SE31520 – Enhancing the CS-Alumni Application

Introduction

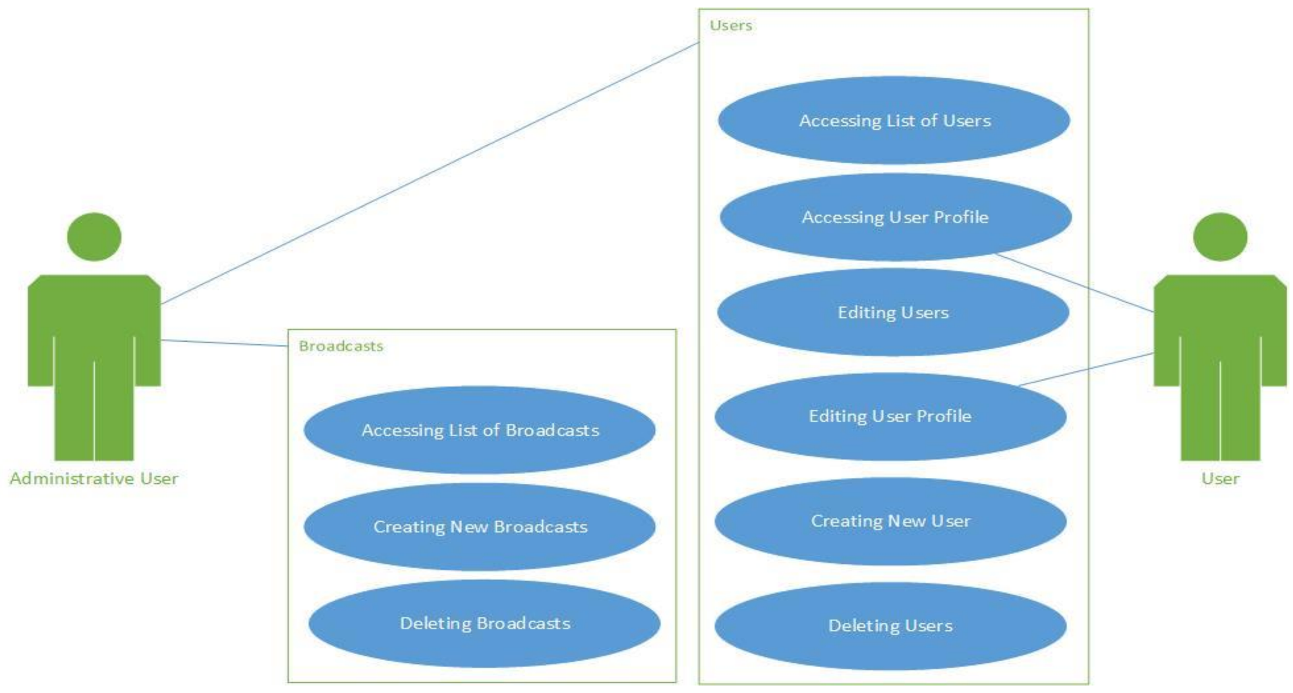
The CS-Alumni application, the application I created an enhancement for, was designed to be used by Computer Science Alumni to be able to sign up for news broadcasts and job offers. It was created in Ruby using the Rails framework, and therefore is a web application

The enhancement I chose to make for the CS-Alumni application was a separate desktop-based application that would connect with the web application in a RESTful manner using JSON as the resource notation. I chose to write it in Java using the RESTlet framework and a combination of the GSON and JSON libraries to parse and create correct JSON objects. This application would only be usable by administrative users of the site due to the nature of what a non-administrative user could access and what an administrative user could access.

Within this document I will detail the architecture, testing and evaluation of both the CS-Alumni application, my application, and how they interface together.

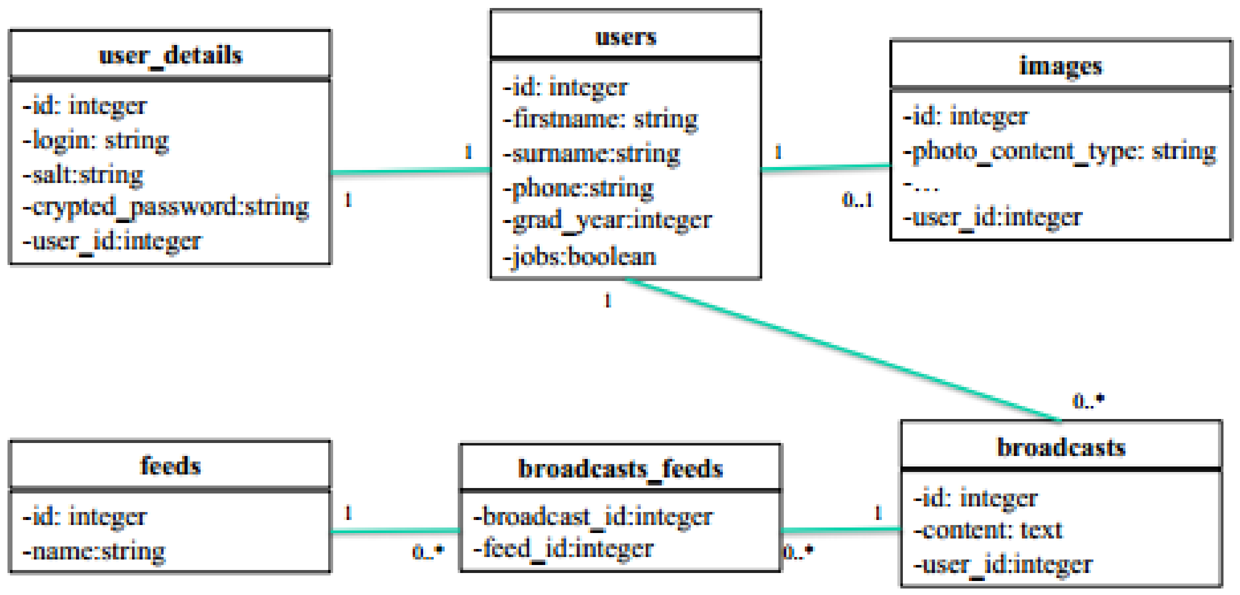
Design and Architecture of Application

There are two main types of users for the CSA Application, these are administrative and non-administrative users. The use- case diagram for these users is as follows:



The extension application is only available to administrative users of the CSA Application as the extent of the functionality they can access is much wider than a non-administrative user and there is, therefore, a business case for developing a separate desktop application for the administrative users.

The CSA application is designed in an MVC manner, which is in line with how the Rails framework is designed to work. The MVC pattern is mainly used for web applications and is a composite design pattern. It has a model, which tends to be a database or some other store for the data the application is using, a view, which is how you will be able to view this data and also manipulate it in some manner, and the controller, which maintains a link between the model and view, handles the functionality for creating displayable data and requests for manipulating the data from the view. In the CS Application, there is a SQLite database that contains this relational database:

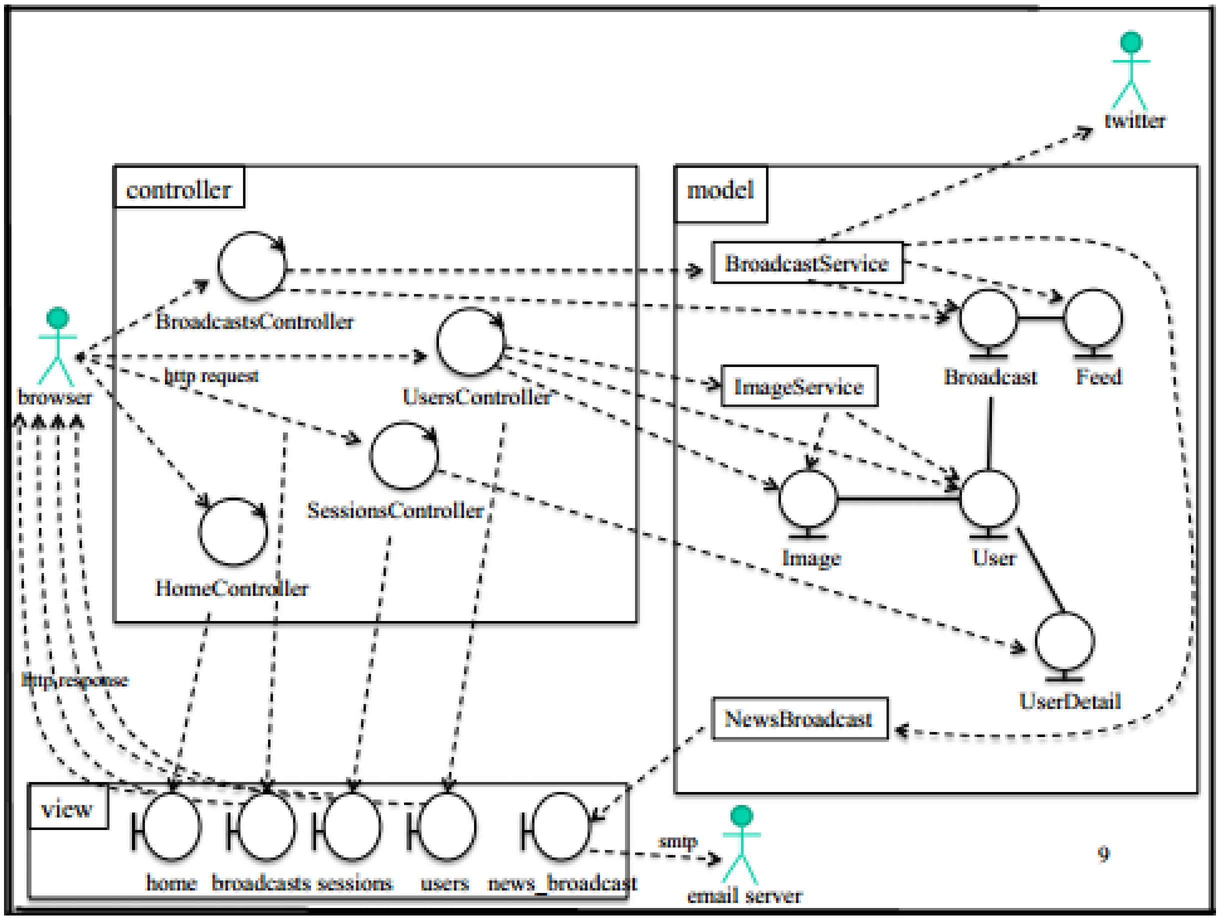


There are also the model classes, such as user and broadcast, which maintain how the data in the database is modelled by the application.

The controllers are separated into user_controller, broadcast_controller, home_controller, session_controller and application_controller. These all handle the various requests between model and view. An example of this would be the user_controller which contains methods that accept requests for data, handle the validation of those requests, and then return the data asked for in a certain format, depending on the type of request.

The views are embedded Ruby files, which contain a mixture of HTML and Ruby. These views relate to each accessible part of the application, and have validation based on what view should be available to which kind of user. For example, on the home page, an administrative user can see the Home, Jobs, Users, Broadcasts and Profile tabs whereas a non-administrative user can only see Home, Jobs and Profile.

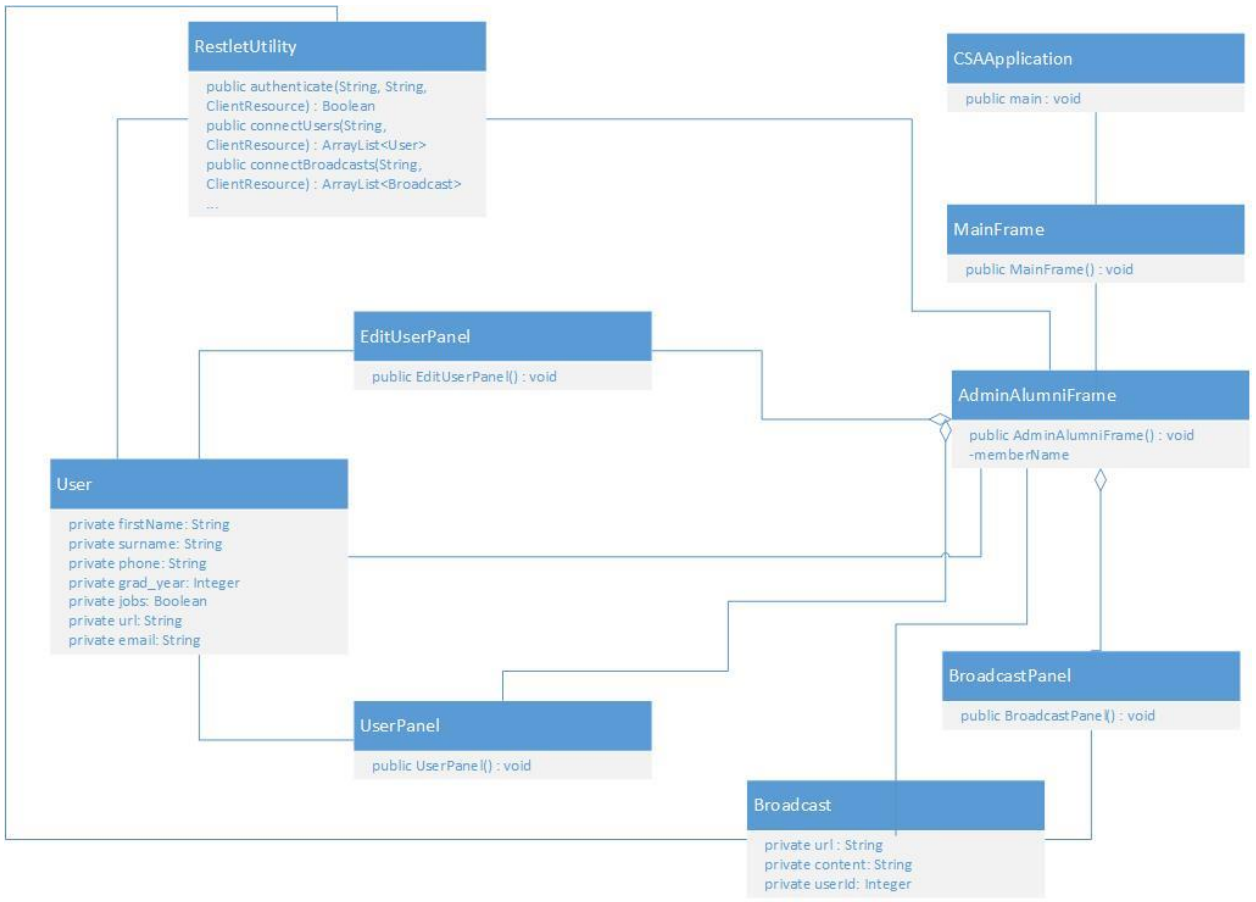
The current class diagram for the CSA Application is as follows, and displays the MVC design:



My extension works as another view for the model and controller. It uses a Swing GUI interface to allow administrative users to access the same functionality on the desktop as they would in browser. I have designed it with a tabbed interface similar to how the browser has a tabbed interface so the experience on desktop is not dissimilar to the experience in browser.

A noticeable difference is the lack of operability by a non-administrative user. I decided upon this for a number of reasons. The main reason was the small amount of functionality a non-administrative user could access which meant that any desktop interface for this kind of user would seem very minimal and unnecessary. An administrative user had access to a lot more functionality that could be worthwhile placing in a desktop application. Another reason was using the RESTful manner of accessing data. With how the CSA application had been built, a user id was required to access any of the non-administrative user's content. Within the application this was trivial as it had direct access to the database. Outside of the application, using JSON representation to access resources via a REST API, it became more difficult to access these resources as the user ids were stored in a place where only administrative users could access. Balancing the difficulty of creating a non-administrative user interface in the desktop application against the practical uses for it seemed to suggest that it was best to create a good administrative user interface rather than lose quality there while trying to build a non-administrative user interface.

The class diagram for my application is as follows, though it will not include the model and controller as these as the same as the previous class diagram, though they do not operate with the browser, but with my extension application:


RestletUtility
public authenticate(String, String, ClientResource) : Boolean
public connectUsers(String, ClientResource) : ArrayList<User>
public connectBroadcasts(String, ClientResource) : ArrayList<Broadcast>
...
CSAApplication
public main : void
MainFrame
public MainFrame() : void
EditUserPanel
public EditUserPanel() : void
AdminAlumniFrame
public AdminAlumniFrame() : void
-memberName
User
private firstName: String
private surname: String
private phone: String
private grad_year: Integer
private jobs: Boolean
private url: String
private email: String
BroadcastPanel
public BroadcastPanel() : void
UserPanel
public UserPanel() : void
Broadcast
private url : String
private content: String
private userId: Integer

As you can see for the class diagram, I chose to separate my REST web connections into a separate utility class. I chose to do this so that if any extensions need to be made in the future, perhaps the addition of the Jobs tab and the operations that come with it or creating the non-administrative user interface, the REST connections are in a place that can be accessed regardless of the GUI class being used rather than specific to the GUI class. I also chose to create data models for the users and broadcasts as these would be easier to maintain and if the Jobs tab were to be added, a Jobs class would be created to maintain the jobs. The separate panels for creating users, editing users and broadcasts were used to ensure easy addition and editing of these models. It meant that instead of adding directly to a table, as you can creating editable tables in Swing, an administrative user could use a user-friendly form instead, and then this data could easily get passed into a JSON object and added to the database using either a POST or PUT request. I have named the AdminAlumniFrame as such in case of the future addition of a UserAlumniFrame.

TEST STRATEGY

For both my application and the CS-Alumni application, unit and functional testing was required. For the CS-Alumni testing, the Rails framework for testing was used and were completed by me after some tweaks were required from the generated Rails tests. For Java application, I used JUnit for the unit testing and will be using a manual test table for the functional testing. Due to its GUI nature, JUnit is a poor framework for this kind of testing, and adding and learning another testing framework seemed the longer of the options. Therefore the test table for functional testing and the screencast will exist as the functional testing for my GUI application.

Unit Testing

Within the CSA application, the classes that needed unit testing were:

User

The test for these were as follows:

Test Number	Test	Input	Output
1	Check to see that a user with empty attributes would create errors based on those empty attributes.	An instance of the User class with all empty attributes.	A list of errors that denoted that the User class was empty.
2	Check to see that the graduate year attribute of a User was between 1970 and the current year.	Changing the user's graduate year attribute to the bounding years (1970 and this year) and bounding erroneous years (1969 and next year)	Assertions that each case was either valid or invalid.
3	Check that the User's email was formatted correctly.	Inserting a valid email and checking that no errors occurred.	Assertion that the user is valid, and therefore the email is valid.

Within my extension application, the classes that needed testing were:

Broadcast

User

RestletUtility

The tests for these were as follows:

Test Number	Test	Input	Output
4	Ensure URL attribute could be set and returned properly within the Broadcast class.	Creating a new Broadcast class and then setting the URL.	Assertion that using the getURL method is equal to the URL entered.
5	Ensure Content attribute could be set and returned properly	Creating a new Broadcast class and then setting the Content.	Assertion that using the getContent method is equal to the content entered.

	within the Broadcast class.		
6	Ensure User ID attribute could be set and returned properly within the Broadcast class.	Creating a new Broadcast class and then setting the User ID.	Assertion that using the getUserId method is equal to the user id entered.
7	Ensure URL attribute could be set and returned properly within the User class.	Creating a new User class and then setting the URL.	Assertion that using the getURL method is equal to the URL entered.
8	Ensure Surname attribute could be set and returned properly within the User class.	Creating a new User class and then setting the Surname.	Assertion that using the getSurname method is equal to the surname entered.
9	Ensure FirstName attribute could be set and returned properly within the User class.	Creating a new User class and then setting the FirstName.	Assertion that using the getFirstName method is equal to the firstname entered.
10	Ensure Email attribute could be set and returned properly within the User class.	Creating a new User class and then setting the Email.	Assertion that using the geEmail method is equal to the email entered.
11	Ensure Jobs attribute could be set and returned properly within the User class.	Creating a new User class and then setting the Jobs.	Assertion that using the getJobs method is equal to the jobs entered.
12	Ensure Grad_Year attribute could be set and returned properly within the User class.	Creating a new User class and then setting the Grad_Year.	Assertion that using the getGrad_Year method is equal to the year entered.
13	Ensure Phone attribute could be set and returned properly within the User class.	Creating a new User class and then setting the Phone.	Assertion that using the getPhone method is equal to the phone number entered.
14	Ensure that using a wrong password/username combination results in an error.	Enter invalid password and username. Requires connection to the web application to be available.	An assertion that the Boolean returned will be false.
15	Ensure that using a correct password/username combination results in no errors.	Enter valid password and username. Requires connection to the web application to be available.	An assertion that the Boolean returned will be true.

16	Ensure that connecting to the broadcasts page, with correct authorisation, returns a non-empty JSON object.	Connect to broadcast page and parse the JSON object into an array.	An assertion that the array created is not empty.
17	Ensure that connecting to the users' page, with correct authorisation, returns a non-empty JSON object.	Connect to users' page and parse the JSON object into an array.	An assertion that the array creates in not empty.
18	Ensure that when creating a new User, it is successfully created and added to the database.	Connect to users' page, post a JSON object that contains all the user's information, and collect the users and check that it contains the new user.	An assertion that the array created contains the new user.
19	Ensure that when creating a new Broadcast, it is successfully created and added to the database.	Connect to users' page, post a JSON object that contains all the user's information, and collect the users and check that it contains the new Broadcast.	An assertion that the array created contains the new Broadcast. (Also needs visual confirmation that it has also been posted to the correct feeds)
20	Ensure that when editing a User, it is successfully patched within the database.	Connect the user's page, using their user id, put a JSON object containing the changed information, collect the users and check that the user's entry contains the new information.	An assertion that the array created contains the user, and that user contains the correct information.

FUNCTIONAL TESTING

Within the CSA Application, the classes that need functional testing were:

Broadcast_Controller

User_Controller

These controlled the main functionality for the application and were therefore suited for the functional testing.

The test for these were as follows:

Test Number	Test	Input	Output
21	Should get the index page for the broadcasts.	Attempt to get index page.	Assert that response is success and broadcasts are not nil.
22	Should get the new broadcast page.	Attempt to get the new broadcast page.	Assert that response is a success.
23	Should create a new broadcast.	Attempt to create a new broadcast.	Assert that the broadcast count has increased by one and you are redirected to the broadcast index page.
24	Should show a broadcast.	Attempt to show a singular broadcast and its details.	Assert that response was successful.
25	Should destroy a broadcast.	Attempt to destroy a broadcast.	Assert that broadcast count has decreased by one and that you are redirected to the broadcasts.
26	Should get the index page for the users.	Attempt to get the index page.	Assert that the response is a success and broadcasts are not nil.
27	Should get the new users page.	Attempt to get the new users page.	Assert that response was a success.
28	Should create a new user.	Attempt to create a new user.	Assert that the user count has increased by one and that you are redirected to the new user’s page.
29	Should show a user.	Attempt to show a singular user and its details.	Assert that response was successful.
30	Should get the edit user page.	Attempt to get the edit user page.	Assert that response was successful
31	Should update a singular user’s details.	Attempt to change a user’s details.	Assert that change was successful and that you are redirected to the user’s page.
32	Should destroy a user	Attempt to destroy a user	Assert that user count has decreased by one and that you are directed to the user list.

Within my extension application, the classes that needed function testing were:

MainFrame

AdminAlumniFrame

EditUserPanel

UserPanel

BroadcastPanel

These were the GUI classes for my application and were therefore manually tested to ensure that they were functioning correctly and functioning correctly with regards to data classes.

These tests were as follows:

Test Number	Test	Input	Output
33	Test that a user can login from the MainFrame	Username and Password.	Redirected to the AdminAlumniFrame.
34	Test whether the table of users has been filled.	JSON object obtained from GET requests to /users path.	Table is filled correctly with the provided data.
35	Test whether the table of broadcasts has been filled.	JSON object obtained from GET request to /broadcasts path.	Table is filled correctly with the provided data.
36	Test whether profile contains correct user details.	JSON object obtained from GET request to the /users/userid path.	List is filled correctly with the provided data.
37	Ability to create new user.	Fill out new user form, found by pressing new user button, and submitting the form as a JSON object.	Table contains new user.
38	Ability to edit a user.	Select user from table, fill out edit user form, found by pressing edit user button, and submitting the form as a JSON object.	Table contains updated user information.
39	Ability to delete user.	Select user from table, press the delete button.	Table no longer contains user.
40	Ensure inability to delete current user.	Select current user from table, press the delete button.	Message appears saying current user cannot be deleted.
41	Ability to create new broadcast.	Fill out new broadcast form, found by pressing the new	Table contains new broadcast. Extension test is to check if

		broadcast button, and submitting the form as a JSON object.	broadcast appears on twitter feed when selected.
42	Ability to delete broadcast.	Select broadcast from table and press delete button.	Table no longer contains broadcasts.

CRITICAL EVALUATION

DESIGN CRITIQUE

After hearing a lot about agile methodologies in the module of the same name, I decided to try to work in an agile manner for this assignment, looking at trying an XP style of coding. I started off with a basic design, knowing what kinds of GUI classes I would need, what classes I would need for the models and how I would utilize the Restlet framework.

Before I began coding the application, I wrote a small program to test the Restlet functionality and then used this to build the methods for obtaining Users and Broadcasts. I wrote the login functionality first, as this was the first step I needed before anything else. After that I worked tab by tab, getting the basic functionality done. I then began refactoring, moving the code into a better structure, adding the RestletUtility class to contain the methods when I noticed repetition in my code.

The downside to this type of agile working is that it leaves you with very little in terms of design, the design is meant to grow itself over time and through refactoring. However this means when it comes to documentation, it becomes difficult to discuss it as a solid design. I think for this kind of assignment, if I did it again, I would choose to do planned development and work with a design planned at the start and a testing strategy planned at the start as well.

DIFFICULTIES/OMISSIONS

As stated multiple times before, I chose not to implement a non-administrative user application. A lot of this was to do with how the CSA application wants you to interface with it. To get hold of any resources that the non-administrative users are able to access requires the user id and yet there is no way of requesting the user id. This would have required altering the CSA application in some manner, changing either authorizations to get hold of the id, creating a new path that would return the user id etc. I decided that this was beyond the scope of the enhancement and there isn't enough functionality that a non-administrative user can take advantage of, so I chose to focus on administrative applications.

I have only been able to get the application working if the CSA application has SSL turned off. This is because after logging into the http://localhost:3000, it redirects to https://localhost:3001 and the authentication doesn't hold. I found this difficult to get around even with the authentication that Restlet has, which made logging in a lot easier and didn't require me to set up a session, merely maintain the same ClientResource. I think given a bit more time and with a bit more understanding of the CSA Application and the Restlet framework, I may have been able to come up with a solution. I think another solution may have been to ensure that it starts on http://localhost:3001 and stays with that URL, as this would have meant I only needed to change one variable in my program to ensure it worked with SSL, but again this would have required some understanding of how SSL was being used in the CSA Application, which I felt was outside of the scope of my enhancement.

I had some trouble early on in the assignment with both the sending of the JSON objects and what the application was sending back. The first problem occurred in that I was unsure of the correct parameters to put in the JSON objects being sent. I solved this by looking at the requests and the parameters used when using the web application. This was problematic when it came to the broadcasts and feeds and the JSON for that was slightly more difficult to build at first, as I wasn't sure how to achieve the correct formatting. This was solved through trial and error. The second problem was due to the pagination of the users and broadcasts. As there was no way of knowing how many pages of users there were while using an outside application, I couldn't really use the current method of accessing the user list. I had to add some additional Ruby code to change how it handled JSON requests for both the users and the broadcasts, so that when a GET request was issued to either the /users of /broadcasts, it returned a list of users or broadcasts that was not paginated and therefore complete.

TECHNOLOGIES USED

I used Java as my programming language. This was due to having a large amount of familiarity with it, especially when it came to GUI programming. I was tempted to use C++ but I found Qt was a rather large, sprawling library and setting it up took up a large amount of time that I realized would be better spent using a language I have already done some GUI programming in before.

I used the Netbeans GUI editor to create the GUI and then edited some of the generated code to make it easier for me to use with the Restlet framework and the custom classes I had made. This is probably the best way to create a Swing application. Because of the rather awkward layout managers that Swing has, it becomes very difficult to create a professional looking program just through programming and without an editor. While the generated code can look messy, it creates a program that actually looks decent, rather than something that seems old and unusable.

As mentioned throughout the program, I chose to make use of the REST framework for Java known as Restlet. I chose this framework specifically for its ability to handle authentication and authorisation well, something that the CSA application uses to protect certain data from non-administrative users. It also has a relatively simple way of creating GET, POST, PUT and DELETE requests and has support for using JSON representation. I did not want to user the standard Java library for HTTP requests as I believe that using libraries, and therefore reusing code that has already been written for the required functionality, is a much better way of creating good applications than attempting to re-invent the wheel and starting from scratch each time you choose to create an application.

I also chose to use the Google GSON library to help create JSON objects to pass as a resource and also to parse JSON objects being obtained as a resource. This was another example of wanting to reuse code that already existed than writing a parser and creator from scratch which would have taken more time from building a robust and decent application. I have also had to include the org/json library to interface between the GSON library and Restlet as Restlet only has support for the org/json library, but I found that library unwieldy to use and found that it kept producing JSON that could not be recognized by the CSA Application.

MARK

I believe that my work is deserving of at least a high 2:1, if not a first. This is because I have chosen a more complicated, in my opinion, enhancement and have finished it to a high standard. I have understood what was required and used that understanding to finish my application. I think some functionality could have been added with more time, which is why I hesitate to say a first specifically, but I feel there is definitely enough done to justify a 2:1.