**Final Documentation**

CS 321-02

Team 4: Film Finder

**Team Members**

Bradley Bowen

Erik Failing

Matthew Fletcher

Will Thomason

**Copyright Bradley Bowen, Erik Failing, Matthew Fletcher, Will Thomason, 2019. This work and the software it references are the intellectual properties of the authors. Permission is granted for these materials to be shared for non-commercial, educational purposes, provided that this copyright statements appears on the reproduced materials and notice is given that copying is by permission of the authors. To disseminate otherwise or to republish requires written permission from the authors**

Table of Contents

[1.0 PROJECT DESCRIPTION 5](#_Toc26427035)

[2.0 PROJECT MANAGEMENT 6](#_Toc26427036)

[History 6](#_Toc26427037)

[Personnel 6](#_Toc26427038)

[Effort 7](#_Toc26427039)

[3.0 USE CASES 8](#_Toc26427040)

[4.0 REQUIREMENTS 10](#_Toc26427041)

[FUNCTIONAL REQUIREMENTS 10](#_Toc26427042)

[INTERFACE REQUIREMENTS 11](#_Toc26427043)

[FUTURE MODIFICATIONS AND EXTENSIONS 12](#_Toc26427044)

[SUMMARY 12](#_Toc26427045)

[ASSOCIATED TESTS 13](#_Toc26427046)

[5.0 DESIGN 17](#_Toc26427047)

[Model Design 17](#_Toc26427048)

[View Design 18](#_Toc26427049)

[Controller Design 19](#_Toc26427050)

[Communication Design 20](#_Toc26427051)

[6.0 IMPLEMENTATION 21](#_Toc26427052)

[Packages and classes we developed 21](#_Toc26427053)

[Utility packages and classes we used 21](#_Toc26427054)

[Design Patterns that influenced your implementation decisions 21](#_Toc26427055)

[Test Plan (what was tested, what was not tested) 22](#_Toc26427056)

[7.0 DISCUSSION 23](#_Toc26427057)

[Appendix A 24](#_Toc26427058)

Table of Figures

[Figure 1 - Requirements Summary Table 13](#_Toc26427059)

[Figure 2 - Browse All Movies Test 13](#_Toc26427060)

[Figure 3 - Save & Load User Profile Test 14](#_Toc26427061)

[Figure 4 - Search for a Movie Test 14](#_Toc26427062)

[Figure 5 - Search for a Movie Test II 15](#_Toc26427063)

[Figure 6 - Save & remove movie to/from WatchList test 16](#_Toc26427064)

[Figure 7 - Go-to and Return-from test 16](#_Toc26427065)

[Figure 8 - UML Diagram 17](#_Toc26427066)

[Figure 9 - Movie Notecard 24](#_Toc26427067)

[Figure 10 - Scraper Notecard 24](#_Toc26427068)

[Figure 11 - User Notecard 25](#_Toc26427069)

[Figure 12 - Search Notecard 25](#_Toc26427070)

[Figure 13 - GUI View Notecard 26](#_Toc26427071)

[Figure 14 - Input Notecard 26](#_Toc26427072)

[Figure 15 - Output Notecard 27](#_Toc26427073)

[Figure 16 - FilmFinder Notecard 27](#_Toc26427074)

# 1.0 PROJECT DESCRIPTION

Finding a movie to watch can be a dilemma. You might know what you’re interested in but not what movie fulfills that desire. You might also be interested in a movie you heard about, but you’re not sure if you’d actually enjoy it. This is where our movie search engine comes in. We scrape popular movie review websites to form a comprehensive and unbiased opinion on each movie. In addition, our software allows the user to use flexible search vectors (like keywords) to find their ideal movie. We also offer niche search options to suit different occasions. Need a movie for date night? How about a movie to watch with your niece? With our search engine, you’ll be able to find whatever kind of movie you want, for any occasion.

# 2.0 PROJECT MANAGEMENT

## History

When we began working on this project, we envisioned it as a program that would allow the user to search for movies, compare them side-by-side, and save the ones that interested them to a WatchList. We were also going to include the option to use filters to arrange the search results in different orders (alphabetically, highest to lowest rating, etc.), but we decided to instead include a wider range of search fields that the user could choose from. For our database, we had originally planned to scrape data about movies every time the program was started in order to keep track of new releases and their ratings, but since another group was focusing on data scraping we decided to prioritize the user interface portion of our project. In place of the dynamic database, we built one using the American Film Institute’s 100 Years...100 Movies (10th Anniversary Edition) list and stored it in a json file. Since we now knew which movies would be in the database, we were able to set a poster that would display alongside each one.

## Personnel

* Matt Fletcher: I am a double major in Mathematics and Computer Science with an emphasis in Cybersecurity and a minor in mechanical engineering. This class was my first real experience with Java. Prior to this, I primarily focused in Python, C++, and FORTRAN. This is also my first team project in an academic environment (I worked on a 2 person team during my internship at IBM this summer). I primarily worked on the search algorithm and database wizard. For the search algorithm, I wrote a cascading partial match filter that would check for substring matches between search terms and movie elements.
* Erik Failing: I am a Computer Science major with a minor in Mathematics. I spend a mild portion of my time coding games in C#. Since C# is very similar to Java, I was already fluent with much of the material we used in creating this application. However, Unity’s way of handling a GUI is vastly different from Java Swing’s method so that took a lot of getting used to. I created our application’s overarching GUI structure and was responsible for everything in MainView and SearchView. I also attended to various tasks in other classes, like rewriting the search logic. In addition, I also ran various tests through our GUI and tackled any bugs that appeared.
* Will Thomason: I am a Computer Science major with a minor in Mathematics. This is also the first time I have coded in Java. Before this I worked almost exclusively in C++. I’ve never worked on a team project like this before, and I’ve never done much GUI programming either. To help get better at the second one, I spent most of my time working on the ResultsView class. Aside from making sure everything showed up that was supposed to, the main thing this entailed was ensuring that the WatchList functioned correctly and communicated with the Model in the way that it was supposed to.
* Bradley Bowen: I am a Computer Science major. I did have some basic experience with Java before this class, but I had not coded in Java since 2015. This also was my first experience working with a software team. Most of my effort was communicating with team members to make sure our different components could correctly integrate and interact with each other. My biggest implementation contribution was creating the initial search function logic in SearchBuilder, and creating the stub functions for the FilmFinder and UserProfile class.

## Effort

The various tasks to be completed were divided amongst the team members. Github was used as a central repository and for version control. Github was also used to keep track of current open issues. Any tasks that were to be completed had an Issue associated with them, so the development team could keep track of progress on each item. We met as a group ten times throughout the semester, with some smaller meetings every now and then for things such as preparing for presentations or making sure two people were building their code in a way that they could be integrated with one another. Furthermore, we communicated on Discord, updating other team members on progress inside individual tasks. We had a total of 10 team meetings throughout the semester, which accounted for about 8 hours of time spent on the project per person. Outside of team meetings, we did around 150 hours of work collectively, either individually or in smaller groups.

# 3.0 USE CASES

Actors - user of the program.

**Use Case 1: Browse All Movies (Display Everything to user)**

*(Feature Updated): capability exists, but doesn’t happen automatically.*

*Precondition: program finds local database and imports movie data.*

1. The user starts the program.
2. All movies are displayed to the user.
3. The user must scroll down the window to see all the movies if they can’t all fit into the window.
4. Optional: program loads a set of movies that are similar to the user’s interests (data from previous searches stored in their profile).
5. Optional: the user could select a button that would enable or disable the user preferences.

**Use Case 2: Search for a movie**

1. The user clicks on the GUI search bar with the mouse.
2. The user clicks on one of the six search criteria: title, minimum rating, Director, Year, Genre, or Actor.
3. The user types out his/her search using the keyboard, and then clicks on the “search” button with the mouse to initiate the search.
4. The program displays the movies found by the search.

(Optional) User chooses to sort movies (ascend and descend)

*FEATURE REMOVED*

* + 1. Alphabetical
    2. Rating Score
    3. Year of Release (or range)

1. The user clicks on the movie found by the search that they would like information about.

*FEATURE UPDATED*

1. Lastly, the user can click on a drop-down menu to choose what order they would like the movies sorted that were found by the search.

*FEATURE REMOVED*

**Use Case 3: Closely compare movies with the “Comparison” button.**

(*Feature Removed): we focused on making our primary search results window work correctly.*

1. The user can choose two or three movies by clicking on them.
2. The user then clicks the “Compare” button. This displays the movies side-by-side so they can directly compare ratings, actors, release date, and all other relevant information.

**Use Case 4: Save movies to the user’s personal WatchList**

1. The user clicks on a movie they are interested in with the mouse.
2. If they want to watch it sometime in the future they will click the “Add to WatchList” button to save it into their profile as a future reminder.
3. After the user exits the program, the program automatically saves all changes, and the WatchList will contain all of the selected movies the next time they start up the program.
4. To view the list, the user clicks the WatchList button, and then all of the movies are displayed.

**Use Case 5: View Search History**

(*Feature Removed): we decided this would not be very desirable to the user.*

1. The user clicks on search history.
2. The user can then scroll down the window to see their searches organized by date searched.

**Use Case 6: Remove from WatchList**

1. The user clicks on the WatchList button.
2. Then, they click on the movie from their WatchList they wish to remove.
3. After the movie is selected, a “Remove from WatchList” button will appear.
4. Finally, the user clicks the button to remove it from the list.
5. The display refreshes, and then the WatchList appears without the selected movie.

# 4.0 REQUIREMENTS

## FUNCTIONAL REQUIREMENTS

1. **Pull (scrape) movie data from internet rating site**
   1. (*Feature Removed) : program only uses local data file.*
   2. The first time a user runs the program the program will pull data from a rating website. This data includes: movie name, release date, director, actors, movie picture (poster), rating score, parental rating, genre, and other relevant data. The user doesn’t have to do anything to make this happen when the program starts up. The only requirement is that the computer must be connected to the internet.
2. **Update movie ratings data**
   1. (*Feature Removed): program only uses local data file.*
   2. Everytime the program is started it will check for rating updates online. It will perform the same operation as requirement 1, but only update the movie rating score.
      1. *(Feature Updated): We focused only on implementing a WatchList feature the user can control rather than automatic search preference collection.*
3. **Load user profile**
   1. Every time the program starts, if the user has had previous sessions, their profile data will be loaded. This data includes their previous search history, their personal WatchList, and (optional) preferences data that is generated from their search history and WatchList.
4. **Browse all movies**
   1. (*Feature Updated): Rather than being a default behavior, the user can perform a search with constraints to view the entire list*
   2. This is the first function the user will see in the view before searching. It loads every movie onto the screen at once, and (optionally) loads a set of recommended movies based on the user’s previous search/WatchList data.
5. **Search for a movie**
   1. The search is one of the primary features of the program. It will be a traditional search bar that can receive text input from the keyboard. There will be a “Search by” button that allows the user to search by the different categories related to movies (ie director, title, actor, release date, alphabetical, parental rating, and review ratings).
6. **Compare movies**
   1. (*Feature Removed): we focused on making our primary search results window work correctly.*
   2. This function places movie data side-by-side for easy comparison. After the movies to be compared have been selected, the program goes to a new screen. The director data of each movie will be on the same horizontal line, review rating will be aligned above another horizontal line below the director line, and so on until all information categories have been handled.
7. **Save movie to WatchList**
   1. This function adds a movie from the total list of movies to a subset list of movies that the user can control. The benefit is so the user doesn’t have to remember all of the movies he/she is interested in watching in the future. Movies can be added or removed from the list. Optionally, this data will also be used to determine the user’s movie preferences.
8. **Remove from WatchList**
   1. Allows the user to remove a movie from their WatchList.
9. View user search history
   1. (*Feature Removed): we decided this would not be very desirable to the user.*
   2. Allows the user to look back at their previous search history ordered by date.
10. **(Optional) User chooses to sort movies (ascend and descend)**
    1. (*Feature Removed): we focused on making fuzzy search and other core   features work well.*
    2. Alphabetical
    3. Rating Score
    4. Year of Release (or range)
11. **Go-to and Return-from**
    1. Basically, this function will handle displaying a new information after a movie has been clicked, or other redirection buttons on the screen have been clicked.
12. **Save user profile data**
    1. *(Feature Updated): Only WatchList is saved now.*
    2. After the program is closed, it will automatically back-up the current state of the user’s WatchList, search history, and preferences.

## INTERFACE REQUIREMENTS

1. **GUI**
   1. Search Input fields
   2. Error output messages
   3. Viewport for individual movie data (Window)
   4. Viewport for list of movies/posters
   5. Viewport when comparing WatchList movies
2. **Input**
   1. Text input (keyboard)
   2. Button input (mouse)
   3. User selected search criteria (year, actor, etc.)
   4. External database movie data (from internet)
3. **Output**
   1. Movie information
   2. Movie posters
   3. Error information
   4. A link chain (or some form of history) for backtracking previous searches the user can click with a mouse.
4. **EXCEPTIONS**
   1. No internet Connection to update ratings (*Feature Removed).*
   2. No Search Results Found

## FUTURE MODIFICATIONS AND EXTENSIONS

Most of our ideas for features that could be implemented in a later version pertain to features that we decided were less of a priority than others for our initial program. The main two that come to mind are the filters for ordering the already constructed ResultsList and adding functionality to FuzzySearch that would allow for matching of similar Strings using their Levenshtein distance. Other features that we thought of initially were a dynamically created database and a Movie Comparison feature. For the database we would need to either find and API for a website that had all of the information we were looking for, or we would need to make the program capable of putting the complete data together from multiple sources like we did for our database this time. For the Movie Comparison, we would need to either redesign our GUI to allow for the side-by-side, or we would want to implement it in a way that opened up another window that was designed for comparison.

## SUMMARY

|  |  |
| --- | --- |
| Requirement Contract | Related Scenario |
| **R1.** The software shall have a Graphical User Interface. | **Use Case 2**: the user searching for a movie using the GUI. |
| **R2.** The software shall implement a Model-View-Control pattern. | *We will implement this in our design phase.* |
| **R3.** The software will offer searching options, and produce nicely formatted text on the screen. | **Use Case 2**: the user searching for a movie, and then seeing the information on the window. |
| **R4.** The software will use json files to store and retrieve information in a structured manner. | **Requirement 12**: Save user profile data when program closes such as the user’s personal “watch list” of movies they created. |
| **R5.** The software will display images of the movies, not just text. | **Interface Requirement 1.c**: there will be a movie poster viewport. |
| **R6.** The software will have access to a collection of movies and their respective descriptions that the user can search through. | **Use Case 2**: it is assumed the user must have movies to search through in order for the software to be useful. |

Figure 1 - Requirements Summary Table

## ASSOCIATED TESTS

1. Browse All Movies

**Set-up**: Are read in from the json file and stored in a MasterList when the program starts. The user then performs a search with no constraints entered.

**Results**: Every Movie within the database is displayed to the user.

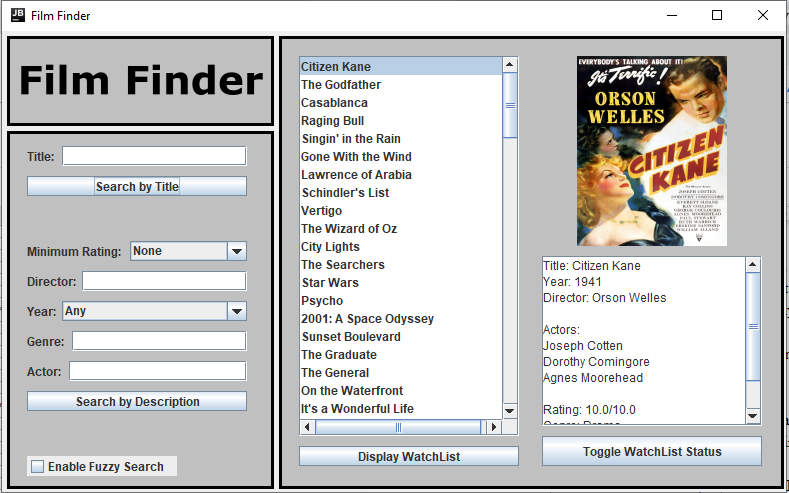


Figure 2 - Browse All Movies Test

**Conclusion**: The database and MasterList work correctly. R1, R3, R4, R5, R6 satisfied.

1. Save & Load user profile

**Set-up**: Movies are added to the user’s WatchList and then the application is closed. The application is then reopened and the WatchList is inspected for the presence of those recently added movies.

**Result**: Newly added movies still display in WatchList after program exits and reopens. R4 satisfied.

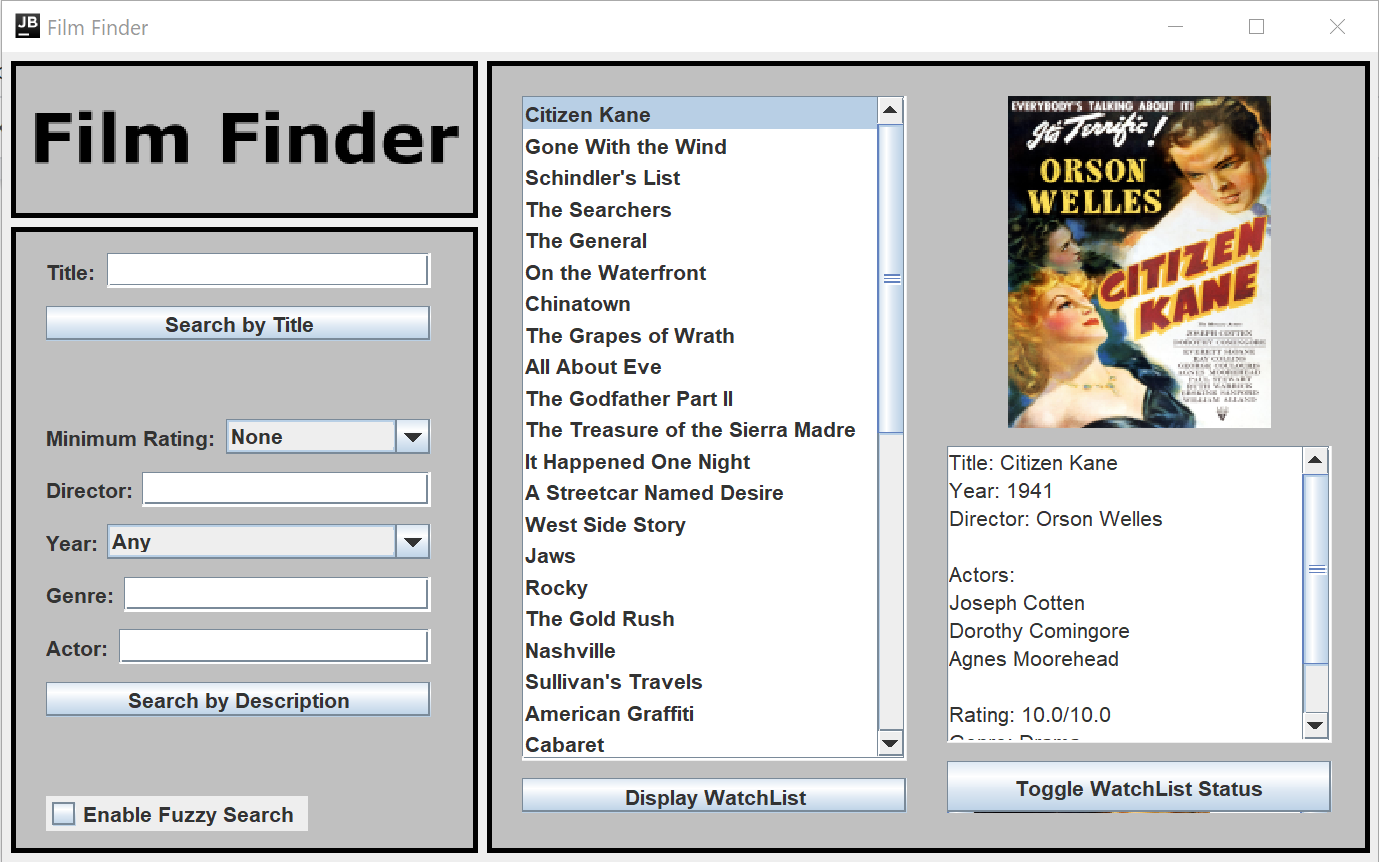


Figure 3 - Save & Load User Profile Test

**Conclusion**: The user’s profile loads correctly.

1. Search for a movie

**Set-up**: A movie currently in our database is searched for using every available search input.

**Result**: Searched for movie is found for every search field.

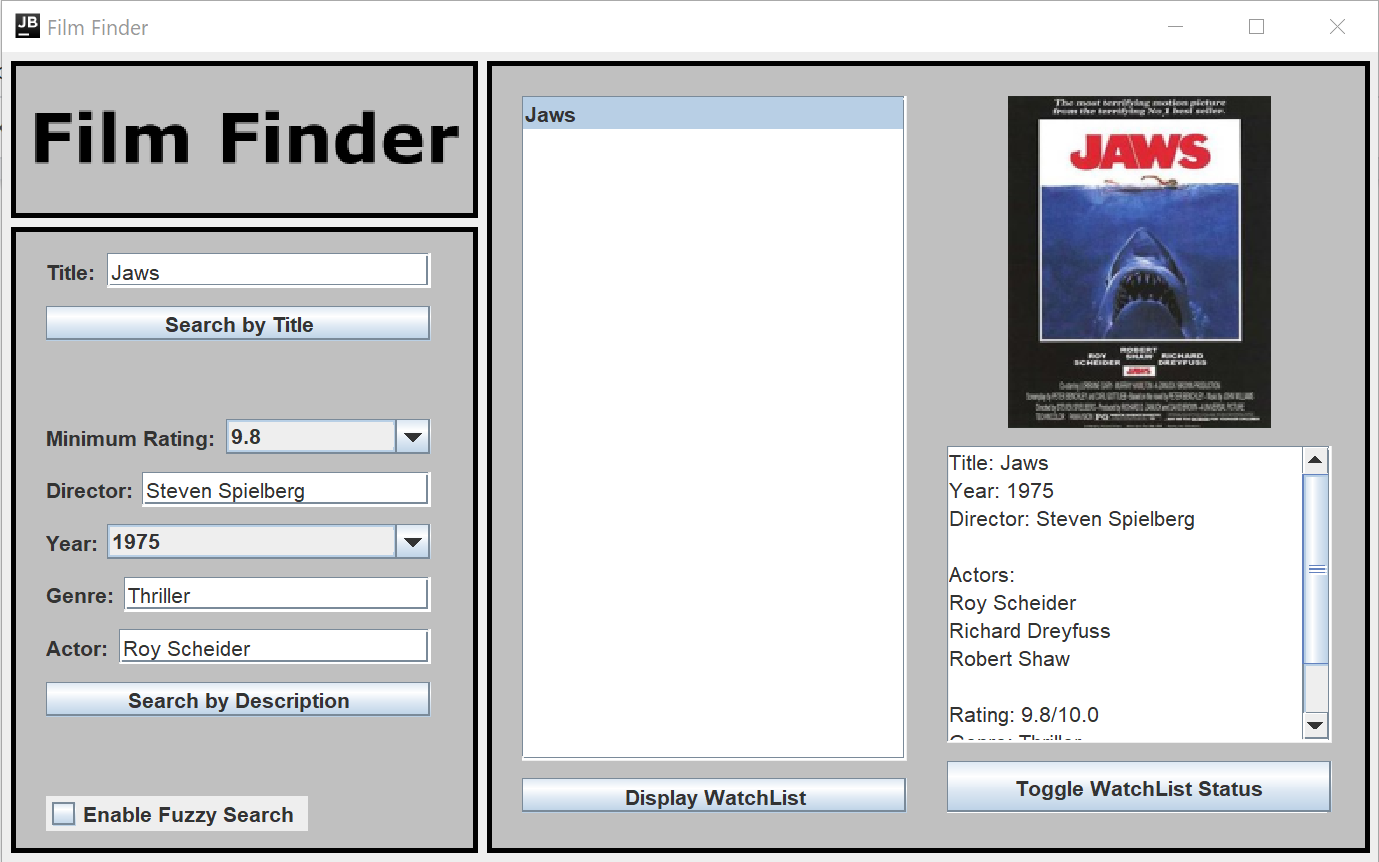


Figure 4 - Search for a Movie Test

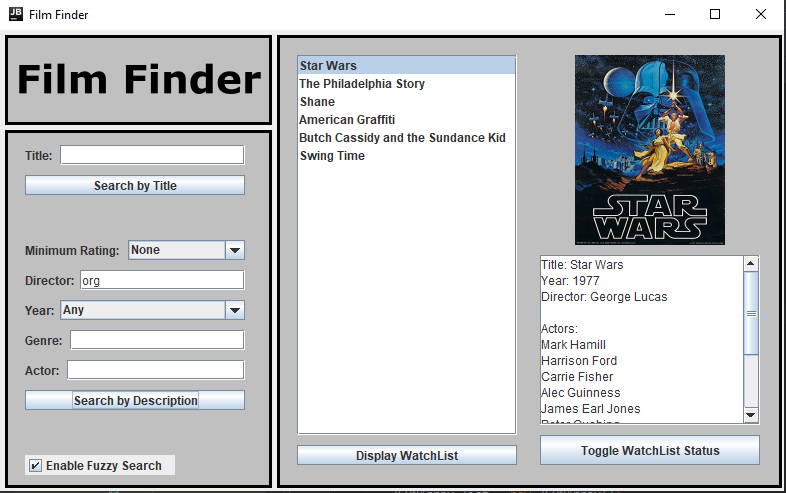


Figure 5 - Search for a Movie Test II

**Conclusion**: Our application accurately searches our database and finds the appropriate movie(s) given a set of parameters. R3 satisfied.

1. Save & remove movie to/from WatchList

**Set-up**: WatchList is displayed and has a movie removed. WatchList is then displayed again to see if movie was removed. If so, movie is searched for and then added to WatchList again. WatchList is then displayed to see if movie was added back.

**Result**: Both operations successful. The movie, All About Eve, was appropriately added to and removed from the WatchList.

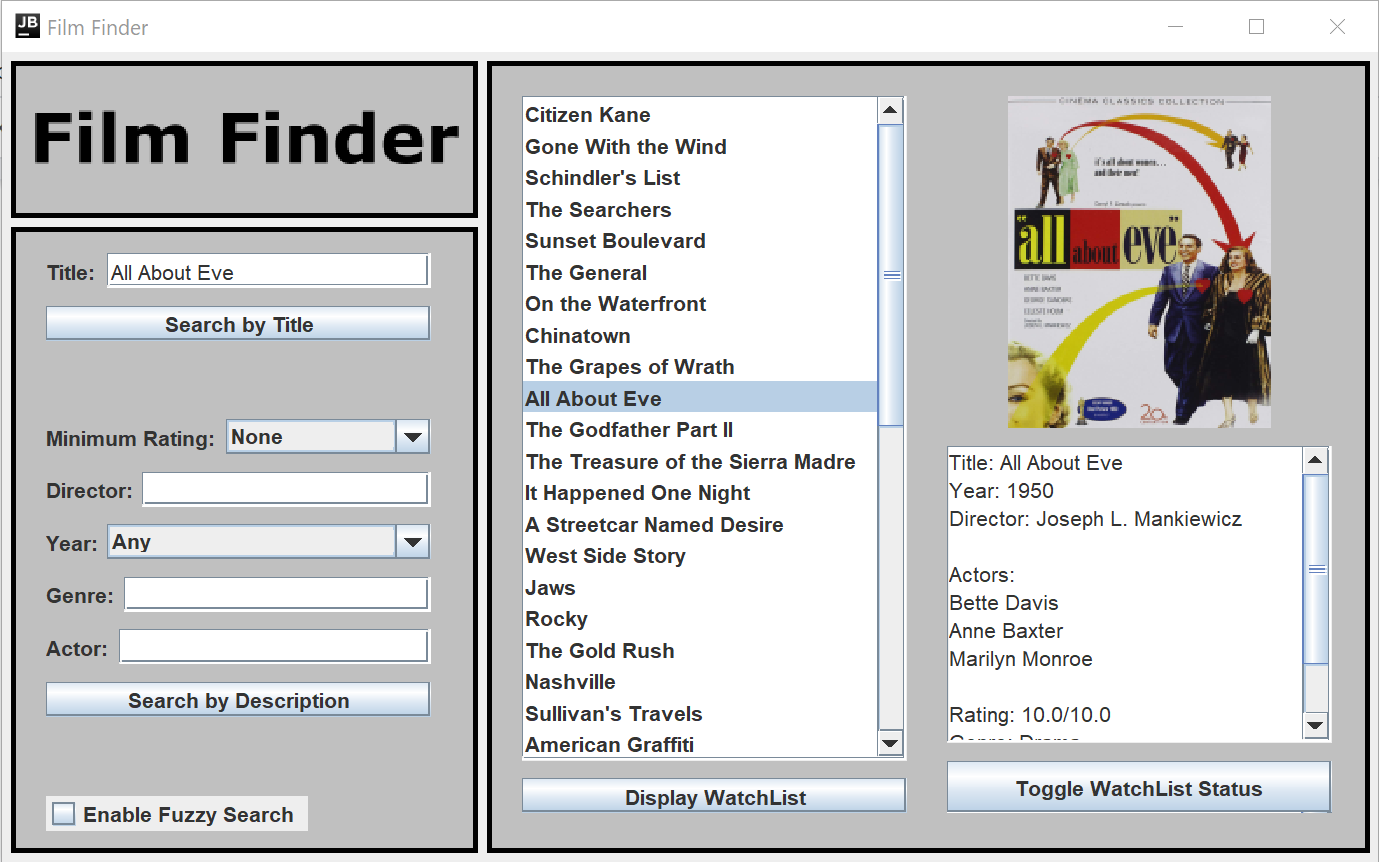


Figure 6 - Save & remove movie to/from WatchList test

**Conclusion**: WatchList adding and removing function work properly. R4 satisfied.

1. Go-to and Return-from

**Set-up**: Pull up the entire database by searching for every movie then attempt to select different movies in the results list.

**Result**: Movies are able to be selected and the UI updates appropriately.

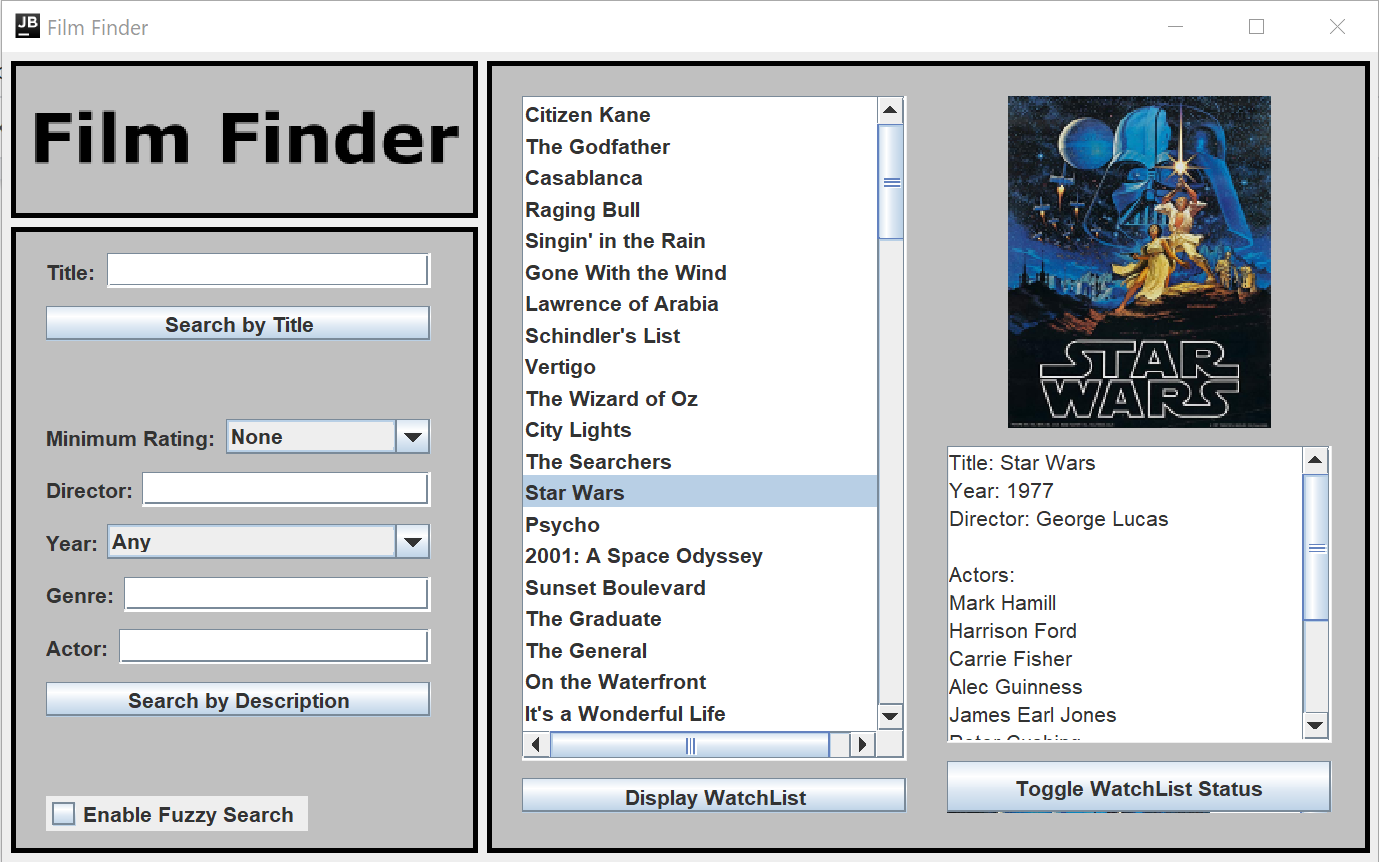


Figure 7 - Go-to and Return-from test

**Conclusion**: Movie selection functions work properly and allow the user to view every movie in the list appropriately. R1 satisfied.

# 5.0 DESIGN

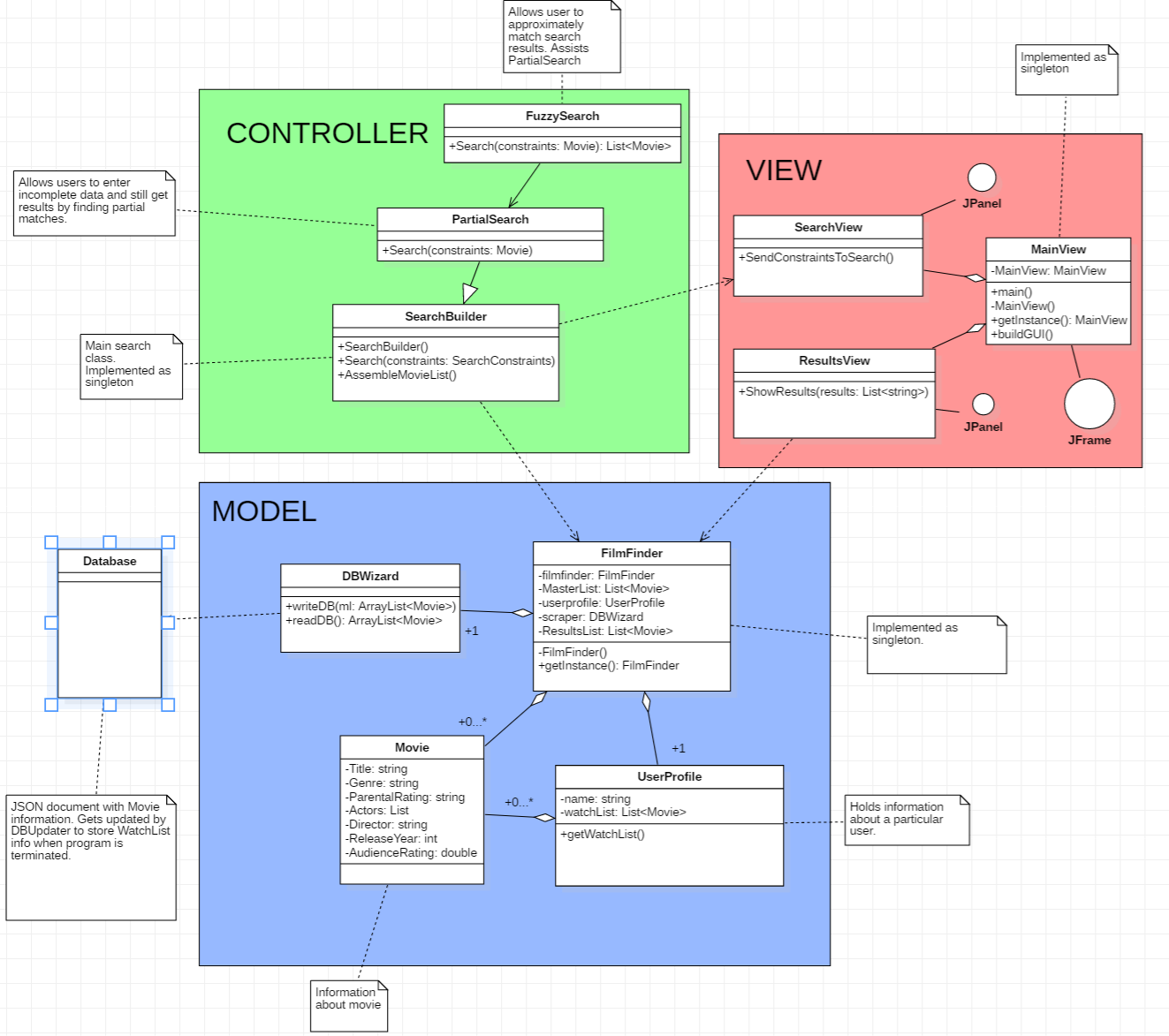


Figure 8 - UML Diagram

Model Design

The model in our program consists of four classes: FilmFinder, UserProfile, Movie, and DBWizard.

1. FilmFinder- This class serves as a middleman between the SearchBuilder and the rest of the classes within the model. When the program initializes, it retrieves Movie information from the Database via the DBWizard class. When a search is done it gives SearchBuilder the information it needs to perform the requested search, and then FilmFinder gets the ResultsList and passes it to the ResultsView object so the Movies be displayed to the user via the GUI. This class also interacts with the UserProfile object (which is used to store the WatchList) and collects the updated WatchList to make the appropriate changes to the Database via DBWizard when the program is terminated.
   1. Interacts With:
      1. Movie- Stores MasterList of Movies and updates their information as needed.
      2. UserProfile- Gives UserProfile movie objects that it needs for a WatchList and knows to update movies within database to maintain the current WatchList.
      3. Scraper- Uses this class to import information to create Movie objects.
      4. DBWizard- This class reads from the database file to create the MasterList, and writes to the file to preserve the MasterList as well as the WatchList.
      5. SearchBuilder- This class receives Movies from FilmFinder and passes back a second list of Movies that correspond to the user’s search constraints
      6. ResultsView- FilmFinder sends Movie information to this class so it can be displayed to the user
2. Movie- This class represents the movies whose data can be displayed by the program. These movies will come from a list of a set size (for example the American Film Institute’s list of Top 100 Movies), and they will be stored in a list within FilmFinder while the program is running. The user can then perform a search, which will put the Movie objects that meet the search criteria into a separate ResultsList, also contained within FilmFinder. The user can also choose to add movies to a list of films they want to watch later, which will be stored within the UserProfile object.
   1. Interacts With:
      1. UserProfile- Movies can be saved in a list of movies the user wants to watch later.
      2. FilmFinder- Movies are saved in a MasterList inside of FilmFinder.
3. UserProfile- This class contains the WatchList of Movies. This information is put there by the FilmFinder, which can also remove Movie objects from the WatchList if the user no longer wants them listed there.
   1. Interacts With:
      1. Movie- Saves Movie objects in a WatchList.
      2. FilmFinder- WatchList is managed by the FilmFinder object. .
4. DBWizard- This class has 2 primary purposes: Generating the master movie list from the database, and writing the necessary changes to the database file. If a movie has been added to (or removed from) the WatchList, it will find the movie’s information in the Database and update its WatchList status.
   1. FilmFinder- DBWizard gives FilmFinder the information it needs to create the MasterList, and FilmFinder passes that list (complete with any changes made) back to DBWizard so the database file is kept up to date.

View Design

The primary component of our program view is the MainView object. The two main parts of the GUI are contained within the SearchView and ResultsView objects.

1. MainView- This is the class representing the GUI that our users will be interacting with. It is made up of two main panels: SearchView and ResultsView. User input is handled using the SearchView panel, and the results from that search are handled by the ResultsView panel.
   1. SearchView- MainView uses SearchView to pass user input to the SearchBuilder.
   2. ResultsView- MainView presents Movie information to the user after it has been passed to ResultsView by FilmFinder.
   3. DBWizard- MainView only interacts with DBWizard when closing the program. When the window is closed, DBWizard executes the writeDB method, writing the MasterList out to the database. When this is done, the window closes and the program terminates.
2. SearchView- This class handles user input pertaining to search parameters. It then passes these parameters to the SearchBuilder in order to find the correct movies to display.
   1. MainView- MainView is the class that contains SearchView and presents it to the user.
   2. SearchBuilder- SearchBuidler receives the search parameters from SearchView and gives FilmFinder the information it needs to create the ResultsList.
   3. Movie- SearchView creates a Movie object and sets its values with the constraints set by the user. This Movie is then passed into SearchBuilder for the comparison
   4. PartialSearch- When the fuzzy toggle is activated, SearchView uses PartialSearch to compare the values of the Movies in the MasterList to the values passed in as search constraints.
3. ResultsView- This class represents the portion of the GUI dedicated to showing the user which movies were found during the search that was done.
   1. MainView- MainView serves as a container for the different parts of the GUI, including ResultsView. It presents the information from ResultsView within a panel.
   2. FilmFinder- FilmFinder contains the list of movies that match the user’s search and passes them to the ResultsView to be displayed to the user. ResultsView also interacts with FilmFinder in order to display and modify the WatchList.
   3. Movie- FilmFinder passes the ResultsList of Movie objects to ResultsView once a search has been completed. ResultsView does not directly modify any of the Movie objects in the program, instead it uses them to build lists of titles and description to display to the user. It then tells FilmFinder which Movies should have their WatchList status modified.

Controller Design

The controller within our program is made up of three classes: SearchBuilder, PartialSearch, and FuzzySearch. SearchBuilder receives user input from SearchView in the form of a Movie and uses that input as parameters in a function call when it tells FilmFinder to build the ResultsList of Movies. PartialSearch supplements SearchBuilder and is used to find substring matches within the MasterList using the search terms. FuzzySearch is where we were going to implement our calculation of Levenshtein distances to determine whether or not something would be considered a match or not.

1. SearchBuilder- This object takes the user input from the various boxes within SearchView and uses them to create the ResultsList that will be passed to ResultsView via FilmFinder.
   1. SearchView- This panel within the GUI receives the user input and passes it to the SearchBuilder, where it is then put into the proper format to execute the search.
   2. FilmFinder- FilmFinder passes the MasterList of Movies to SearchBuilder. If a Movie matches the constraints passed in by SearchView, that Movie is copied into a separate ResultsList which gets passed to FilmFinder once all matches have been found.
   3. FuzzySearch- Makes SearchBuilder capable of having FilmFinder find movies that closely match user input but aren’t exactly the same.
2. PartialSearch- This object is used to see if the search constraints passed in are contained within the appropriate field of the Movies within the MasterList.
   1. SearchView- This panel within the GUI receives the user input and passes it to the PartialSearch, where it is then put into the proper format to execute the search.
   2. FuzzySearch- This class assisted PartialSearch’s search logic and returned a narrowed down list of movies
   3. SearchBuilder- PartialSearch inherits from SearchBuilder and overrides the search method.
3. FuzzySearch- Contains information used to make a search capable of having FilmFinder create a ResultsList with close matches to user input if no exact ones are found (i.e. name that the user input was spelled incorrectly).
   1. PartialSearch- When search in PartialSearch is called, FuzzySearch assists and finds close

Communication Design

As shown in the UML diagram, FilmFinder serves a central hub for all of the components of the program. It is in charge of managing all of the objects within the Model (like UserProfile and the Movie objects), and it also connects the Model to the Controller and View modules of the design. Its relationship with SearchBuilder allows it to receive the parameters that the user wishes to use when searching for movies; and its connection to ResultsView allows it to display the results of the search to the UserProfile. SearchBuilder is also connected to the SearchView object, which is what allows it to receive the search terms that the user enters into the GUI.

# 6.0 IMPLEMENTATION

**Database:** The database was chosen to be written in a json format. This was chosen due to the ease of use, as well as the availability of the json-simple library, an open-source API for Java to both read and write a JSON object.

## Packages and classes we developed

* DBWizard
* FilmFinder
* FuzzySearch
* MainView
* Movie
* PartialSearch
* ResultsView
* Scraper
* SearchBuilder
* SearchView
* UserProfile

## Utility packages and classes we used

* import java.io.\*;
* import java.io.FileNotFoundException;
* import java.io.FileReader;
* import java.util.Scanner;
* import org.json.simple.JSONArray;
* import org.json.simple.JSONObject;
* import org.json.simple.parser.JSONParser;
* import org.json.simple.parser.ParseException;
* import javax.swing.\*;
* import java.awt.\*;
* import java.awt.event.ActionEvent;
* import java.awt.event.ActionListener;
* import java.util.ArrayList;

## Design Patterns that influenced your implementation decisions

**Singleton -** Early on in development we decided to use the singleton design pattern for certain key classes that would never have more than one instance created. This allowed us to easily reference one part of our program from a completely different part. Although singletons have their drawbacks, our application is small enough to avoid most of them. Now, nearing the end of our project, we still feel that using singletons was a great decision.

**Decorator -** Towards the end of development we had a few movies in our database with exceptionally long names. These names would break past the far right side of our display and the user would never be able to see the entire name. To fix the issue, we implemented a decorator pattern that adds scroll bars to our lists.

## Test Plan (what was tested, what was not tested)

Over the course of our project we did fairly rigorous statement testing. Before every commit to github, we required each team member to thoroughly test their code before uploading. Even still, some bugs did get through to the main build.

We tested in multiple stages. First, the code was tested by the programmer in local scope immediately after they finished coding. Then the code was implemented into the full application and tested for any integration errors. Finally, the code would be pushed to our build on git and other members of the group would test it.

We tested thoroughly in our command line and then, when we had a working GUI, we tested thoroughly through that instead.

The only thing we didn’t get to test thoroughly was fuzzy searching and that was just because we ran out of time.

# 7.0 DISCUSSION

One of the more interesting problems we encountered during this project came when performing multiple consecutive searches. We displayed the results by calling a function in the GUI, but since it had to have action listeners we couldn’t return the function to terminate it. As more searches were done, there would be multiple instances of the showMoviesText function running on top of each other. The biggest problem with this came when trying to change the WatchList status of a Movie because the program would attempt to do it to each “layer” of the function. In order to fix this, we had to restructure the program to ensure that only the correct movie was being removed instead of just using the selected index. This left a problem where the program would work, but there would still be multiple layers of results that would throw IndexOutOfBounds exceptions whenever one list wasn’t as long as the one currently being viewed. Since this error would only occur within layers that no longer needed to be displayed, we were able to handle the exception by terminating that instance of the function.

# Appendix A

**CRC Cards-** these are the original cards we created. We used them to help us begin discovering what classes we needed, including their respective responsibilities (Left column) and dependencies (right column). We also used them to help us visualize the expected flow of the program, and in what order it would pass through the objects.

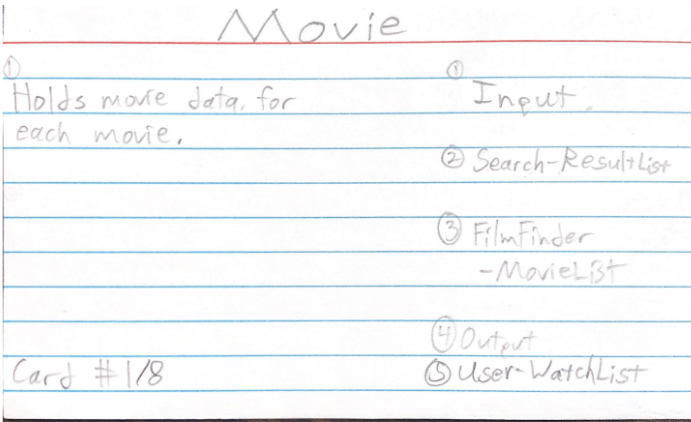


Figure 9 - Movie Notecard

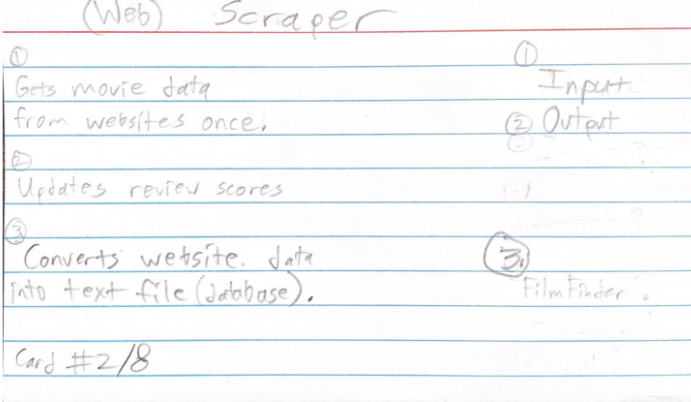


Figure 10 - Scraper Notecard

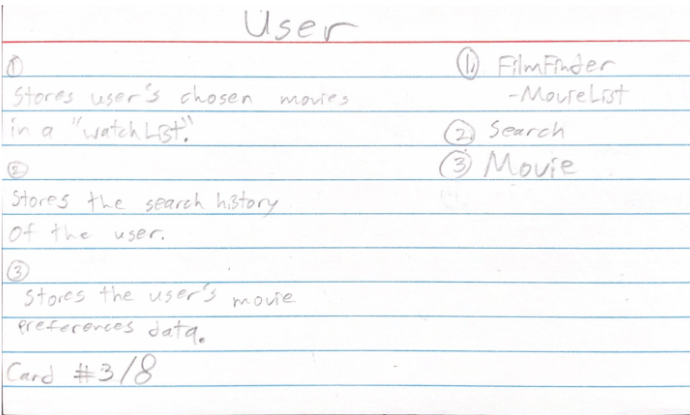


Figure 11 - User Notecard

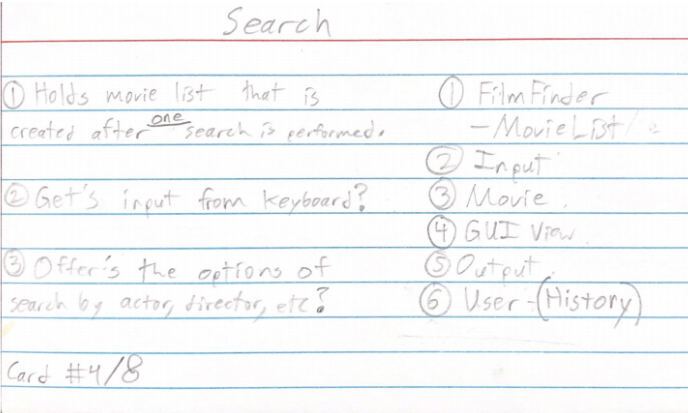


Figure 12 - Search Notecard

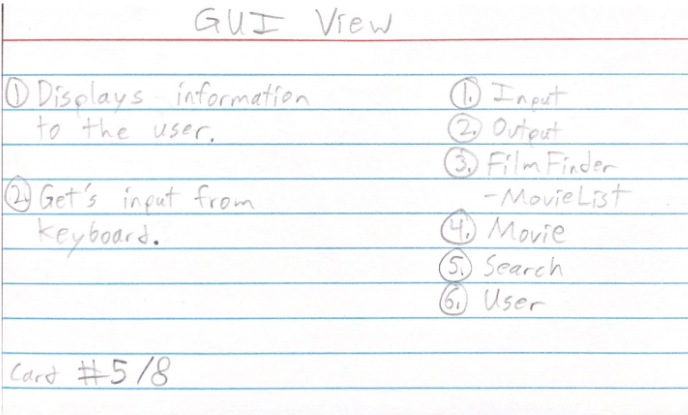


Figure 13 - GUI View Notecard

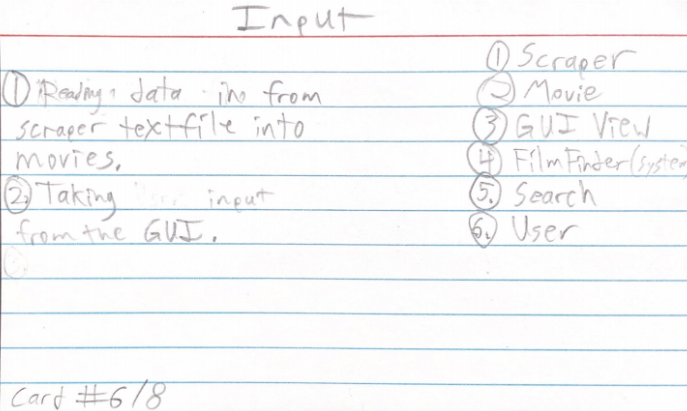


Figure 14 - Input Notecard

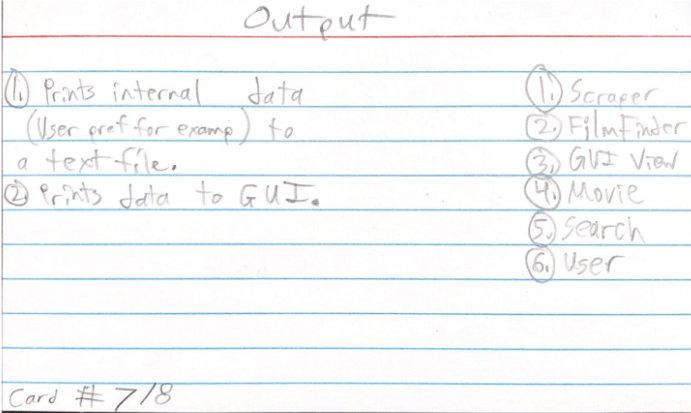


Figure 15 - Output Notecard

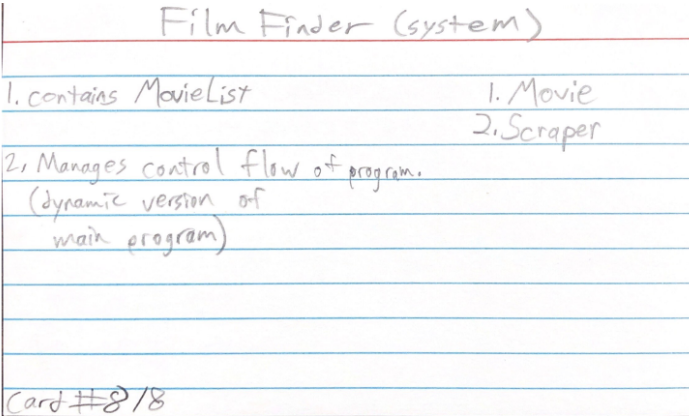


Figure 16 - FilmFinder Notecard