**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.

Android applications are built upon several elements that work by themselves or together. When any application is launched, what one sees on the screen is either a product of an Activity or Fragment class. These serve as the basis for any application and can handle all business logic and view interactions. Both Activities and Fragments exist and progress 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 and 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. It is paramount that all developers pay respect to each life-cycle as they flesh out their application. Null pointers can be frequent if the developer is calling on a Fragment from an Activity. 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 also allows developers to creates classes that execute tasks, handle business logic, or interpret view interactions, all while not being bound to the screen. These are known as services. Like Activities and Fragments, services have their own set of life-cycle methods and are functionally similar. Developers employ services to complete a task that doesn’t depend on user interaction. For instance, if there was a need to download a file over FTP, then the system would spawn a service to achieve this task. Regardless of whether the user navigates away from the application, the actions of the service wouldn’t be hindered. This is different from Activities and Fragments, because their state changes when the user navigates away from the application.

Android has several folders that house resources relevant 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 and 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, so that the developer can target the view 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 relies on 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.

Now that the basis of android is understood, we can talk about how we built our application. Hana was constructed 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. In sum, our application uses: Activities, Fragments, and Services as well as the Utility, Observable, and MVP patterns.

**User Stories**

1. I want to quickly add a task
2. I want to be able to add a detailed task
3. I want to be able to use the Pomodoro technique
4. I want to track my Pomodoro timer from the notification bar
5. I would like to be able to delete or complete tasks
6. I want to see an encouraging welcome screen
7. I want to order the tasks I add by priority
8. I want to see an attractive UI
9. I want to utilize the “10-minute hack”

Our app currently serves the following features: Get it Done (GTD) by David Allen, the Pomodoro Technique, and the 10-minute hack. User story 6 & 8 are achieved when the application is launched. The user is greeted with a welcome screen, followed by being presented with the GTD module. The view was custom tailored by Malachi and Aleks to be both visually stimulating and pleasing. They did this by addressing XML elements and their underlying view code to follow our color and animation scheme. Stories 1, 2, 5, and 7 are handled by the GTD module and its subsequent sub-views. The application allows seamless use of the module regarding settings, whether it be obtaining or viewing the tasks he or she created. Stories 3 & 4 becomes active when the user navigates from any module to the Pomodoro module, and initiates a new countdown. This spawns a service which keeps track of the state of each view (screen and notification), and works through the presenter when storing user settings. When launched, the user can configure and control the timer from the screen, or through notification. Lastly, user story 9 provides a simple interface to turn on or off the module, as well as defining when the user should be prompted.

**OOD**