**Team YMCA - Design Manual**

Authored by Charles Hennessey and Yash Mittal

**Introduction**

For our CSCI205 final project, my team and I built Hana, a multi-purpose productivity app that allows users to leverage different physiologically verified techniques in their daily lives. For our purposes, we needed to deliver a software product that users can easily use at moment’s notice, carry around with them always, and provided a degree a customizability that the market doesn’t freely provide. To satisfy these requirements, we choose to adopt the Android mobile platform. Android powered smartphones provide developers with a rich library of tools that enabled our team to deliver the features that both the customer and development team felt where necessary.

Hana was built using the Model-View-Presenter (MVP) architecture. MVP allowed development of the app to be modular, scalable, easily testable, and most importantly, clean. While many popular and modern applications choose to use various frameworks such as RxJava or dependency injection like Dagger2, we choose to keep the design as close to Java as possible. This ensured that each member of the team, regardless of prior Android experience, could make effective contributes after learning the basics of Android development. Our implementation of MVP had several basic assumptions:

* The model stored our data, and was only accessible through our presenter.
* Presenters serve as the connection between the model and view. They contain logic that decouples events and interactions that the view intercepts, and serve to preserve state. Presenters can exist without a view(technically), but views cannot exist without their presenter. Because we choose not to use dependency injection, Presenters, like all objects, are created in their view’s respective onCreateXXX() method.
* Views are the logic that controls an XML layout. Thus, notification, activities, dialogs, and fragments are all considered views. When designing our views, we allowed them to service and control logic that explicitly defined what user saw. So, if one needed to change screens, or update text on screen, the view would handle such occurrences. All other logic was passed to the view’s presenter.
* All views have references to their corresponding presenter.

With these assumptions in mind our development process became very stream-lined. A common architecture meant that each member of our team understood where each dependency existed for each module. Any new addition, improvement, or tweak to one module has no effect or barring on any other part of the application. Thus, our team could divide and conquer the different features needed.

Hana also employs two additional patterns, the Utility and Observable pattern. The Utility pattern is a given, but the application uses three different interfaces that when implemented, listen for an action triggered by another class. Usually, these reactions are to events the user would trigger when interfacing with view elements on the screen. For instance, if one were to dismiss an element on the screen, it would trigger an event and all listeners will receive it and process it as such.

With a basic understanding of how we structured our app, it is appropriate to mention briefly how the Android operating system works regarding our application. When any application is launched, what one sees on the screen is either an Activity or Fragment class. Forgetting about MVP for a moment, these serve as the basis for any application. Both Activities and Fragments exist and cycle through life-cycle methods. These various methods are triggered by the OS depending on the state of the application. For example, if the app is alive, but not visible, then it is considered to be in a paused state, thus onPause() would be called. While both Activities and Fragments have life-cycle methods, they are independent of one another. Generally, Activities house Fragments; thus, the developer must pay respect to each life-cycle as they flesh out their application. During the first life-cycle event, onCreate()/onCreateView(), instance variables, and the XML reference are created. A java developer can draw an analogy to the main method, because is it here that the program begins its core logic depending on architecture.

Android has several folders that house resources reveleant to the application. These can include layout files (XML), icons, pictures, color specifications, or anything else that the developer might need. The most important aspect for this paper is the XML layout. Like JavaFX, XML in Android is where the developer defines the aspects of the user interface. They have a choice of manually writing the view elements, or using the built-in interface builder to drag in view elements as set their properties in the properties menu. When building XML layouts, unless the developer wishes to keep their visual static, they must define an ID, in text, to that XML view element to have access to it in code. In JavaFX, one can simply build a controller with method calls to each UI element based on their ID. In Android, one must specify an ID, define the object reference in code based on the XML, and then assign a listener of sorts depending on the type of view object. Android heavily uses the Observable pattern, thus most view elements have an interface or corresponding anonymous inner class that one can use to react to a view event.

Our app currently serves the following features: Get it Done (GTD) by David Allen, the Pomodoro Technique, and the 10-minute hack. When the application is launched, the user is greeted with a welcome screen, followed by being presented with the GTD module.

//Talk about android artectiure.

**User Stories**

**OOD**