CS3217 Project – Final Report

NUS Reach

Project Group 7

Lu Xiaodi (U097000R)
Raunak Rajpuria (A0074900L)
Ishaan Singal (A0078549L)
Zhou Biyan (U094837M)

Table of Contents

1. REQUIREMENTS **4**

1.1 OVERVIEW **4**
1.2 REVISED SPECIFICATIONS **5**
1.2.1 EVENT LIST VIEW 5
1.2.2 DYNAMIC MAP OF NUS CAMPUS 5
1.2.3 EVENT FILTERS 5
1.2.4 IVLE INTEGRATION 6
1.2.5 EVENT MANAGEMENT 6
1.2.6 EVENT RECOMMENDATION 7
1.2.7 SYNC WITH CALENDAR 7
1.2.8 FACEBOOK INTEGRATION 7
1.2.9 BUS DIRECTION 8
1.2.10 CREATE PRIVATE EVENTS 8
1.3 USER MANUAL **9**
1.3.1 NUS REACH AT A GLANCE 9
1.3.2 CREATE EVENT 11
1.3.3 BROWSE EVENT DETAIL 12
1.3.4 SHARE EVENT ON FACEBOOK 12
1.3.5 GET NEAREST BUS STOP 13
1.3.6 FIND BUS DIRECTIONS 13
1.4 PERFORMANCE **15**

2. TESTING **16**

2.1 STRATEGY **16**
2.2 TEST RESULTS **17**

3. DESIGN **18**

3.1 OVERVIEW **18**
3.1.1 TOP LEVEL ORGANIZATION 18
3.1.2 USE OF LIBRARIES AND THIRD-PARTY APIS 20
3.1.3 DESIGN PATTERNS 20
3.1.4 INTERESTING DESIGN ISSUES 20
3.2 RUNTIME & MODULE STRUCTURES **22**
3.2.1 CENTRAL CONTROL CLASSES 22
3.2.2 EVENT CLASSES 23
3.2.3 MAP CLASSES 25
3.2.4 USER CLASSES 26
3.2.5 ROUTE PACKAGE 28
3.2.6 IVLE PACKAGE 28
3.2.7 FACEBOOK PACKAGE 28

4. REFLECTION **30**

4.1 EVALUATION **30**
4.2 LESSONS **31**
4.2.1 KNOWN BUGS AND LIMITATIONS 31

APPENDIX 32

MODULE SPECIFICATIONS 32
TEST CASES 55

1. Requirements

1.1 Overview

NUS Reach is aimed to help the NUS community, including students and faculty, to know about what is happenings around the NUS campus. Currently, the only way to view all the events is through IVLE or through the NUS calendar of events, none of which are user friendly. Furthermore, given the large size of the campus, locating certain events is a common problem faced by many NUS students. This application will make this experience a lot more intuitive by displaying all the events on the NUS map, while also helping to enhance social interaction within the NUS community. The application will also prove to be an essential utility for incoming students to get to know the popular events happening across the campus.

Currently, there are two available applications which targets NUS students. None of these display upcoming events as proposed by us:

- NUS map: A campus directory that displays a detailed NUS campus map with the added functionality of locating various building/rooms/facilities on the map.
- Around NUS: Provides detailed NUS maps for the three campuses along with a guide for dining and other facilities/services on campus

Our application provides a dynamic experience customized purely for the user. Apart from the various filters available, the user can enter his interests and the application displays the events related to his/her interest on the map (or alternatively, in a list form). If the user is interested in career talks, the map will display all the career talks happening on that day, along with his other interests. In addition, the user can also create events that can be viewed by everyone in NUS. While doing so, the user has the option of publishing it on IVLE (official event) or making it an informal student event.

The application also finds the shortest path and the bus directions from the user's current location to any location or any event that he wants to go. NUS Reach also gives the user an option of signing into his Facebook account. The user can then share the events he is attending, or the ones he finds interesting, to Facebook.

The application is perfectly suited for an iPad. Apart from its ideal size for a landscape NUS map, it provides great portability to the user. The map navigation on an iPad is a lot more intuitive (with the pinch and pan gestures) as opposed to a desktop computer. This can be extended to other gestures like drawing a rectangle to specify the area of interest and long pressing to open a mini-popup menu. Also, event schedules can be directly synced with the iPad Calendar, which is difficult to achieve from web applications.

1.2 Revised Specifications

The following is the list of the detailed specifications for the behavior of the application:

1.2.1 Event List View

Description and Priority

We will have a list view to show all events, which will supplement the display of events on the map. This is an essential, high priority feature.

Functional Requirements

- The user should be able to view all events in a list form
- The user should be able to apply the filters to this list of events
- The user should be able to view each event from this list itself

1.2.2 Dynamic Map of NUS Campus

Description and Priority

This map would be used as the main interface of our application and will be used to display the campus-wide events. This is an essential, high priority feature.

Functional Requirements

- The map should be limited to the NUS Campus
- The user can zoom in and out of the map. Zooming in should display more details and zooming out should display less details on the map
- The user can pan across the map
- The user can see his/her location on the map
- The user can see events (based on filters applied) on the map

1.2.3 Event Filters

Description and Priority

The application will have the following filters to allow the user to customize the type of events that are displayed on the event-map:

    - By date
    - By category
    - By location

This is an essential, high priority feature.

Functional Requirements

- The user can select any one of the filters. When the user switches between filters, the map view should remain the same (ie zoom level and location)
- By Date: The user can select any date in the future (from the current date). Only one day's events can be displayed at one time.
- By Category: The user can select multiple categories of events (eg welcome teas, career talks)
- By Location: The user can select a location. The map will zoom to the selected location and display all the corresponding events

1.2.4 IVLE Integration

Description and Priority

The application will be integrated with IVLE to support IVLE events. This is an essential, high priority feature.

Functional Requirements

- The users will use their IVLE login name and password to login to the application.
- The application will import all the IVLE events and locate these events on map.
- The events created by the user will be submitted to IVLE events.

1.2.5 Event Management

Description and Priority

The application will support the following two broad types of events:

- Official Events: These events are currently supported by IVLE. The creation of these events requires approval from NUSSU.
- Student Events: informal events that can be create by students at all times.

This is an essential, high priority feature.

Functional Requirements

- The user can create new events, edit created events and delete events created by himself.
- The user can specify the event's type (official or student) and post it to IVLE (requires IVLE integration feature) or Facebook (requires Facebook Integration feature)
- The user can browse a list of events created by him.
- The user can mark events to attend
- The user can browse a list of events he has registered for.
- The user can report student events and choose from a list of reasons if he feels the event is inappropriate

- An event will be removed from the map once the number of reports it receives reaches a threshold.

1.2.6 Event Recommendation

Description and Priority

The application records the user's interests towards various types of the events. By default, it displays only user's preferred types of events. This is an essential, high priority feature.

Functional Requirements

- During first-time login, the application allows users to choose what are the types of events that is of user's interest. User selects event type from a predefined list.
- Users are allowed to change their interest at any time.
- By default, the application displays events only of types which are in user's interest list.
- Based on his location, the user will be notified of the recommended events occurring near him. User can choose to toggle this function on/off.

1.2.7 Sync with Calendar

Description and Priority

The application will sync all the events the user has registered to with the iPad calendar. This is an essential, medium priority feature.

Functional Requirements

- The user has an option of syncing his events with the iPad calendar. If this option is enabled, the sync will take place
- Everytime the user marks an event as 'Attending', the application will enter that event in the user's iPad Calendar
- This does not interfere with Calendar's functionality of sending reminders/emails etc

1.2.8 Facebook Integration

Description and Priority

The application will be integrated with Facebook to view friends and share events. This is an essential, medium priority feature.

Functional Requirements

- The user can choose to connect to Facebook.
- The user can post an event of interest on Facebook.

1.2.9 Bus Direction

Description and Priority

This application will display the suggested internal shuttle buses that can be taken from the user's location to the event's venue. This is an additional, low priority feature.

Functional Requirements

- The user can enter the location of the place he wants to (in NUS) and the application will tell him the buses he can take to reach his destination from his current location

1.2.10 Create Private Events

Description and Priority

The application will provide additional support for event categories, where private events can be created (only be viewed by friends). This is an additional, low priority feature.

Functional Requirements

- The user can create private events that will only be displayed to his friends
- This event will not show on the map of the users that are not part of the private event

1.3 User Manual

1.3.1 NUS Reach at a Glance

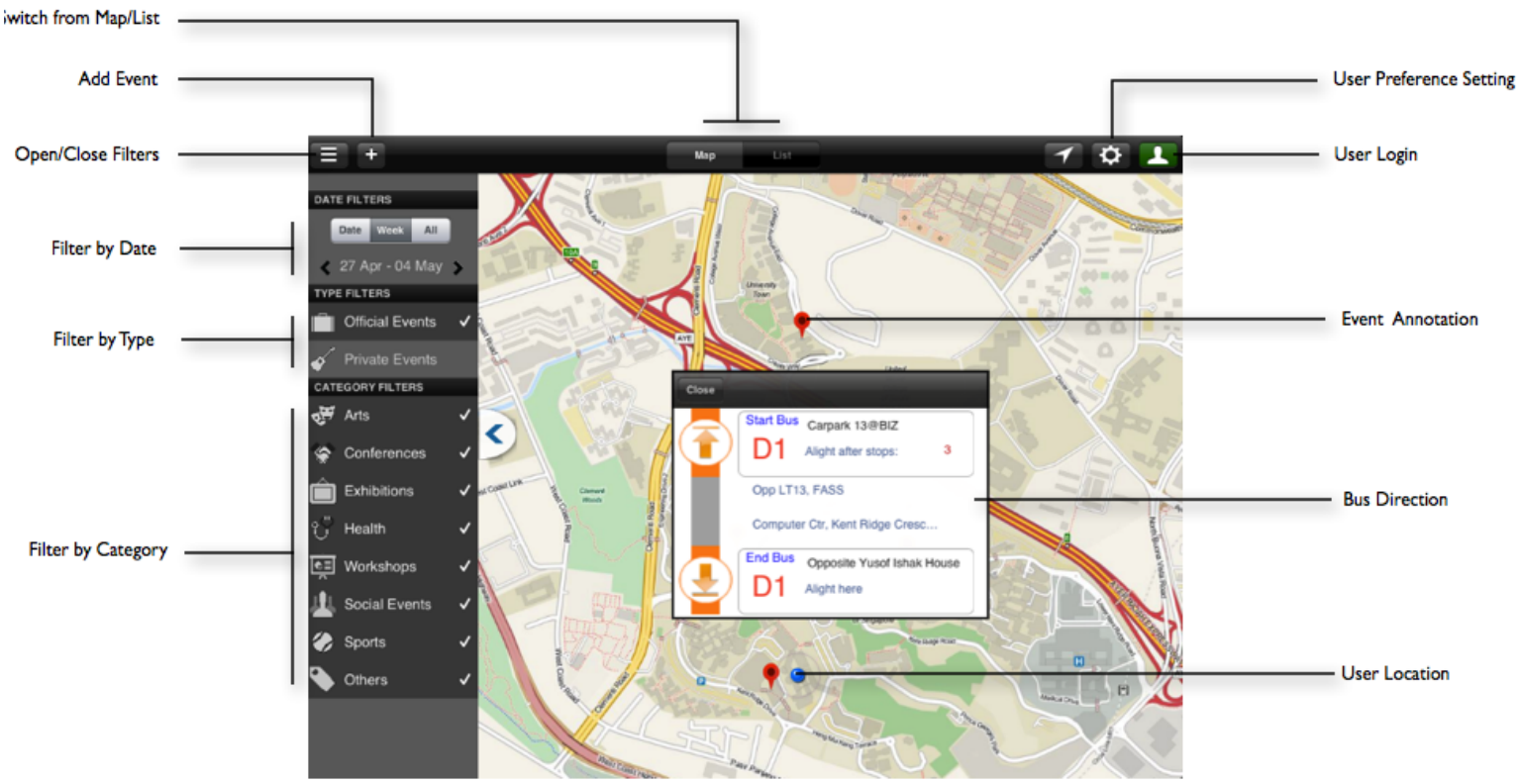


Figure 1. Map View

The following is the list of features available in the map view of our application:

- **Switch from Map/List:** Switch from map view to list view. On map view, events will be shown on different locations of the campus. On list view, event abstract will be shown in a list.
- **Add Event:** Create private event on map.
- **Open/Close Filter:** Open/Close side panel filters.
- **Filter By Date:** Filter events on map/list according to event date. There are three types of date filters. Filtering by one date, by a period (a week) or no date filter
- **Filter by Type:** An event is divided into two types, official event and private event. Official events are events from IVLE and private events are events created by user using this application.
- **Filter by Category**: An event can have one of the 8 categories: Arts, Conferences, Exhibitions, Health, Workshops, Social Events, Sports and Others.
- **User Preference Setting:** Choose interested event categories using this settings.

- **User Login:** To create an event, IVLE login is required. To share an event, Facebook login is required. User login provides login pages for these two logins.
- **Event Annotation:** Events on map are shown as red annotations on the map. Tap on the event annotation, a list of events on the current location will be shown.
- **Bus Direction:** Long press on the destination location and select Get Direction will show you the bus to take to get to the destination from your current location.
- **User Location:** The blue dot shows your current location and pressing this button centres the map to your location

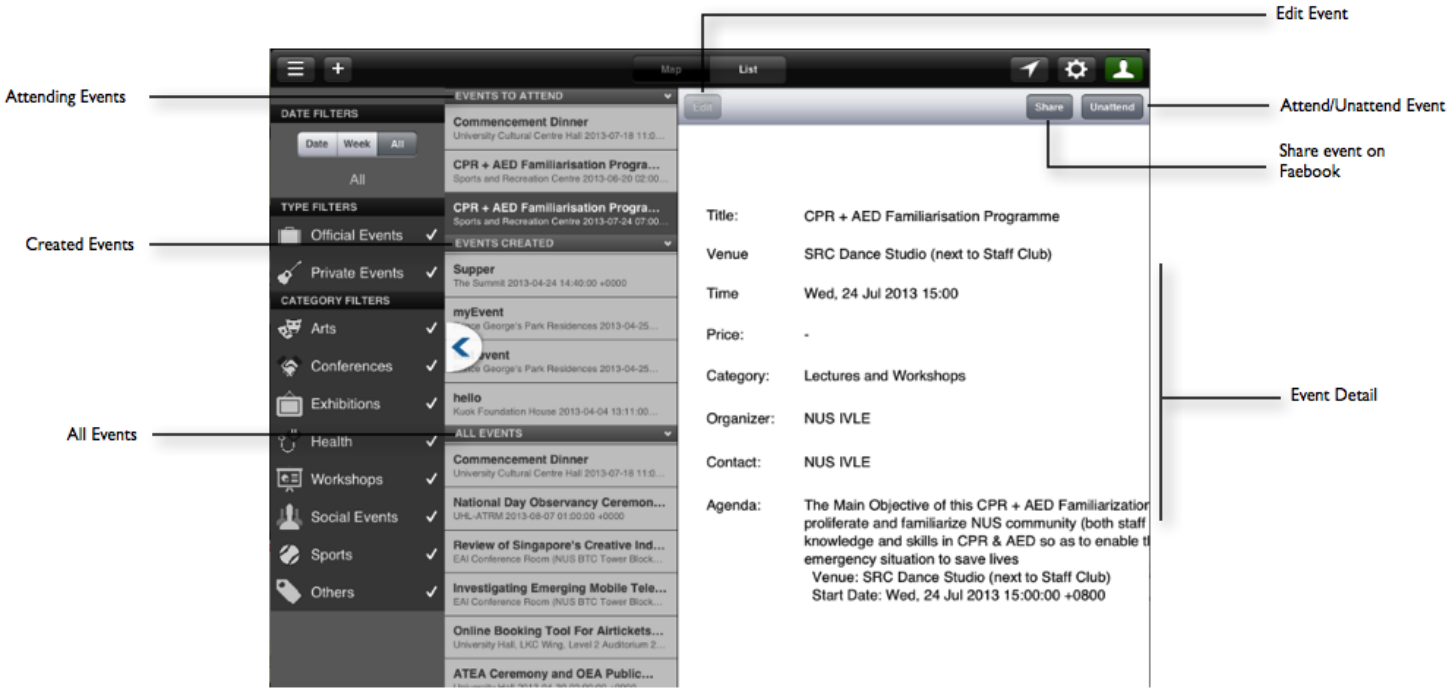


Figure 2. List View

- **Attending Events:** Under this section, events that you want to attend are listed.
- **Created Events:** Under this section, events created by you are listed.
- **All Events:** Under these section, all events that satisfy the filters are listed.
- **Edit Event:** Press the edit event button to edit event details. Only events created by you can be edited.
- **Attend/Unattend Event:** Press the attend/unattend button to add the event to the attending events list. These events will be synced to your iPad calendar.
- **Share Event on Facebook:** Press the share button to share the event on your Facebook page.
- **Event Detail:** Event details are shown on the right panel.

1.3.2 Create Event

The event created by you will be classified as private event. There are two ways to create an event. Press on the Add Event button to create event, or long press on the location on the map to create the event.

Figure 3. Long press and add button to create event.

- Long press on the location and select Create Event to create event on the selected location. The event create page will shown.
- Tap on Add Event button to create event.
- The location field will be automatically filled. To change the location, tap on the venue field, you can select the location from selection popover. Similarly, tap on the category field, you can select the category from selection popover.

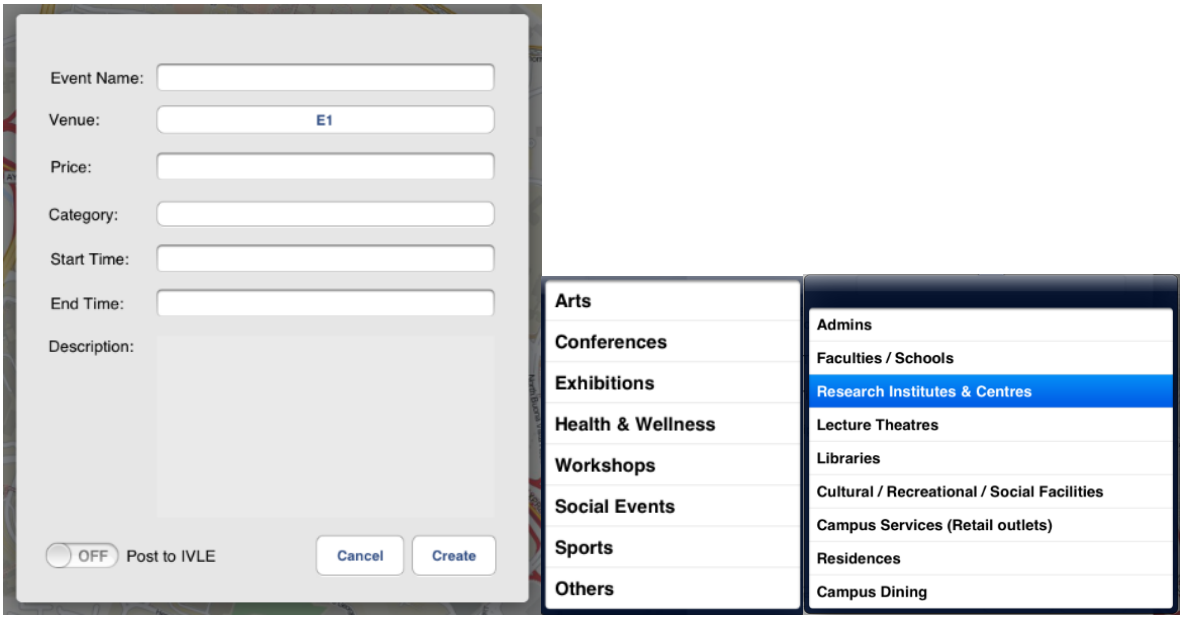


Figure 4. Event creation page, category selection popover, venue selection popover (from left to right).

1.3.3 Browse Event Detail

To see events on a location, tap on the event annotation at the location on the map. A list of events of shown on the popover. Tap on event cell to see the event details.

Figure 5. Tap on event annotation to open a list of events on the location.

1.3.4 Share Event on Facebook

After getting the event detail, you will notice the share button on the page. Tap on the Share button to share the event to Facebook.

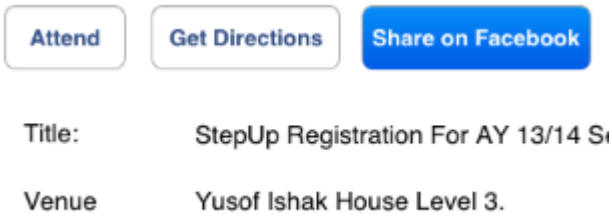


Figure 6. Share button on event detail page.

Figure 7. Post the event to Facebook popover.

1.3.5 Get Nearest Bus Stop

To get the nearby bus stops from your current location, tap on the location annotation (blue dot) to find the nearest bus stops. The figure on the right shows the list of bus available at the bus stop.

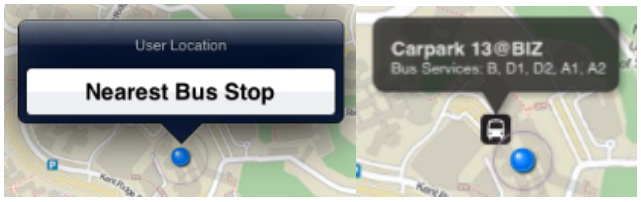


Figure 8. Tap on the location annotation to get the nearest bus stops.

1.3.6 Find Bus Directions

- Long press on the destination location to open the popover. Tap on the Direction button to get the bus directions from your current location to destination location.
- Or tap the Get Direction button on the event detail page to get the bus directions from your current location to destination location.

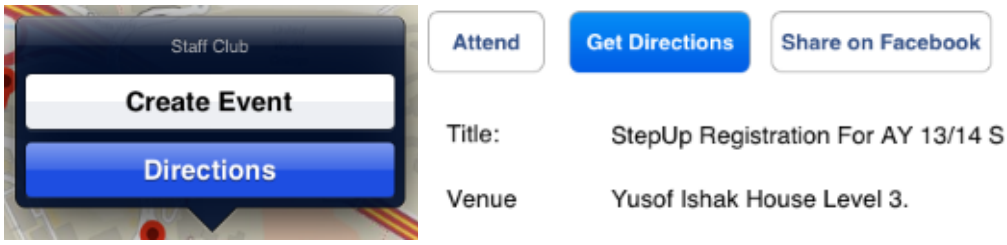


Figure 9. Long press to open popover or tap on get direction button on event detail page.

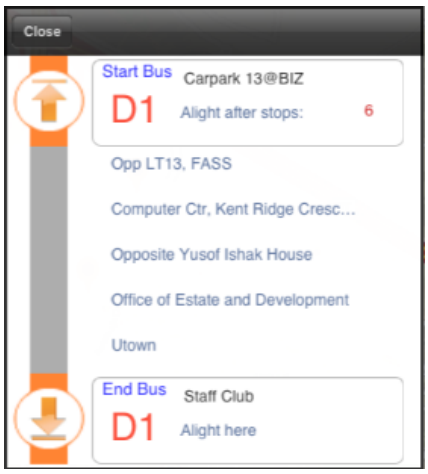


Figure 10. The bus directions from Carpark 13@BIZ to Staff Club.

1.4 Performance

Since our application does not deal with updating of the screen constantly (as in a game, or video related applications), there aren't any constraints regarding the frame rates while updating the display. The constraints are related to network and storage issues.

For best performance, the user should have good connectivity as some of the functionality depend on internet access (pullings of events, userlocation etc). Also, in order to provide the best experience, at the start of the application, all the events are pulled from IVLE and database. Although this is a time consuming process, this is run in the background. Once all the events are loaded, all the further actions are taken directly on these events.

To avoid lag, all the actions that deal with the database (or network related functions) are run in background, where the UI is updated accordingly after these batches have been run.

On the side bar, there are certain filters that the user can apply. If the user selects too many options in a very short time where some of them need to retrieve a new list from the database, the application may have a certain lag. Similarly, when the user wants to share an event on facebook, there is a certain lag as the facebook session is first created/loaded.

Although not a direct performance issue, all the events posted to IVLE may take upto 7 days as the administrator needs to approve its validity.

2. Testing

2.1 Strategy

Our team decided the bottom-up testing strategy for this project. As part of this strategy, every team member continuously tested his/her part as part of the development process. At this level, each member also conducted unit testing. This type of regression testing helped to ensure that our individual parts function properly. As we integrated in parts after each of the milestones, a two-phase integration was conducted to ensure that the system does not suffer from big-bang breakdown. Besides, we also conducted functional testing to validate our project. The following is a short description of each of the approaches that we adopted:

Unit Testing: Although our project does not deal with many computations on a set of data, we conducted unit testing for our individual parts by testing the models and the observer pattern as well as using assertions to check that the correct data is returned from the APIs. This was of great importance in the process of regression testing during development. This kind of testing helped to ensure that our individual components were bug-free before the integration process.

Integration Testing: Given that each of us worked on a specific component, integration testing played an important role in our testing strategy. In order to avoid a big-bang scenario, we planned to perform integration testing comprehensively. Differences in understanding or assumptions made may result in incompatibility of the various components developed individually. As a result, integration testing is a very important aspect of the development life cycle, and should be performed earlier rather than at the end. This is done to ensure that we do not end up with a system where all the components work individually, but fail when we put everything together. We gave significant man hours for this task and were able to successfully integrate all the components together.

Functional Testing: During development, we tested every feature and checked whether they worked according to the specification. The test cases in the appendix cover this part of the testing process. Functional testing is important as it helps us to verify that our system is ready for release, as per the requirements. A maintained suite of functional tests:

- Captures user requirements in a useful way
- Gives the entire team (users and developers) confidence that the system meets those requirements

Performance Testing: Once integration testing and functional testing were completed, we conducted performance testing to ensure that the app does not crash when subjected to different workloads and under different background conditions. Stress testing was a crucial part of this process. We also made use of the Instruments tool in Xcode to do performance testing. This helped us to analyse the memory consumption at different usages of the application, while also measuring the response times for our application. This phase of testing is quite important to ensure the robustness, and improve the overall user experience, for our app.

2.2 Test Results

Based on the priority of the specifications, the functional testing was performed. Hence for functionality like filters of date and categories, the testing is thorough and all the faults have been eliminated. The date filter itself has 3 filters: daily, week range and all dates. All these three date filters have also been tested and errors in these sections should be not expected. However, this assumes that the events that are being filtered have all the relevant fields in the correct format otherwise an exception will be thrown.

Protocols have been used extensively in the system. However, testing these delegates was difficult as the both the delegate and the target had to be checked. There were cases of delegate not being set correctly (wrong target). However, the process of integration testing ultimately help to resolve this, and we are now confident regarding the delegation patterns that we have used in this application.

Certain checks and assertions have been kept to check for the representation invariants. However these are not robust as they should have been. The creation of events should contain more checks in order to verify the validity of the event. This is the reason why certain newly-created errors do not show up on the map (elaborated in the reflections).

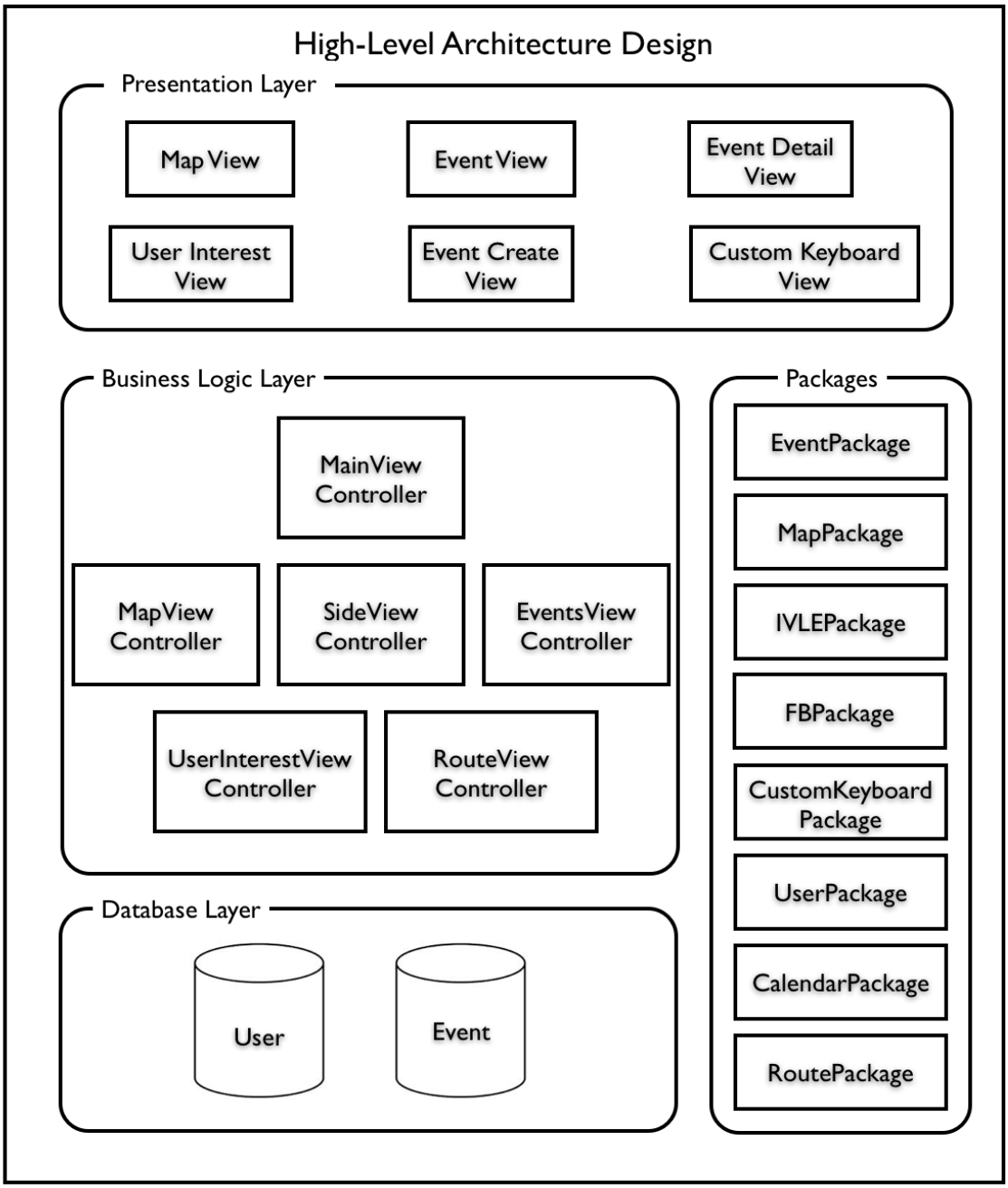
Some of the low priority features such as bus directions, and user preferences, have not been as widely tested as they should have been. Faults might appear under certain boundary conditions as the Bus directions feature is made very flexible with the user being able to get directions to any location. Errors should not be expected with direct routing (non changeovers), whereas changeovers requires a much complex algorithm that might throw exceptions.

The functional and integration testing helped us to eliminate many of the faults and helped in identifying unexpected behaviour. However, these are tedious and time-consuming and were only performed after a particular time-frame. Similarly, performance testing indicated what changes needed to be made in order for the application to perform in the desired manner, but did not help in identifying errors. When some processes, such as pulling events from the database, were found to be time consuming, they were shifted to be performed in the background so that they did not affect the other functionalities.

3. Design

3.1 Overview

3.1.1 Top Level Organization



The presentation layer consists of the following main views:

- **Map View**: This view is used to display the NUS campus map and all the related functionalities.

- **Event View:** This view provides an alternative to the map view by displaying the events in a list form.
- **Event Create View:** This view displays a form for the user to create a new event.
- **Event Detail View:** This view displays the details of individual events. It can be accessed from the map view as well as the event view.
- **User Interest View:** This view displays the collection of category preferences which the user can save to his profile.
- **Custom Keyboard View:** This view is used to display a custom keyboard with date/time inputs, and is accessed from the Event Create view.

The business logic layer consists of following classes to achieve the main functionality of the application:

- **Main View Controller:** This is the central logic controller of the application, which interacts and controls the other controllers to achieve the various functionalities. It is the initial view controller called in the application, and sets up the different models, views and controllers required in the application.
- **Map View Controller:** This is the view controller associated with the map view. It acts as a façade between the Main View controller and the Map package.
- **Side View Controller:** This controller handles the logic for the display and behavior of the side filter bar in the application.
- **Events View Controller:** This controller is the interface for the Events package which handles all the event-related views such as the event list view, event create view and the event detail view, and their related functionalities.
- **User Interest View Controller:** This controls the user interest view and its functionalities such as saving/loading the user's preferred categories of events.
- **Route View Controller:** This controls the logic and display of the bus routes to an event location.

Apart from the above layers, the design consists of a number of packages, which include classes that provide logical support for the various functionalities and deals with the external APIs. Chief packages include the following:

- Event Package
- Map Package
- IVLE Package
- Facebook Package
- Custom Keyboard Package
- User Package
- Calendar Package
- BusRouter Package

In the bottom layer, our app requires a database server to store and retrieve data related to the user, his preferences, the app-created events as well as the events which the user chooses to attend.

3.1.2 Use of Libraries and Third-party APIs

This app makes use of MapQuest API, Facebook API and IVLE API. On the database side, we are using Parse. The following is a short description of why we chose the APIs for our project:

- **IVLE API:** We need the IVLE API to provides login functionality, thus freeing us from having to let users create new accounts from scratch.
- **Facebook API:** To enable users to connect to Facebook, we need to make use of the Facebook API. From iOS 6 onward, Facebook provides a native API which is convenient for us to use. We use the Facebook API to allow users to share events from our application by posting the event details on their Facebook wall.
- **MapQuest API:** MapQuest is a free, widely-used mapping service owned by AOL. It provides a native API for iOS. We choose to use MapQuest instead of Google Maps as it provides greater level of details for the NUS campus buildings.
- **Parse API:** Parse provides a backend database server to store all the app data online, with minimal backend implementation; the Parse SDK provides a very easy-to-use interface to post and pull events from the cloud database. All the data (events and user data) are stored in *Parse.*
- **TapkuCalendar:** TapkuCalendar provides a calendar view that can be displayed conveniently and has delegates to inform the program which date was selected. This is used for selecting the dates when creating an event, as well as when the user wants to change the date filter.

3.1.3 Design Patterns

In this application, we have made use of the delegation pattern extensively. This is because we use Main View Controller as the central control, which coordinates the actions of the other controllers. For example, if the user uses a long press gesture on the map to create a new event, then the MapViewController sends a message to its delegate i.e. the MainViewController, which then calls the appropriate method in the EventCreateViewController.

We have also used the Façade pattern to reduce the dependencies in our design. Instead of allowing the Main View Controller to interact with the individual classes in each package, the façade class provides an interface to the underlying implementation. It also deals with the third-party libraries and APIs

The CalendarManager class, which handles the syncing of events with the iPad Calendar, is a Singleton class. The single instance stores the identifier of the NUS Reach calendar in iCal, and handles the adding and removal of events from this calendar.

3.1.4 Interesting Design Issues

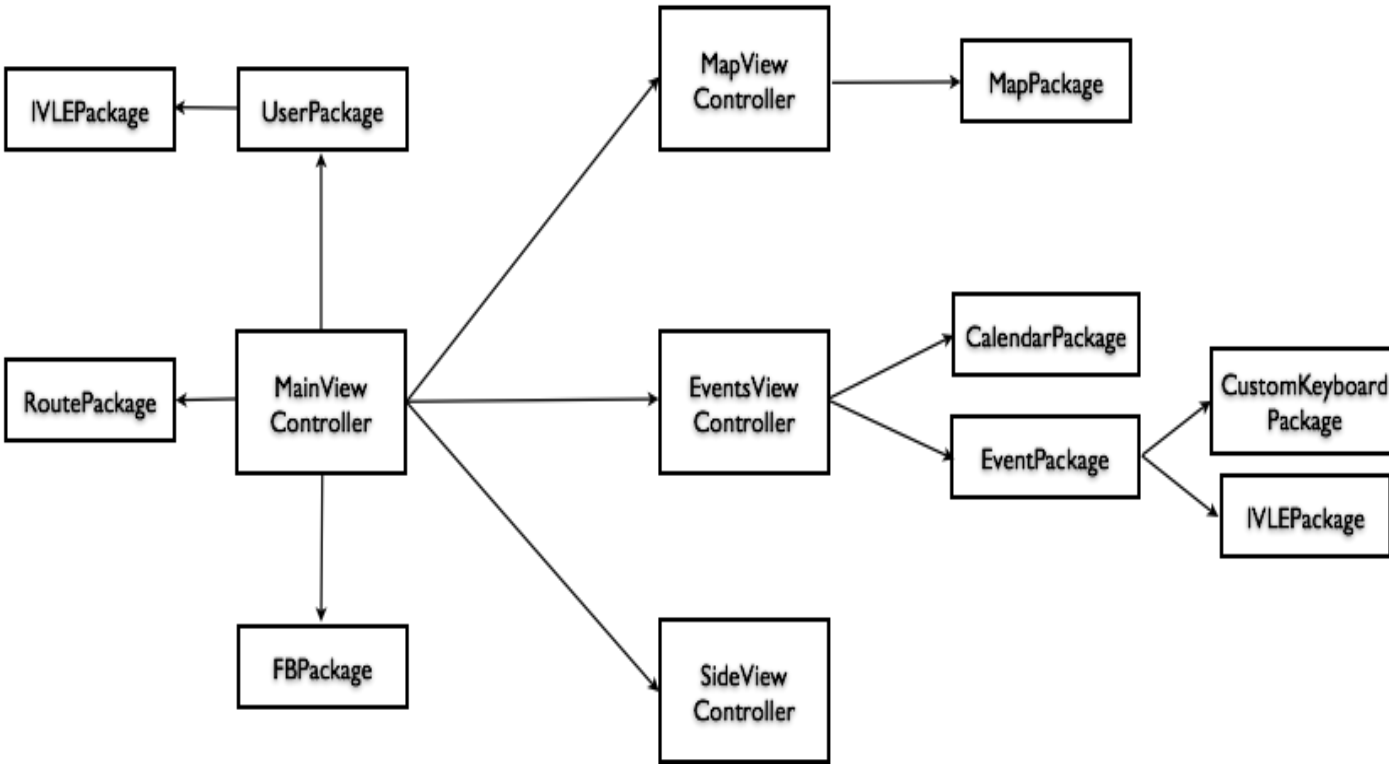
This section highlights some of the issues which we encountered during the design process and the design choices that we made in these regards:

- **Choice of Map API:** As opposed to using a static picture, we preferred to have a dynamic map API that would allow us to have more functionalities such as displaying more/less details on the map when the user zooms in/out. After researching we found 2 map tools which we could use: OneMap and MapQuest. Both these map providers had quite detailed maps of the NUS campus. OneMap has a better locating system compared to MapQuest (it locates buildings more accurately and seems to have coordinates of more buildings in their database), but it only provides a web API. On the other hand, MapQuest does provide a native iOS API. This would make map customization and adding various views and overlays easier. Therefore, we chose MapQuest over OneMap. MapQuest's lack of building coordinates in their database does not pose a big challenge as we have store a complete set of geo-coordinates for all the NUS buildings.
- **Map Display:** We want the user to only see the NUS Map (Kent Ridge campus), and not be able to drag or zoom the map outside the campus. This was difficult to control as the gestures were directly controlled by the MapQuest SDK, and we had to understand the processes underneath.
- **Map Annotations:** To add the overlays and annotations on the map, a custom layer couldn't be made as there were problems of the layer going out-of-sync with the map. Hence, we decided to use the annotations feature of MapQuest, and implement a custom callout when the annotation was tapped. The callout is difficult to customize as this was also implemented as part of their SDK, and only the annotation pin could be directly modified. This was done eventually be subclassing the MapQuest API's classes.
- **EventCreateViewController:** In our initial design, the event creation functionality was handled by the EventsViewController. However, we chose to separate this functionality and create a separate class for this functionality as the EventsViewController is responsible for displaying all the events in a list form, and the creation of a new event should be handled by a separate module.
- **Asynchronous Event Pulling:** When the application is loaded, it pulls all events from IVLE and from our database, to display on the map. Although this is done asynchronously and does not affect the other functionalities of the application, it would present a problem if the user changes the filters while the event is being loaded. In order to solve this, we display the Side Filter View only after the events have been loaded.
- **Picker vs Table Dropdown:** When creating a new event, the user needs to select a location for the event. However, given the hierarchical structure of locations available (Category -> Sub-Category -> Location), it would not be very intuitive to use the standard picker control. Therefore, we chose to create a custom table dropdown display, which would allow hierarchy navigation. However, this dropdown is currently specific to this application. In future implementations, we will change its design to make it generic.
- **Custom Keyboard:** While considering the user experience aspect of our application, we realized that the user might not want to enter the date through a keyboard when creating events, while a date picker does not fit well within our page. To tackle this, we decided to

use a custom keyboard that will display a calendar and a time picker, thus providing a more intuitive input method.

3.2 Runtime & Module Structures

The top level organization in the previous section described the overall architecture of the application. Each of the components, along with its module structure, has been described in greater detail in this section. The following is a module-dependency diagram for our overall system:



3.2.1 Central Control Classes

The central control is the starting point of the application, as it delegates all the tasks to the respective view controllers based on the current state of the application. In accordance with the MDD above, the following components are required to support the functionalities of the central control:

- **MainViewController**: As described in the previous section, this class is responsible for interacting and controlling the other view controllers to achieve the various functionalities of the application. It is the delegate of a number of these view controllers. These view controllers inform the MainViewController of any action performed (for example, the user presses on the ‘Get Directions’ button on the Event Detail View). The

MainViewController then calls the appropriate methods in the corresponding view controllers to complete the action (in the above example, the MainViewController calls the classes in the RoutePackage to display the bus directions.

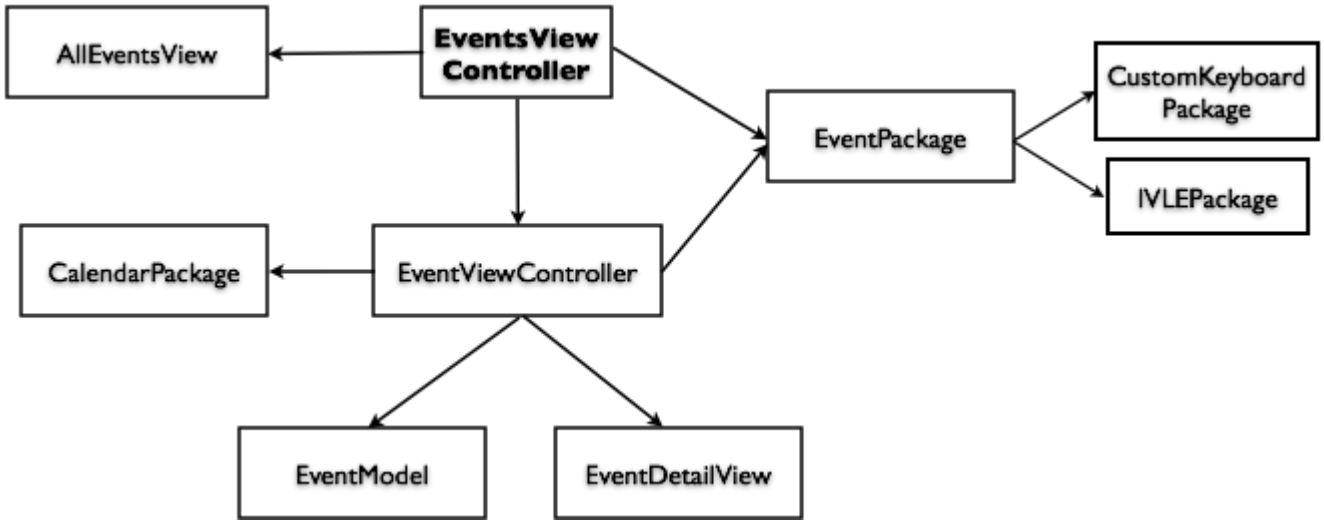
- **SideViewController**: This class is a subclass of UIViewController and is responsible for the display and behavior of the side filter bar. The user interacts with this bar to apply various filters to the events being displayed. All the user interaction with the sidebar is handled by this class, which then informs the MainViewController to update the dynamic view accordingly.

We choose to have MainViewController as the central logic controller of the application as it helps to reduce dependencies in our design. The MainViewController itself does not link with the map and events packages. It is the individual components that deal with the packages to perform the appropriate action. Thus, each component functions independently. A possible disadvantage of such a design is that the MainViewController becomes a bit bulky, as it has to deal with a number of components. An alternative design could be to allow greater interaction between the components. While this would simplify the MainViewController, it would also lead to increased coupling in our design.

3.2.2 Event Classes

These classes manage the creation, editing and display of events. In this app, we enable the user to create both IVLE events (official IVLE events which go through the same approval process) and student events (unofficial events which go into our database and does not need to be approved before being displayed on the map). Users can edit or delete those events which he created and which have still not expired. On the map view, events will be displayed as pins. Once a user clicks on the pin, the details of the events will be displayed. Last but not least, users can filter the events according to date, event types and category.

The following module-dependency diagram given below interactions among the various Events classes:



The following are the main components which will deal with the event functionality:

- **EventsViewController:** This is a subclass of UIViewController. It controls the display of events in the list view. It stores a list of EventViewControllers (i.e. it stores a collection of events) and also controls the instantiation of new EventViewControllers. Upon first load, it will ask EventManager for the list of Events and create the respective EventViewControllers. This class is the main interface between the MainViewController and the rest of the Event classes.

- **EventViewController:** This is a subclass of UIViewController. It controls the actions related to individual events, such as storing the event details in the Event Model, and maintaining an EventDetailView object of the event. It uses the delegation pattern to inform the MainViewController when an action is taken on an event. It also interacts with the EventPackage and CalendarPackage.

- **EventPackage:**
    - EventFilter: This class is used to generate filter patterns. When the app first loads, filter patterns are generated from the user's stored preferences. Additionally, when using the app, users can apply various other filters (filters of date, event category and event types) as well.

    - EventManager: This class handles pulling, saving events and loading event details from the database (for Private Events) and IVLE Event Organizer (for Official Events). Also, it reads filter patterns from EventFilter, applies it on events, and return the list events which should be displayed to the user.

    - EventCreateViewController: This is a subclass of UIViewController. It contains all the functionality related with the display and logic for the creation/editing of an event. It uses the CustomKeyboardPackage to display the custom keyboard view

for the date and time inputs. It also uses two popover classes to display dropdowns for the venue and category inputs.

- **CalendarPackage:** The class in this package handles integration with the iPad Calendar. Whenever the user chooses to attend an event, this class syncs the schedules of the event with the user's Calendar application.

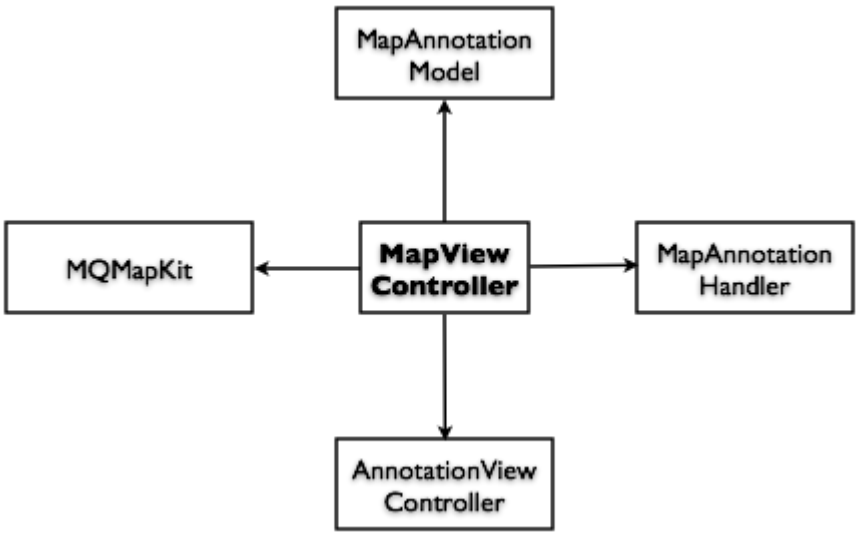
- **EventModel:** the EventModel stores the details of an event. For example, event title, event date, event category, description, creator, attendees, etc. Each event is uniquely identified by its eventID.

The above design for the Event classes helps us to separate the classes in the EventPackage from the EventsViewController, EventViewController and the EventModel. The classes in the package are used by the EventsViewController and EventViewController to get data related to events. We feel that such a design helps to reduce coupling and increase cohesion. Besides, such a design is also in conformity with the MVC design pattern, where the Event model and the Event views does not interact with each other, and all the views are controlled by view-controllers. Given that the entire Events component functions independently, such a design also aids in division of responsibility among team members as one member can focus on this component.

3.2.3 Map Classes

These classes control the display of the NUS map, along with its various overlays. For this project, we have decided to use the base map provided by MapQuest.

The following module-dependency diagram given below interactions among the various Events classes:



The following are the main components which will deal with the map functionality:

- **MapViewController:** This is a sub-class of UIViewController which controls the UIView which is used to display the map on the screen. It responds to user actions and gestures with the help of the other classes in the Map package. This class is the main interface between the MainViewController and the rest of the map classes.

- **MQMapKit Framework:** As mentioned above, this framework is used to obtain the map tiles and support the various functionalities related to the map such as zoom, rotation, panning, adding overlays etc.

- **AnnotationViewController:** This is a subclass of UITableViewController. It creates and controls the table that is visible when each annotation pin is tapped on the map.

- **MapAnnotationHandler:** This class handles the adding of each event as a map annotation by parsing the string value of the venue of each event into the respective location coordinates.

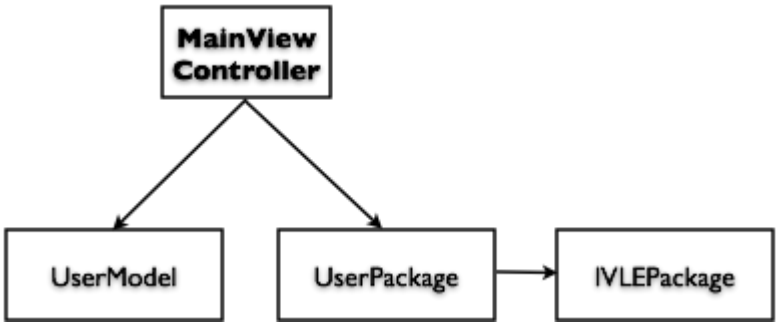
- **MapAnnotationModel:** This class is a subclass of MQPointAnnotation (the annotation class of MapQuest SDK). This class is used to store additional details like the actual EventModel being represented by each annotation.

In the above design, we need to extend the MQPointAnnotation class of MapQuest in order to store the EventModel associated with each annotation. This is needed to populate the annotation table which is displayed when a pin is tapped on the map. This component functions independently as well. The MainViewController needs to provide the MapViewController with the list of EventViewControllers representing the events to be displayed on the map. Whenever the user changes the filters applied, a new list is provided, and the MapViewController updates its display accordingly.

3.2.4 User Classes

These classes control the data specific to the user, such as use login and user's preferences of events. The user-specific data is stored in a server, so that whenever the user logs into the system (from any device), he can have the same experience and would not have to re-customize all his settings. *Parse* database, as mentioned earlier for storing events, is used to store the user's data.

The following module dependency diagram shows the main components that deal with the User-specific data:



- **User Package:**
    - UserOptionsViewController: This class is used to display the login fields on the main screen of the application. Two login fields will be displayed: one for IVLE and one for Facebook, where logging into IVLE is compulsory to create events and logging into Facebook is optional.

    - UserLoginViewController: The class handles the login/logout request for IVLE. It has a delegate to inform the respective controller that the user has been successfully logged in. Based on the login with IVLE, it sends the token to IVLEManager to be saved in a file. The data regarding the user-login is stored in the class *UserModel*.

    - UserInterestViewController: This is a subclass of UIViewController and is responsible for his preferences for the types of events he is interested in. This page is displayed when the settings button is tapped at the top toolbar in the mainview. The user's preferences entered here are stored in the class *UserModel* and synchronised with the database server. The map is updated accordingly to display the new filtered events.

- **UserModel**: The *UserModel* will be used to store the user-specific data like the login details and his interests (to be used in his current session). The UserModel will store the references of the events created by the user, and subscribe to (marked as 'Attending') by the user.

It is important to note there are two main views that are required for the Users (the login and the interests). Both the views use this *UserModel* and modify it accordingly. Also, the database is synchronised with the UserModel at all times, in order to have the latest list of user registered and created events (as there are issues that the user might face with lack of network etc).

The design here is in conformity with that of the other discussed above (Events, Map etc). This too follows the MVC pattern in a similar manner with minimal coupling between the view and the model.

3.2.5 Route Package

These classes handle the computations of route between two given points and displays the route in a table form. It provides the most efficient route in terms of the NUS shuttle buses that can be taken to reach the destination.

The following are the main components which will deal with the route functionality:

- **BusRouter:** Computes the route between two points, where one point is userlocation and the other can either be the building name or location coordinates. MainViewController uses this class and links with the RouteViewController to display the route computed.

- **RouteViewController:** A class used by MainViewController to display the route in a table form. It uses a dictionary with the route data (start and end points, bus numbers etc) and parses it to fill the cells of the table. It also alters the table format if the route contains a changeover.

3.2.6 IVLE Package

These classes handle all the IVLE related functionality. We use the IVLE API to enable user login, pull events from IVLE and post event to IVLE events organizer.

The following describes the class which will deal with the IVLE functionality:

- **IVLEManager**: This class contain basic login, event pulling and event posting methods. *UserLoginController* will interact with this class after creating the user IVLE login page, where this class would validate the user login information and get the IVLE token. Similarly, *EventsController* will use this class to pull events from IVLE and *EventController* will use the post events functionality.

- **IVLEParser:** A helper class for IVLEManager that parses the data received from different url requests. It handles the parsing of username, user login, and user token.

3.2.7 Facebook Package

This class handles the Facebook related functions. We will use the Facebook API to enable integration with Facebook in the form of logging in and sharing events to Facebook.

The following describes the class which will deal with the Facebook functionality:

- **FacebookManager**: This class contains user connection, user validation and event posting methods. UserLoginController interacts with this class to create user Facebook login page and validate user login information. MainViewController uses the sharing events functionality provided by this class.

The IVLEManager and FacebookManager classes deal solely with the functionalities directly related to loading data or posting data to IVLE or Facebook. Other modules can use these functionalities through these classes. We feel that such a design helps to reduce coupling and increase cohesion.

4. Reflection

4.1 Evaluation

Given the limited time and the tight schedule under which this application was developed, we are happy to have completed all the functional specifications with efficiency and accuracy. The order of implementation was based on the priority of the specification. Hence, the most important features were first identified and implemented. Prior to implementation, our team had spent a significant amount of time in designing our system. Given that we each of us was clear in terms of the design, and our design allowed us to work on different components independently, the implementation of the high priority features was smooth. These features were then enhanced with medium & low priority features like custom keyboard, sync with iPad calendar and bus directions.

Although we are pleased with the overall design of our application, there are a few features of the design that can be improved. Key among them are as follows:

- Our design for the location dropdown is very rigid. Currently, there are separate classes to deal with the individual levels of the location selection hierarchy. In future versions, we will change this to a more robust design that can be extensible beyond our application by using a single view controller for hierarchical selection.
- In our current design, the MainViewController is long as it implements the delegate methods for a number of other view controllers. This design could perhaps be improved to improve its modularity.
- Currently, our application pulls events from IVLE and from the database when it is started. However, in future implementations, we will try to store a local cache of limited number of events on the iPad. This would ensure that the application would be able to display the upcoming events even when the iPad is not connected to the network.

Since we have used the delegation pattern extensively in our application, our design required all the different components to be in sync with each other. This presented a problem initially, when we worked on different components separately. In these cases, creating stubs for the delegate methods was important. Once we began integrating the various components, then we could replace the stubs with the actual implementations, with confidence that the integration would not break the system.

Once the functionalities and packages were implemented, we shifted our focus to polishing our app. A lot of time was invested in fine-tuning and improving the user interface and user interaction with the application. Performance testing was done simultaneously with the regression testing so as to notice any significant issues in terms of updating the views. This process helped to ensure that we were able to develop a reliable application which meets all the required specifications.

4.2 Lessons

This project was a great learning opportunity for us. We were able to apply a number of software engineering principles during the course of the design and implementation. The use of these principles made the entire development process much easier for us. Some of the important lessons are as follows:

- It is important to spend time on the analysis and design aspects. This includes things such as preparing the module specifications as well as the test cases even before the implementation begins. This would help the entire team to be clear of the requirements and help them to work independently. We had prepared the test cases before hand. This helped to ensure that we conformed to our specifications well, as it provided a reference throughout the implementation process. We should have done the same for module specifications.
- It is important to set the coding standards among the team members. Every member may have different coding styles . Therefore, in order to ensure that the coding style is uniform throughout the project, we agreed upon the basic standards to follow. Yet, there were a few irregularities which we will try to avoid in future implementations.
- Documenting one's code is crucial in a team project. As each member works on a different component, he should simultaneously document it. This lesson was reinforced when we used third-party APIs in our application. For example, the documentation provided by MapQuest is not very extensive. Therefore, we had to spend quite a bit of time while integrating their API.

4.2.1 Known Bugs and Limitations

Although we have worked on removing most of the bugs from our application, there are some known bugs which we will resolve in the next version of the application:

- The application allows the user to enter blank values when creating a new event. This, in turn, presents problems when loading the events from the database. We need to set checks to prevent this in the EventCreateViewController.
- The Bus Router sometimes returns a null value for the suggested bus. This may occur when there is a change of bus required to reach the destination.
- When the user logs out of the application, or changes his preferences from the settings, the list of events displayed on the map is not updated properly. This can be solved by reloading the events list when the user performs the above actions.

Thus, we have learned a number of valuable lessons during this project. Given a second chance, we would have spent some more time on testing the low-priority and medium-priority features, which would have helped to eliminate the bugs mentioned above.

Appendix

Module Specifications

MainViewController.h

```
/*
 This class is the main root controller of all the views and models. It handles the
 delegates of all other controllers and takes the necessary actions.
 It also handles the main user interaction and the actions to be taken
 This class integrates all the othe classes, as it implements the delegates from
 the other classes.
 */



#import <UIKit/UIKit.h>
#import "MapViewController.h"
#import <QuartzCore/QuartzCore.h>
#import "SideFilterViewController.h"
#import "DatabaseHandler.h"
#import "UserLoginViewController.h"
#import "EventsViewController.h"
#import <EventKit/EventKit.h>
#import <EventKitUI/EventKitUI.h>
#import "UserModel.h"
#import "RouteViewController.h"
#import "UserOptionsViewController.h"
#import "UserInterestViewController.h"
#import "FBManager.h"

@interface MainViewController : UIViewController <UIGestureRecognizerDelegate,
EventsViewDelegate, FilterDelegate,MapViewUpdater, RouterDelegate, UserOptionsDelegate,
UserLoginDelegate, UserInterestDelegate, UINavigationControllerDelegate, FacebookLoginViewUpdater>

@property (weak, nonatomic) IBOutlet UIView *mapView;
@property (strong, nonatomic) IBOutlet UIView *sideView;
@property (strong, nonatomic) IBOutlet UIImageView *slideArrow;
@property MapViewController *mvController;

@property (strong, nonatomic) IBOutlet UIView *filterView;
@property (strong, nonatomic) IBOutlet UIBarButtonItem *userButton;

@property (strong, nonatomic) IBOutlet UIToolbar *controlToolbar;
@property (strong, nonatomic) IBOutlet UISegmentedControl *displayTypeSegment;
@property (strong, nonatomic) IBOutlet UIActivityIndicatorView *mapActivityView;

@property (strong,nonatomic) UserModel *userModel;

//REQUIRES: The user location to be present on the map
//EFFECTS: Sends an action to the MapViewController to center the map based on
//the user's current location
- (IBAction)myLocationButtonPressed:(id)sender;
```

```
//EFFECTS: Toggles the view type between the MapView and the Events List View
- (IBAction)displayTypeSegmentChanged:(id)sender;

//EFFECTS: Toggles the sidebar - slide in and out from the left
- (IBAction)sidebarPressed:(id)sender;

//EFFECTS: Shows the user login popover with the IVLE and Facebook login buttons
- (IBAction)userButtonPressed:(id)sender;

//EFFECTS: Modally load another viewcontroller that has a list of categories for
//the user to select, which will be stored in the database if the user is logged in
//MODIFIES: the UserModel - usermodel's userPreferences array will be modified
- (IBAction)settingsButtonPressed:(id)sender;

@end
```

MapViewController.h

```
/*
 This class controls the main map view of the application.
 It interacts with the MapQuest API to display the campus map.
 This class also controls the annotations and gestures which have been added onto the map.
 It communicates with the other classes through delegates when the user long presses on
 the map to create an event or get directions to a location
 */

#import <UIKit/UIKit.h>
#import <MQMapKit/MQMapKit.h>
#import <QuartzCore/QuartzCore.h>
#import "Constants.h"
#import "EventViewController.h"
#import "AnnotationViewController.h"
#import "MapAnnotationHandler.h"

//a protocol to inform when user requests an action through gestures
@protocol MapViewUpdater <NSObject>
@optional
//sends a delegate to inform the relevant controller when directions to a particular
//spot on map is requested
- (void)longpressShowRouteDetails:(CLLocationCoordinate2D)location;

//sends a delegate to inform the relevant controller when an event is to be created
//directly from the map at a location
- (void)longpressCreateEvent:(NSString*)location;
@end


@interface MapViewController : UIViewController <UIGestureRecognizerDelegate, MQMapViewDelegate, AnnotationViewDelegate,
CLLocationManagerDelegate, UIActionSheetDelegate>

@property MQMapView *mapView;
@property (weak) id<MapViewUpdater>delegate;
```

```
//REQUIRES: The array to have EventViewControllers with valid EventModels
//EFFECTS: Resets the annotations being shown on the map, and re-loads the annotions
//provided as a parameter in 'annotationList. Only those events with a valid
//MODIFIES: the annotations on the map
- (void)loadAnnotations:(NSArray*)annotationList;

//REQUIRES: An EventVIewController with a valid EventModel
//EFFECTS: Adds an annotation to the map, it an annotation already existed at the location,
//it adds the event to the annotation table
- (void)addAnnotation:(EventViewController*)annotation;

//REQUIRES: The user location to be present on the map
//EFFECTS: Centers the map based on the user's current location
- (void)goToUserLocation;

//REQUIRES: Some EventDetailView to be modally shown on the screen
//EFFECTS: Dismisses the detail view controller
- (void)dismissDetailController;

@end
```

SideViewController.h

```
/* This class controls the display of the sidebar view in the app
 It also manages the behavior of the filters applied by the user.
 It communicates with the other classes through delegates when a filter is modified by the user
 */

#import <UIKit/UIKit.h>
#import <QuartzCore/QuartzCore.h>
#import "Constants.h"
#import <TapkuLibrary/TapkuLibrary.h>



@protocol FilterDelegate <NSObject>
//filter applied on the event type - ivle events or student events
- (void)eventTypeAdded;

//filter applied on the event type - ivle events or student events, where one of them is removed
- (void)eventTypeRemovedWithSelectedEvent:(eventCategory)eventType;

//filter applied on the event categories, and the modified list is sent
- (void)filtersModified:(NSArray*)newFilters;

//filter applied on the data - new dates are sent (one date, date range or no date)
- (void)dateModified:(NSDictionary*)newDate;
@end

@interface SideFilterViewController : UITableViewController <UITableViewDelegate,
UITableViewDataSource, UITextViewDelegate, TKCalendarMonthViewDelegate>
```

```
@property (weak) id<FilterDelegate>delegate;
@property NSArray *selectedPreferences;
@property eventCategory eventType;

//EFFECTS: sets the frame of the tableview of this controller
//MODIFIES: the frame of the tableview
- (void)setTablesFrame: (CGRect)thisTableFrame;

//EFFECTS: changes the filters on the side view controller
//MODIFIES: self.selectedPreferences
- (void)userInterestsChanged:(NSArray*)interests;

@end
```

EventsViewController.h

```
/*This class is responsible for handling for all the event based functionalities
 It stores a list of all the individual events which have been pulled from IVLE/database
 It also controls the display of the list view in the app
 It is responsible for linking the event with their detail view, and related functionalities such as the event edit, attend buttons etc.
 It communicates with various other classes through delegates in the following cases:
  - new event created
  - share events to Facebook
  - show directions to event
 */

#import "EventViewController.h"
#import "EventCreateViewController.h"
#import "EventFilter.h"
#import "EventManager.h"
#import <QuartzCore/QuartzCore.h>

@protocol EventsViewDelegate <NSObject>

//provides the new EventViewController that was created
- (void)newEventCreated:(EventViewController*)newEventController;

//provides the largest event id as and when encountered (from the database)
- (void)largestId:(NSString*)currentID;

//asks for a list of EventViewControllers to be shown in the eventsListView
- (NSArray*)getEventControllers;

//provides the EventViewController that is currently being seen
- (void)setTarget:(EventViewController*)event;

//provides the EventViewController and the event details that the user wants to share
- (void)shareEvent:(EventViewController*)event;

//asks for the target event set on which to perform certain action
- (EventViewController*)getTarget;
```

```
//provides the location of the event for which the user wants to see directions
- (void)showRouteForEvent:(NSString*)location;
@end

@interface EventsViewController : UIViewController <UITableViewDelegate, UITableViewDataSource, EventCreateViewDelegate>

@property (weak) id<EventsViewDelegate>delegate;
@property (nonatomic, readwrite) EventViewController *targetEvent;
@property (nonatomic, readonly) EventCreateViewController *eventCreate;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *eventListEditBtn;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *eventListAttendBtn;
@property (strong, nonatomic) IBOutlet UIBarButtonItem *eventListFbBtn;
@property (weak, nonatomic) IBOutlet UITableView *eventsListView;
@property (weak, nonatomic) IBOutlet UIView *eventDetailView;

- (id)init;

//REQUIRES: A selected Event
//EFFECTS: Opens the fields to edit the Event
//MODIFIES: The EventModel of the particular event is modified
- (IBAction)editEventList:(id)sender;

//REQUIRES: A selected Event
//EFFECTS: Stores the event in the iPad Calendar and sends the information to database
- (IBAction)attendEventList:(id)sender;

//REQUIRES: A selected Event
//EFFECTS: Sends a delegate to share the event details through facebook
- (IBAction)shareEventList:(id)sender;

//EFFECTS: Loads all the events and from the database
// based on the attendList and the createdList, marks the events at attending/unattending
// & Edit
//MODIFIES: self.eventControllers
- (void)loadAllEventsForUser:(NSArray*)attendingList :(NSArray*)createdList;

//REQUIRES: eventType - {ivle, usercreate, both}
//         date = {single date, date range or nil}
//EFFECTS: returns an array of EventViewControllers that contain that events based on the filteres applied
- (NSArray*)loadEvents:(NSDictionary*)date withEventType:(eventCategory)eventType Categories:(NSArray*)categories;

//REQUIRES: newEvents to be a list of EventViewControllers with valid EventModels
//EFFECTS: Reloads the eventlist table with the new list of events, usually after a filter is applied
//MODIFIES: the eventsListView displaying the list of events
- (void)reloadData:(NSArray*)newEvents;
@end
```

EventCreateController.h

```
/*
 This class controls all the functionality related with the creation and editing
 of an event.
```

```
The view is linked with StoryBoard.
It is also linked with other popover classes to display 2 dropdowns for Categories
and Locations
It displays a custom keyboard for the input of start & end date
Also, a pre-defined venue can be set if the user has used long-press to create
an event at a particular location.

It has a delegate to inform the relevant controllers when an action is taken on
the event:
- new event created
- event edited (only applies to those events that the user created)
- event deleted (only applies to those events that the user created)
- updateEventList (to update the list of events when a newevent is created)
*/

#import <UIKit/UIKit.h>
#import "EventViewController.h"
#import "KeyboardInputView.h"
#import "LocationPickerViewController.h"
#import "CategoryPickerViewController.h"

@protocol EventCreateViewDelegate <NSObject>
//provides the new EventViewController that was created
- (void)newEventCreated:(EventViewController*)newEventController;

//provides the largest event id as and when encountered (from the database)
- (void)largestId:(NSString*)currentID;

//asks for a list of EventViewControllers to be shown in the eventsListView
- (NSArray*)getEventControllers;

//asks for the target event set on which to perform certain action
- (EventViewController*)getTarget;

//provides the EventViewController that is currently being seen
- (void)setTarget:(EventViewController*)event;

//provides the eventcontroller that is being edited
- (void)editEvent:(EventViewController*)event;

//provides the eventcontroller that is being shared
- (void)shareEvent:(EventViewController*)event;

//provides the eventcontroller that is to be removed
- (void)removeEvent:(EventViewController*)event;

//provides the location of the event to which directions are require
- (void)showRouteForLocation:(NSString*)location;

//delegates when an event is modified and the list should be updated
- (void)updateEventList;
@end

@interface EventCreateViewController : UIViewController <LocationPickerDelegate, LocationSubCategoryDelegate,
LocationSelectDelegate, CategoryPickerDelegate, EventViewDelegate, UITextViewDelegate, UITextFieldDelegate>
```

```
@property id<EventCreateViewDelegate> delegate;
@property (nonatomic, readwrite) NSString *largestID;
@property (nonatomic, strong) EventViewController *targetEvent;
@property (weak, nonatomic) IBOutlet UISwitch *postIVLEBtn;
@property (weak, nonatomic) IBOutlet UIButton *eventCreateCancelBtn;
@property (strong, nonatomic) IBOutlet UIView *eventCreateView;
@property (strong, nonatomic) IBOutlet UIScrollView *eventCreateScrollView;
@property (weak, nonatomic) IBOutlet UITextField *titleField;
@property (weak, nonatomic) IBOutlet UITextField *priceField;
@property (weak, nonatomic) IBOutlet UIButton *categoryBtn;
@property (weak, nonatomic) IBOutlet UITextView *descriptionField;
@property (weak, nonatomic) IBOutlet UIButton *saveBtn;
@property (strong, nonatomic) IBOutlet UITextField *startTimeField;
@property (strong, nonatomic) IBOutlet UITextField *endTimeField;
@property (weak, nonatomic) IBOutlet UIButton *venueButton;

//EFFECTS: dismisses the modal view controller when the cancel button is pressed
- (IBAction)cancelEventCreate:(UIButton *)sender;

//EFFECTS: sends a delegate to inform the relevant controller to save the event
- (IBAction)saveEvent:(id)sender;

//EFFECTS: displays a dropdown of categories
- (IBAction)displayCategoriesList:(id)sender;

//EFFECTS: displays a dropdown of locations for events
- (IBAction)displayDropDown:(id)sender;


//a pre-defined venue can be set if the user has used long-press to create
//an event at a particular location
- (void)setPredefinedVenue:(NSString *)venue;
@end
```

EventViewController.h

```
/*
 This class controls the actions for each particular event.
 It stores the event details into the EventModel
 It maintains an EventDetailView object of the event (that is shown each time the
 event is tapped)
 It has a delegate to inform the relevant controllers when an action is taken on the
 event (edit, shared, get directions)
 It also offers support for posting the event on ivle and saving an event to the database
 */

#import "EventManager.h"
#import "EventModel.h"
#import "EventDetailView.h"
#import "CalendarManager.h"
```

```
@protocol EventViewDelegate <NSObject>
//provides the eventcontroller that is being edited
- (void)editEvent:(id)eventController;

//provides the eventcontroller that is being shared
- (void)shareEvent:(id)eventController;

//provides the location of the event to which directions are require
- (void)showRouteForLocation:(NSString*)location;
@end

@interface EventViewController : UIViewController <DetailViewDelegate>

@property (weak) id<EventViewDelegate> delegate;
@property (nonatomic, readonly) EventModel *model;
@property (nonatomic) EventDetailView *detailView;
@property (nonatomic) BOOL isUserAttending;
@property (nonatomic) BOOL isUserCreated;

//REQUIRES: a valid EventModel
//EFFECTS: initializes the detailview and self.model with the model provided
- (id)initWithModel:(EventModel*)model delegate:(id)delegate;

//REQUIRES: valid strings for each of the section
//EFFECTS: initializes the detailview and self.model based on each of the parameteres
- (id)initWithTitle:(NSString*)title eventid:(NSString*)eventID category:(int)c venue:(NSString*)v start:(NSDate*)s end:(NSDate*)e
price:(NSString*)p description:(NSString*)d organizer:(NSString*)organizer contact:(NSString*)contact tag:(NSString*)tag
delegate:(id)delegate;

//EFFECTS: saves the created event to the database
- (void)save;

//EFFECTS: posts the event to IVLE based on the event details entered by the user
- (void)postToIVLE;

@end
```

EventDetailView.h

```
/*
 The class that shows the details of each event. This is usually called as a pagesheet
 and the view is initialized through the EventDetails.xib
 It has a delegate to inform the relevant controller when a button a pressed:
 - attend, edit, get directions, shareevent
 */

#import <UIKit/UIKit.h>

@protocol DetailViewDelegate
@optional

//delegate to inform when the attendbutton is pressed
```

```
- (void)attendEvent;

//delegate to inform when the editbutton is pressed
- (void)editEvent;

//delegate to inform when the directions button is pressed
- (void)showRoute;

//delegate to inform when the share button is pressed
- (void)shareEvent;
@end

@interface EventDetailView : UIView

@property (weak) id <DetailViewDelegate> delegate;
@property (nonatomic, readonly) CGFloat width;
@property (nonatomic, readonly) CGFloat height;
@property (strong, nonatomic) IBOutlet UILabel *titleValue;
@property (strong, nonatomic) IBOutlet UILabel *venueValue;
@property (strong, nonatomic) IBOutlet UILabel *timeValue;
@property (strong, nonatomic) IBOutlet UILabel *priceValue;
@property (strong, nonatomic) IBOutlet UILabel *categoryValue;
@property (strong, nonatomic) IBOutlet UILabel *organizerValue;
@property (strong, nonatomic) IBOutlet UILabel *contactValue;

@property (strong, nonatomic) IBOutlet UIScrollView *descriptionScroll;
@property (strong, nonatomic) IBOutlet UIButton *editBtn;
@property (strong, nonatomic) IBOutlet UIButton *attendBtn;
@property (strong, nonatomic) IBOutlet UIButton *routeBtn;
@property (strong, nonatomic) IBOutlet UIButton *fbBtn;

//EFFECTS: initlizes the detail view with the details provided in the parameter
- (id)initWithWidth:(CGFloat)w height:(CGFloat)h title:(NSString*)title venue:(NSString*)venue time:(NSString*)time
price:(NSString*)price category:(NSString*)category organizer:(NSString*)organizer contact:(NSString*)contact
description:(NSString*)description;

//EFFECTS: modifies the button title from 'Attend' to 'Unattend'
//MODIFIES: attendBtn
- (void)userAttending;

//EFFECTS: sends a delegate when the edit button is pressed
- (IBAction)editBtnPressed:(id)sender;

//EFFECTS: sends a delegate when the attend button is pressed
- (IBAction)attendBtnPressed:(id)sender;

//EFFECTS: sends a delegate when the directions button is pressed
- (IBAction)routeBtnPressed:(id)sender;

//EFFECTS: sends a delegate when the share button is pressed
- (IBAction)facebookBtnPressed:(id)sender;

@end
```

EventFilter.h

```
/*
 This class has various filters to according to which the all the events are filtered
 The filtered events are returned as an array.
 */

#import <Foundation/Foundation.h>
#import "EventViewController.h"

@interface EventFilter : NSObject

@property (nonatomic, readonly) int category;
@property (nonatomic, readonly) NSString *price;
@property (nonatomic, readonly) NSString *keyword;
@property (nonatomic, readonly) NSArray *dates;
@property (nonatomic, readonly) NSArray *tag;
@property (nonatomic, readonly) NSString *userid;

//EFFECTS: it sets the filters fields based on the parameteres
//MODIFIES: the properties of this instance variable
- (void)setCategory:(int)category price:(NSString*)price date:(NSArray*)date tag:(NSArray*)tag;

//EFFECTs: returns an array of EventViewControllers based on the filter that is set
//  here it uses all the filters that are set in this instance
- (NSArray*)filter:(NSArray*)events;

//EFFECTs: returns an array of EventViewControllers based on the category filter that is set
- (NSArray*)filterByCategory:(NSArray*)events;

@end
```

EventManager.h

```
/*
 The is the facade that links with the backend for the events.
 It does the pulling of events from ivle and database
 It does the pushing of events from ivle and database
 It also pushes and pulls user created + user attended events to/from database
 */

#import <Foundation/Foundation.h>
#import "DatabaseHandler.h"
#import "IVLEManager.h"
#import "EventModel.h"
#import "Constants.h"

@interface EventManager : NSObject

//EFFECTS: initalizes the internal data
- (id)init;
```

```
//EFFECTS: initializes the internal data based on the ivle object provided
- (id)initWithIVLE:(IVLEManager*)ivle;

//EFFECTS: returns all the events pulled from IVLE
- (NSArray*)getEventsFromIVLE;

//EFFECTS: returns all the events pulled from database
- (NSArray*)getEventsFromDatabase;

//EFFECTS: saves the given event model by linking with the database
//MODIFIES: event table in database
- (void)save:(EventModel*)model;

//EFFECTS: removes the given event model from the database
//MODIFIES: event table in database
- (void)remove:(EventModel*)model;

//EFFECTS: sends a post request to ivle for the given event model by linking with IVLE manager
- (void)postToIVLE:(EventModel*)model;

//EFFECTS: adds a row in the "Attending" table for which user is attending which event
//MODIFIES: attending table in database
- (void)saveAttend:(EventModel*)model id:(NSString*)user;

//EFFECTS: removes a row in the "Attending" table for the given user
//MODIFIES: attending table in database
- (void)removeAttend:(EventModel*)model id:(NSString*)user;

//EFFECTS: adds a row in the create table to know which user created which events
//MODIFIES: "create" table in database
- (void)saveCreate:(EventModel*)model id:(NSString*)user;

//EFFECTS: removes a row from the create table based on the event and the user
//MODIFIES: "create" table in database
- (void)removeCreate:(EventModel*)model id:(NSString*)user;

@end
```

DatabaseHandler.h

```
/*
 This class handles all the backend requests to send to the database
 It handles insertion, retrieval and deleting of events.
 An edited event is first deleted and then readded in the database.
 */

#import <Foundation/Foundation.h>
#import <Parse/Parse.h>

@interface DatabaseHandler : NSObject
```

```
//REQUIRES: a valid tablename and dictionary of data, where each key is the column name
//EFFECTS: if the tablename does not exist, it will create a new table in the database
//else add to the existing database
+ (void)insertRow:(NSDictionary*)data inTable:(NSString*)tableName;

//REQUIRES: a valid tablename
//EFFECTS: if the table exists, it will return all the rows data in a dictionary form,
//each key is the column name
+ (NSArray*)getAllRowsFromTable:(NSString*)tableName;

//REQUIRES: a valid tablename
//EFFECTS: if the table exists, it will delete the rows whose values correspond to that of
//the dictionary provided
+ (void)deleteRowWithData:(NSDictionary*)date FromTable:(NSString*)tableName;

@end
```

UserInterestViewController.h

```
/*
 This class handles the settings page of the user. It displays the possible Preferences
 along with his selected preferences and send a delegate to the relevant controller
 when the preferenes are selected and done is pressed
 */

#import <UIKit/UIKit.h>
#import <FacebookSDK/FacebookSDK.h>
#import <QuartzCore/QuartzCore.h>
#import "UserModel.h"
#import "UserLoginViewController.h"
#import "FBManager.h"
#import "CategoriesCell.h"

@protocol UserInterestDelegate <NSObject>
//informs the updated preferences of the user through the UserModel
- (void)userInterestsChanged:(UserModel*)user;
@end

@interface UserInterestViewController : UIViewController <UICollectionViewDataSource, UICollectionViewDelegate,
UICollectionViewDelegateFlowLayout, UINavigationControllerDelegate>
@property (strong, nonatomic) IBOutlet UICollectionView *interestCollection;
@property (weak) id<UserInterestDelegate> delegate;

//dismiss the view controller
- (IBAction)doneBtnPressed:(id)sender;

//dismisses the view controller wihout editing any preferences
- (IBAction)skipBtnPressed:(id)sender;


@end
```

UserLoginViewController.h

```
/*
 The class handles the login/logout request for IVLE. It has a delegate to inform
 the respective controller that the user has been successfully logged in.
 Baseed on the login with IVLE, it sends the token to IVLEManager to be saved in
 a file
 */

#import "IVLEManager.h"
#import "UserModel.h"

@protocol UserLoginDelegate <NSObject>
@optional
//sends a delegate when the user is successfully logged in
- (void)userLoggedIn;
@end

@interface UserLoginViewController : UIViewController <UIWebViewDelegate>

@property (weak) id<UserLoginDelegate>delegate;
@property (strong, nonatomic) IBOutlet UITextField *uidField;
@property (strong, nonatomic) IBOutlet UITextField *pwdField;
@property (strong, nonatomic) IBOutlet UILabel *errorLabel;

//EFFECTS: initialises the login with the given usermodel
- (id)initWithUser:(UserModel*)usermodel;

//EFFECTS: tries to log the user in and returns whether successful
- (BOOL)runLoginRequest;

@end
```

UserOptionsViewController.h

```
/*
 This class shows the popover view for the User logins. The two logins shown
 and controlled are facebook and ivle. Based on which login the user chooses, the
 action is delegated accordingly.
 It has a delegate to inform the relevant controller when the user logs in/out of ivle
 and facebook.

 */
```

```
#import <UIKit/UIKit.h>
#import <QuartzCore/QuartzCore.h>
#import <FacebookSDK/FacebookSDK.h>
#import "AppDelegate.h"
@protocol UserOptionsDelegate <NSObject>
//informs when ivle login is pressed
- (void)ivleLoginPressed;

//informs when ivle logout button is pressed
- (void)ivleLogoutPressed;

//informs when facebook button is pressed
- (void)facebookLoginPressed;
@end

@interface UserOptionsViewController : UIViewController <FBLoginViewDelegate>

@property (weak) id<UserOptionsDelegate>delegate;

@property (strong, nonatomic) IBOutlet UIButton *facebookBtn;
@property (strong, nonatomic) IBOutlet UIButton *ivleButton;
@property (strong, nonatomic) IBOutlet UIButton *ivleLoginButton;
@property (strong, nonatomic) IBOutlet UIButton *facebookLoginButton;

//EFFECTS: sends a delegate to inform whether the ivle log in or logout action is to be taken
- (IBAction)ivleLoginButtonPressed:(id)sender;

//EFFECTS: sends a delegate to inform
- (IBAction)facebookLoginButtonPressed:(id)sender;

//EFFECTS: functions to toggle the ivle title (between login and logout)
//MODIFIES: ivle button
- (void)setIvleButtonTitle:(NSString*)title;

//EFFECTS: functions to toggle the facebook title (between login and logout)
//MODIFIES: facebook button
- (void)setFacebookButtonTitle: (NSString*)title;

@end
```

AnnotationViewController.h

```
/*
 This class creates the table that is visible when each annotation pin is tapped
 It has an array (allElements) that has all the elements to be displayed
 It has a delegate to inform the relevant controller when a cell is selected
```

```
  and more details about that view is to be shown
 */

#import <UIKit/UIKit.h>
#import "MapAnnotationModel.h"

@protocol AnnotationViewDelegate
@optional
//informs which cell was tapped in the annotation to display the event detail accordingly
- (void)didTapAccessory:(MapAnnotationModel*)modelTapped;
@end

@interface AnnotationViewController : UITableViewController <UITableViewDataSource, UITableViewDelegate>
@property NSArray *allElements;
@property (weak) id<AnnotationViewDelegate> delegate;
@end
```

MapAnnotationHandler.h

```
/*
 This class handles the adding of each event on the map annotation by parsing
 the venue of each event into the respective location coordinates and adding
 in a dictionary.
 This dictionary has the venue as its key, and all the associated events as the values
 */

#import <Foundation/Foundation.h>
#import "MapAnnotationModel.h"
#import "EventModel.h"

@interface MapAnnotationHandler : NSObject

@property NSMutableDictionary* locationAnnotationSet;
@property NSDictionary* locationCoordinates;
@property NSDictionary* fullList;

//EFFECTS: adds the annotation to the dictionary of events and returns false if location of event is invalid
//MODIFIES: locationAnnotationSet - model is added, with the model venue as the key
- (BOOL)addAnnotation:(EventModel*)model;

//EFFECTS: returns all the events for a given location from the stored dictionary
// it parses the location provided into the key for the dictionary and returns accordingly
- (NSArray*)getEventsForLocation:(NSString*)location;

@end
```

MapAnnotationModel.h

```
/*
 This class is a subclass of MQPointAnnotation (the annotation class of MapQuest SDK)
 The annotation will stores additional details like the actual event at the annotation
 */

#import <Foundation/Foundation.h>
#import <MQMapKit/MQMapKit.h>
#import "EventModel.h"

@interface MapAnnotationModel : MQPointAnnotation
@property EventModel *event; //the event that represents this annotation

//EFFECTS: a new initializer that also takes the EventModel as a parameter
//MODIFIES: the event model of this instance
- (id)initWithCoordinate:(CLLocationCoordinate2D)coord title:(NSString *)aTitle subTitle:(NSString *)aSubtitle
model:(EventModel*)aModel;

@end
```

BusRouter.h

```
/*
 This class computes the route between two given points.
 One point needs to be the user location and the other point can be either a building name
 or the location destination coordinate
 It has a class method that computes the distance between two given location
 coordinates as the coordinates are in longitude & latitude using CLLocation.
 It has a class metho to find the nearest builidng from a given location coordinate.
 */

#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>

@interface BusRouter : NSObject

//EFFECTS: initializes the internal dictionaries by reading all the plist files
- (id)init;

//EFFECTS: computes the route between the two points and returns in the form of a dictionary
//  Keys in the dictionary: "startPoint", "endPoint", "changePoint", "busNumber", "stopNumbers"
- (NSDictionary*)routeBetweenPoints:(CLLocationCoordinate2D)start Venue:(NSString*)end;

//EFFECTS: computes the route between the two points and returns in the form of a dictionary
//  Keys in the dictionary: "startPoint", "endPoint", "changePoint", "busNumber", "stopNumbers"
- (NSDictionary*)routeBetweenPoints:(CLLocationCoordinate2D)start End:(CLLocationCoordinate2D)end;

//EFFECTS: returns the nearest building string based on the location coordinates provided
+ (NSString*)findNearestBuilding:(CLLocationCoordinate2D)location;

//EFFECTS: returns the double distance between two given coordinates
+ (double)getDistance:(CLLocationCoordinate2D)a To:(CLLocationCoordinate2D)b;
```

```
@end
```

RouteViewController.h

```
/*
 This class is responsible for the creation of the Directions view based on the
 directions computed by another class. It takes in the directions and displays the
 details in a table form.
 It has a delegate to inform the relevant controller when the view was closed.

 */

#import <UIKit/UIKit.h>

@protocol RouterDelegate <NSObject>
//inform the relevant controller when the directions view was closed.
- (void)routeClosed;
@end

@interface RouteViewController : UIViewController <UITableViewDataSource, UITableViewDelegate>

@property (weak) id<RouterDelegate> delegate;
@property (strong, nonatomic) IBOutlet UITableViewCell *routeCell;
@property (strong, nonatomic) IBOutlet UITableView *routeTable;

//this route needs to be set when the view is being shown. The table cells are
//created based on this Dictionary route
@property NSDictionary *route;
@property CGFloat viewHeight;

//EFFECTS: closes the route view thats shown
- (IBAction)closeBtnPressed:(id)sender;

@end
```

CalendarManager.h

```
/*
 This class is used to manage the interactions with the iPad Calendar and sync events with the calendar
 It implements a Singleton pattern as the default calendar controls all interactions with iCal.
 */

#import <Foundation/Foundation.h>
#import <EventKit/EventKit.h>

@interface CalendarManager : NSObject
```

```
//EFFECTS: Returns the default calendar of the iPad, by checking for the singleton
//class and returning it
+ (CalendarManager*)defaultCalendar;

//EFFECTS: Adds an event in the iPad calendar of the device with the calendar name 'NUS Reach'
- (void)addEventWithTitle:(NSString*)title startDate:(NSDate*)sDate endDate:(NSDate*)eDate location:(NSString*)location
description:(NSString*)description eventID:(NSString*)eventID;

//EFFECTS: Removes the given event from the calendar of the device based on the provided eventID
-(void)removeEventWithEventID:(NSString*)eventID;

@end
```

KeyboardInputView.h

```
/*
 This class implemens a custom keyboard that can be used as an input view for the
 entering of Start/End Date Time of events
 It uses the Tapku library to show the calendar and uses its delegates to find the
 selected dates.
 */

#import <UIKit/UIKit.h>
#import <TapkuLibrary/TapkuLibrary.h>

@interface KeyboardInputView : UIView
@property TKCalendarMonthViewController *calendarView;
@property (strong, nonatomic) IBOutlet UILabel *timeLabel;
@property (strong, nonatomic) IBOutlet UIDatePicker *timePicked;
@property (strong, nonatomic) IBOutlet UIView *calendarHolder;

//EFFECTS: Returns the current selected date and time in the keyboard
- (NSString*)getDateAndTime;

@end
```

IvleManager.h

```
/*
 This class is responsible for all actions with IVLE.
 It pulls all the IVLE events (through the RSS events) and parsing them in the relevant forms.
 It also saves and loads the user IVLE token once he logs in, or if he has a valid session
 It retrieves the username & userid of the user that is currently signed in
 It has functions to post an event to IVLE
 */

#import <Foundation/Foundation.h>
#import "Constants.h"
```

```
#import "IVLEUserParser.h"

@interface IVLEManager : NSObject <NSXMLParserDelegate> {

    // it parses through the document, from top to bottom...
            // we collect and cache each sub-element value, and then save each item to our array.
            // we use these to track each current item, until it's ready to be added to the "stories" array
            NSString * currentElement;
            NSMutableString *currentTitle, *currentDate, *currentDescription, *currentLink, *currentCategory, *currentVenue,
*currentTime;

    // a temporary item; added to the "stories" array one at a time, and cleared for the next one
            NSMutableDictionary * item;

    // category list
    NSDictionary * catDictionary;
    // current category
    NSString * currCategory;
    id __unsafe_unretained ivleViewUpdaterDelegate;
}

@property(strong, nonatomic) NSString* usrToken;
@property(strong, nonatomic) NSString* usrId;
@property(strong, nonatomic) NSMutableArray* allEvents;

//EFFECTS: pulls all the events from IVLE through the RSS feeds and returns an array of the dictionary
- (NSArray*) pullAllEvents;

//EFFECTS: posts an event to IVLE based on the given parameters of the event provided
//NOTE: it might take upto 6 days for the event to show unser IVLE as it is verified
- (BOOL) postNewEventUsingTitle:(NSString*)eventTitle Venue:(NSString*)eventVenue Price:(NSString*)eventPrice
Category:(NSString*)categoryStr StartTime:(NSDate*)startTime EndTime: (NSDate*)endTime Description: (NSString*)description;

//EFFECTS: saves the usertoken currently present in self.userToken
- (void)saveUsrToken;

//EFFECTS: loads the usertoken currently present in the file in documents
- (void)loadUsrToken;

//EFFECTS: removes the usertoken currently present in the file in documents
- (void)removeUsrToken;

//EFFECTS: based on the session and token, checks whether the user is actually logged in
- (BOOL) validate;

//EFFECTS: based on the session and token, retrieves the username of user logged in and returns it
- (NSString*) getUserName;

//EFFECTS: based on the session and token, retrieves the userid of user logged in and returns it
- (NSString*) getUserId;

//EFFECTS: a test method to check whether IVLE post works or not
-(void) testPost;

@end
```

IVLEParser.h

```
/*
 This class is a helper class for IVLE manager to parse the user data and check for
 various functions. It uses the usertoken to check whether the user is logged in and
 to parse its username & userid
 */

#import <Foundation/Foundation.h>

@interface IVLEUserParser : NSObject<NSXMLParserDelegate>{
    NSString* usrToken;
}
@property(strong, nonatomic) NSString* usrToken;



//EFFECTS: initialises the parser with the usertoken to validate
-(id) initWithUserToken: (NSString*)aUsrToken;

//EFFECTS: based on the url provided, checks whether the usertoken is valid
-(BOOL) parseValidationFromURL:(NSURL*)url;

//EFFECTS: based on the url provided, parses the username and returns it
-(NSString*) parseUserNameFromURL:(NSURL*)url;

//EFFECTS: based on the url provided, parses the userid and returns it
-(NSString*) parseUserIdFromURL:(NSURL*)url;

//EFFECTS: parses the data that is provided and checks whether post was successful
-(BOOL) parseEventPostResponseFromData:(NSData*)data;

@end
```

FBManager.h

```
/*
 This class handles the login/logout and posting events on facebook.
 It also has a delegate to inform the relevant controller that the user is logged in
 and that the buttons should be updated accordingly (title login/logout)
 */

#import <Foundation/Foundation.h>
#import "AppDelegate.h"

@protocol FacebookLoginViewUpdater
-(void)updateFBLoginBtn;
@end
```

```
@interface FBManager : NSObject
@property (weak) id <FacebookLoginViewUpdater> loginButtonUpdater;

//EFFECTS: initalizes the session and sends a delegate based on whether the session is set
- (id)init;

//EFFECTS: returns whether logged in or not
+ (BOOL)isSessionOpen;

//EFFECTS: handles the login and sends the delegate accordingly
- (void)buttonClickHandler:(void (^)(id)) block;


//REQUIRES: user to be logged into facebook
//EFFECTS: shares the event on facebook by posting on the user wall
- (void)shareButtonAction:(NSDictionary*)shareData;

@end
```

LocationPickerViewController.h

```
/*
 This class is used to control the location dropdown, which is presented to the user when creating an event.
 It is a sub-class of UITableViewController and controls the display of the location category table.
 It informs the EventCreateViewController of the user's selection through delegates
 */

#import <UIKit/UIKit.h>
#import "LocationSubCategoryViewController.h"
#import "LocationSelectViewController.h"

@protocol LocationPickerDelegate <NSObject>
//informs which subcategories are to be displayed
- (void) selectedCategoryWithSubCategories:(NSArray*)subCategories;

//informs which locations are to be displayed
- (void) selectedCategoryWithLocations:(NSArray *)locations;
@end


@interface LocationPickerViewController : UITableViewController
@property NSArray* locationCategories;
@property (weak) id<LocationPickerDelegate> delegate;

@end
```

CategoryPickerViewController.h

```
/*
 This class is used to control the category dropdown, which is presented to the user when creating an event.
 It is a sub-class of UITableViewController and controls the display of the main category table.
 It informs the EventCreateViewController of the user's selection through delegates
 */

#import <UIKit/UIKit.h>

@protocol CategoryPickerDelegate <NSObject>
//informs the category that was selected from the dropdown
- (void) selectedCategory:(NSString*)category;
@end

@interface CategoryPickerViewController : UITableViewController

@property NSMutableArray* categoryList;
@property (weak) id<CategoryPickerDelegate> delegate;
@end
```

LocationSubCategoryDelegate.h

```
/*
 This class is used to control the location dropdown, which is presented to the user when creating an event.
 It is a sub-class of UITableViewController and controls the display of the sub category table.
 It informs the EventCreateViewController of the user's selection through delegates
 */

#import <UIKit/UIKit.h>
#import "LocationSelectViewController.h"

@protocol LocationSubCategoryDelegate <NSObject>
//informs about the locations in the subcategory selected
- (void) selectedSubCategoryWithLocations:(NSArray *)locations;
@end

@interface LocationSubCategoryViewController : UITableViewController
@property NSArray* locationSubCategories;
@property (weak) id<LocationSubCategoryDelegate> delegate;
@end
```

LocationSelectViewController.h

```
/*
 This class is used to control the location dropdown, which is presented to the user when creating an event.
 It is a sub-class of UITableViewController and controls the display of the locations table.
 It informs the EventCreateViewController of the user's selection through delegates
 */

#import <UIKit/UIKit.h>

@protocol LocationSelectDelegate <NSObject>
```

```
//informs the location that was selected from the dropdown
- (void) selectedLocation:(NSString*)location;
@end

@interface LocationSelectViewController : UITableViewController
@property NSArray* locationList;
@property (weak) id<LocationSelectDelegate> delegate;
@end
```

Test Cases

Black-Box Testing:

- Test Application Display:
    - The application should be loaded in landscape view
    - Upon rotating the device to portrait mode, the application should remain in landscape mode

- Test EventManager:
    - Test whether events are pulled from the database and IVLE
    - Test whether events are properly filtered according EventFilter pattern

- Test EventFilter
    - Test whether filter pattern affects the events returned by EventManager

- Test EventCalendar
    - Test whether events are properly synced with iCal

- Test EventViewController
    - Test whether event detailed view is displayed properly
    - Test whether event edit view is displayed properly and will correctly update the event details.

- Test EventsViewController
    - Test whether changing event properties will update the respective event properties correctly
    - Test whether event container view is at the right location and displayed properly
    - Test whether the EventListView is formatted properly with all the events loaded according to the filter applied
    - Test whether the EventList is customized for the user and shows the correct user attending and created events/

- Test Events List displays:
    - The table-style list of events should appear, with the user-attended events first, then the user-created events and then all the vents pulled based on the filter. Here, the user has an option of editing his created events or unregistering to his ‘attending’ events.

- Test EventCreateViewController
    - Test whether event creating view is displayed properly

    - Test whether event creating view will create a new event with specified details.
    - Test whether event editing view is displayed properly

- Test Map Display:
    - Test whether the map is displayed when the application is loaded
    - Test that the user is able to zoom in/out of the map using the pinch gesture
    - Test whether the user is able to move around the map using the pan gesture
    - Test whether annotations are shown on the map, with a table filled with events
    - Test whether these annotations are dynamically updated in real time based on the filters are applied
    - Test whether the user location is correctly displayed on the map
    - Test whether tapping the user location opens a popover
    - Test whether long pressing at any point on the map opens a popover with giving the options of creating an event at the location or getting directions to there

- Test Directions Display:
    - Test that navigation directions are displayed properly on the map with the correct buses and number of stops to the destination

- Test User Login:
    - After tapping on the user button on the controller toolbar at the top, a popover should open giving the option of logging in with IVLE and Facebook.
    - At subsequent launches:
      - if the user has already logged into the system, the application should automatically log the user into the system (IVLE and/or Facebook)
      - if the user has never logged into the system, the same login page should be displayed

- Test User Logout:
    - The option to logout should only be display if the user is logged into the system
    - After the user logs out of the system, the initial login page should be displayed

- Test User Preferences:
    - After pressing the settings button in the controller toolbar at the top, a view should be launched that gives the user an option to select multiple interests.
    - The user should be able to go back to the main view (ie. not select any preference).
    - Once a user has selected his preferences, filters should be applied based on these settings
    - If the user is logged in, these preferences should be remembered and loaded each time the user launches the application

- Test IVLE login:

    - Test whether the wrong combination of username & password will successfully login to IVLE (It should not)
    - Test whether the correct combination of username & password will successfully login to IVLE (It should)

- Test IVLE event pulling:
    - Test whether future events can be pulled from the Event Organizer

- Test IVLE event posting:
    - Test whether user can post event to IVLE using the application (Note: the events can take upto 7 days to be published on IVLE)
    - Test whether an event will be posted even if the user does not provide the complete information required for posting

- Test Facebook connection:
    - Test whether the wrong information will connect to Facebook
    - Test whether the correct information will connect to Facebook
    - Test whether user particulars can be successfully loaded from Facebook

- Test Facebook event sharing:
    - Test whether user can share event to Facebook using the application
    - Test whether user will fail post event if requirements is not met

- Test Calendar sync:
    - Test whether the event marked is attending is added to the calendar under the name “NUS Reach”
    - Test whether the event marked is unattend is removed from the calendar

- Test Bus directions:
    - Test whether directions between two points with a common bus is shown
    - Test whether directions between two points with no common bus is shown with a changeover
    - Test whether the nearest busstop is picked from the location provided
    - Test whether the opposite busstop for a location is picked while computing the route

Glass-Box Testing:

- Test EventManager:
    - Test whether events pulled from the database, IVLE or Facebook are well formatted
    - Test whether events are stored to database after event creation
    - Test whether event details are successfully updated after event edition
    - Test whether events are removed from database after event deletion

    - Test whether events are properly filtered according EventFilter pattern by checking the event data against the criteria

- Test EventFilter
    - Test whether filter pattern are well formatted according to selected criteria

- Test EventCalendar
    - Test whether events are properly synced with iCal by comparing the event dates

- Test EventViewController
    - Test whether the event edit view will properly edit the event model

- Test EventModel
    - Test whether event model stored the correct details for an event
    - Test whether event model is updated after event editing

- Test EventsViewController
    - Test whether the list of events for the given user are updated based on the user (attended and created)

- Test EventCreateController
    - Test whether event creation will create a new event model and add to database
    - Test whether event editing will edit the given model and update in database

- Test Map Zoom:
    - Test that the user is not able to zoom in to the map beyond a particular level
    - Test that the user is not able to zoom out of the map beyond a particular level

- Test Map Pan:
    - Test that the map does not move beyond the boundaries of the NUS campus
    - Test that the map does not lag while the user moves it around

- Test Event Display:
    - Test that zooming does not affect the positions of the existing events
    - Test that moving around the map does not affect the positions of the existing events

- Test Directions Display:
    - Test that zooming does not affect the positions of the navigation directions
    - Test that moving around the map does not affect the positions of the navigation directions

- Test User Re-Login:

    - The database should not store any user login related data and the token in the user's device should be overwriten each time he logs in

- Test User Settings:
    - The changes made to the preferences here should be directly displayed on the map as soon as the user closes this window.

- Test IVLE login:
    - Test whether the IVLE token is returned upon successful login
    - Test whether error message is returned upon unsuccessful login

- Test IVLE event pulling:
    - Test whether the returned event response is in the correct format

- Test IVLE event posting:
    - Test whether the post even request is in the correct format

- Test Facebook connection:
    - Test whether the Facebook token is returned upon successful connect
    - Test whether error message is returned upon unsuccessful connect

- Test Facebook event sharing:
    - Test whether the post request of event sharing is well formatted

- Test Calendar sync:
    - Test whether each time an event is added, it is added to the same NUS Reach calendar and only one instance of the calendar is used (singleton)

- Test Bus directions:
    - Test whether the given route contains the least number of changes and busstops for the given two points