Project Title: Plantam
Authored by: Brian Cieplicki, Isabella Dahilig, Bemnet Demere, Anh Nguyen, Binh Vo
Group Number: 4

.CPSC310 Project Report December 19, 2017.

• ***Project Summary*:**

The Plantam app provides a community–within the scope of our project, the Trinity college community–with a synchronized platform of public events for each individual of Trinity to copy into their personal planner with additional helpful planning management tools.

Within the application there are two important features to the user–the personal and public calendar. Each user has a unique account which is tied to a trinity email to ensure privacy, and each user can only view his or her personal calendar. Users can view their personal calendar from a monthly and daily view. Users can also view, create, delete, or modify events shown on the personal daily list. In addition, users can view public events from an event list by categories such as academic, social, sports, etc. Users can also view each public event and copy the public event to their personal calendars. Any user can create a public event.

• ***Problem Statement*:**

Trinity college is enriched with engaging academics, professional athletics, extracurricular interests, and cultural organizations; however, these components, comprised of the core values of the college, fall under various departments of the college, and each department utilizes different platforms of communication, such as social networks or email chains. Information does not permeate effectively throughout campus. As a result, this unintentionally forms niches in smaller branches of the college, and events lose potentially interested participants. Students are deprived of the opportunity to broaden their horizons socially or academically as liberal art colleges advocates.

These setbacks directly impact students' experience at Trinity college and the connections they can create with others on campus. We intend to provide a universal platform with proper publication of all events for students and staff of Trinity college, in order to encourage student involvement on campus. This Android application is designed for an individual of Trinity campus to easily access public campus events and plan their personal time along with it.

• ***Project Features*:**

- High-priority features:
    - add public events to personal calendar
    - sign in for each account
    - view personal daily and monthly schedule

    - view campus events with details (dates, locations, hours)
    - delete an unwanted event from personal calendar
    - edit details of an event in personal calendar (dates, time, location)
    - reminders for personal plans
    - user can add events into the public calendar
- Medium-priority features:
    - create an account for a new user
    - display a list of public events that the user subscribes to
    - display location of events upon the user's request
- Low-priority features:
    - reset password if user forgets

An integral change in our application, which isn't necessarily reflected in the change of our applications' features, is how the user views the public and personal calendars. Our initial designs of the application possessed a key feature in which users can view the personal calendar, the public calendar, and simultaneously view the public and personal calendar. After numerous design attempts, it became apparent that compiling events from two different databases into a seamless calendar, i.e. handling duplicate events, overlapping events, and so on, would be very inefficient. Thus, we decided to completely separate viewing public and private calendars to avoid mixing calendar events and previously mentioned complications. Although this feature was originally high priority, the application still provides an open platform for any individual belonging to the Trinity community can access any public event.

Another relatively large modification was of the initial feature, where an authorized user can create public events. However, issues revolving around what specifically is an "authorized user" arose, whether these individuals should be implemented as a User with additional functionalities of editing public calendar or as an entirely different object, perhaps called Authority. Both options complicated the intricacies of our application. Ultimately, we decided on allowing any user from creating a public event and added an additional field to the Event object, so that the Event with a creator tag is encapsulated and saved into FireBase. Thus, anyone attempting to inappropriate material, is discouraged to do so and will suffer consequences if done.

A few lower priority features were removed to fit the overall scope of our project. The following feature, in which informs the user if other contacts in user's phone have similar activities, was not implemented because the storage of each user's personal calendar information is stored locally, and we also wanted to avoid storing private calendars on the FireBase database to avoid any complications. Thus, we decided it was more advantageous to remove this feature. Another feature, where the user can view open hours of stores and other facilities, was removed due to time.

• ***System Design Overview and Component Diagram*:**
The link below depicts the components of our system and the relationships established between various components.

https://www.lucidchart.com/documents/view/5bf43f2a-f52b-4626-ab66-b7dd9309a7e0

Beginning with the front-end of our system, the User Interface is responsible for allowing the user to view and navigate through the public and private calendars and to provide user input to the AddEvent component and both EditEvent components . When the UI supplies user input of a new event to AddEvent, the AddEvent ensures the validity of user input and encapsulates the input into an EventForm for further encapsulation. The UI component also listens for any changes made to existing events, and when changes are observed, the UI supplies the changed information to the EditEvent component corresponding to the proper database. The EditEvent components verify user input info and directly modifies the targeted event within its database.

The AddEvent components supplies the event information within EventForm such that the Event class can encapsulate the data to an Event object, depicted by the EventForm component. From here, the Event Object can be encapsulated using the FireBase component, specifically the FBDatabase class. The FireBase component then sends the encapsulated event to the FireBase storage. When the Event component sends an Event object to the SQLite database, the SQLite component has an additional class for encapsulating the event into an entry within the SQLite database.

• ***UML Class Diagram*:**
The link below depicts the components of our system and the relationships established between various components.

https://www.lucidchart.com/documents/view/51364a89-bf67-40a6-b48b-411d79af80fb

• ***Software Design Principles*:**

- ***Tell, don't ask*:** The AddEventActivity class allows the user to fill out event information; and the AddEventActivity class utilizes the AddEventTemplate class to check for validity of information. Thus the AddEventActivity does not need to check with the AddEventTemplate on the status of the information provided.
- ***High cohesion, low coupling*:** Our UML diagram showcases the overall high cohesion of classes in particular the interface activity component, in which the tasks of filling out, validating, and correctly setting the event fields are separated into EditEventDialog, AddEventActivity and AddEventTemplate classes. There is low coupling in the code because there is no unnecessary access of information of classes. For example, there are

two very similar objects EventInfo and EventForms but the two are involved in separate components of the program to avoid unnecessary dependencies.

- ***Encapsulate what varies:*** The Event class encapsulates both SQLite and Firebase events into an Event object our application can use, so that other classes using Events are not aware of the details of encapsulating an Event into a specific database entry. In addition, the User class works the same.
- ***Program to interfaces, not implementations:*** The FBdatabase and SQLiteEventDatabase both extend to an abstract database instead of inheriting a database superclass. Because Firebase and SQLite store information differently, extracting and adding information should be handled differently. In addition, the type of information being stored in SQLite is different from what is stored in Firebase, so the two database classes should not be considered the same type of object.
- ***Favor composition over inheritance:*** Except in the cases involving Android programming, composition was heavily prioritized. For example the EditEventActivity and AddEventActivity both use EventForm to modify event information. These classes could have used inheritance to in order to extract information from EventForm, but instead of reduplicating code with inheritance, the EditEventActivity and AddEventActivity classes do not have unnecessary exposure to EventForm fields.
- ***Single Responsibility Principle:*** One example of single responsibility principle is found in the FetchUser class. The User class is used as a component of the Event class, and is also used specifically for logging into the application. When an individual attempts to login, the application checks with Firebase whether this user already has an account. The User class does not always need to fetch User information from Firebase, since the User class is also used in Event class. Thus, retrieving user information is delegated to the FetchUser class.
- ***Open/Closed Principle:*** As mentioned earlier, the AddEventTemplate handles validity of information input from the user, so that AddEventActivity does not have to handle input error control.
- ***Interface Segregation Principle:*** Our application does not necessarily utilize many interfaces; however, since the components such as database storage and user navigation in our application are so different, there can be no “God” object that controls the majority of the classes.
- ***Dependency Inversion Principle:*** The EventForm class translates time from a visual clock the user can click on to a date field into a date object since the higher-level modules, EditEventActivity and AddEventActivity will never need the visual clock component of the EventForm class. While the higher-lever modules use the EventForm, they will never need to know the implementation of it.

- ***DRY:*** Although the User class is used for both events and login activity, two entirely different components, the User object is used for both classes rather than creating separate user objects.
- ***YAGNI:*** Initially we created a DatabaseEntry enumerations class to represent the Firebase event, however once we realized the extraction process for Firebase storage required its own class to encapsulate and decapsulate events, we erased the class.
- ***KISS*****:** Our application focuses on maintaining personal and public calendars, and add-on features such as location, subscription, and notification alarm were added later on.
- ***Dependency Injection*****:** The Event class uses EventInfo and User to instantiate an Event object, rather than instantiating every field such as what an event such as time, date, and location.
- ***Principle of Least Knowledge*****:** Although there is high cohesion, there is no unnecessary communication between classes. For example, the SQLiteDatabase class has a DatabaseHelper, which formats the event into a tabled entry. The DatabaseHelper only assists the SQLiteDatabase class, so the DatabaseHelper doesn't know about the User class or other classes that relates to Event class.
- ***The Hollywood Principle*****:** The FBDatabase uses an observer, and the observer is triggered when change is made. Then, the FBDatabase alerts the dependent classes on modifications that each dependent class should make. The act of notifying dependent classes to take action reflects the Hollywood Principle.
- ***Principle of Least Astonishment*****:** The application classes and its functions are properly named to associate with its function.

• ***<u>Design Patterns</u>*****:**

- ***Observer Pattern***: The FBDatabase class notifies any dependent classes on updates the FBDatabase provided—preventing dependent classes from having to 'ask' for data, instead they should be 'subscribers' to the database.
- ***Strategy Pattern:*** Our application did not have many varying classes of a similar framework, which would need to implement this overarching framework in unique ways. Thus, our application did not require this design pattern.
- ***The Decorator Pattern***: We did not have any instances, where a number of subclasses interact in an overlapping, codependent way in which one's functionality would need to be expanded to incorporate that of others.
- ***Singleton Pattern:*** - The FBDatabase and SQLiteDatabase classes ensures that only one instance of each database is used for our application. The singleton pattern prevents multiple databases from being created and guarantees that all data is saved locally in SQLite or virtually in the Firebase storage.
- ***Factory Method:*** In EditEventDialog class, we implement the factory method newInstance() to create a new instance of the fragment based on the provided parameters.

- ***Adapter Pattern***: We did not encounter any instances in which necessary interfaces were incompatible with each other and had to be altered to allow this interaction to happen.
- ***Facade Pattern*****:** The FBDatabase class acts as a facade for the Firebase system in order to provide encapsulation so that the application that uses Firebase's routines need not know about the technicality of the platform and our application, while taking advantage of the functionalities provided by Firebase, only gets to use a simple interfacea
- ***Proxy Pattern*** - We had no instances in this application design where a particular object needed to be controlled remotely in order to monitor its access and behavior.
- ***Template Method Pattern*** - Seen in the use of the AddEventTemplate class, to validate and extract information from the AddEventActivity and EditEventDialog classes. This is efficient because it delegates this behavior in a separate class in a predefined way.
- ***Iterator Pattern*****:** The only set of iterable elements our application deals with is the set of events stored in a calendar, but because we access these events primarily through our Databases, there is no need to use an Iterator to sift through these elements.

• ***Refactoring and Testing*****:**

From numerous attempts at creating a workable product compatible with two different databases, refactoring code was a regular part of our process throughout the semester. Implementing the most efficient and accurate design was crucial to our application in order to ensure events were saved in the proper storage database as well as retrieved appropriately. For example, in the beginning, the private and personal calendar was just a calendar object, and the Event object was split into two type of Events, personal events and public events. Thus within our code we were extracting events from Firebase and encapsulating them to the SQLite database in order to display all events. However this seemed redundant, so this is when we decided to split up SQLite and FireBase databases to reduce complications and ensure user privacy.

For testing purposes, we tested each functionality in isolation, ensuring that as we added functionalities, each one was validated. Fortunately, testing with user interface helps with discovering bugs. However, once we began adding multiple functionalities at a time, then runtime errors were appearing. To ensure functionality is working, we would attempt different functionalities at in random order to see when and where the code will break. In addition, we ensured that the basic navigation through the application worked so that the high-priority features were working.

• ***Technology Requirements*****:**

- *Android Studio*:
    - Android studios will be used to handle front end development. We will use XML to layout the user interface components and Java to provide functionality.

- *Firebase* is a web application for mobile development :
    - Firebase Database stores a list of public events by category. Any user can create a public event, or add public event to his or her private calendar.
    - Firebase Authentication allows our app to securely save user data such as account information.
    - Firebase Cloud Messaging provides a reliable delivery of messages used for notification and subscription components of our application.
- *SQLite* is an in–process library that implements a serverless SQL database engine.
    - SQLite stores added events and provides for additional functions such as removing and modifying stored personal events. In addition, SQLite retrieves events for display in calendar view.
- *Google Mobile Services* is a collection of Google API's that help support functionality across devices.
    - Google Mobile Services provides location details such as displaying global coordinates on a map for each event. The user has ability to view location on a map with this added library.
- *Jsoup* is a java library used for extraction and modification of HTML web documents.
    - Jsoup parses through the Trinity college website to identify and extract public events.
- Hardwares:
    - Android phones and Laptops

• *Team Contributions*:

Everyone contributed equally because our software needs us to talk and be interactive with each other, especially planning out our designs. The most important component in this project is working with databases. We decided not only storing data locally but also storing it online. We decided to split our groups into two groups:

1. SQLite Local Database(Anh, Brian)
2. Firebase Realtime Database(Binh, Bemnet, and Izzi)

We would have weekly meeting to talk about our progress and updates on the database. During this phase, we would research about our respective assigned database and implemented it by using their API. Each group would meet up on their own time to talk about their designs and implementations. When implementing, organizing data, pushing the most important features or designs onto the database, we usually meet up and implement the designs so everyone can understand and progress at the same pace.

Other features within our app such as Notifications(Brian), Webscraping TrinityToday & Sync events to the local database(Anh), User Authentication (Binh), View Layouts (Izzi), Events (Bemnet). We would like to give thanks to Bemnet. He plays an important role in our project because of his experiences working with group projects. He is like a 'Scrum Leader' who would

try to keep everyone on pace. We would use Trello to set out TODO tasks and assign them to each member. Overall, our group has learned a lot from working with each other. Whether what results we'll get, we can all honestly say it was an experience of a lifetime. Even though at times where we felt it was impossible, we trusted each other and have opened ears for feedback. The project taught us a lot. It wasn't only about being a better programmer, but a better friend.