Team YMCA - Design Manual

Authored by Charles Hennessey

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 and provided a degree of customization 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 in a system. When any application is launched, what one sees on the screen is either the product of an Activity or Fragment class. These serve as the basis for any application and can handle all business logic or 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. By design, Activities house Fragments. It is paramount that all developers pay respect to each life-cycle as they flesh out their application. Unlike development for desktop or enterprise applications, mobile must be able to react and handle situations that conserve power, memory, and CPU usage. While Activities and Fragments offer powerful ways to process user interaction, it is not the only method developers can employ.

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, but certainly can be a result of one. 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. 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 frameworks 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 contributions 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 chose 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, notifications, activities, dialogs, and fragments are all considered views. When designing our views, we allowed them to service and control logic that explicitly defined what the 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 handling 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 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

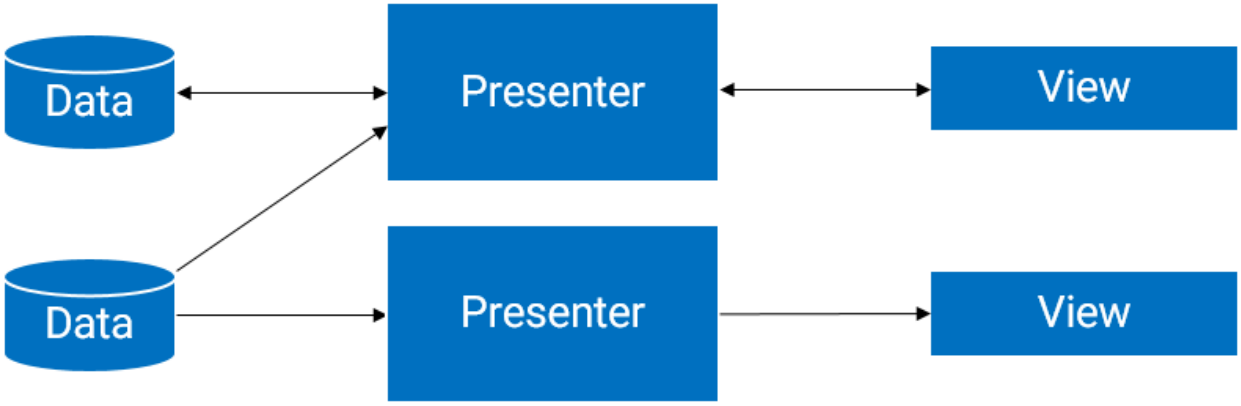
1. I want to quickly add a task
2. I want to be able to add a detailed task
3. I want to be able to use the Pomodoro technique
4. I want to track my Pomodoro timer from the notification bar
5. I would like to be able to delete or complete tasks
6. I want to see an encouraging welcome screen
7. I want 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 Things Done (GTD) by David Allen, the Pomodoro Technique, and the 10-minute hack. User stories 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. User stories 1, 2, 5, and 7 are handled by the GTD module and its subsequent sub-views, which were implemented by Yash and I (Charles). The application allows seamless use of the module regarding settings, whether it be obtaining or viewing

the tasks the user created. Stories 3 & 4 become active when the user navigates from any module to the Pomodoro module and initiates a new session. 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 a Pomodoro timer from the screen, or through the notification. Lastly, user story 9, built by Malachi, implements the 10-minute hack. This module provides a simple interface to enable or disable itself, as well as defining when the user should be prompted.

OOD

MVP and its challenges



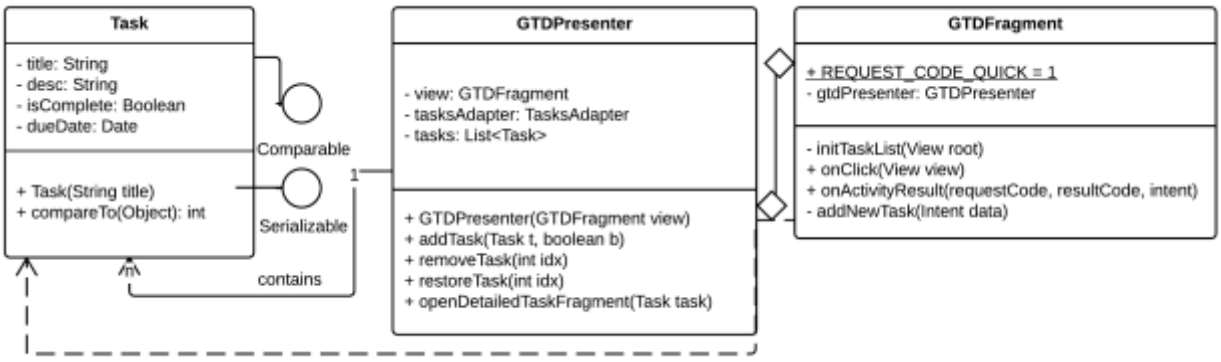
As stated previously, our application uses the MVP design pattern, but we also chose to not implement popular libraries such as RxJava and Dagger2. While challenging, this allowed a lower barrier of entry for non-Android developers, but required more experienced team-mates to be clever with their implementation. In a classical MVP android application, the presenter should exist outside of the Activity/Fragment lifecycle, and instead will persist whether any configuration changes occur (destroys views), threads are stopped, or the device runs out of memory. In addition, the presenter should be flexible enough that it doesn't store an object reference to the view itself to ensure that no memory leaks occur. These events are common in modern android development, and have plagued professionals for years.

However, we moved forward with our decision and decided to promote an encapsulation strategy upon the creation of a new module. Our team experimented with several hacker solutions to simplify the usage we needed. We found that it was not viable to pass object references of the presenters with active view variables within them. This occurred because to pass data references around in Android, you create a class of type Intent and pass a bundle of objects that a view can interpret on creation. Views themselves cannot be serialized, which resulted in subsequent null pointer errors. Ideally, when the application is loaded, our master view the NavActivity, would interpret user selection of a module, create and instantiate the new view, and bind the correct presenter to the view. Because this was not possible, each view had to encapsulate itself from the others, which meant that it was responsible for the creation of its presenter. With this interpretation, the NavActivity was only responsible for handling views related to the Fragments. Thus, we didn't compromise the idea of MVP, rather we made it more modular.

Another challenge relating to this issue dealt with Fragment persistence and handling the life-cycle events of all the sub-views each Fragment had to display. Originally, we asserted that upon any change of the view, the old view would be destroyed and replaced with the new, user selected

replacement. This worked fine until we needed further persistence after the user chose to leave a given module. The solution arrived when the decision was made to not replace views, but rather use a system of showing and hiding. To do this, the NavActivity had to create and hold the references of each Fragment during application start up. Those objects would never mutate or be destroyed during runtime. This differs from before, where we only created references when we needed them, and destroyed them when we didn't. By storing the Fragments and not destroying them, we did not have to explicitly handle the destruction of views so frequently. This significantly reduced the number of null-pointer references and memory leaks in our application.

GTD



Model:

Starting with GTD, the team decided that it was important to provide local persistence in the form of a SQL database for the application. Specifically, the GTD module required the team to store a user defined task, as well as subtasks (U.S. 1 &2). To make things simpler, we used a third-party application known as SugarORM, which is a wrapper for a local SQL-Lite database. To use SugarORM in our application, we simply had to extend the classes that mocked the data object that we wished to create. This allowed the team to employ functionality that explicitly defined the specific characteristics of our data(tasks), and the proper functionality needed to create, add, update, and delete said data. Thus, both Task, and Subtask, extend from SurgarORM. There was no need to manually create or drop SQL tables. Instead, if we wanted to create a Task, we created the appropriate object, passed all relevant member data if supplied, and called save().

When the user indicated that they wanted to save a Task, we used a local array-list data structure that stored a list of Tasks. This is by convention when using a RecycleView, which we will discuss later. Adding, creating, updating, and deleting tasks meant we referenced the specific task in the list by index, and called the proper method depending on operation (U.S. 5). If we needed to create or destroy a Subtask, we executed a similar process, but instead we specified a subtask id to target. Whenever the user leaves the application, all active Tasks are already stored in the local SQL database. Upon restart, these are loaded by the presenter, and populated accordingly by index, or by priority defined in a Task onto the screen.

Presenter:

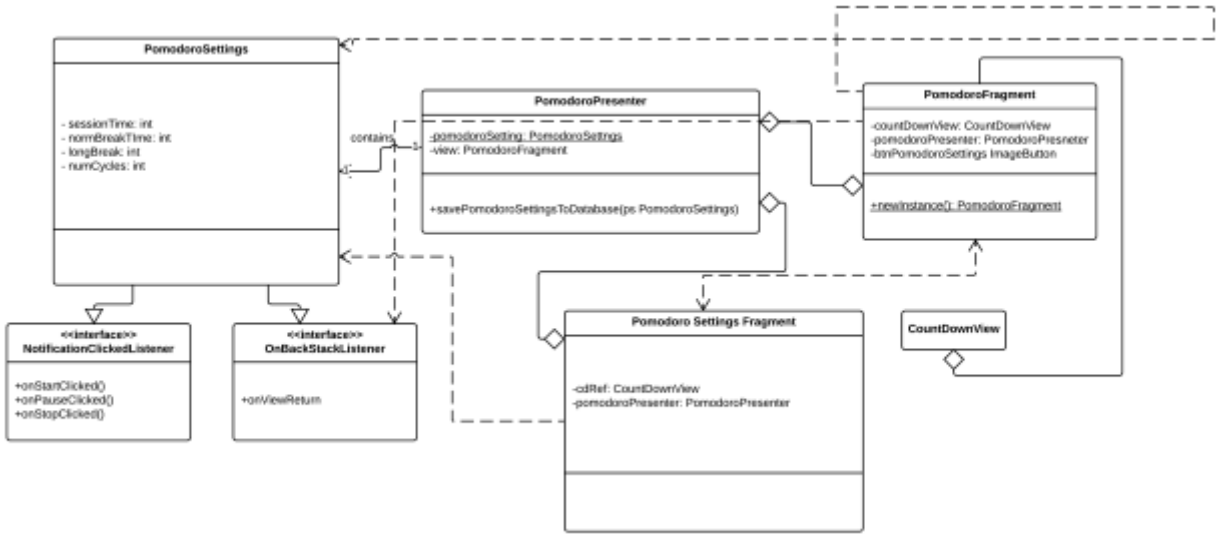
The GTD module has two presenters, the GTDPresenter, and the DetailTaskPresenter. These serve similar functionality in that both respond to view events that concern data. We

separated their functionality because it is possible for a Task to exist without a Subtask, but the latter isn't true for the former. If the user choses to invoke functionality related to the subtasks, the GTDPresenter will respond by populating the screen and allowing the user to make any changes as they wish. We stated previously that the view handles and processes view interactions, but this is a data driven event, and is processed as one because the outcome of this process involves a change in the dataset.

View:

Because of the nature of Android development, there are quite a few elements that compose the view portion of the GTD module. For our purposes, we will focus on the core element, the GTDFragment. This Fragment shows a floating action button, which reacts to user input when clicked. From there, the user has a choice to add a Quick-Task, which is one with no details other than a title, or one that is more fleshed out. Depending on the decision, the system will react by informing the model that the Task needs to be stored, or that we need to show a view that allows for further details before saving. Once this is complete, the presenter will respond to the GTDFragment, and inform it that the underlying data set has been updated. The data held in the model is connected to an adapter (stored in the presenter) and allows the facilitation of it to appear on the screen. Because we list out each Task, it is by convention that we employ an underlying array-list, connected to the adapter when using a RecyclerView. Any click in the list will invoke a system response that supplies an integer index of the position on the screen. This is the same index as the array list, making access trivial.

Pomodoro & CountDown Library



Model:

The class that represents the model of the Pomodoro module is PomodoroSettings, which holds the default configuration any user needs when using the technique for personal use. The constants defined reflect the research we obtained regarding the topic. Unlike the GTD-Model, there is only one stored database reference that needs to be accessed, and other operations such as save() and delete() remain the same.

Presenter:

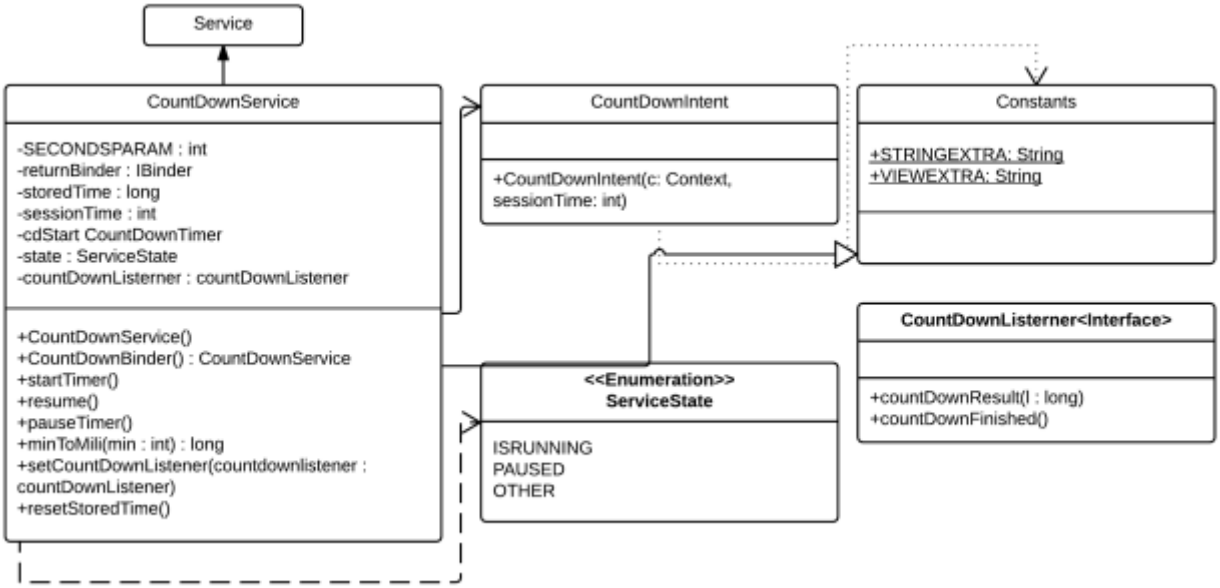
The Pomodoro module only has one presenter, and is quite simple. It facilities the updates needed to the local model, and responds to the CountDownListener interface, which is a component of the CountDown library which we will discuss shortly. Whenever a user completes a work session, break session, or cancels a session, it advances the counter. This internal counter is used to determine which of the three sessions is next, depending on their usage.

View:

The primary view of this module is the PomodoroFragment, and it is responsible for two subviews: PomodoroSettingsFragment, and a custom view, the CountDownView. To reduce complexity, we chose to build in most of the view logic directly into the CountDownView (U.S. 3). Thus, the PomodoroFragment is only responsible for reacting to a button click for the settings and resetting the UI upon session completion, or after a settings update. When the settings button is clicked, we use the PomodoroSettingsFragment to overlay the PomodoroFragment view with several options that allow the user to define a custom session. Consistent with the approach deployed within the rest of the project, any update of data from the view makes it to the presenter and into the database.

The CountDownView is a custom view, implementing its own logic, a series of listeners, and handles its sister-view, the notification. When this view object is created, it acquires the correct settings from the presenter, and implements the CountDown library to track progress. At the same time, a custom notification is spawned, with functionality that is directly tied to the CountDownView. We needed to ensure that responses from the notification (start, pause, and cancel session) were valid and consistent with the view displayed on screen (U.S. 4). Thus, the CountDownView facilitated all interactions of those events. So for instance if the user choses to pause the timer, whether it be from the notification or the view itself, the corresponding onPauseClicked listener method reacts to that event and processes it.

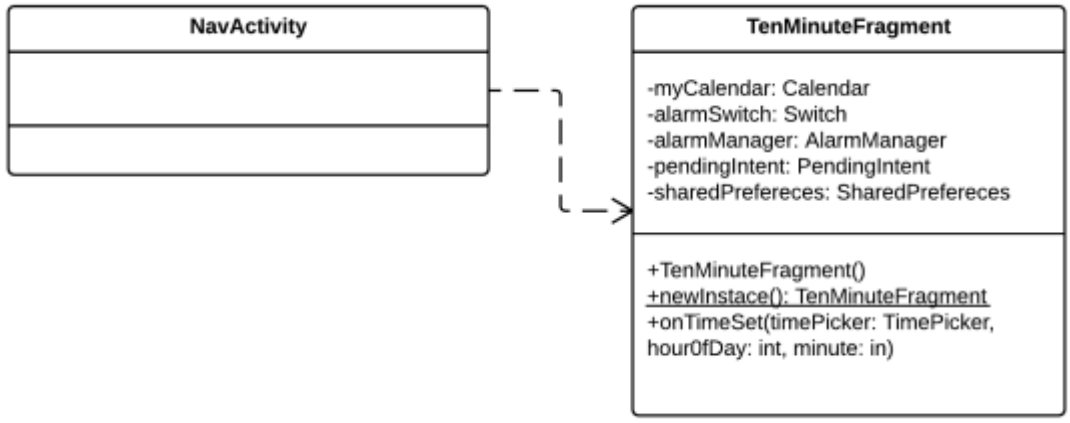
Library



The CountDown library was created when we realized there wasn't a sufficient open source solution that satisfied our needs. We needed to develop a creative solution that allowed the processing of time to continue regardless of the state of our application. To achieve this, we built a custom Android service that utilizes the CountDownTimer, an abstract class that's part of the Android SDK. Implementing a service was necessary because we didn't want this to be bound to any specific view; rather we simply wanted to react to an update. This meant that if a view were to go through its full life-cycle of being created and destroyed, the service would remain alive. Reconnecting to the service and correctly updating the views became a seamless process.

Usage of the service is quite straight forward. When the developer writes code that successfully connects to the service, they have access to starting, pausing, and stopping the timer. Additionally, it supports dynamic updating of the internal clock (if we reset the timer in Pomodoro for instance). During each of these three methods, we need to re-create a new instance of the internal CountDownTimer. This is a limitation of the library because CountDownTimer itself is an abstract class and cannot be instantiated. It works by creating a new session, defining two anonymous inner-classes. onTick() is the update of time in milliseconds, and onFinish() is called when the CountDownTimer is complete. To make this compatible with *any* implementation, we defined a CountDownListener, which defines similar methods. The trick here was getting an abstract class to act like a formal object, and using the service to handle all subsequent interactions. The library also provides an enum, ServiceState, that allows the developer to easily check whether the timer is in a running, paused, or other state. This was useful if one wanted to reconnect to the service and resume where they left off. Such was the case earlier in our project upon Fragment destruction, for instance.

10-Minute Hack



Model, Presenter:

There is no model or presenter in this module.

View:

The primary element of the module is the TenMinuteFragment, which processes whether the user wants the module enabled, and when they want their notification delivered. Why no presenter or model you might ask? This is because the team chose to utilize the alarm manager and the shared preferences manager. Both are system wide services that all applications and developers have access too, thus they are not application specific. Hence, there is no need to implement a model or a view, because we are not explicitly defining data, but a system service.

If the user choses to enable this module, they very simply define a system level alarm that gets processed by the alarm manager. When triggered, a notification will appear informing the user that one needs to process their time consistent with the requirements of the 10-Minute hack (U.S. 9).

Conclusion

Overall, the project went well. Throughout this entire process, we heavily relied on documentation, stack overflow, and expert opinions such as Common's Ware Busy Guide to Android Development. I provided a series of videos throughout the development on my contributions and explanations regarding the code design. If you wish, you can observe the beginnings of our application through my view-point. Some videos are educational, others are walk-throughs, and several were live coding.

YouTube Links:

- https://www.youtube.com/watch?v=Kcxkwxco4Xc
- https://www.youtube.com/watch?v=bQvIFtTm198
- https://www.youtube.com/watch?v=7m7L1_lD5cY
- https://www.youtube.com/watch?v=Fj-xqMTypY0
- https://www.youtube.com/watch?v=mzCcbtXPsWA
- https://www.youtube.com/watch?v=gG1nuORDSHY
- https://www.youtube.com/watch?v=WAW5jTZTzgY
- https://www.youtube.com/watch?v=_2OZop-wMzY
- https://www.youtube.com/watch?v=72GHIBw-UWs

Favorite resources:

- https://github.com/konmik/konmik.github.io/wiki/Introduction-to-Model-View-Presenter-on-Android
- https://github.com/commonsguy
- http://stackoverflow.com/users/115145/commonsware
- https://guides.codepath.com/android