well
  - Ian and Michael have done a lot of work on implementing initial functionality
  - Most high priority functionality working
  - Glenn has completed rough sequence diagrams for all remaining use cases (still not on computer)
  - Tiaki has developed a number of test cases
- Discussed refinements to use case diagrams
- Discussed a number of questions about requirements that have arisen in the last week
- Tasks to be completed for next week
  - further implementation (finish high priority and do medium priority) – Ian and Michael
  - refined use case diagrams - Glenn
  - sequence diagrams on computer - Glenn
  - continue on test cases - Tiaki
  - extra documentation - All

Week 8 – 14/9

- Progress reports
  - Everything seems to be on track to finish with time to spare
  - Ian and Michael have managed to get ahead with coding, almost all done, just low priority functionalities to implement
  - Tiaki going well with test cases, should be finished after the weekend
  - Glenn has started producing final UML diagrams in Rational Rose
  - Doco going well, moving onto final versions and the final report
- Decided to split Wrappers class into two – MovieController and UserController, makes more sense and makes code more manageable – diagrams will need to be modified
- Tasks to be completed for next week
  - finish implementation - Ian and Michael
  - finish UML - Glenn
  - testing when possible – Tiaki with help from Ian and Michael
  - Start on various part of report - All

Week 9 – 19/9

- Present to tutors on Thursday
- Progress reports
  - Michael and Ian have finished code
  - Testing is going well

  - Glenn has finished most of the final versions of the UML diagrams
  - Need more work on the final report
- Discussed how we should divide up the remaining sections of the report
  - Ian will write about functionality
  - Michael will write about Iteration and introduction
  - Glenn will write up design
  - Tiaki will do testing section
- Other than that everyone will help with testing and debugging

Week 9 – 19/9

- Almost done now
- Worked on a script/checklist for the presentation
- Discussed what else needs doing for the report

Appendix 3: Work Diary Samples

Sample from Glenn's Work Diary:

4/8

- 30min – informal group meeting in lab time

23/8

- 30min – reading spec

24/8

- 1h – formal group meeting
- To do for next meeting:
  - use case diagrams
  - class diagram
  - sequence diagram for high priority use cases

25/8

- 1.5h – design work as a group in lab time, produced use case and class diagrams on paper

28/8

- 30min – typed up meeting minutes and work diary

29/8

- 1hr – worked on sequence diagrams for high priority functionality with Tiaki (only on paper)

31/8

- 1hr – formal group meeting
- To do for next meeting:
  - more sequence diagrams (medium/low priority)

3/9

- 1hr – coding password encryption and validation functions

4/9

- 30min – testing and debugging password encryption and validation functions

7/9

- 30min – formal group meeting
- To do for next week:
  - refine use case diagram (add discussed includes and extensions)
  - prepare UML diagram in rational rose

8/9

- 2hr – group work in lab time (mostly code inspection and debugging)

Sample from Michael's Work Diary:

8/9

- Corrected problem with ID's together with Ian (3 hours)
- Informal meeting with group in lab, spent time debugging and planning design for second iteration (1 hour)

9/9

- Added the rest of the function and menu bodies for interface, completing first draft for it (not integrated yet, simply available to be called) (2 hours)
- Integrated Glenn's password functions into the interface (1 hour)

10/9

- Tested integration with password functions, no defects found (1 hour)
- Published Problem and Position statements onto SVN (1 hour)

11/9

- Began testing on interface functions, defect found for login, not registering lockout. Fixed (2 hours)

12/9

- Integrated functions for editing and searching from Ian's code (3 hours)
- Began testing this integration (1 hour)

13/9

- Testing with Ian, found defects in integration:
  - Search not only searching for relevant users if required. Fixed by adding bind statement (2 hours)
  - Prompting error if string entered instead of int. Fixed using cin.clear(). (1 hour)
  - Trying to get struct from search in interface, where it should be getting vector. Fixed (1 hour)
  - Searches not getting required results, due to invalid joins. Fixed (1 hour)

14/9

- Formal group meeting, discussed progress and found defects, decided to continue on as going (1 hour)
- Planned to start on integrating for final iteration

15/9

- Informal meeting in lab, discussed how we were to start documentation (1 hour)

16/9

- Integrated Tiaki's preliminary import function with interface, not finished yet (2 hours)

Appendix 4: Final Product Code Listing

main.cpp

The main function simply create an instance of the Interface and calls its logon function.

```
#include "interface.h"

int main()
{
	Interface emovies;
	emovies.logon();

	return 0;
}
```

Interface Class

The Interface class used to present the menus and choices to the user to allow them to call the controller's functions.

interface.h

```
/***
*       Authors:        Michael Boge (menus/forms and design), Glenn Harper (password functions)
*       Modification Date:      21/9/2011
*       File Description:       This file contains the definition of the Interface class used to
*                                       present the menus and choices to the user to allow them to call the
*                                       controller's functions
***/

#ifndef INTERFACE_H
#define INTERFACE_H

#include "userController.h"
#include "movieController.h"
#include "userInfo.h"
#include "movieInfo.h"

const int SHIFT = 23;           //must be positive int between 1 and 61 inclusive

class Interface
{
        public:
                Interface();
                ~Interface();
                void logon();
        private:
                userController* userctrl;
                movieController* moviectrl;
                database *dBase;
                //Holds the information of the user who is currently logged in
                userInfo currUser;

                void mainMenuUser();
                void mainMenuAdmin();
                void displayHelp();

                //Movie functions
                bool createMovie();
                bool editMovie();
                bool deleteMovie();
                bool showAllMovies();
                bool showAllUserMovies();
                bool searchMovies();
                void resetMovie(movieInfo&);
                void readMovie(movieInfo&, bool);
                void displayMovie(movieInfo&);

                //User functions
                bool createUser();
```

```
        bool editUser();
        bool deleteUser();
        bool changePassword();
        void resetUser(userInfo&);
        void readUser(userInfo&, string&, bool);
        void displayUser(userInfo&);

        //Login functions - Written by Glenn
        string encryptPassword(string);
        bool validatePassword(string);

#ifdef TEST
        friend class eMoviesTest;
#endif
};

#endif
```

interface.cpp

```
/***
*       Authors:        Michael Boge (menus/forms and design), Glenn Harper (password functions)
*       Modification Date:      21/9/2011
*       File Description:       This file contains the implementation of the Interface class used to
*                                       present the menus and choices to the user to allow them to call the
*                                       controller's functions
***/

#include <iostream>
#include <unistd.h>             //For getPass()
#include <cctype>
#include "interface.h"
#include "movieInfo.h"
#include "userController.h"
#include "movieController.h"
#include "userInfo.h"

using namespace std;

Interface::Interface()
{
        string db = "";

        while (db == "")        //Ensures a database name is entered
        {
                cout << "Please enter the database name (a .db file) to load, or the name of the database you wish to
create: ";
                getline(cin, db);
                for(unsigned int i = 0; i < db.length() - 3; i++)
                {
                        if(!isalnum(db[i]) && db[i] != '_' && db[i] != '-')
                        {
                                cout << "Invalid database name" << endl;
                                db = "";
```

```
                    break;
                }
            }
            if(db[db.length()-3] != '.' && db[db.length()-2] != 'd' && db[db.length()-1] != 'b')
            {
                cout << "Invalid database name, must end name with .db extension" << endl;
                db = "";
            }
        }
        char* conversion = new char [db.size()+1];
        strcpy(conversion,db.c_str());
        dBase = new database(conversion);
        userctrl = new userController(dBase);   //Calls database constructor to open/create the database file
        moviectrl = new movieController(dBase);
}

Interface::~Interface()
{
        delete userctrl;            //Call deconstructor to close database (saving data)
        delete moviectrl;
        delete dBase;
}

void Interface::logon()
{
        string username;
        string password = "";
        string cryptedPass;
        int numLoops = 0;

        enum status {USER_VALID = 0,ADMIN_VALID = 1,INVALID = 2,LOCKED = 3};    //Values returned by
                                                                                //the user controller login function
        status loginStatus = INVALID;

        while(numLoops < 3 && loginStatus == INVALID)   //Loops 3 times or until the user is locked or logs in
                                                        // succesfully
        {
            username = "";
            while(username == "")
            {
                cout << "Please enter username: ";
                getline(cin, username);
            }
            password = getpass("Please enter password: ");  //Get the password for login
            if (validatePassword(password))
            {
                cryptedPass = encryptPassword(password);        //Encrypt the password and pass it to the login
                                                                // function for comparison
                loginStatus = static_cast<status>(userctrl->user_login(username,cryptedPass));  //Call the login
                                                                // function, getting back a return relative to the situation
                switch (loginStatus)
                {
                    case USER_VALID:        //Login
                        currUser.username = username;
                        cout << "Logging in as user" << endl;
                        mainMenuUser();
```

```
                    break;
                case ADMIN_VALID:       //Will call the relevant menu depending on if they are a user or
                                                    //admin
                    currUser.username = username;
                    cout << "Logging in as administrator" << endl;
                    mainMenuAdmin();
                    break;
                case INVALID:   //Incorrect username, or password for a correct username
                    cout << "Invalid username or password." << endl;
                    numLoops++;
                    break;
                case LOCKED:    //Returned after 3 attempts
                    cout << "User has made the maximum 3 attempts at password, account locked.\nPlease
contact the administor to unlock your account." << endl;
                    numLoops++;
                    break;
                default:
                    cout << "An error has occurred with the database" << endl;
                    numLoops++;
                    break;
            }
        }
        else
        {
            cout << "Invalid password entered." << endl;
            numLoops++;
        }
    }
    if(numLoops >= 3)
        cout << "Unable to login after 3 attempts, eMovies exiting..." << endl;

}

//Menu displayed to the user
void Interface::mainMenuUser()
{
    while (true)    //Loops until program is quit
    {
        char choice = ' ';

        cout << endl << "1: Show All Movies" << endl;
        cout << "2: Show All Movies Created by this User" << endl;
        cout << "3: Search For Movies" << endl;
        cout << "4: Change Password" << endl;
        cout << "I: Import Movies" << endl;
        cout << "H: Show Help" << endl;
        cout << "Q: Quit" << endl;
        cout << "Please enter a menu selection: ";
        choice = cin.get();
        cin.ignore(100, '\n');
        cout << endl;
        switch(choice)
        {
            case '1':
                showAllMovies();
                break;
```

```
                case '2':
                        showAllUserMovies();
                        break;
                case '3':
                        searchMovies();
                        break;
                case '4':
                        changePassword();
                        break;
                case 'I':
                case 'i':
                        moviectrl->importMovies(currUser.username);
                        break;
                case 'H':
                case 'h':
                        displayHelp();
                        break;
                case 'q':
                case 'Q':
                        cout << "Closing Program...";
                        return;
                default:
                        cout << "Invalid choice." << endl;
        }

    }

}

//Menu and choices displayed to the admin
void Interface::mainMenuAdmin()
{
    while(true)    //Loops until program is quit
    {
        char choice = ' ';

        cout << endl << "1: Create Movie" << endl;
        cout << "2: Edit Movie" << endl;
        cout << "3: Delete Movie" << endl;
        cout << "4: Show All Movies" << endl;
        cout << "5: Show All Movies Created by this User" << endl;
        cout << "6: Search Movie" << endl;
        cout << "7: Add User" << endl;
        cout << "8: Edit User" << endl;
        cout << "9: Remove User" << endl;
        cout << "0: Change Password" << endl;
        cout << "I: Import movies" << endl;
        cout << "H: Show Help" << endl;
        cout << "Q: Quit" << endl;
        cout << "Please enter a menu selection: ";
        choice = cin.get();
        cin.ignore(100, '\n');
        cout << endl;
        switch(choice)
        {
            case '1':
```

```
                createMovie();
                break;
            case '2':
                editMovie();
                break;
            case '3':
                deleteMovie();
                break;
            case '4':
                showAllMovies();
                break;
            case '5':
                showAllUserMovies();
                break;
            case '6':
                searchMovies();
                break;
            case '7':
                createUser();
                break;
            case '8':
                editUser();
                break;
            case '9':
                deleteUser();
                break;
            case '0':
                changePassword();
                break;
            case 'I':
            case 'i':
                moviectrl->importMovies(currUser.username);
                break;
            case 'H':
            case 'h':
                displayHelp();
                break;
            case 'q':
            case 'Q':
                cout << "Closing Program..." << endl;
                return;
            default:
                cout << "Invalid choice." << endl;
        }

    }

}

void Interface::resetMovie(movieInfo& movieTemp)
{
    movieTemp.title = "";
    movieTemp.release = 0;
    movieTemp.runtime = 0;
    movieTemp.directors.clear();
    movieTemp.genres.clear();
```

```
	movieTemp.countries.clear();
	movieTemp.keywords.clear();
}

bool Interface::createMovie()
{
	movieInfo movieTemp;

	resetMovie(movieTemp);  //Reset for inputting

	readMovie(movieTemp, false);
	movieTemp.username = currUser.username;
	if(moviectrl->create_movie(movieTemp))
	{
		cout << "Movie created successfully" << endl;
		return true;
	}
	else
	{
		cout << "Movie could not be created" << endl;
		return false;
	}

}

bool Interface::editMovie()
{
	movieInfo movieTemp;
	vector<movieInfo> results;

	resetMovie(movieTemp);

	cout << "Please Enter the Movie Title:" << endl;
	getline(cin, movieTemp.title);
	cout << "Please Enter the Movie Release Year:" << endl;
	cin >> movieTemp.release;
	if(cin.fail())
	{
		cin.clear();
		cin.ignore(100, '\n');
		cout << "Invalid release year entered." << endl;
		return false;
	}
	cin.ignore(100, '\n');

	results = moviectrl->search_movie(movieTemp);   //Search for movie to edit

	if(results.empty())     //Returned if not found
	{
		cout << "Movie could not be found." << endl;
		return false;
	}

	cout << "Movie Returned" << endl;
	displayMovie(results[0]);
```

```
	movieTemp = results[0];

	//Read in movie details
	readMovie(results[0], false);

	if(moviectrl->edit_movie(results[0],movieTemp)) //Pass to controller for editing
	{
		cout << "Movie edited successfully" << endl;
		return true;
	}
	else
	{
		cout << "Movie could not be edited" << endl;
		return false;
	}

}

bool Interface::deleteMovie()
{
	char choice = ' ';
	movieInfo movieTemp;
	vector<movieInfo> results;

	resetMovie(movieTemp);

	cout << "Please Enter the Movie Title:" << endl;
	getline(cin, movieTemp.title);
	cout << "Please Enter the Movie Release Year:" << endl;
	cin >> movieTemp.release;
	if(cin.fail())
	{
		cin.clear();
		cin.ignore(100, '\n');
		cout << "Invalid release year entered." << endl;
		return false;
	}
	cin.ignore(100,'\n');

	results = moviectrl->search_movie(movieTemp);   //Search for movie to delete

	if(results.empty())     //If not found return false
	{
		cout << "Movie could not be found." << endl;
		return false;
	}

	cout << "Movie Returned" << endl;
	displayMovie(results[0]);
	//Display movie in question and ask for permission to delete
	cout << "Are you sure you would like to delete? (y/n) ";
	choice = cin.get();
	cin.ignore(100, '\n');
	if(choice != 'y' && choice != 'Y' && choice != 'n' && choice != 'N')
	{
		cout << "Invalid choice made, movie will not be deleted." << endl;
```

```
        }
        if(choice == 'Y' || choice == 'y')
        {
                if(moviectrl->delete_movie(results[0]))
                {
                        cout << "Movie deleted successfully" << endl;
                        return true;
                }
                else
                {
                        cout << "Movie could not be deleted" << endl;
                        return false;
                }
        }
        return true;
}

void Interface::readMovie(movieInfo& movieTemp, bool search)
{
        string tempIn;

        while(true)     //Continues until query is executed
        {
                char choice = ' ';

                cout << endl << "1: Add/Change Title" << endl;
                cout << "2: Add/Change Release Year" << endl;
                cout << "3: Add/Change Runtime" << endl;
                cout << "4: Add Director" << endl;
                cout << "5: Add Genre" << endl;
                cout << "6: Add Country" << endl;
                cout << "7: Add Keyword" << endl;
                cout << "8: Execute Query" << endl;
                cout << "Please enter a menu selection: ";
                choice = cin.get();
                cin.ignore(100, '\n');
                cout << endl;
                switch(choice)
                {
                        case '1':
                                cout << "Title: ";
                                getline(cin, movieTemp.title);
                                if (!search)
                                        moviectrl->setKeywords(movieTemp);
                                break;
                        case '2':
                                cout << "Release Year: ";
                                cin >> movieTemp.release;
                                if(cin.fail())
                                        cin.clear();
                                cin.ignore(100, '\n');
                                break;
                        case '3':
                                cout << "Runtime: ";
                                cin >> movieTemp.runtime;
                                if(cin.fail())
```

```
                cin.clear();
            cin.ignore(100, '\n');
            break;
        case '4':
            cout << "Director: ";
            getline(cin, tempIn);
            movieTemp.directors.push_back(tempIn);
            break;
        case '5':
            cout << "Genre: ";
            getline(cin, tempIn);
            movieTemp.genres.push_back(tempIn);
            break;
        case '6':
            cout << "Country: ";
            getline(cin, tempIn);
            movieTemp.countries.push_back(tempIn);
            break;
        case '7':
            cout << "Keyword: ";
            getline(cin, tempIn);
            movieTemp.keywords.push_back(tempIn);
            break;
        case '8':
            if(!search)     //If editing or creating (not searching however), the movie must have a defined title
                            // and release year
            {
                if(movieTemp.title == "" || movieTemp.release == 0)
                    cout << "Need to define a title and release year" << endl;
                else
                    return;
            }
            else
                return;
        default:
            cout << "Invalid choice." << endl;
    }

}

}

void Interface::displayMovie(movieInfo& movieOutput)
{
    //Displays all movie information
    cout << "Title: " << movieOutput.title << endl;
    cout << "Release Year: " << movieOutput.release << endl;
    cout << "Runtime: " << movieOutput.runtime << endl;
    cout << "Added By: " << movieOutput.username << endl;
    cout << "Director(s): " << endl;
    if (movieOutput.directors.size()==0)
        cout << "\t(none)" << endl;
    for(unsigned int i=0; i<movieOutput.directors.size(); i++)
    {
        cout << "\t" << movieOutput.directors[i] << endl;
    }
```

```
	cout << "Genre(s): " << endl;
	if (movieOutput.genres.size()==0)
		cout << "\t(none)" << endl;
	for(unsigned int i=0; i<movieOutput.genres.size(); i++)
	{
		cout << "\t" << movieOutput.genres[i] << endl;
	}
	cout << "Country(s): " << endl;
	if (movieOutput.countries.size()==0)
		cout << "\t(none)" << endl;
	for(unsigned int i=0; i<movieOutput.countries.size(); i++)
	{
		cout << "\t" << movieOutput.countries[i] << endl;
	}
	cout << "Keyword(s): " << endl;
	if (movieOutput.keywords.size()==0)
		cout << "\t(none)" << endl;
	for(unsigned int i=0; i<movieOutput.keywords.size(); i++)
	{
		cout << "\t" << movieOutput.keywords[i] << endl;
	}
	cout << endl;
}


void Interface::resetUser(userInfo& userTemp)
{
	userTemp.username = "";
	userTemp.fullname = "";
	userTemp.admin = false;
}

bool Interface::createUser()
{
	userInfo userTemp;
	string password = "";

	resetUser(userTemp);	//Reset for inputting

	readUser(userTemp,password,true);
	if(userctrl->create_user(userTemp, password))
	{
		cout << "User created successfully" << endl;
		return true;
	}
	else
	{
		cout << "User could not be created" << endl;
		return false;
	}

}

void Interface::readUser(userInfo& userTemp, string& password, bool create)
{
	string tempPass = "";
```

```
while(true)     //Displays menu until the query is executed
{
	char choice = ' ';

	cout << endl << "1: Add/Change Username" << endl;
	cout << "2: Add/Change Password" << endl;
	cout << "3: Add/Change Full Name" << endl;
	cout << "4: Add/Change Admin/User Permissions" << endl;
	cout << "5: Execute Query" << endl;
	cout << "Please enter a menu selection: ";
	choice = cin.get();
	cin.ignore(100, '\n');
	cout << endl;
	switch(choice)
	{
		case '1':
			cout << "Username: ";
			getline(cin, userTemp.username);
			break;
		case '2':
			tempPass = getpass("Password: ");
			if(validatePassword(tempPass))
				password = encryptPassword(tempPass);
			else
				cout << "Invalid password entered." << endl;
			break;
		case '3':
			cout << "Full Name: ";
			getline(cin, userTemp.fullname);
			break;
		case '4':
			cout << "Admin(A) or User(U): ";
			choice = cin.get();
			cin.ignore(100, '\n');
			if(choice == 'A' || choice == 'a')
				userTemp.admin = true;
			else
				userTemp.admin = false;
			break;
		case '5':
			if(create)      //The user is being created, so it must have a new username and password
			{
				if(userTemp.username == "" || password == "")
					cout << "Need to define a username and password" << endl;
				else
					return; //Is all valid
			}
			else
				if(userTemp.username == "")     //Edited user must have a non-blank username
					cout << "Need to define a username" << endl;
				else
					return;
			break;
		default:
			cout << "Invalid choice." << endl;
```

```
            }

        }

}

void Interface::displayUser(userInfo& userTemp)
{
        cout << "Username: " << userTemp.username << endl;
        cout << "Full Name: " << userTemp.fullname << endl;
        cout << "Account Type: ";
        if(userTemp.admin)
                cout << "Administrator" << endl;
        else
                cout << "User" << endl;
}

bool Interface::editUser()
{
        char choice = ' ';
        string password = "";
        userInfo userTemp;
        resetUser(userTemp);

        cout << "Please enter the username of the user to be edited:" << endl;
        getline(cin, userTemp.username);

        if(userTemp.username == "")
        {
                cout << "Invalid username." << endl;
                return false;
        }
        if(userTemp.username == currUser.username)
        {
                cout << "Cannot edit self" << endl;
                return false;
        }

        userctrl->search_user(userTemp);
        if(userTemp.username == "")
        {
                cout << "User could not be found" << endl;
                return false;
        }

        displayUser(userTemp);
        cout << "Reset the password attempts for the user (y/n)? " << endl;
        choice = cin.get();
        cin.ignore(100, '\n');
        if(choice != 'y' && choice != 'Y' && choice != 'n' && choice != 'N')
        {
                cout << "Invalid choice made, password will not be reset." << endl;
        }
        //Read in user details to edit
        readUser(userTemp, password, false);
        if(choice == 'y' || choice == 'Y')
```

```
	{
		if(userctrl->edit_user(userTemp,password,true))
		{
			cout << "User edited successfully" << endl;
			return true;
		}
		else
		{
			cout << "User could not be edited" << endl;
			return false;
		}
	}
	if(userctrl->edit_user(userTemp,password,false))
	{
		cout << "User edited successfully" << endl;
		return true;
	}
	else
	{
		cout << "User could not be edited" << endl;
		return false;
	}

}

bool Interface::deleteUser()
{
	userInfo userTemp;
	resetUser(userTemp);
	char choice = ' ';

	cout << "Please enter the username of the user to be edited:" << endl;
	getline(cin, userTemp.username);


	if(userTemp.username == "")
	{
		cout << "Invalid username." << endl;
		return false;
	}
	if(userTemp.username == currUser.username)
	{
		cout << "Cannot delete self" << endl;
		return false;
	}

	userctrl->search_user(userTemp);
	if (userTemp.username == "")
	{
		cout << "User could not be found." << endl;
		return false;
	}

	displayUser(userTemp);  //Display the user to be deleted

	//Confirm choice
```

```
	cout << "Are you sure you would like to delete this user? (y/n) ";
	choice = cin.get();
	cin.ignore(100, '\n');
	if(choice != 'y' && choice != 'Y' && choice != 'n' && choice != 'N')
	{
		cout << "Invalid choice made, user will not be deleted." << endl;
		return true;
	}
	if(choice == 'Y' || choice == 'y')
	{
		if(userctrl->delete_user(userTemp.username))
		{
			cout << "User deleted successfully" << endl;
			return true;
		}
		else
		{
			cout << "User could not be deleted" << endl;
			return false;
		}
	}
	return true;

}

bool Interface::changePassword()
{
	userInfo userTemp;
	resetUser(userTemp);
	string newPass, cryptedPass;

	userTemp.username = currUser.username;
	userctrl->search_user(userTemp);	//Get the current user information

	newPass = getpass("New Password: ");	//Get new password

	if (validatePassword(newPass))
	{
		cryptedPass = encryptPassword(newPass);
		if(userctrl->edit_user(userTemp, cryptedPass, false))
		{
			cout << "Password changed" << endl;
			return true;	//Returns true for successful change, and false for error
		}
		else
		{
			cout << "Password could not be changed" << endl;
			return false;
		}
	}
	else
	{
		cout << "Invalid new password, unable to change" << endl;
		return false;
	}
```

```
}

//Returns true for matching searches, and false for no matches
bool Interface::showAllMovies()
{
	movieInfo tempMovie;
	vector<movieInfo> results;
	resetMovie(tempMovie);  //Sets all fields to null, so will be a search for all

	results = moviectrl->search_movie(tempMovie);   //Returned a vector of movies

	if(results.empty())
	{
		cout << "No movies found" << endl;
		return false;
	}
	else
	{
		for(vector<movieInfo>::iterator it = results.begin(); it != results.end(); it++)
		{
			displayMovie(*it);	//Loops through and displays all movies
		}
		return true;
	}
}

//Returns true for matching searches, and false for no matches
bool Interface::showAllUserMovies()
{
	movieInfo tempMovie;
	vector<movieInfo> results;
	resetMovie(tempMovie);
	tempMovie.username = currUser.username; //So only movies related to this user are returned

	results = moviectrl->search_movie(tempMovie);   //Returned a vector of movies

	if(results.empty())
	{
		cout << "No movies found" << endl;
		return false;
	}
	else
	{
		for(vector<movieInfo>::iterator it = results.begin(); it != results.end(); it++)
		{
			displayMovie(*it);	//Loops through and displays all movies
		}
		return true;
	}
}

//Returns true for matching searches, and false for no matches
bool Interface::searchMovies()
{
	movieInfo tempMovie;
	vector<movieInfo> results;
```

```
        resetMovie(tempMovie);

        cout << "Enter the details to be searching for: " << endl;
        readMovie(tempMovie, true);

        results = moviectrl->search_movie(tempMovie);   //Returned a vector of movies

        if(results.empty())
        {
                cout << "No movies found matching criteria" << endl;
                return false;
        }
        else
        {
                for(vector<movieInfo>::iterator it = results.begin(); it != results.end(); it++)
                {
                        displayMovie(*it);      //Loops through and displays all movies
                }
                return true;
        }

}


//encrypts a password using a caesar cipher - will only work for passwords containing
//only upppercase letters, lowercase letter and digits
string Interface::encryptPassword (string password)
{
   int len, index;
   string encrypted;
   char characters[63] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
   char ch;

   len = password.length();
   for (int i = 0; i < len; i++)
   {
      ch = password[i];

      if (isdigit(ch))
      {
         index = ch - 48;
      }
      else if (isupper(ch))
      {
         index = ch - 55;
      }
      else
      {
         index = ch - 61;
      }
      index = (index + SHIFT) % 62;
      encrypted.push_back(characters[index]);
   }
   return encrypted;
}
```

```
//returns true if password is a valid password otherwise returns false.
//a valid password must be at least 8 characters (letters or numbers only,
//have at least 1 letter and 1 number
bool Interface::validatePassword (string password)
{
   bool hasDigit = false, hasLetter = false;
   int len;

   len = password.length();
   if (len < 8)
   {
      return false;
   }
   else
   {
      for (int i = 0; i < len; i++)
      {
         if (isdigit(password[i]))
         {
            hasDigit = true;
         }
         else if (isalpha(password[i]))
         {
            hasLetter = true;
         }
         else
         {
            return false;
         }
      }
      return (hasDigit && hasLetter);
   }
}

void Interface::displayHelp()   //Simply display block help, describing functions and how to use them
{
      cout << "*** Welcome to eMovies Help ***" << endl << endl;
      cout << "This program is used to store and manage movie information." << endl << endl;
      cout << "* Main Menu *" << endl << "From the main menu, the user can access a number of tasks, " << endl;
      cout << "these are accessed by entering the number or letter next to the desired action " << endl;
      cout << "and pressing enter. The selected action will then prompt for more information if needed." << endl;
      cout << "Available Choices" << endl;
      cout << "(for regular users)\n";
      cout << " - 1. Show All Movies - Displays all the movies in the eMovies profile\n";
      cout << " - 2. Show All Movies Created by this User - Displays all the movies this user has created\n";
      cout << " - 3. Search For Movies - Displays all movies that match the criteria entered by the user\n";
      cout << " - 4. Change Password - Allows the current user to change their password\n";
      cout << "(for administrators)\n";
      cout << " - 1. Create Movie - Adds a movie into the eMovies profile, information entered by the user\n";
      cout << " - 2. Edit Movie - Changes the details of a selected movie, information entered by the user\n";
      cout << " - 3. Delete Movie - Removes a selected movie from the eMovies profile\n";
      cout << " - 4. Show All Movies - Displays all the movies in the eMovies profile\n";
      cout << " - 5. Show All Movies Created by this User - Displays all the movies this user has created\n";
      cout << " - 6. Search For Movies - Displays all movies that match the criteria entered by the user\n";
      cout << " - 7. Create User - Adds a user into the eMovies profile, information entered by the user\n";
      cout << " - 8. Edit User - Changes the details of a selected user, information entered by the user\n";
```

```
    cout << " - 9. Remove User - Removes a selected movie from the eMovies profile\n";
    cout << " - 0. Change Password - Allows the current user to change their password\n";
    cout << "(for both)\n";
    cout << " - I. Import Movies - Imports the movies stored in the movies.list and genres.list files\n";
    cout << " - H. Show Help - Displays this help\n";
    cout << " - Q. Quit and Save - Saves and exits the program\n" << endl;
    cout << "* User and Movie Information Entry Forms *" << endl;
    cout << "If the user selects a create, edit or search function they will then be navigated to another\n";
    cout << "menu, where they can enter information on that user/movie. The information to be entered is\n";
    cout << "again selected by selecting a number next to the desired choice and then pressing enter.\n";
    cout << "If the user makes a mistake on entry, they simply select that option again and enter it.\n";
    cout << "This will overwrite the entry they had previously made for that choice.\n";
    cout << "To create/edit/search for the data entered in this form the user simply selects \'Execute Query\'\n";
    cout << "The desired action will then be perform using the entered information." << endl;
    cout << endl << "Press enter to leave help and go back to the main menu" << endl;
    cin.ignore(100, '\n');
}
```

Database Class

The Database Class control access to the database. It provides function for opening and closing the database and running database queries.

database.h

```
#ifndef DATABASE_H
#define DATABASE_H

#include <string.h>
#include <vector>
#include <sqlite3.h>
using namespace std;

class database
{
	public:
		database(char* filename);
		~database();

		vector<vector<string> > query(const char* query, sqlite3_stmt *statement);
		sqlite3* getdb();
	private:
		bool open(char* filename);
		bool newdb(char* filename);
		void close();
		sqlite3 *sqldb;
};

#endif
```

database.cpp

```
#include <iostream>
#include <sqlite3.h>
#include <cstdlib>
#include <string.h>
#include "database.h"
using namespace std;

database::database(char* filename)
{
	sqldb = NULL;
	if (!open(filename)) // attempts to open existing database, if failed creates a new database
	{
		if (newdb(filename)) // creates default tables in new database
			exit(1);
	}
}

database::~database()
```

```
{
	close(); // closes the database
}

bool database::open(char* filename)
{
	if (sqlite3_open_v2(filename, &sqldb, SQLITE_OPEN_READWRITE, NULL) == SQLITE_OK) // attempts
                                                                                    //to open existing database
		return true;
	else
		return false;
}

bool database::newdb(char* filename)
{
	string sqlquery;
	sqlite3_stmt *stmt;

	if (sqlite3_open_v2(filename, &sqldb, SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, NULL)
!= SQLITE_OK) // creates new database
	{
		cerr << "Database creation error, exiting..." << endl;
		return 1;
	}

	/* The following inserts the default tables into the newly created database */
	sqlquery = "CREATE TABLE users (username TEXT,password TEXT,fullname TEXT,admin
INTEGER,passtries INTEGER)";
	// Create default tables
	if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return 1;
	}
	query(sqlquery.c_str(),stmt);

	sqlquery = "CREATE TABLE movies (username TEXT,movieid INTEGER,title TEXT,release TEXT,runtime
INTEGER)";
	// Create default tables
	if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return 1;
	}
	query(sqlquery.c_str(),stmt);

	sqlquery = "CREATE TABLE directorindex (directorid INTEGER,movieid INTEGER)";
	// Create default tables
	if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return 1;
	}
	query(sqlquery.c_str(),stmt);

	sqlquery = "CREATE TABLE genreindex (genreid INTEGER,movieid INTEGER)";
```

```
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return 1;
}
query(sqlquery.c_str(),stmt);

sqlquery = "CREATE TABLE countryindex (countryid INTEGER,movieid INTEGER)";
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return 1;
}
query(sqlquery.c_str(),stmt);

sqlquery = "CREATE TABLE keywordindex (keywordid INTEGER,movieid INTEGER)";
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return 1;
}
query(sqlquery.c_str(),stmt);

sqlquery = "CREATE TABLE directors (directorid INTEGER,director TEXT)";
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return 1;
}
query(sqlquery.c_str(),stmt);

sqlquery ="CREATE TABLE genres (genreid INTEGER,genre TEXT)";
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return 1;
}
query(sqlquery.c_str(),stmt);

sqlquery = "CREATE TABLE countries (countryid INTEGER,country TEXT)";
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return 1;
}
query(sqlquery.c_str(),stmt);

sqlquery = "CREATE TABLE keywords (keywordid INTEGER,keyword TEXT)";
// Create default tables
if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
```

```
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return 1;
	}
	query(sqlquery.c_str(),stmt);

	sqlquery = "INSERT INTO users VALUES ('admin','x095AOPQ','Administrator',1,0)";
	// Create default tables
	if (sqlite3_prepare_v2(sqldb,sqlquery.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return 1;
	}
	query(sqlquery.c_str(),stmt);

	return 0;
}

sqlite3* database::getdb()
{
	return sqldb; // returns a pointer to the database object
}

vector<vector<string> > database::query(const char* query, sqlite3_stmt *statement)
{
	vector<vector<string> > results;

	int cols = sqlite3_column_count(statement);
	int result = 0;
	while (true)
	{
		result = sqlite3_step(statement); // steps through the SQL statement

		if (result == SQLITE_ROW)
		{
			vector<string> values;
			for(int col = 0; col < cols; col++)
			{
				if(sqlite3_column_text(statement, col) != NULL)
					values.push_back((char*)sqlite3_column_text(statement, col)); // adds column name to
                                                                                  // results
				else
					values.push_back((char*)""); // if no column name is available, add nothing
			}
			results.push_back(values); // adds each data element to the string matrix
		}
		else
			break;
	}

	sqlite3_finalize(statement);

	string error = sqlite3_errmsg(sqldb);
	if (error != "not an error")
		cout << query << " " << error << endl; // prints out any SQL error messages
```

```
    return results;
}

void database::close()
{
    if (sqldb!=NULL)
    {
        sqlite3_close(sqldb); // close the database
        sqldb = NULL; // invalidate the database pointer
    }
    else
    {
        cout << "database close error" << endl; // database is already closed
        exit(1);
    }
}
```

UserController Class

The UserController class prepares user data from the Interface to be run in database queries by the database class and interprets the results returned from such queries.

userController.h

```
#ifndef USERCONTROLLER_H
#define USERCONTROLLER_H

#include <string>
#include <vector>
#include <sqlite3.h>
#include "database.h"
#include "userInfo.h"
using namespace std;

class userController
{
	public:
		userController(database*);

		bool create_user(userInfo&, string password);
		bool edit_user(userInfo&, string password, bool passtries);
		bool search_user(userInfo&);
		bool delete_user(string username);
		int user_login(string username, string password);
	private:
		bool user_increment(string username);
		database *db;
#ifdef TEST
		friend class eMoviesTest;
#endif
};
#endif
```

userController.cpp

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <fstream>
#include <sstream>
#include <cctype>
#include <sqlite3.h>
#include "database.h"
#include "userController.h"
#include "userInfo.h"
#include "movieInfo.h"
using namespace std;

userController::userController(database* dBase)
{
	db = dBase; // stores reference to database
}

bool userController::create_user(userInfo& user, string password)
{
	string query;
	sqlite3_stmt *stmt = NULL;
	vector<vector<string> > results;

	sqlite3_reset(stmt); // initialise statement

	query = "SELECT * FROM users WHERE username = ?";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return false;
	}

	if (sqlite3_bind_text(stmt,1,user.username.c_str(),user.username.size(),SQLITE_STATIC) != SQLITE_OK)
	{
		cout << "could not bind username." << endl;
		return false;
	}
	results = db->query(query.data(),stmt); // test if user already exists, using username

	if (!results.empty())
	{
		cout << "User already exists." << endl;
		return false;
	}

	if (user.admin) // set admin privilages if appropriate
		query = "INSERT INTO users VALUES (?,?,?,1,0)";
	else
		query = "INSERT INTO users VALUES (?,?,?,0,0)";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
```

```
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return false;
	}
	//prepare the sql statement
	if (sqlite3_bind_text(stmt,1,user.username.c_str(),user.username.size(),SQLITE_STATIC) != SQLITE_OK)
	{
		cout << "could not bind username." << endl;
		return false;
	}
	if (sqlite3_bind_text(stmt,2,password.c_str(),password.size(),SQLITE_STATIC) != SQLITE_OK)
	{
		cout << "could not bind password." << endl;
		return false;
	}
	if (sqlite3_bind_text(stmt,3,user.fullname.c_str(),user.fullname.size(),SQLITE_STATIC) != SQLITE_OK)
	{
		cout << "could not bind fullname." << endl;
		return false;
	}

	char* conversion = new char [query.size()+1];
	strcpy(conversion,query.c_str());
	db->query((char*)conversion,stmt);	//insert user into database

	return true;
}

bool userController::edit_user(userInfo& user, string password, bool passtries)
{
	int parameter_count = 1;
	sqlite3_stmt *stmt = NULL;
	vector<vector<string> > results;
	string query;

	sqlite3_reset(stmt); // initialise statement

	// create the query for editing, depending on if the password is to be changed, or if the password attempts
should be reset
	query = "UPDATE users SET ";
	if (user.admin) // conserves admin privilages
	{
		if (password!="")	//change password
			query+="password = ?, ";
		if (passtries)  //will reset password attempts
			query+="fullname = ?, admin = 1, passtries = 0 WHERE username = ?";
		else
			query+="fullname = ?, admin = 1 WHERE username = ?";
	}
	else
	{
		if (password!="")	//change password
			query+="password = ?, ";
		if (passtries)  //will reset password attempts
			query+="fullname = ?, admin = 0, passtries = 0 WHERE username = ?";
		else
```

```
                query+="fullname = ?, admin = 0 WHERE username = ?";
        }

        //prepare the sql statement
        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }
        if (password!="")       // checks for blank password (defensive coding)
        {
                if (sqlite3_bind_text(stmt,parameter_count,password.c_str(),password.size(),SQLITE_STATIC) !=
SQLITE_OK)
                {
                        cout << "could not bind password." << endl;
                        return false;
                }
                parameter_count++;
        }
        if (sqlite3_bind_text(stmt,parameter_count,user.fullname.c_str(),user.fullname.size(),SQLITE_STATIC) !=
SQLITE_OK)
        {
                cout << "could not bind fullname." << endl;
                return false;
        }
        parameter_count++;
        if (sqlite3_bind_text(stmt,parameter_count,user.username.c_str(),user.username.size(),SQLITE_STATIC) !=
SQLITE_OK)
        {
                cout << "could not bind username." << endl;
                return false;
        }

        db->query(query.c_str(),stmt);  // creates user

        return true;
}

bool userController::search_user(userInfo& tempUser)
{
        string query = "SELECT fullname, admin FROM users WHERE username = ?";
        sqlite3_stmt *stmt = NULL;
        vector<string> row;
        vector<vector<string> > results;

        sqlite3_reset(stmt);// initialise statement

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }
        //prepare the sql statement
        if (sqlite3_bind_text(stmt,1,tempUser.username.c_str(),tempUser.username.size(),SQLITE_STATIC) !=
SQLITE_OK)
        {
```

```
            cout << "could not bind username." << endl;
            return false;
        }
        results = db->query(query.data(),stmt); // searches for user
        if (results.empty()) // if the user cannot be found, return nothing
        {
            tempUser.username = "";
            return true;
        }
        vector<vector<string> >::iterator it = results.begin();
        row = *it;
        tempUser.fullname = row[0];
        if (row[1]=="0")        // set returned user's admin privilages as appropriate
            tempUser.admin = false;
        else
            tempUser.admin = true;

        return true;
}

bool userController::delete_user(string username)
{
        string query = "DELETE FROM users WHERE username = ?";
        sqlite3_stmt *stmt = NULL;
        vector<vector<string> > results;

        sqlite3_reset(stmt); // initialise statement

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad" << endl;
            return -1;
        }
        //prepare the sql statement
        if (sqlite3_bind_text(stmt,1,username.c_str(),username.size(),SQLITE_STATIC) != SQLITE_OK)
        {
            cout << "could not bind username." << endl;
            return false;
        }
        db->query(query.c_str(),stmt);  // deletes user from users table based on username

        return true;
}

int userController::user_login(string username, string password)
{
        string query = "SELECT fullname,admin,passtries FROM users WHERE username = ? AND password = ?";
        sqlite3_stmt *stmt = NULL;
        vector<string> row;
        vector<vector<string> > results;
        int retAdmin;
        userInfo tempUser;

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad" << endl;
```

```
            return -1;
        }
        // get user based on username and password
        if (sqlite3_bind_text(stmt,1,username.c_str(),username.size(),SQLITE_STATIC) != SQLITE_OK)
        {
            cout << "could not bind username." << endl;
            return -1;
        }
        if (sqlite3_bind_text(stmt,2,password.c_str(),password.size(),SQLITE_STATIC) != SQLITE_OK)
        {
            cout << "could not bind password." << endl;
            return -1;
        }
        results = db->query(query.c_str(),stmt);
        if (results.size()>0) // if matched a user in database
        {
            vector<vector<string> >::iterator it = results.begin();
            row = *it;
            if (atoi(row[2].c_str())>=3)    //if number of password attempts >= 3
                return 3; // return account locked
            retAdmin = atoi(row[1].c_str()); // value whether user is admin or not

            tempUser.username = username;
            tempUser.fullname = row[0];
            tempUser.admin = retAdmin;
            edit_user(tempUser,"",true); // resets password tries

            return retAdmin; // return the logged in user
        }
        else
        {
            user_increment(username); // increments login try if exists
            return 2; // return invalid user-pass
        }
}

//increments the number of pass tries for the user
bool userController::user_increment(string username)
{
        string query = "UPDATE users SET passtries = (passtries + 1) WHERE username = ?";
        sqlite3_stmt *stmt = NULL;
        vector<vector<string> > results;

        sqlite3_reset(stmt); // initialise statement

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad" << endl;
            return false;
        }

        if (sqlite3_bind_text(stmt,1,username.c_str(),username.size(),SQLITE_STATIC) != SQLITE_OK)
        {
            cout << "could not bind username." << endl;
            return false;
        }
```

```
    char* conversion = new char [query.size()+1];
    strcpy(conversion,query.c_str());
    db->query((char*)conversion,stmt); // increments the users password attempts

    return true;
}
```

MovieController Class

The MovieController class prepares movie data from the Interface to be run in database queries by the database class and interprets the results returned from such queries.

movieController.h

```
#ifndef MOVIECONTROLLER_H
#define MOVIECONTROLLER_H

#include <string>
#include <vector>
#include <sqlite3.h>
#include "database.h"
#include "movieInfo.h"
using namespace std;

class movieController
{
	public:
		movieController(database*);

		bool create_movie(movieInfo& movie);
		bool delete_movie(movieInfo&);
		vector<movieInfo> search_movie(movieInfo&); // WIP
		bool edit_movie(movieInfo&,movieInfo&);
		movieInfo get_movie(int);
		void importMovies(string username);
		void setKeywords(movieInfo&);
	private:

		void genreImport(movieInfo& movie);
		bool isNumeric(const char* input);
		void trimstr(string& str);
		database *db;
#ifdef TEST
		friend class eMoviesTest;
#endif

};

#endif
```

movieController.cpp

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <fstream>
#include <sstream>
#include <cctype>
#include <sqlite3.h>
#include "database.h"
#include "movieController.h"
#include "userInfo.h"
#include "movieInfo.h"
using namespace std;

movieController::movieController(database* dBase)
{
	db = dBase; // stores database pointer
}

//returns a selected movie based on movieid
movieInfo movieController::get_movie(int movieid)
{
	movieInfo movie;
	sqlite3_stmt *stmt = NULL;
	vector<string> row;
	vector<vector<string> > results;
	string query = "SELECT title, release, runtime, username FROM movies WHERE movieid = ?";

	sqlite3_reset(stmt); // initialise statement

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return movie;
	}

	if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
	{
		cout << "could not bind movieid." << endl;
		return movie;
	}

	/* Gets other details of movie from movieid and parse into struct movieInfo */
	results = db->query(query.c_str(),stmt);
	vector<vector<string> >::iterator it = results.begin();
	row = *it;
	movie.title = row[0];
	movie.release = atoi(row[1].c_str());
	movie.runtime = atoi(row[2].c_str());
	movie.username = row[3];

	query = "SELECT directors.director FROM directors INNER JOIN directorindex ON directorindex.directorid
= directors.directorid WHERE directorindex.movieid = ? ORDER BY directors.director";
	//Prepare sql statement that returns all directors related to that movie
```

```
	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return movie;
	}

	if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
	{
		cout << "could not bind movieid." << endl;
		return movie;
	}

	results = db->query(query.c_str(),stmt);
	// Adds directors from database to movie that matches the movieid
	for (vector<vector<string> >::iterator it = results.begin(); it != results.end(); it++)
	{
		row = *it;
		movie.directors.push_back(row[0]);
	}

	query = "SELECT genres.genre FROM genres INNER JOIN genreindex ON genreindex.genreid =
genres.genreid WHERE genreindex.movieid = ? ORDER BY genres.genre";
	//Prepare sql statement that returns all genres related to that movie
	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return movie;
	}

	if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
	{
		cout << "could not bind movieid." << endl;
		return movie;
	}

	results = db->query(query.c_str(),stmt);
	// Adds genres from database to movie that matches the movieid
	for (vector<vector<string> >::iterator it = results.begin(); it != results.end(); it++)
	{
		row = *it;
		movie.genres.push_back(row[0]);
	}

	query = "SELECT countries.country FROM countries INNER JOIN countryindex ON countryindex.countryid
= countries.countryid WHERE countryindex.movieid = ? ORDER BY countries.country";
	//Prepare sql statement that returns all countries related to that movie
	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return movie;
	}

	if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
	{
		cout << "could not bind movieid." << endl;
		return movie;
```

```
	}

	results = db->query(query.c_str(),stmt);
	// Adds countries from database to movie that matches the movieid
	for (vector<vector<string> >::iterator it = results.begin(); it != results.end(); it++)
	{
		row = *it;
		movie.countries.push_back(row[0]);
	}

	query = "SELECT keywords.keyword FROM keywords INNER JOIN keywordindex ON
keywordindex.keywordid = keywords.keywordid WHERE keywordindex.movieid = ? ORDER BY
keywords.keyword";
	//Prepare sql statement that returns all keywords related to that movie
	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return movie;
	}

	if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
	{
		cout << "could not bind movieid." << endl;
		return movie;
	}

	results = db->query(query.c_str(),stmt);
	// Adds keywords from database to movie that matches the movieid
	for (vector<vector<string> >::iterator it = results.begin(); it != results.end(); it++)
	{
		row = *it;
		movie.keywords.push_back(row[0]);
	}

	return movie;
}

bool movieController::create_movie(movieInfo& movie)
{
	string query;
	int movieid; // retrieved movie identifier
	sqlite3_stmt *stmt = NULL;
	vector<string> row;
	vector<vector<string> > results;
	int directorid, genreid, countryid, keywordid;

	sqlite3_reset(stmt); // initialise statement

	query = "SELECT * FROM movies WHERE title = ? AND release = ?";
	//Check to see if the movie already exists
	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return false;
	}
```

```
if (sqlite3_bind_text(stmt,1,movie.title.c_str(),movie.title.size(),SQLITE_STATIC) != SQLITE_OK)
{
	cout << "could not bind movie title." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,2,movie.release) != SQLITE_OK)
{
	cout << "could not bind movie release." << endl;
	return false;
}
results = db->query(query.data(),stmt);

if (!results.empty()) // Errors if movie already exists
{
	cout << "Movie already exists." << endl;
	return false;
}

query = "SELECT MAX(movieid) FROM movies";

// Get unique identifier from table
if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad in movies" << endl;
	return false;
}
results = db->query(query.c_str(),stmt);
/* If the movies table is not empty, get the largest movieid and add 1, creating a unique identifier, else set 0 */
if (results.size()>0)
{
	vector<vector<string> >::iterator it = results.begin();
	row = *it;
	movieid = atoi(row[0].c_str())+1;
}
else
	movieid = 0;

query = "INSERT INTO movies VALUES (?,?,?,?,?)";
// construct statement to insert the values into the movie table
if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad in movies2" << endl;
	return false;
}

if (sqlite3_bind_text(stmt,3,movie.title.c_str(),movie.title.size(),SQLITE_STATIC) != SQLITE_OK)
{
	cout << "could not bind title." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,4,movie.release) != SQLITE_OK)
{
	cout << "could not bind release." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,5,movie.runtime) != SQLITE_OK)
```

```
{
        cout << "could not bind runtime." << endl;
        return false;
}
if (sqlite3_bind_text(stmt,1,movie.username.c_str(),movie.username.size(),SQLITE_STATIC) !=
          SQLITE_OK)
{
        cout << "could not bind username." << endl;
        return false;
}
if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
{
        cout << "could not bind movieid." << endl;
        return false;
}

db->query(query.c_str(),stmt);   // inserts movie data into movies table

// Insert directors
for (unsigned int i=0; i<movie.directors.size(); i++)   // while there are directors to insert
{
        query = "SELECT directorid FROM directors WHERE director = ?";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }

        if (sqlite3_bind_text(stmt,1,movie.directors[i].c_str(),movie.directors[i].size(),SQLITE_STATIC) !=
                        SQLITE_OK)
        {
                cout << "could not bind movie title." << endl;
                return false;
        }
        results = db->query(query.data(),stmt);
        /* If the director already exists, only add a director index, otherwise add an index and the director */
        if (!results.empty())
        {
                vector<vector<string> >::iterator it = results.begin();
                row = *it;
                directorid = atoi(row[0].c_str());
        }
        else    // needs to be added
        {
                query = "SELECT MAX(directorid) FROM directors";

                if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
                {
                        cout << "Error: SQL Prepared Bad in movies" << endl;
                        return false;
                }

                // Get unique identifier from table
                results = db->query((char*)"SELECT MAX(directorid) FROM directors",stmt);
                if (!results.empty())
```

```
            {
                vector<vector<string> >::iterator it = results.begin();
                row = *it;
                directorid = atoi(row[0].c_str())+1;
            }
            else
                directorid = 0;

            query = "INSERT INTO directors VALUES (?,?)";

            if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
            {
                cout << "Error: SQL Prepared Bad in directors" << endl;
                return false;
            }

            if (sqlite3_bind_int(stmt,1,directorid) != SQLITE_OK)
            {
                cout << "could not bind directorid." << endl;
                return false;
            }
            if (sqlite3_bind_text(stmt,2,movie.directors[i].c_str(),movie.directors[i].size(),SQLITE_STATIC) !=
                    SQLITE_OK)
            {
                cout << "could not bind directors." << endl;
                return false;
            }

            // Insert into directors table
            db->query((char*)"INSERT INTO directors VALUES (?,?)",stmt);
        }

        query = "INSERT INTO directorindex VALUES (?,?)";
        // add link between director and movie
        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad" << endl;
            return false;
        }

        if (sqlite3_bind_int(stmt,1,directorid) != SQLITE_OK)
        {
            cout << "could not bind director ID." << endl;
            return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
            cout << "could not bind movie ID." << endl;
            return false;
        }
        // Insert into directors index table
        db->query(query.c_str(),stmt);
    }

    // Insert genres
    for (unsigned int i=0; i<movie.genres.size(); i++)      // while there are genres to insert
```

```
{
	query = "SELECT genreid FROM genres WHERE genre = ?";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return false;
	}

	if (sqlite3_bind_text(stmt,1,movie.genres[i].c_str(),movie.genres[i].size(),SQLITE_STATIC) !=
			SQLITE_OK)
	{
		cout << "could not bind movie title." << endl;
		return false;
	}
	results = db->query(query.data(),stmt);
	/* If the genre already exists, only add a genre index, otherwise add an index and the genre */
	if (!results.empty())
	{
		vector<vector<string> >::iterator it = results.begin();
		row = *it;
		genreid = atoi(row[0].c_str());
	}
	else	// needs to be added
	{
		query = "SELECT MAX(genreid) FROM genres";

		if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
		{
			cout << "Error: SQL Prepared Bad in movies" << endl;
			return false;
		}

		// Get unique identifier from table
		results = db->query((char*)"SELECT MAX(genreid) FROM genres",stmt);
		if (!results.empty())
		{
			vector<vector<string> >::iterator it = results.begin();
			row = *it;
			genreid = atoi(row[0].c_str())+1;
		}
		else
			genreid = 0;

		query = "INSERT INTO genres VALUES (?,?)";

		if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
		{
			cout << "Error: SQL Prepared Bad in genres" << endl;
			return false;
		}

		if (sqlite3_bind_int(stmt,1,genreid) != SQLITE_OK)
		{
			cout << "could not bind genreid." << endl;
			return false;
```

```
            }
            if (sqlite3_bind_text(stmt,2,movie.genres[i].c_str(),movie.genres[i].size(),SQLITE_STATIC) !=
                  SQLITE_OK)
            {
                  cout << "could not bind genres." << endl;
                  return false;
            }

            // Insert into genres table
            db->query((char*)"INSERT INTO genres VALUES (?,?)",stmt);
      }

      query = "INSERT INTO genreindex VALUES (?,?)";
      // add link between genre and movie
      if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
      {
            cout << "Error: SQL Prepared Bad" << endl;
            return false;
      }

      if (sqlite3_bind_int(stmt,1,genreid) != SQLITE_OK)
      {
            cout << "could not bind genre ID." << endl;
            return false;
      }
      if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
      {
            cout << "could not bind movie ID." << endl;
            return false;
      }
      // Insert into genres index table
      db->query(query.c_str(),stmt);
}

// Insert countries
for (unsigned int i=0; i<movie.countries.size(); i++)   // while there are countries to insert
{
      query = "SELECT countryid FROM countries WHERE country = ?";

      if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
      {
            cout << "Error: SQL Prepared Bad" << endl;
            return false;
      }

      if (sqlite3_bind_text(stmt,1,movie.countries[i].c_str(),movie.countries[i].size(),SQLITE_STATIC) !=
                  SQLITE_OK)
      {
            cout << "could not bind movie title." << endl;
            return false;
      }
      results = db->query(query.data(),stmt);
      /* If the country already exists, only add a country index, otherwise add an index and the country */
      if (!results.empty())
      {
            vector<vector<string> >::iterator it = results.begin();
```

```
        row = *it;
        countryid = atoi(row[0].c_str());
}
else    // needs to be added
{
        query = "SELECT MAX(countryid) FROM countries";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad in movies" << endl;
                return false;
        }

        // Get unique identifier from table
        results = db->query((char*)"SELECT MAX(countryid) FROM countries",stmt);
        if (!results.empty())
        {
                vector<vector<string> >::iterator it = results.begin();
                row = *it;
                countryid = atoi(row[0].c_str())+1;
        }
        else
                countryid = 0;

        query = "INSERT INTO countries VALUES (?,?)";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad in countries" << endl;
                return false;
        }

        if (sqlite3_bind_int(stmt,1,countryid) != SQLITE_OK)
        {
                cout << "could not bind countryid." << endl;
                return false;
        }
        if (sqlite3_bind_text(stmt,2,movie.countries[i].c_str(),movie.countries[i].size(),SQLITE_STATIC) !=
                SQLITE_OK)
        {
                cout << "could not bind countries." << endl;
                return false;
        }

        // Insert into countries table
        db->query((char*)"INSERT INTO countries VALUES (?,?)",stmt);
}

query = "INSERT INTO countryindex VALUES (?,?)";
// add link between country and movie
if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
        cout << "Error: SQL Prepared Bad" << endl;
        return false;
}
```

```
        if (sqlite3_bind_int(stmt,1,countryid) != SQLITE_OK)
        {
                cout << "could not bind country ID." << endl;
                return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
                cout << "could not bind movie ID." << endl;
                return false;
        }
        // Insert into countries index table
        db->query(query.c_str(),stmt);
}

// Insert keywords
for (unsigned int i=0; i<movie.keywords.size(); i++)    // while there are keywords to insert
{
        query = "SELECT keywordid FROM keywords WHERE keyword = ?";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }

        if (sqlite3_bind_text(stmt,1,movie.keywords[i].c_str(),movie.keywords[i].size(),SQLITE_STATIC) !=
                        SQLITE_OK)
        {
                cout << "could not bind movie title." << endl;
                return false;
        }
        results = db->query(query.data(),stmt);
        /* If the keyword already exists, only add a keyword index, otherwise add an index and the keyword */
        if (!results.empty())
        {
                vector<vector<string> >::iterator it = results.begin();
                row = *it;
                keywordid = atoi(row[0].c_str());
        }
        else    // needs to be added
        {
                query = "SELECT MAX(keywordid) FROM keywords";

                if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
                {
                        cout << "Error: SQL Prepared Bad in movies" << endl;
                        return false;
                }

                // Get unique identifier from table
                results = db->query((char*)"SELECT MAX(keywordid) FROM keywords",stmt);
                if (!results.empty())
                {
                        vector<vector<string> >::iterator it = results.begin();
                        row = *it;
                        keywordid = atoi(row[0].c_str())+1;
```

```
                }
                else
                        keywordid = 0;

                query = "INSERT INTO keywords VALUES (?,?)";

                if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
                {
                        cout << "Error: SQL Prepared Bad in keywords" << endl;
                        return false;
                }

                if (sqlite3_bind_int(stmt,1,keywordid) != SQLITE_OK)
                {
                        cout << "could not bind keywordid." << endl;
                        return false;
                }
                if (sqlite3_bind_text(stmt,2,movie.keywords[i].c_str(),movie.keywords[i].size(),SQLITE_STATIC)
                                != SQLITE_OK)
                {
                        cout << "could not bind keywords." << endl;
                        return false;
                }

                // Insert into keywords table
                db->query((char*)"INSERT INTO keywords VALUES (?,?)",stmt);
        }

        query = "INSERT INTO keywordindex VALUES (?,?)";
        // add link between keyword and movie
        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }

        if (sqlite3_bind_int(stmt,1,keywordid) != SQLITE_OK)
        {
                cout << "could not bind keyword ID." << endl;
                return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
                cout << "could not bind movie ID." << endl;
                return false;
        }
        // Insert into keywords index table
        db->query(query.c_str(),stmt);
    }

    return true;    // successfully created
}

bool movieController::delete_movie(movieInfo& movie)
{
    int movieid;
```

```
string query;
sqlite3_stmt *stmt = NULL;
vector<string> row;
vector<vector<string> > results;

sqlite3_reset(stmt); // initialise statement

query = "SELECT movieid FROM movies WHERE title = ? AND release = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return false;
}

if (sqlite3_bind_text(stmt,1,movie.title.c_str(),movie.title.size(),SQLITE_STATIC) != SQLITE_OK)
{
	cout << "could not bind title." << endl;
	return false;
}

if (sqlite3_bind_int(stmt,2,movie.release) != SQLITE_OK)
{
	cout << "could not bind release." << endl;
	return false;
}

results = db->query(query.c_str(),stmt);
if (results.size()>0)   // check if it exists
{
	vector<vector<string> >::iterator it = results.begin();
	row = *it;
	movieid = atoi(row[0].c_str());  // get the unique identifier of the movie to be deleted
}
else
{
	cout << "Error: Movie does not appear to exist" << endl;
	return false; // movie not found
}

query = "DELETE FROM movies WHERE movieid = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return false;
}

if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
{
	cout << "could not bind movieid." << endl;
	return false;
}

db->query(query.c_str(),stmt);  // delete the movie from the movies table
```

```
query = "DELETE FROM directorindex WHERE movieid = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return false;
}

if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
{
	cout << "could not bind movieid." << endl;
	return false;
}

db->query(query.c_str(),stmt); // delete director references to the movie

query = "DELETE FROM genreindex WHERE movieid = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return false;
}

if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
{
	cout << "could not bind movieid." << endl;
	return false;
}

db->query(query.c_str(),stmt); // delete genre references to the movie

query = "DELETE FROM countryindex WHERE movieid = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad" << endl;
	return false;
}

if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
{
	cout << "could not bind movieid." << endl;
	return false;
}

db->query(query.c_str(),stmt); // delete country references to the movie

query = "DELETE FROM keywordindex WHERE movieid = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad KEYWORDS" << endl;
	return false;
}
```

```
	if (sqlite3_bind_int(stmt,1,movieid) != SQLITE_OK)
	{
		cout << "could not bind movieid." << endl;
		return false;
	}

	db->query(query.c_str(),stmt); // delete keyword references to the movie

	return true;
}

vector<movieInfo> movieController::search_movie(movieInfo& movie)
{
	string query = "";
	bool SQLhit = false; // used to construct the SQL query
	int parameter_index = 1;
	sqlite3_stmt *stmt = NULL;
	vector<string> row;
	vector<vector<string> > results;
	vector<movieInfo> ret;

	ret.clear(); // initialize the returning vector of movies

	sqlite3_reset(stmt); // initialise statement

	// Construct query
	if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!="")
	{
		// ifs will add the correct parts to the statement if are searched for (not default values)
		query = "SELECT movies.movieid FROM movies";
		if (movie.title!="")
		{
			query+=" WHERE title = ?";
			SQLhit = true;
		}
		if (movie.runtime!=0)
		{
			if (SQLhit)
				query+=" AND ";
			else
				query+=" WHERE ";
			query+="runtime = ?";
		}
		if (movie.release!=0)
		{
			if (SQLhit)
				query+=" AND ";
			else
				query+=" WHERE ";
			query+="release = ?";
		}
		if (movie.username!="")
		{
			if (SQLhit)
				query+=" AND ";
			else
```

```
                query+=" WHERE ";
            query+="username = ?";
        }
    }
    if (movie.directors.size()>0)   // add in directors for countries to the sql statement (if any)
    {
        if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!="") // tests if query has already
                                                                                    // started
            query+=" AND movies.movieid IN ( ";
        query+="SELECT movieid FROM movies WHERE movieid IN (SELECT directorindex.movieid FROM
directorindex INNER JOIN directors ON directorindex.directorid = directors.directorid WHERE ";
        for (unsigned int i=0; i<movie.directors.size(); i++)
        {
            if (i!=0) // tests if query has already started
                query+=" OR ";
            query+="directors.director = ?";
        }
        query+=")";
        if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!="") // tests if query has already
                                                                                    //started
            query+=")";
    }
    if (movie.genres.size()>0)      // add in searches for genres to the sql statement (if any)
    {
        if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!=""||movie.directors.size()>0)
                                                                    // tests if query has already started
            query+=" AND movies.movieid IN ( ";
        query+="SELECT movieid FROM movies WHERE movieid IN (SELECT genreindex.movieid FROM
genreindex INNER JOIN genres ON genreindex.genreid = genres.genreid WHERE ";
        for (unsigned int i=0; i<movie.genres.size(); i++)
        {
            if (i!=0) // tests if query has already started
                query+=" OR ";
            query+="genres.genre = ?";
        }
        query+=")";
        if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!=""||movie.directors.size()>0) //
tests if query has already started
            query+=")";
    }
    if (movie.countries.size()>0)   // add in searches for countries to the sql statement (if any)
    {
        if
(movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!=""||movie.directors.size()>0||movie.genres.s
ize()>0) // tests if query has already started
            query+=" AND movies.movieid IN ( ";
        query+="SELECT movieid FROM movies WHERE movieid IN (SELECT countryindex.movieid FROM
countryindex INNER JOIN countries ON countryindex.countryid = countries.countryid WHERE ";
        for (unsigned int i=0; i<movie.countries.size(); i++)
        {
            if (i!=0) // tests if query has already started
                query+=" OR ";
            query+="countries.country = ?";
        }
        query+=")";
```

```
            if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!=""||movie.directors.size()>0||movie.genres.size()>0) // tests if query has already started
                query+=")";
        }
        if (movie.keywords.size()>0)    // add in searches for keywords to the sql statement (if any)
        {
            if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!=""||movie.directors.size()>0||movie.genres.size()>0||movie.countries.size()>0) // tests if query has already started
                query+=" AND movies.movieid IN ( ";
            query+="SELECT movieid FROM movies WHERE movieid IN (SELECT keywordindex.movieid FROM keywordindex INNER JOIN keywords ON keywordindex.keywordid = keywords.keywordid WHERE ";
            for (unsigned int i=0; i<movie.keywords.size(); i++)
            {
                if (i!=0) // tests if query has already started
                    query+=" OR ";
                query+="keywords.keyword = ?";
            }
            query+=")";
            if (movie.title!=""||movie.runtime!=0||movie.release!=0||movie.username!=""||movie.directors.size()>0||movie.genres.size()>0||movie.countries.size()>0) // tests if query has already started
                query+=")";
        }

        if (query=="")  // if there are no specific cases, search for all
            query = "SELECT movieid FROM movies";
        query+=" ORDER BY movies.title";

        // Prepare statement
        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad" << endl;
            return ret;
        }

        // Bind variables
        if (movie.title!="")    //bind title if needed
        {
            if (sqlite3_bind_text(stmt,parameter_index,movie.title.c_str(),movie.title.size(),SQLITE_STATIC) !=
                    SQLITE_OK)
            {
                cout << "could not bind movie title." << endl;
                return ret;
            }
            parameter_index++;
        }
        if (movie.runtime!=0)   //bind runtime if needed
        {
            if (sqlite3_bind_int(stmt,parameter_index,movie.runtime) != SQLITE_OK)
            {
                cout << "could not bind movie runtime." << endl;
                return ret;
            }
            parameter_index++;
```

```
    }
    if (movie.release!=0)   //bind release if needed
    {
        if (sqlite3_bind_int(stmt,parameter_index,movie.release) != SQLITE_OK)
        {
            cout << "could not bind movie release." << endl;
            return ret;
        }
        parameter_index++;
    }
    if (movie.username!="") //bind username if needed, for 'search for movies created by this user'
    {
        if
(sqlite3_bind_text(stmt,parameter_index,movie.username.c_str(),movie.username.size(),SQLITE_STATIC) !=
          SQLITE_OK)
        {
            cout << "could not bind movie title." << endl;
            return ret;
        }
        parameter_index++;
    }
    for (unsigned int i=0; i<movie.directors.size(); i++)   //bind each director if needed
    {
        if
(sqlite3_bind_text(stmt,parameter_index,movie.directors[i].c_str(),movie.directors[i].size(),SQLITE_STATIC) !=
          SQLITE_OK)
        {
            cout << "could not bind director." << endl;
            return ret;
        }
        parameter_index++;
    }
    for (unsigned int i=0; i<movie.genres.size(); i++)      //bind each genre if needed
    {
        if
(sqlite3_bind_text(stmt,parameter_index,movie.genres[i].c_str(),movie.genres[i].size(),SQLITE_STATIC) !=
          SQLITE_OK)
        {
            cout << "could not bind genre." << endl;
            return ret;
        }
        parameter_index++;
    }
    for (unsigned int i=0; i<movie.countries.size(); i++)   //bind each country if needed
    {
        if
(sqlite3_bind_text(stmt,parameter_index,movie.countries[i].c_str(),movie.countries[i].size(),SQLITE_STATIC) !=
          SQLITE_OK)
        {
            cout << "could not bind country." << endl;
            return ret;
        }
        parameter_index++;
    }
    for (unsigned int i=0; i<movie.keywords.size(); i++)    //bind each keyword if needed
    {
```

```
            if
(sqlite3_bind_text(stmt,parameter_index,movie.keywords[i].c_str(),movie.keywords[i].size(),SQLITE_STATIC) !=
              SQLITE_OK)
            {
                  cout << "could not bind keyword." << endl;
                  return ret;
            }
            parameter_index++;
      }

      results = db->query(query.c_str(),stmt);        //execute search

      for (vector<vector<string> >::iterator it = results.begin(); it != results.end(); it++) // get results into form of a
                                                                                              //vector of movieInfo structs
      {
            row = *it;
            ret.push_back(get_movie(atoi(row[0].c_str()))); // adds all results to vector
      }

      return ret;
}

bool movieController::edit_movie(movieInfo& movie, movieInfo& old)
{
      string query;
      int movieid; // retrieved movie identifier
      sqlite3_stmt *stmt = NULL;
      vector<string> row;
      vector<vector<string> > results;
      int directorid, genreid, countryid, keywordid;

      query = "SELECT movieid FROM movies WHERE username = ? AND title = ? AND release = ? AND
runtime = ?";
      // query to find the movie
      if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
      {
            cout << "Error: SQL Prepared Bad in movies2" << endl;
            return false;
      }

      if (sqlite3_bind_text(stmt,2,old.title.c_str(),old.title.size(),SQLITE_STATIC) != SQLITE_OK)
      {
            cout << "could not bind title." << endl;
            return false;
      }
      if (sqlite3_bind_int(stmt,3,old.release) != SQLITE_OK)
      {
            cout << "could not bind release." << endl;
            return false;
      }
      if (sqlite3_bind_int(stmt,4,old.runtime) != SQLITE_OK)
      {
            cout << "could not bind runtime." << endl;
            return false;
      }
      if (sqlite3_bind_text(stmt,1,old.username.c_str(),old.username.size(),SQLITE_STATIC) != SQLITE_OK)
```

```
{
	cout << "could not bind username." << endl;
	return false;
}

results = db->query(query.c_str(),stmt); // get movies unique identifier for movie to be changed
if (results.size()>0)
{
	vector<vector<string> >::iterator it = results.begin();
	row = *it;
	movieid = atoi(row[0].c_str());
}
else
{
	cout << "Error: Movie does not appear to exist" << endl;
	return false; // error if movie not found and return false
}

query = "SELECT movieid FROM movies WHERE title = ? AND release = ?";

if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad in movies2" << endl;
	return false;
}
if (sqlite3_bind_text(stmt,1,movie.title.c_str(),movie.title.size(),SQLITE_STATIC) != SQLITE_OK)
{
	cout << "could not bind title." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,2,movie.release) != SQLITE_OK)
{
	cout << "could not bind release." << endl;
	return false;
}

results = db->query(query.c_str(),stmt);	// search to see if the movie is being changed to something which
						// already exists
if (results.size()>0)
{
	vector<vector<string> >::iterator it = results.begin();
	row = *it;
	if (atoi(row[0].c_str())!=movieid)	// errors if movies are not edited correctly (defensive coding)
	// i.e. it is being changed to something which already exists
	{
		cout << "Error: Changing title and release year to a movie that already exists" << endl;
		return false;
	}
}

query = "UPDATE movies SET username = ?, title = ?, release = ?, runtime = ? WHERE movieid = ?";
// setup query to update the movie
if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad in movies" << endl;
	return false;
```

```
}
if (sqlite3_bind_text(stmt,2,movie.title.c_str(),movie.title.size(),SQLITE_STATIC) != SQLITE_OK)
{
	cout << "could not bind title." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,3,movie.release) != SQLITE_OK)
{
	cout << "could not bind release." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,4,movie.runtime) != SQLITE_OK)
{
	cout << "could not bind runtime." << endl;
	return false;
}
if (sqlite3_bind_text(stmt,1,movie.username.c_str(),movie.username.size(),SQLITE_STATIC) !=
	SQLITE_OK)
{
	cout << "could not bind username." << endl;
	return false;
}
if (sqlite3_bind_int(stmt,5,movieid) != SQLITE_OK)
{
	cout << "could not bind movieid." << endl;
	return false;
}

db->query(query.c_str(),stmt);  // update the movie

// Insert directors if required
for (unsigned int i=0; i<movie.directors.size(); i++)
{
	query = "SELECT directorid FROM directors WHERE director = ?";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad" << endl;
		return false;
	}

	if (sqlite3_bind_text(stmt,1,movie.directors[i].c_str(),movie.directors[i].size(),SQLITE_STATIC) !=
			SQLITE_OK)
	{
		cout << "could not bind movie title." << endl;
		return false;
	}
	results = db->query(query.data(),stmt); //search if director already exists

	if (!results.empty())   // if directory already exists use its id
	{
		vector<vector<string> >::iterator it = results.begin();
		row = *it;
		directorid = atoi(row[0].c_str());
	}
	else    // otherwise create the director
```

```
{
	query = "SELECT MAX(directorid) FROM directors";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad in movies" << endl;
		return false;
	}

	// Get unique identifier from table
	results = db->query((char*)"SELECT MAX(directorid) FROM directors",stmt);	//get the next
                                                                                // director id, or 0 if none exist
	if (!results.empty())
	{
		vector<vector<string> >::iterator it = results.begin();
		row = *it;
		directorid = atoi(row[0].c_str())+1;
	}
	else
		directorid = 0;

	query = "INSERT INTO directors VALUES (?,?)";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad in directors" << endl;
		return false;
	}

	if (sqlite3_bind_int(stmt,1,directorid) != SQLITE_OK)
	{
		cout << "could not bind directorid." << endl;
		return false;
	}
	if (sqlite3_bind_text(stmt,2,movie.directors[i].c_str(),movie.directors[i].size(),SQLITE_STATIC) !=
		SQLITE_OK)
	{
		cout << "could not bind directors." << endl;
		return false;
	}

	// Insert into directors table
	db->query((char*)"INSERT INTO directors VALUES (?,?)",stmt);
}

query = "SELECT * FROM directorindex WHERE directorid = ? AND movieid = ?";
// check to see if the link already exists between director and movie
if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
	cout << "Error: SQL Prepared Bad in movies" << endl;
	return false;
}

if (sqlite3_bind_int(stmt,1,directorid) != SQLITE_OK)
{
	cout << "could not bind directorid." << endl;
```

```
            return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
            cout << "could not bind movieid." << endl;
            return false;
        }

        // Get unique identifier from table
        results = db->query(query.c_str(),stmt);
        if (results.empty())    // if there is no link add it
        {
            query = "INSERT INTO directorindex VALUES (?,?)";

            if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
            {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
            }

            if (sqlite3_bind_int(stmt,1,directorid) != SQLITE_OK)
            {
                cout << "could not bind director ID." << endl;
                return false;
            }
            if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
            {
                cout << "could not bind movie ID." << endl;
                return false;
            }
            // Insert into directors index table
            db->query(query.c_str(),stmt);
        }
}

// Insert genres if required
for (unsigned int i=0; i<movie.genres.size(); i++)
{
        query = "SELECT genreid FROM genres WHERE genre = ?";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad" << endl;
            return false;
        }

        if (sqlite3_bind_text(stmt,1,movie.genres[i].c_str(),movie.genres[i].size(),SQLITE_STATIC) !=
                SQLITE_OK)
        {
            cout << "could not bind movie title." << endl;
            return false;
        }
        results = db->query(query.data(),stmt); //search if genre already exists

        if (!results.empty())   // if genre already exists use its id
        {
```

```
        vector<vector<string> >::iterator it = results.begin();
        row = *it;
        genreid = atoi(row[0].c_str());
}
else    // otherwise add the genre
{
        query = "SELECT MAX(genreid) FROM genres";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad in movies" << endl;
                return false;
        }

        // Get unique identifier from table
        results = db->query((char*)"SELECT MAX(genreid) FROM genres",stmt);     //get the next genre id,
                                                                                //or 0 if none exist
        if (!results.empty())
        {
                vector<vector<string> >::iterator it = results.begin();
                row = *it;
                genreid = atoi(row[0].c_str())+1;
        }
        else
                genreid = 0;

        query = "INSERT INTO genres VALUES (?,?)";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad in genres" << endl;
                return false;
        }

        if (sqlite3_bind_int(stmt,1,genreid) != SQLITE_OK)
        {
                cout << "could not bind genreid." << endl;
                return false;
        }
        if (sqlite3_bind_text(stmt,2,movie.genres[i].c_str(),movie.genres[i].size(),SQLITE_STATIC) !=
                SQLITE_OK)
        {
                cout << "could not bind genres." << endl;
                return false;
        }

        // Insert into genres table
        db->query((char*)"INSERT INTO genres VALUES (?,?)",stmt);
}

query = "SELECT * FROM genreindex WHERE genreid = ? AND movieid = ?";
// check to see if the link already exists between genre and movie
if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
{
        cout << "Error: SQL Prepared Bad in movies" << endl;
        return false;
```

```
        }

        if (sqlite3_bind_int(stmt,1,genreid) != SQLITE_OK)
        {
                cout << "could not bind genreid." << endl;
                return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
                cout << "could not bind movieid." << endl;
                return false;
        }

        // Get unique identifier from table
        results = db->query(query.c_str(),stmt);
        if (results.empty())    // if there is no link add it
        {
                query = "INSERT INTO genreindex VALUES (?,?)";

                if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
                {
                        cout << "Error: SQL Prepared Bad" << endl;
                        return false;
                }

                if (sqlite3_bind_int(stmt,1,genreid) != SQLITE_OK)
                {
                        cout << "could not bind genre ID." << endl;
                        return false;
                }
                if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
                {
                        cout << "could not bind movie ID." << endl;
                        return false;
                }
                // Insert into genres index table
                db->query(query.c_str(),stmt);
        }
}

// Insert countries if required
for (unsigned int i=0; i<movie.countries.size(); i++)
{
        query = "SELECT countryid FROM countries WHERE country = ?";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }

        if (sqlite3_bind_text(stmt,1,movie.countries[i].c_str(),movie.countries[i].size(),SQLITE_STATIC) !=
                        SQLITE_OK)
        {
                cout << "could not bind movie title." << endl;
                return false;
```

```
}
results = db->query(query.data(),stmt); //search if country already exists

if (!results.empty())   // if country already exists use its id
{
	vector<vector<string> >::iterator it = results.begin();
	row = *it;
	countryid = atoi(row[0].c_str());
}
else
{
	query = "SELECT MAX(countryid) FROM countries";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad in movies" << endl;
		return false;
	}

	// Get unique identifier from table
	results = db->query((char*)"SELECT MAX(countryid) FROM countries",stmt);	//get the next
											//country id, or 0 if none exist
	if (!results.empty())
	{
		vector<vector<string> >::iterator it = results.begin();
		row = *it;
		countryid = atoi(row[0].c_str())+1;
	}
	else
		countryid = 0;

	query = "INSERT INTO countries VALUES (?,?)";

	if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
	{
		cout << "Error: SQL Prepared Bad in countries" << endl;
		return false;
	}

	if (sqlite3_bind_int(stmt,1,countryid) != SQLITE_OK)
	{
		cout << "could not bind countryid." << endl;
		return false;
	}
	if (sqlite3_bind_text(stmt,2,movie.countries[i].c_str(),movie.countries[i].size(),SQLITE_STATIC) !=
		SQLITE_OK)
	{
		cout << "could not bind countries." << endl;
		return false;
	}

	// Insert into countries table
	db->query((char*)"INSERT INTO countries VALUES (?,?)",stmt);
}

query = "SELECT * FROM countryindex WHERE countryid = ? AND movieid = ?";
```

```
        // check to see if the link already exists between country and movie
        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad in movies" << endl;
                return false;
        }

        if (sqlite3_bind_int(stmt,1,countryid) != SQLITE_OK)
        {
                cout << "could not bind countryid." << endl;
                return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
                cout << "could not bind movieid." << endl;
                return false;
        }

        // Get unique identifier from table
        results = db->query(query.c_str(),stmt);
        if (results.empty())    // if there is no link add it
        {
                query = "INSERT INTO countryindex VALUES (?,?)";

                if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
                {
                        cout << "Error: SQL Prepared Bad" << endl;
                        return false;
                }

                if (sqlite3_bind_int(stmt,1,countryid) != SQLITE_OK)
                {
                        cout << "could not bind country ID." << endl;
                        return false;
                }
                if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
                {
                        cout << "could not bind movie ID." << endl;
                        return false;
                }
                // Insert into countries index table
                db->query(query.c_str(),stmt);
        }
}

// Insert keywords if required
for (unsigned int i=0; i<movie.keywords.size(); i++)
{
        query = "SELECT keywordid FROM keywords WHERE keyword = ?";

        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
        }
```

```
if (sqlite3_bind_text(stmt,1,movie.keywords[i].c_str(),movie.keywords[i].size(),SQLITE_STATIC) !=
            SQLITE_OK)
{
    cout << "could not bind movie title." << endl;
    return false;
}
results = db->query(query.data(),stmt); //search if keyword already exists

if (!results.empty())   // if keyword already exists use its id
{
    vector<vector<string> >::iterator it = results.begin();
    row = *it;
    keywordid = atoi(row[0].c_str());
}
else
{
    query = "SELECT MAX(keywordid) FROM keywords";

    if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
    {
        cout << "Error: SQL Prepared Bad in movies" << endl;
        return false;
    }

    // Get unique identifier from table
    results = db->query((char*)"SELECT MAX(keywordid) FROM keywords",stmt); //get the next
                                                                            //keyword id, or 0 if none exist
    if (!results.empty())
    {
        vector<vector<string> >::iterator it = results.begin();
        row = *it;
        keywordid = atoi(row[0].c_str())+1;
    }
    else
        keywordid = 0;

    query = "INSERT INTO keywords VALUES (?,?)";

    if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
    {
        cout << "Error: SQL Prepared Bad in keywords" << endl;
        return false;
    }

    if (sqlite3_bind_int(stmt,1,keywordid) != SQLITE_OK)
    {
        cout << "could not bind keywordid." << endl;
        return false;
    }
    if (sqlite3_bind_text(stmt,2,movie.keywords[i].c_str(),movie.keywords[i].size(),SQLITE_STATIC)
            != SQLITE_OK)
    {
        cout << "could not bind keywords." << endl;
        return false;
    }
```

```
            // Insert into keywords table
            db->query((char*)"INSERT INTO keywords VALUES (?,?)",stmt);
        }

        query = "SELECT * FROM keywordindex WHERE keywordid = ? AND movieid = ?";
        // check to see if the link already exists between director and movie
        if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
        {
            cout << "Error: SQL Prepared Bad in movies" << endl;
            return false;
        }

        if (sqlite3_bind_int(stmt,1,keywordid) != SQLITE_OK)
        {
            cout << "could not bind keywordid." << endl;
            return false;
        }
        if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
        {
            cout << "could not bind movieid." << endl;
            return false;
        }

        // Get unique identifier from table
        results = db->query(query.c_str(),stmt);
        if (results.empty())    // if there is no link add it
        {
            query = "INSERT INTO keywordindex VALUES (?,?)";

            if (sqlite3_prepare_v2(db->getdb(),query.c_str(),-1,&stmt,0) != SQLITE_OK)
            {
                cout << "Error: SQL Prepared Bad" << endl;
                return false;
            }

            if (sqlite3_bind_int(stmt,1,keywordid) != SQLITE_OK)
            {
                cout << "could not bind keyword ID." << endl;
                return false;
            }
            if (sqlite3_bind_int(stmt,2,movieid) != SQLITE_OK)
            {
                cout << "could not bind movie ID." << endl;
                return false;
            }
            // Insert into keywords index table
            db->query(query.c_str(),stmt);
        }
    }

    return true;
}


//Functions for importing
```

```
void movieController::genreImport(movieInfo& movie)
{
	int lastspace;
	bool done = false;
	string line, genre, title;
	ifstream fin;

	fin.open("genres.list"); // open genres list for import
	while (fin.good())
	{
		getline(fin,line);
		lastspace = max(line.find_last_of(' '),line.find_last_of('\t'));
		title = line.substr(0,lastspace); // gets the title without the year
		trimstr(title); // trim whitespace
		while (title==movie.title && fin.good())	// while it is for the current movie add these genres
		{
			genre = line.substr(lastspace+1);
			movie.genres.push_back(genre);  //add to record
			done = true; // flag to test for genre's found

			getline(fin,line);
			lastspace = max(line.find_last_of(' '),line.find_last_of('\t'));
			title = line.substr(0,lastspace);
			trimstr(title);
		}
		if (done)
			return;
	}
}

void movieController::importMovies(string username)
{
	int lastspace;
	char choice;
	string line, year;
	ifstream fin;
	vector<movieInfo> results;
	movieInfo tempMovie;

	fin.open("movies.list"); // opens movies list for importing
	while (fin.good())
	{
		/* Initialize temporary movie */
		tempMovie.title = "";
		tempMovie.release = 0;
		tempMovie.genres.clear();
		tempMovie.keywords.clear();
		getline(fin,line);
		lastspace = max(line.find_last_of(' '),line.find_last_of('\t'));
		year = line.substr(lastspace+1);	// year will be all characters after last whitespace
		if (fin.good())
		{
			tempMovie.title = line.substr(0,lastspace);
			trimstr(tempMovie.title);
			if (isNumeric(year.c_str())) // test for valid year
				tempMovie.release = atoi(year.c_str());
```

```
                else
                        tempMovie.release = 0;
                tempMovie.runtime = 0;
                genreImport(tempMovie);
                setKeywords(tempMovie);
                results = search_movie(tempMovie);      // search for duplicates
                if(!results.empty())    //if a duplicate is found prompt
                {
                        choice = ' ';
                        while(choice != '1' && choice != '2')
                        {
                                cout << "Duplicate \"" << tempMovie.title << "\" found\n1. Skip\n2. Merge\n3. Abort"
                                        << endl;
                                choice = cin.get();
                                cin.ignore(100, '\n');
                                switch (choice)
                                {
                                        case '1':
                                                break;
                                        case '2':
                                                edit_movie(tempMovie,results[0]); // edits existing movie if found
                                                break;
                                        case '3':
                                                return;
                                }
                        }
                }
                else
                {
                        tempMovie.username = username;  //set the creator
                        create_movie(tempMovie);        //add movie to database
                }
        }
    }
}

bool movieController::isNumeric(const char* input)
{
    double dTestSink;
    istringstream iss(input);

    iss >> dTestSink;

    // was any input successfully consumed/converted?
    if (!iss)
        return false;

    // was all the input successfully consumed/converted?
    return (iss.rdbuf()->in_avail()==0);
}

void movieController::trimstr(string& str)
{
    while (isspace(str[str.size()-1]))
        str.erase(str.size()-1);
}
```

```
void movieController::setKeywords(movieInfo& movieTemp)
{
	int len = movieTemp.title.length();
	int i = 0;
	//Common words that aren't to be allocated as keywords
	string commonWords[23] = {"the", "of", "to", "and", "a", "in", "is", "it", "that", "for", "on", "are", "with", "as",
				"I", "be", "have", "this", "or", "had", "by", "can", "but"};
	while(i < len)
	{
		string keyword = "";
		while(!isspace(movieTemp.title[i]) && movieTemp.title[i] != '\"' && i < len)	//Get the string
		{
			keyword += movieTemp.title[i];
			i++;
		}
		if(keyword != "")	//If there was a new word
		{
			bool valid = true;
			for(int j = 0; j < 23; j++)	//Test if it is a common word
			{
				if(keyword == commonWords[j])
					valid = false;
			}
			if(valid)	//If not it can be a keyword so add
				movieTemp.keywords.push_back(keyword);
		}
		i++;
	}
}
```

Struct Definitions

These are simple structs used for passing movie and user data between the Interface and the UserController and MovieController classes.

userInfo.h

```
/*
	Basic structure for passing User information
*/

#ifndef USERINFO_H
#define USERINFO_H

#include <string>
using namespace std;

struct userInfo
{
	string username;
	string fullname;
	bool admin;
};

#endif
```

movieInfo.h

```
/*
	Basic structure for passing Movie information
*/

#ifndef MOVIEINFO_H
#define MOVIEINFO_H

#include <vector>
#include <string>
using namespace std;

struct movieInfo
{
	string title;
	int release;
	int runtime;
	vector<string> directors;
	vector<string> genres;
	vector<string> countries;
	vector<string> keywords;
	string username;
};

#endif
```

Appendix 5: Support Code Samples

emoviestest Class

The emoviestest class and associated main file was used for CppUnit testing of the classes used in the project.

```
/*
 * File:   cppunit.h
 * Author: Tiaki Rice
 *
 * Created on 17 September 2011, 12:47 PM
 */

#ifndef EMOVIESTEST_H
#define EMOVIESTEST_H

#include "interface.h"
#include "userController.h"
#include "movieController.h"
#include "database.h"
#include <cppunit/TestCase.h>
#include <cppunit/extensions/HelperMacros.h>

class eMoviesTest : public CppUnit::TestFixture
{
	CPPUNIT_TEST_SUITE (eMoviesTest);
	CPPUNIT_TEST (create_user);
	CPPUNIT_TEST (validatePassword);
	CPPUNIT_TEST (searchMovie);
	CPPUNIT_TEST (importMovies);
	CPPUNIT_TEST (edit_user);
	CPPUNIT_TEST (search_user);
	CPPUNIT_TEST (delete_user);
	CPPUNIT_TEST (user_login);
	CPPUNIT_TEST (create_movie);
	CPPUNIT_TEST (delete_movie);
	CPPUNIT_TEST (edit_movie);
	CPPUNIT_TEST (get_movie);
	CPPUNIT_TEST_SUITE_END ();
private:
	Interface *interf;
	database *db;
	userController *uc;
	movieController *mc;
	void create_user();
	void validatePassword();
	void searchMovie();
	void importMovies();
	void edit_user();
	void search_user();
	void delete_user();
	void user_login();
```

```
        void create_movie();
        void delete_movie();
        void edit_movie();
        void get_movie();
public:
        void setUp();
        void tearDown();
};

#endif


/*
 * File:   emoviestest.cpp
 * Author: Tiaki Rice
 *
 * Created on 17 September 2011, 12:47 PM
 */

#include <iostream>
#include <stdio.h>
#include <vector>
#include <cstring>
#include <sstream>
#include "interface.h"
#include "emoviestest.h"
#include "userInfo.h"

using namespace std;

CPPUNIT_TEST_SUITE_REGISTRATION (eMoviesTest);

void eMoviesTest::setUp()
{
        stringstream stream;
        char *dbName = new char[16];

        stream << "dat.db\n";
        strcpy(dbName, "db.db");
        interf = new Interface(stream);
        db = new database(dbName);
        mc = new movieController(db);
        uc = new userController(db);
}

void eMoviesTest::tearDown()
{
        delete interf;
        delete db;
        delete mc;
        delete uc;
        remove("db.db");
}

void eMoviesTest::create_user()
{
```

```
	//Create a new user and add it
	userInfo info;
	info.username = "bobdole";
	info.fullname = "Bob Dole";
	info.admin = false;
	cout << "\nCreating a new user bobdole" << endl;
	CPPUNIT_ASSERT(uc->create_user(info, "password1"));
	cout << "Attempting to create a copy of an existing user" << endl;
	CPPUNIT_ASSERT_EQUAL(uc->create_user(info, "password1"), false);

}

void eMoviesTest::searchMovie()
{
	vector<movieInfo> results;
	movieInfo movie;
	bool isEmpty;

	//create existing movie
	movie.title = "bobmovie";
	movie.release = 1999;

	//Search for movie that doesnt exist
	CPPUNIT_ASSERT((mc->search_movie(movie)).empty());

	//Add movie to database
	CPPUNIT_ASSERT(mc->create_movie(movie));

	//Search for just added movie
	CPPUNIT_ASSERT(!(mc->search_movie(movie)).empty());
	//Search for the movie by title only
	movie.release = 0;
	CPPUNIT_ASSERT(!(mc->search_movie(movie)).empty());
	//Search for the movie by release year only
	movie.title = "";
	movie.release = 1999;
	CPPUNIT_ASSERT(!(mc->search_movie(movie)).empty());
}

void eMoviesTest::importMovies()
{
	//create a string for username to get credit and a movie
	//that exists in the movie list file
	string username = "bobdole";
	movieInfo movie;
	bool isEmpty;

	//create existing movie
	movie.title = "Banana Boy (2003)";
	movie.release = 2003;

	//Search for movie that doesnt exist
	CPPUNIT_ASSERT((mc->search_movie(movie)).empty());

	//Import the movie list
	mc->importMovies(username);
```

```
    //Search for the same movie that now should exist
    CPPUNIT_ASSERT(!(mc->search_movie(movie)).empty());
}

void eMoviesTest::validatePassword()
{
    //Check a valid password
    CPPUNIT_ASSERT(interf->validatePassword("password1"));
    //Check various invalid passwords
    CPPUNIT_ASSERT(interf->validatePassword("invalidpass"));
    CPPUNIT_ASSERT(interf->validatePassword("invalid1!"));
    CPPUNIT_ASSERT(interf->validatePassword("pass word1"));
    CPPUNIT_ASSERT(interf->validatePassword(""));
    CPPUNIT_ASSERT(interf->validatePassword("short1"));
}
```

testscript.sh

The following shell script was used to test the Final product as a whole by running a series of test cases contain in an input file.

```
#!/bin/bash

#       Tiaki Rice
#       18/9/2011
#       Run test cases from the file "in" and prints the results to "out"

    fname="in"
    NUMBER_OF_TEST_CASES=2
    testCase=""

    exec<$fname

    : > out

    for i in {1..2}
    do
        while read ARGS
        do
            if [ "$ARGS" = "---" ]
            then
                break
            fi
            testCase="$testCase\n$ARGS"
        done
        echo -e "\n---\nTest Case $i\n---\n" >> out
        echo -e $testCase > tempCase
        emovies >> out < tempCase
        rm tempCase
    done
```