List of Features

Organizers

- **View All Events**
    - Sign up for an event
    - Cancel the event
    - Schedule new events
    - Assign speaker to event
- **View My Events**
    - Unregister for an event
- **Create Account**
    - Create a new Attendee, VIP Attendee, Organizer, Speaker account
- **Manage My Account**
    - Change your name
- **View Contact List**
    - See Chat History
    - See Unread/Read Messages
    - Set messages as unread
- **Social Networking**
    - Recommends friends based on number of common events
    - Message recommended friend
- **Create Room**
    - Enter a new room into the system
- **Message All**
    - Message all Attendees, Speakers, VIP Attendees
- **Analytics**
    - Provides some stats about the current conference

Attendees

- **View All Events**
    - Sign up for an event
- **View My Events**
    - Unregister for an event
- **Manage My Account**
    - Change your name
- **View Contact List**
    - See Chat History
    - See Unread/Read Messages
    - Set messages as unread
- **Social Networking**
    - Recommends friends based on number of common events
    - Message recommended friend

VIP Attendees

- **View All Events**
    - Sign up for an event
- **View My Events**

    - Unregister for an event
- **Manage My Account**
    - Change your name
- **View Contact List**
    - See Chat History
    - See Unread/Read Messages
    - Set messages as unread
- **Social Networking**
    - Recommends friends based on number of common events
    - Message recommended friend

Speaker

- **View My Events**
    - Unregister for an event
    - Message all attendees in your event
- **Manage My Account**
    - Change your name
- **View Contact List**
    - See Chat History
    - See Unread/Read Messages
    - Set messages as unread

Design Patterns

- **Dependency Injection:**
    - **Why:** We used dependency injection in order to make our code in certain classes less coupled
    - **How:** We implemented dependency injection when we create an Object outside a class, and then passed into the class to use (in constructors, methods, etc.).
    - **Where:**
        - This happens in all the classes in the conference directory (View in MVP pattern). We passed objects (UserController, etc.) using the .putExtra() method between these classes so we didn't have to read/write to our files and instantiate new objects everytime we go to a new activity in our app
        - Dependency Injection also happens in the presenter directory (Presenter in MVP pattern). In some methods we pass in objects of use case classes to decouple the classes
        - Dependency Injection also happens in the use case classes. In some methods we pass in an instance of an entity in order to update/change values in entities.
- **Iterator** (in Phase 1)**:**
    - **Why:** We wanted a way to iterate over the user prompts read in from a file to make login more user friendly
    - **How/Where:** Implemented Iterator<String> in the UserPropertiesIterator class

Reasons For Not Using Certain Design Patterns

- **Simple Factory/Abstract Factory Design Pattern:**
    - We decided as a group to not use a factory design pattern because we didn't have any complex inheritance structures in the entity layer
    - For example, we only had a User class in the entity layer and we distinguish between the different types of Users in the use case layer (AttendeeManager, OrganizerManager, etc.)
    - We implemented Event in a similar way as well
    - So when we were creating the various User accounts/Events, we didn't have to potentially interact with many different classes (just create type User/Event)
- **Strategy:**
    - In our code, there was one potential place we thought of to use the Strategy Design pattern
    - In our Analytics feature, we use the built-in Collections.sort() in order to sort all the events based on number of attendees
    - We thought it was not necessary to utilize the strategy pattern since we were only sorting with respect to number of attendees and we did not necessarily need multiple implementations of sort

<u>Design Decisions</u>

- **Model View Presenter:**
    - We decided to implement this architecture because we decided to use an Android GUI
    - **Model:** model directory, underlying program (mostly from phase 1)
    - **View:** conference directory and layout xml files, what the user sees
    - **Presenter:** presenter directory, responsible for communicating between View and Model ("controller" in Clean Architecture)
    - In order to continue to follow Clean Architecture, we had the classes in conference directory (View in MVP) implement and depend on an interface from Presenter classes (Dependency Inversion)

- **Distinguishing Types at the Use Case Layer:**
    - Rather than have classes for Attendee, VIP Attendee, Organizer and Speaker, in phase 2 we decided to just have one class for all objects (User)
    - We distinguished between the types of users by storing them in the appropriate use case class (AttendeeManager, VipManager, OrganizerManager, SpeakerManager)
    - We made a similar decision for the distinguishing between types of Events (VIP and normal events)

- **Fixed the Controller Layer:**
    - In Phase 1, we made the mistake of having the Controller classes depend on Entities
    - This time around, Controller classes only interact with the user ID (integer value) when we needed to refer to an Entity object

- **Changed ArrayList to List**
    - Changed the types of our list variables to List<> to be more abstract