Healthy Eating App

Detailed Design Description

Jun Ho Lee 20230435

Atulan Zaman Deep

Rashna Razdan

Nadeem Ahmad

June 23rd, 2013

Abstract

Table of Contents

List of Figures

List of Tables

1 Introduction

The purpose of this report is to provide detailed design models of the Healthy Eating application software. In particular, the work done in Part 1 of the project was reflected and expanded on. The scope of this report is limited to the software architecture summary, identification of modules and control flow, structural design specification and behavioural design specification. It should be noted that names of only the core classes are included. The key data structures and algorithms are mentioned for only the notable classes. Furthermore the report focuses on the implementation and reasoning behind the usage of only the most prominent design patterns and identification of the essential modules.

2 Software Architecture Summary

The software architectural style was changed from the repository style (as identified in Part 1 of the Project) to Model-View-Presenter. The reason behind this change was due to the fact that the team decided to use the Google Web Toolkit (GWT), a Java software development framework. Using this framework made it easier to focus on implementing design patterns and optimizing the Healthy Eating application. Furthermore, the framework ensured the presence of cross browser compatibility and other non-functional 'nice to have' features. The following text will provide brief overview of the newly adapted software architecture. The diagram below visualizes this new architectural style:

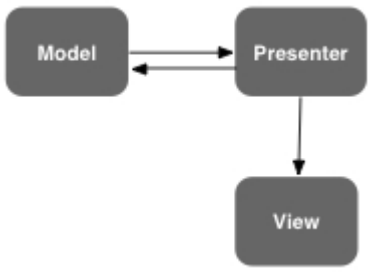

Figure 1 Diagram Showing the MVP architectural style

The Model, View and Present can each be thought of as different subsystems that can be represented as separate packages. In the case of the Healthy Eating application the Model would consist of the Database Access Object where all queries to the database would be made. The View would consist of the different pages that comprise the application. The Presenter would house all the application's logic including handling when to perform data synchronization and controlling view transitions. The main connectors include remote procedure calls to the server (between Presenter and Model), SQL queries to the database, call back routines and method calls between all three subsystems.

3 Identifying Modules and Control Flow

3.1. Explanation of the Rationale Used in Module Identification and Encapsulation, and Selection of Applicable Design Patterns

Modules were created in adherence to the Model-View-Presenter architectural style. Consequently, notable functionalities within the application had a View and Presenter class. For instance, Login had a LoginView class and a LoginPresenter class. The View class' identification was made possible by the fact that GWT uses widgets in order to develop the application. Each widget is an individual class with certain attributes. In order to maintain cohesion and decrease coupling, widgets that served the same purpose were made part of one custom class. This custom class represented a widget composed of other widgets. It should be noted that this rationale also ensured that encapsulation was achieved seeing as each module contains the necessary resources to perform its main function (i.e. LoginView's main function is to display Login information; this class has all the resources it needs to do so). Furthermore, the Presenter class simply has the logic associated with the corresponding View class. For instance, the LoginPresenter has the logic for user authentication.

Moreover, two structural design patterns, Composite and Proxy, were applied to refine the design of the software. The Composite design pattern was chosen because the application had widgets made up of other widgets. The following UML diagram further highlights the way the design pattern was used in the design:

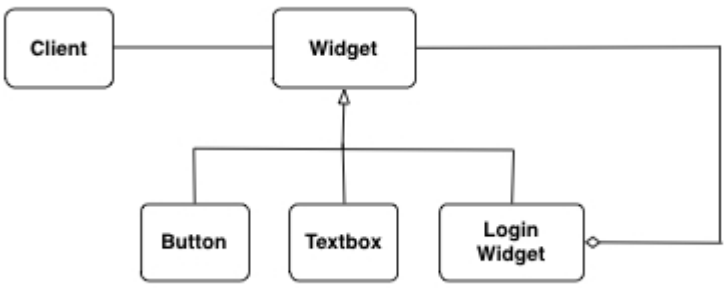

Figure 2 Example of use of the Composite design pattern in the application

In addition, the Proxy design pattern was used. In particular a protection proxy was utilized for authentication. Via the proxy design pattern, administrators were allowed unrestricted access whereas normal users were allowed to see only a subset of the Access object. The rationale behind the use of this pattern was to provide different actors with different accesses to the same object. The UML diagram below highlights how the design pattern was used in the application:

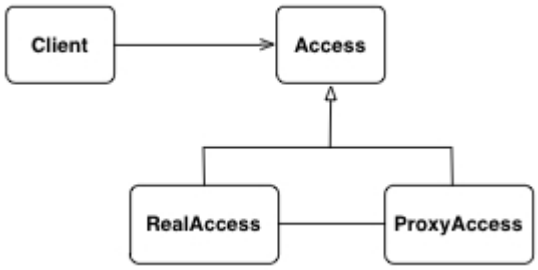

Figure 3 Example of use of the Proxy design pattern in application

Since this is a dynamic web application where any update to the model might affect clients depending on whether the client is using the volatile object, we used the observer pattern. In our case the observer pattern is applicable in the Leaderboards module where any change on the clients side for the friends of the user would cause updates to the Leaderboard of the client side. Therefore, when the clients are on the Leaderboard page, act as subscribers to the subject models, which are the calorie data for the friends of the user. This means that for example if a logged in user has 5 friends using the application, that means the user client is subscribed with the user data of the friends which act as subjects for the observer model. We use the "push-update notification" implementation of the observer pattern.

3.2 Overall Control Flow

The overall control flow is centralized. The reason behind this is that the view transition logic is controlled by an onValueChange() method, this provides a centralized way of navigating within the app. For instance the onValueChange() method receives an event of type ValueChangeEvent. The value of this event is then stored inside a string token. Subsequently a series of if-else statements determine what the new view should be, based on what the token value is. In summary, seeing as navigating the whole application is controlled by one method, the control flow design is centralized.

3.3 Detailed Module Description

Since we used the Model View Presenter architecture for our design, all the client windows have a "View" class and "Presenter" class and they collectively form individual modules for the client side. The View classes contain the objects in the user interface, such as buttons, text boxes, and widgets. The Presenter module controls the events that happen due to the action of each object in the View module.

We divided our class diagram into 4 modular subsystems due their functional independence and the

subsystems themselves own individual modules. The description of the important modules and the details of the external APIs used by these modules are organized according to the subsystems. The detailed descriptions of the class diagrams are provided in a later section.

HomeLogin Subsystem (Figure 4)

The HomeLogin subsystem has the modules responsible for handling the login use cases, which is possible in two ways, either using facebook or using a regular login. The *HealthyEatingApp* module contains the two login widgets. The *FBLogin* widget provides the external API with Facebook for logging in using the facebook credentials. The *LoginView* module takes care of registering the users by invoking a separate widget, and also contains the widget for authenticating users not using the Facebook API to login. Both the *Facebook_Login* module and the *LoginWidget* connect with the *Homepage* Module, which contains the widgets, that provides the features for the application.

GraphWidget Subsystem (Figure 5)

The Graph_Widget subsystem is responsible for the feature to display the history of calories consumed by the user. The visualization modules come in three different modules all of which inherit properties from an abstract class name *Visualization.* The *Line_Graph* module visualizes the data in the form of a line graph, and the rendering of the graph is implemented by an external interface provided by the GWT dynamic graph widget. The *Pie_Graph* module visualizes the data in the form of a pie-chart to divide the weekly consumption into slices divided according to a timing scheme of days, weeks or months. It is also implemented by a Dynamic Pie-Chart widget provided by the GWT. The *Bar_Chart* Module implements the visualization as a series of bar charts according to a chosen timing scheme. A Dynamic Bar Graph widget provided by GWT also implements it. All three visualization modules use the super class methods to make queries to the database to populate the data. The OnEventHandler() method for the share button calls the external Facebook API to share the graph on Facebook as a status update.

LogWidget Subsystem (Figure 6)

The *Log Widget* subsystem has two modules: "*Log Widget*" and "*Food Item Browser*". The *Log Widget* module includes the view and presenter classes for viewing the form where users can enter the logs for entering the food information. The *Log Widget* module calls the *Food Item Browser* module when a new log is to be entered. The *Food Item Browser* module uses an external GWT widget called "Cell Browser" which lets the user pick the food item from an available set of categories, which are fetched from the database. The population of the food database into the Cell Browser object is done *Concurrently* when the Log Widget form is loaded so that the user does not have to wait long for

querying the database after the button is clicked to add an entry.

SocialWidget (Figure 7)

The *SocialWidget* subsystem simply has the one module that implements the social interaction part of the application where users can see the shared calorie data of the their friends on Facebook. This implements a dynamic list, which updates dynamically as friends of the user input new logs to update their calorie information on their local clients. This module uses the external API provided by Facebook to fetch data regarding the friends of the user followed by another query to fetch the calorie data of the users.

4 Structural Design Specification

4.1 Class Diagrams

The following diagrams illustrate the relationships between each class, per view of our M-V-P model.

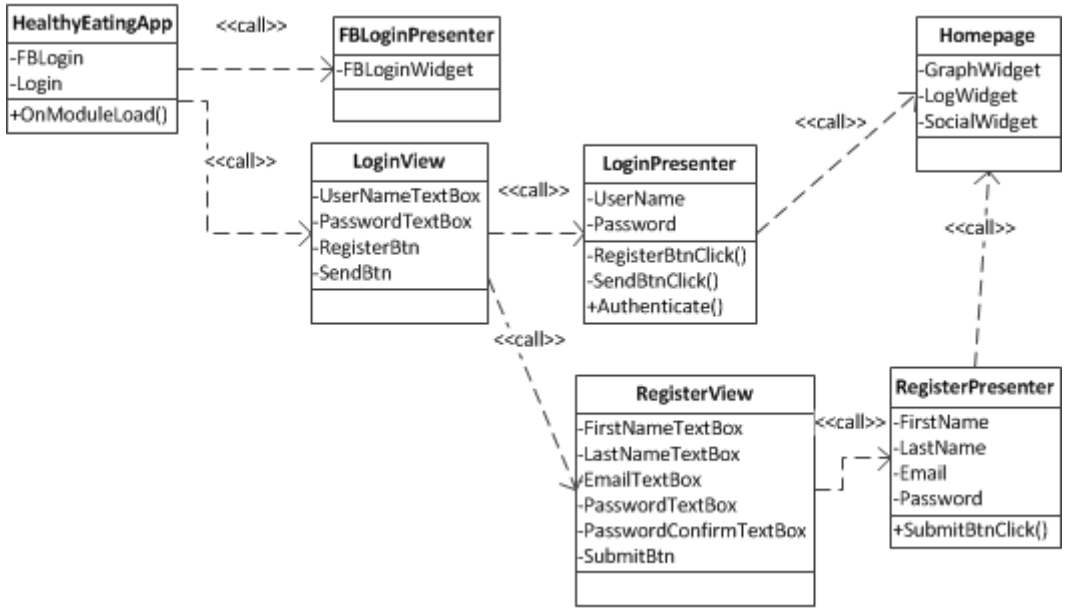

Figure 4 Home Login Class Diagram

Figure 4 illustrates the connections between each class belonging to the Home Login view. The two login subsections, Facebook Login and general Login, each have a view and the associated presenter class. Register is a subclass of Login, and therefore has a separate view and presenter classes.

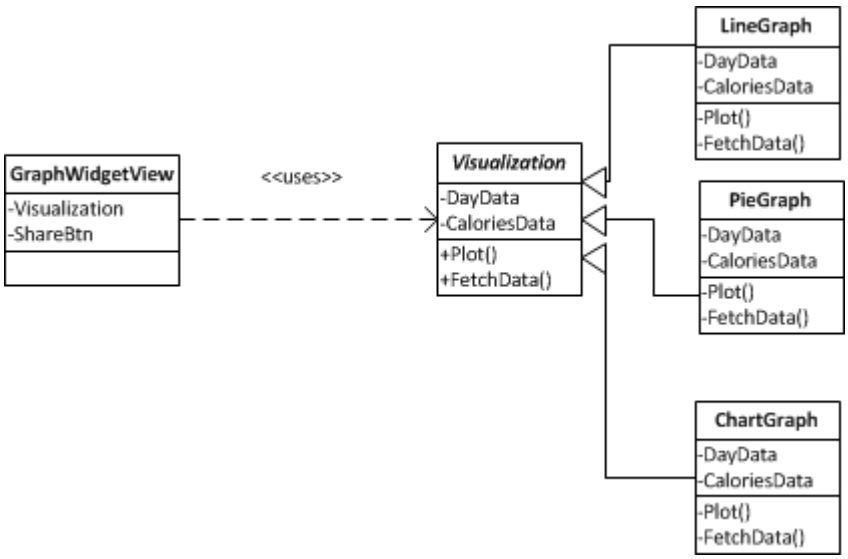

Figure 5 Graph Widget Class Diagram

Figure 5 is the class diagram for the Graph Widget. Here, an interface “Visualization” is used for all three types of graphs that this widget supports.

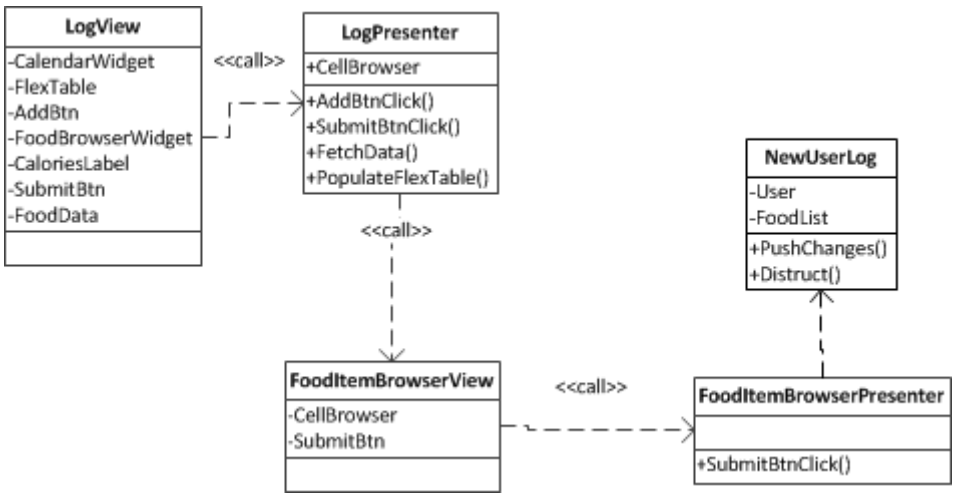

Figure 6 Log View Class Diagram

Figure 6 is the representation of classes associated with the Log View. Upon the constructor call of the Log Presenter class, the flex table, which is enclosed in the cell browser pop up, is immediately populated. When the user clicks on the "Add" button, the cell browser pop up is called, and the flex table is painted on it. From here, user can make the desired selection of the food item, and at the click of the submit button, the pop up closes and New User Log object is initialized to store the changes to the database.

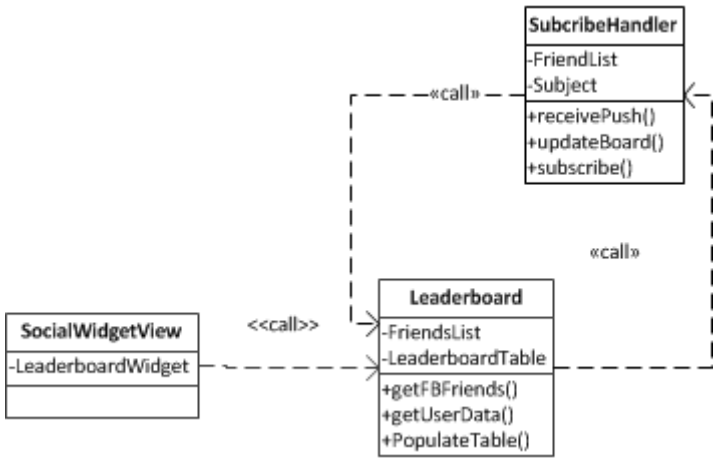

Figure 7 Social Widget Class Diagram

Figure 7 shows the class diagram of the Social Widget. Here, the Leaderboard object employs a handler called "SubscribeHandler" to request and fetch data from the database.

4.2 Description of applied GoF Design Pattern

The GoF patterns applied in this project are the Facade Pattern, Composite Pattern, Proxy Pattern, and Observer Pattern.

The *Facade Pattern* is implied in the implementation of the MVP pattern where every client view implements the Viewer Interface, and every Presenter Class implements the Presenter interface given by the GWT environment. The interfaces for the classes are not shown in the diagram to preserve simplicity because a lot of the classes implement the Presenter interface. Using the interface ensure that the View Classes that call the Presenter Classes implement a standard set of methods that external classes call.

The *Composite Pattern* is used in multiple places in this design. One of the most important application of the composite pattern for the application is the use of widgets to implement the modules that implement the features This gives great object reusability and decoupling and causes objects to have a recursive object ownership structure. An example of the composite is the in the implementation of the *HealthyEatingApp* class in the class diagram of the subsystem *Home_Login.* This class owns the *FBLogin* and *Login* as objects of the class where those objects themselves are objects that implement certain use cases. Again the *Login* class owns the *Register* object, which implements another use case. Another implementation of the composite pattern is in the implementation of the hierarchy in the *GraphWidget* subsystem where all the visualization types inherit properties from the super class *Visualization.*

The *Proxy Pattern* is implemented in the *LogWidget* Subsystem where whenever the user submits new calorie data, the information is stored in a temporary object that store the user id and the new food information. The stored information is pushed to the server after a certain amount of debouncing when the information is written to the database and the all the relevant clients are notified using the Observer pattern. Since changes do not happen very happen to clients this pattern does not significantly affect performance and it preserves the synchronization in a safe way.

As stated earlier, the *Observer Pattern* is used in the *SocialWidget* subsystem to update the social leaderboard when there is new log input from any of the friends of the user. To implement the pattern, the logged in user viewing the social leaderboard is subscribed to the client of the all the friends of the user. Therefore whenever there is a change from PushChanges method is executed in the LogWidget subsystem at the friend's client end, this causes the user's *SubscribeHandler* class to receive the changes that happened in the subject class and push those changes to the local client's views.

4.3 Relevant Data Structures and Algorithms as Pseudocode

Since each subsystem implements a feature of different complexity they each have a different data

structures to implement the features. The data structures and algorithms for the subsystems are described below.

Home Login: The Home Login subsystem is basically a wrapper class that contains the LoginWidget which implements uses the database to authenticate new uses and register new users and uses the Facebook external API to authenticate Facebook users, therefore it does not really use any concrete data structures or algorithm, but simply uses forms and external API to update the database.

GraphWidget: The data structure involved with this subsystem is one that can be passed to the Visualization subclasses to visualize the data in the appropriate way. Since we are using an external API to render the visualizations we are basically fetching the data from the database and transforming it to fit the data structure required by the external interface.

LogWidget: In the LogWidget subsystem the main data structure is contained in the CellBrowser object which is loaded concurrently with the loading of the form so that it can passed to the user quickly when the user want to fill in the food data. Once again, since this is an API provided by GWT the data is prefetched and transformed to fit the data structure required for this API. The other data structures used for this subsystem is the temporary cache object "NewUserLog" that acts as the client side proxy object that stores changes for a fixed amount of time before pushing the changes to the remote object. For this data structure the userID for the logged user is stored as well as all the new or edited food data that the user enters. This is also the required data structure type for the SubscribeHandler class for the SocialWidget subsystem that implements the SubscribeHandler interface.

SocialWidget: The SocialWidget subsystem implements the social leaderboard feature and for that it uses the SubscribeHandler class that uses the FriendList Data Structure that basically uses the same object as the one created by the Proxy client classes.

5 Behavioural Design Specification

5.1 Interaction Diagram

The figures below illustrate interaction diagrams for each subsection of the project.

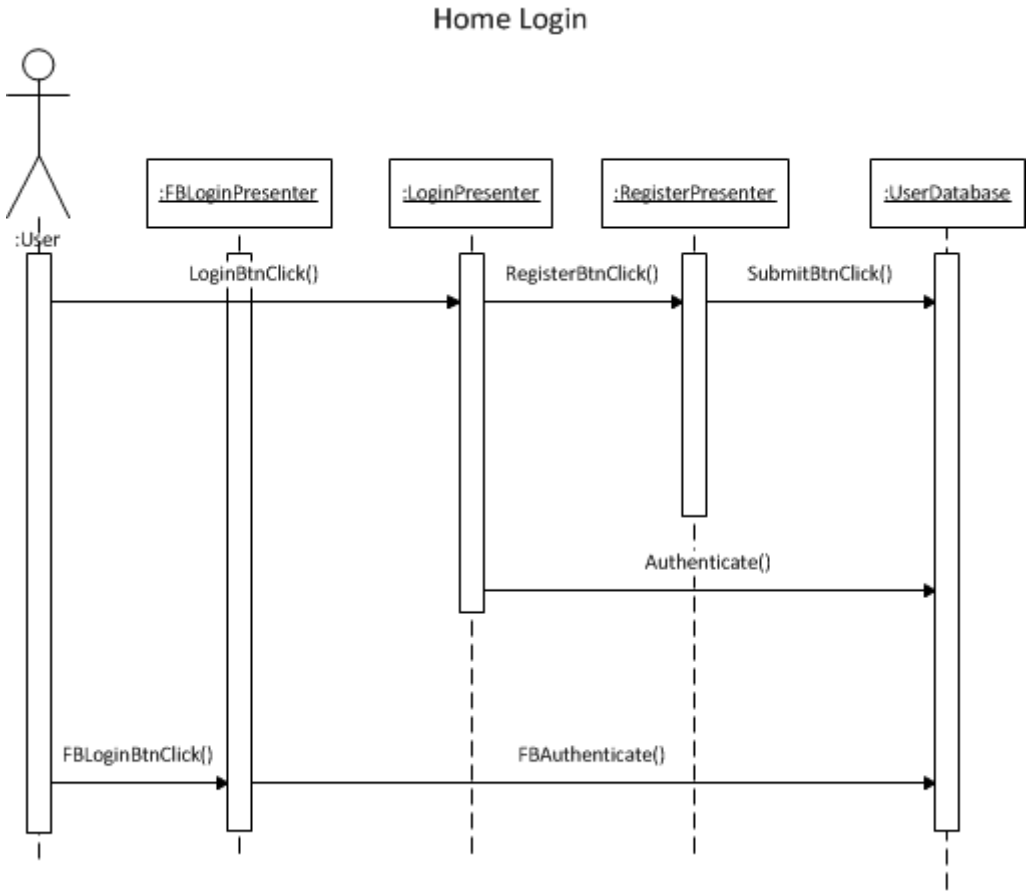

Figure 8 Home Login Interaction Diagram

The interactions between classes for the Home Login view are displayed in Figure 8. User navigates through the classes using the button features on each page. The event handlers are included in the presenter classes for each view.

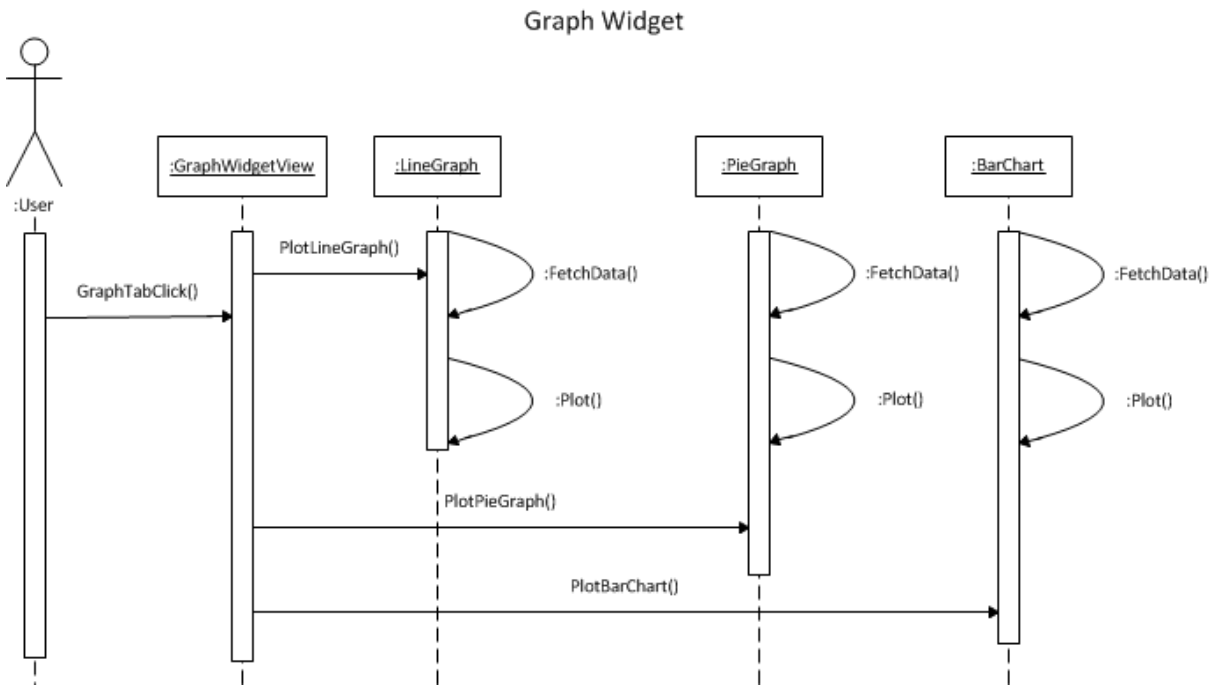

Figure 9 Graph Widget Interaction Diagram

Figure 9 represents the interaction diagram for the graph widget. When the user selects which type of graph to plot, the appropriate graph object (LineGraph, PieGraph, or BarChart) will fetch data from the database and plot the graph.

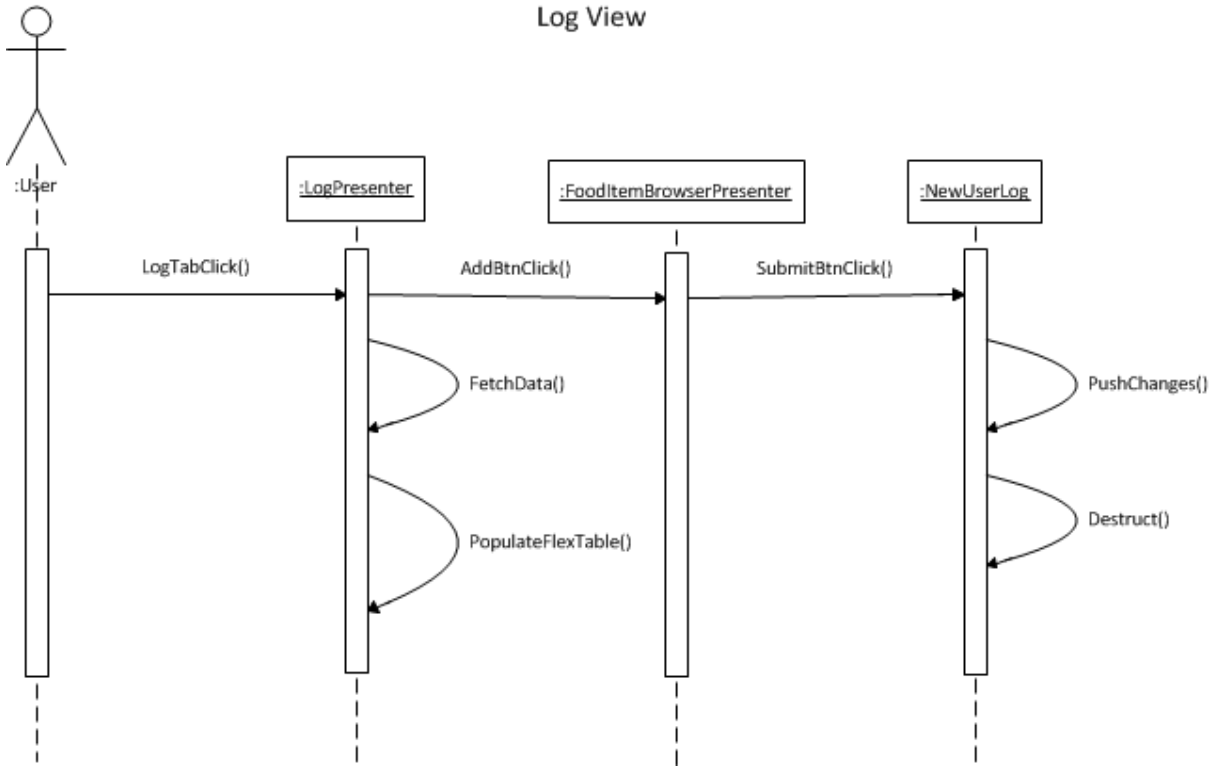

Figure 10 Log View Interaction Diagram

Figure 10 is the log widget interaction diagram. Again, the primary method of interaction between classes is through the usage of the button feature. Within the log view, the progression of the user action is straightforward, culminating in the NewUserLog object pushing the changes made by the user to the database.

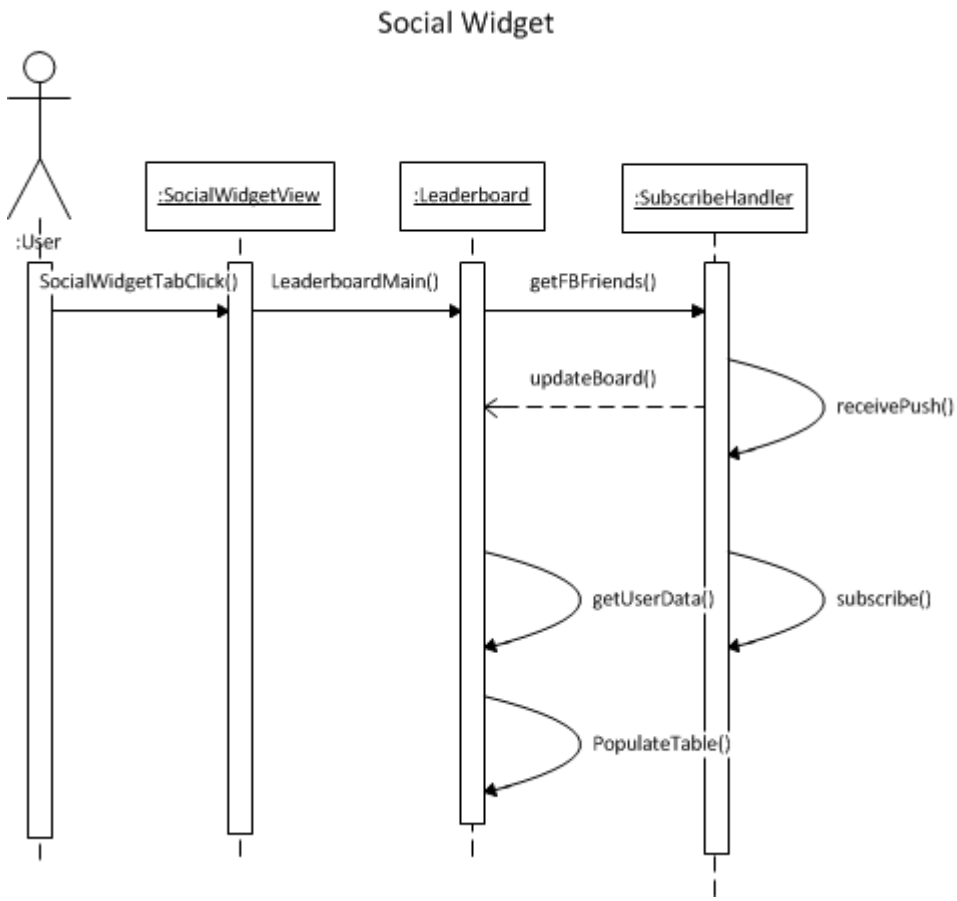

Figure 11 Social Widget Interaction Diagram

The social widget interaction diagram is depicted in Figure 11. Here, the observer pattern is used where the SubscribeHandler gets notified of changes from the database, and updates the Leaderboard accordingly using the updateBoard() function.

5.2 Access Control and Security

In order to address access control there are two different actors that need to be taken into account for this particular application. The difference in the two users is the method that they log in with. One of the ways to log in using the native log in built for the application and the second method is to use the Facebook login. The security requirements of the system are demonstrated in the access matrix below.

Table 1 Security Requirements

	FBUser	**RegularUser**
FBLogin	view()	
Login		view()
Register		<<create>> edit() view()
Homepage	view()	view()
GraphWidget	view()	view()
LogWidget	<<create>> edit() view()	<<create>> edit() view()
SocialWidget	<<create>> edit() view()	<<create>> edit() view()