**Design**

**Advanced Software Engineering – COMS W4156**

**Team Name:** PurplePoster

**Team Members:** Abhas Bodas (ab3599@columbia.edu)

Jared Pochtar (jrp2181@columbia.edu)

Maria Taku (mat2185@columbia.edu)

Suhan Canaran (sc3055@columbia.edu)

**Date:** October 29, 2012

**1. Class Diagram**

Image 1 shows the UML Class Diagram for the PurplePoster project. Since this project is being done using an Agile-type, iterative software development life cycle, we decided to model only the more core features of the system in this class diagram. In other words, this class diagram represents what is needed in order to make a fully functioning, first iteration of our project – something which could be given to customers, were this a real-life situation. However, this class diagram does not model any of the non-essentials features of the system (e.g., search abilities, Facebook integration). The assumption is that such features could either be added in later iterations of the project, or could even be added in the first iteration if time permits.

The class diagram illustrates 7 main classes: PurplePoster, AliasToMovie, Movie, Comment, Homepage, FilmingPage, and User.

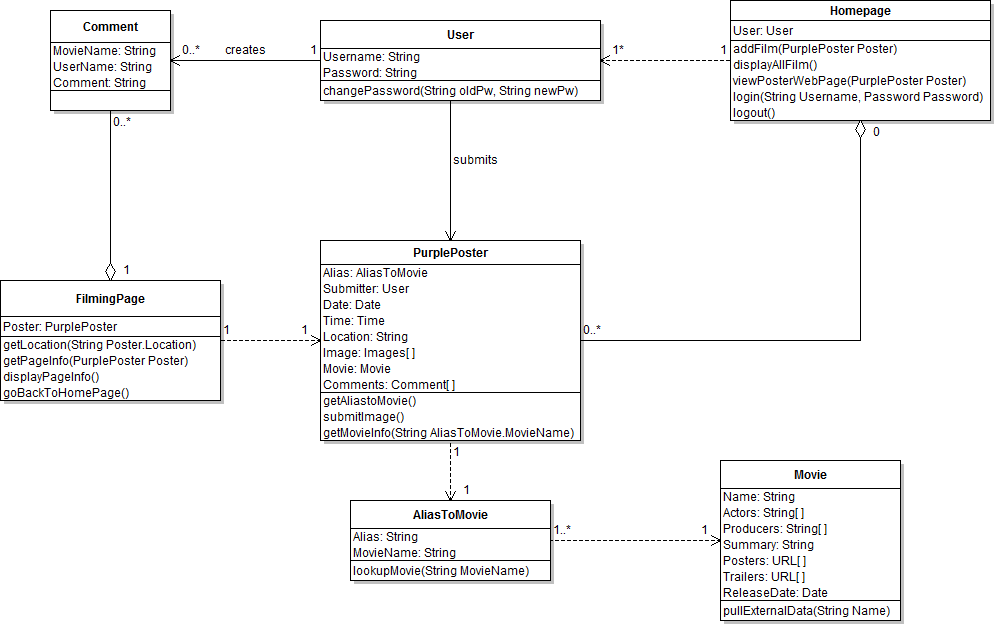


Image 1: Purple Poster Class Diagram

The PurplePoster class can be used to create objects that correspond to a particular purple poster; in other words, these objects correspond to a specific filming location for a movie. As illustrated in the class diagram, attributes related to the specific filming location would thus include data like the date, time, filming location, images, and comments. The last-mentioned attributes, the comments are stored by the PurplePoster class using a list of objects of the Comment class. Details of the Comment class are covered later. There is also an AliasToMovie attribute which is discussed in more detail in the next paragraph. This class supports methods such as submitImage (allows a user to submit an image of the filming location) addComments (allows a user to add comments about the filming location), and getMovieInfo (looks up the movie related to this purple poster/filming location).

The next class is the AliasToMovie class. As shown in the Image 1, since a single movie can have several filming locations (can have several PurplePoster objects) we decided to create a separate Movie class. We then created the AliasToMovie class whose purpose is to provide a mapping between PurplePosters objects and their associated movies. The attributes of the AliasToMovie class are thus strings identifying (1) an alias and (2) its associated movie. Since each PurplePoster is associated with exactly one movie, there is a 1:1 relationship between the PurplePoster and AliasToMovie classes.

The Movie class can be used to create objects for particular movies. Since one movie can be associated with one or more PurplePoster objects, there is a 1:many relationship between the Movie class and the AliasToMovie class. The Movie class can store attributes related to the movie itself, such as actors and directors, etc. The Movie class supports methods such as pullExternalData (pulls information related to the movie from an external source, such as the Rotten Tomatoes API).

The Homepage class is used to render the homepage of the website. The Homepage will contain a list of all PurplePosters (to be exact, it contains a list of links to these PurplePosters’ individual webpages). Thus, we decided to model the Homepage class as an aggregate of PurplePosters objects. Since the Homepage may have zero PurplePoster (if the site is completely new and has never been used before) or more, the Homepage to PurplePoster relationship has been modeled as a zero:many relationship. The Homepage has an attribute called “User” which corresponds to a user currently logged into the PurplePoster system. The Homepage class supports methods such as addFilm (used to add a new PurplePoster to the system), displayAllFilms (displays a listing of links to all PurplePoster’s webpages on the homepage), and viewPosterWebPage (allows the user to see a webpage for a particular purple poster), and login/logout (allows a users to login/logout, respectively).

The FilmingPage class is used to render the specific webpages for each purple poster dynamically, based on the desired PurplePoster. In other words, a FilmingPage object is created for each PurplePoster object, and this FilmingPage object is responsible for creating a webpage displaying the information of that PurplePoster object. Since each FilmingPage is associated with exactly one PurplePoster, there is a 1:1 relationship between the FilmingPage class and the PurplePoster class. The FilmingPage class includes as attributes a PurplePoster object--this object will be used by the filming page to display information related to a particular film location (e.g., to a particular PurplePoster object) as well to look up the film location’s corresponding movie (e.g., via the getPageInfo(PurplePoster poster) method). The FilmingPage class supports objects such as getPageInfo (gets the purple poster and movie information related to that film’s webpage), getLocation (displays a map showing where the filming location is geographically located), displayPageInfo (displays the information retrieved via the getPageInfo function), and goBacktoHomePage (renders the HomePage).

The Comment class is used to store a comment about a particular PurplePoster. It stores the comment text, the name of the movie, and the name user who made the comment. Each PurplePoster object stores a list of Comments, comprised of objects of the Comment class.

**2. Sequence Diagrams**

Image 2.1 shows the Sequence Diagram for submitting an image of a particular filming location (e.g., of a particular PurplePoster object) to the system. In other words, this sequence diagram can correspond to the requirement of “Submit Pictures.” The sequence diagram illustrates two objects: a FilmingPage object called “FP” and a PurplePoster object called “poster.” The process begins with FP calling the submitImage function. Specifically, since submitImage is actually a function of the PurplePoster class, the exact function called is FP.poster.submitImage (where, as illustrated in the class diagram, “poster” is the PurplePoster attribute of FP).

In response, poster attempts to add the submitted image to Image[] (e.g., to its array of images). FP will then refresh the PurplePoster’s webpage as necessary. For example, if the image was submitted successfully, FP can then refresh the webpage to display this newly-submitted image. If an error occurs and/or poster is unable to successfully add the submitted image, then FP may not need to refresh the webpage.

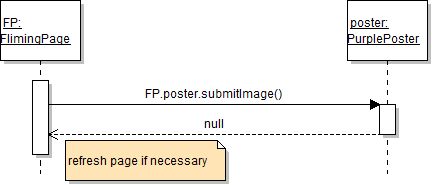


Image 2.1: Sequence Diagram illustrating how to submit an image of a PurplePoster

Image 2.2 shows the Sequence Diagram for retrieving the Movie object that is related to a particular PurplePoster. As shown in the diagram, there are three objects: a PurplePoster object called “poster,” an AliasToMovie object named “alias,” and a Movie object named “movie.”

To execute the request, poster first calls the function getMovieInfo(). In response, alias then calls the function lookupMovie(). This then causes movie to find the movie related to poster. Movie then returns the movie object (e.g., or a reference to this movie object) to alias. Alias then forwards the movie object back to poster.

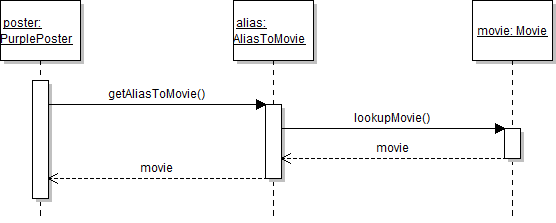


Image 2.2: Sequence Diagram illustrating how to retrieve a movie object.

Image 2.3 shows a sequence diagram illustrating how to view the webpage of a particular purple poster’s webpage. In other words, this sequence diagram corresponds to the requirement of “Viewing Information on Film Locations.”

Since the links to all purple posters’ webpages are located on the homepage, the sequence diagram starts with a Homepage object called “session.” From session, the viewPosterWebPage function is called, which is used to render/direct the user to the purple poster’s webpage. In order to display the purple poster’s webpage, the FilmingPage object, “FP,” calls the getPageInfo() in order to retrieve the requisite PurplePoster and Movie objects. The interaction between the PurplePoster object, AliasToMovie object, and Movie object then operate in the same manner as the Sequence Diagram of Image 2.2, resulting in the PurplePoster object retrieving the necessary Movie object. The PurplePoster object then passes itself (e.g., or references to itself) back to FP. Since the PurplePoster object includes as attributes the purple poster information as well the movie object, FP can then use this information to build the purple poster’s webpage.

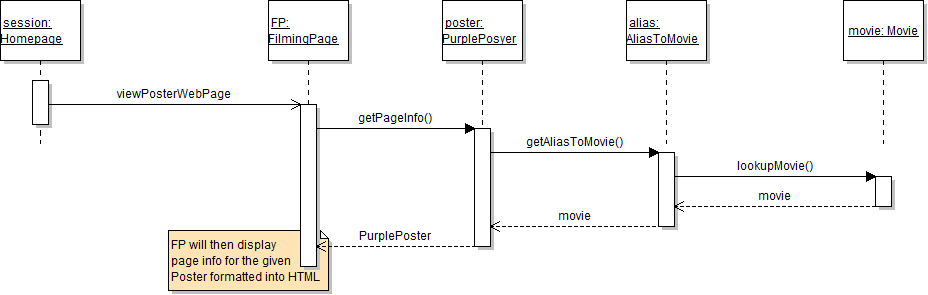


Image 2.3: Sequence Diagram illustrating how to view a purple poster’s webpage

**3. State Diagrams**

The following section contains state diagrams for a medium and a high priority task for our project. The first task illustrated in Image 3.1 is a medium priority task allowing a user to view PurplePosters and Movie details that have been reported in the past. After login, the user is greeted with an idle home page. In the most basic case (e.g., our first iteration of the project), the Homepage displays a list of previously-reported purple posters; the user can then manually read through and choose a desired past purple poster.  In later iterations of this project, the home page supports a search option, for looking up PurplePosters based on the movie name or other details. Whether the user manually reads through the list of purple posters or uses a search function, the result leads to the FilmingPage about the movie, which fetches details from our database that contains data retrieved from RottenTomatoes. The process ends with the user logging out of the website.

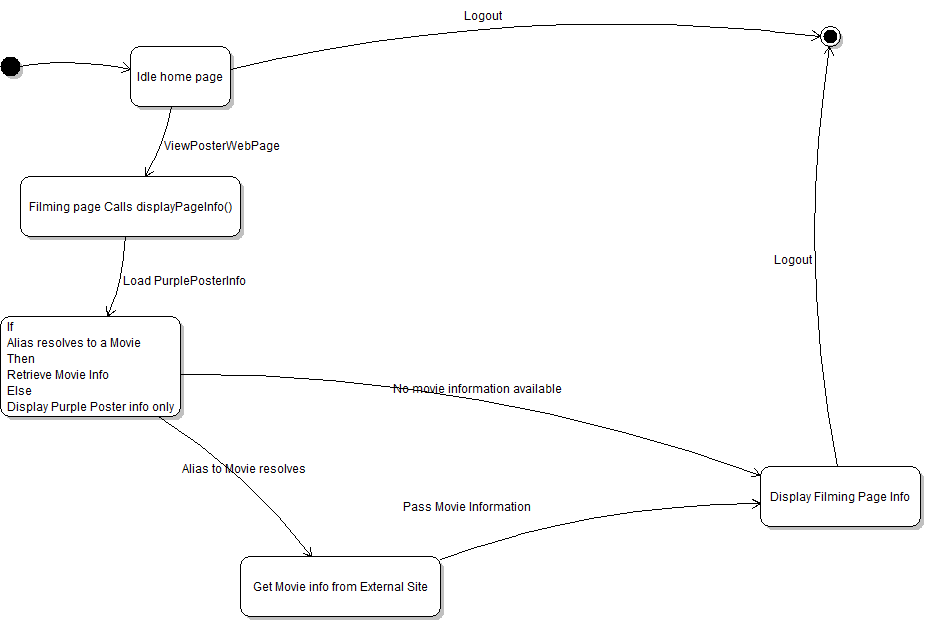


Image 3.1 State Diagram illustrating the process of searching for PurplePosters

The second task illustrated in Image 3.2 is the high priority task allowing a user to report filming of a movie. After login, the home page has an option to report filming along with a list of previously-reported PurplePosters.  On proceeding with this  option, the user fills out details in a form, such as the alias name, location, movie name, actors etc. On  submission of these details, an alias-to-movie lookup is performed and movie details (if available) are fetched from the RottenTomatoes API and stored on our database along with the PurplePoster details.  The home page with the updated list of PurplePosters is displayed. The process ends with the user logging out of the website.

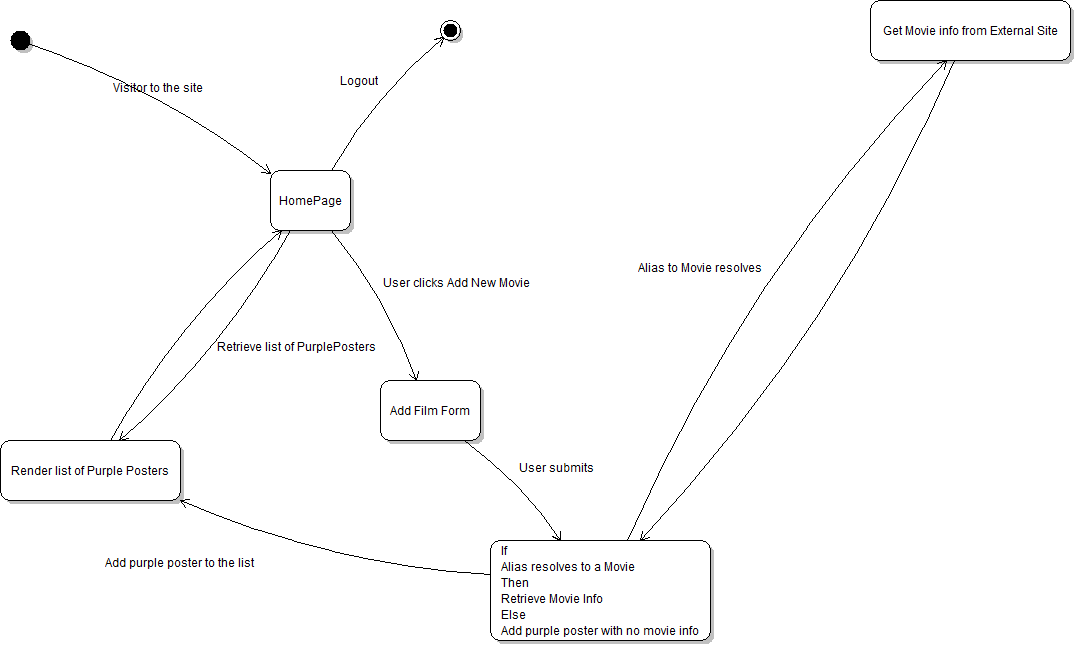


Image 3.2 State Diagram illustrating the process of reporting a new PurplePoster

**4. Other Diagrams: UML Use Case Diagrams**

For our two “other” diagrams, we decided to create UML Use Case Diagrams. This choice was made since it provided a convenient, high level way of displaying the general functionality of our system. As such, it gives the reader a much better idea of the overall functions and benefits of the system, as opposed to the more focused Sequence Diagrams and State Diagrams, etc.

Image 4.1 shows the Use Case diagram where the main actor is the actual user of the system himself (e.g., someone who is using the Purple Posters Website). The diagram then illustrates the various system services that are provided to the user. For example, Image 4.1 illustrates that the user can perform various actions such as viewing the website’s homepage, viewing the webpage of a particular purple poster, add new information about a purple poster, and/or submit images of a filming location (e.g., of a purple poster).

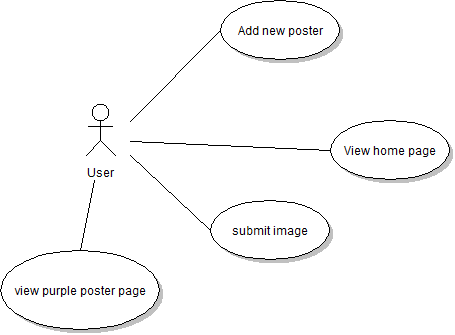


Image 4.1: A Use Diagram illustrating the services provided to a user of the system.

Image 4.2 shows a Use Case illustrating the high level interaction between the purple poster system and an external system providing ancillary information about movies (e.g., such as the Rotten Tomatoes API). In this case, the two actors in the Use Case diagram are a PurplePoster Webpage (e.g., to be more precise, the Movie object associated with a FilmingPage object, where the Movie object could call a function such as PullExternalData() to communicate with the Rotten Tomatoes API) and the external system (e.g., the Rotten Tomatoes API). These two actors then interact via a “pull data” relationship in order to allow the PurplePoster webpage to gather ancillary information about its movie.

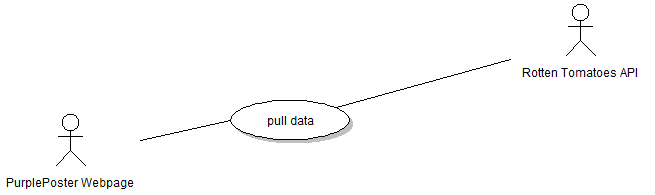


Image 4.2: Use Case Diagram illustrating the interaction between a purple poster   
webpage and an external system.

**5. Controversies**

None.