ILLO ARCHITECTURAL DESIGN

Root Digital

The Layered architectural pattern is what will be used in the development of Illo. This pattern involves separating and organizing your applications different systems into separate layers. This is to prevent the issue of layer interface changes only affecting the adjacent layer. This architectural pattern is most commonly associated and used in the development of mobile (or in this case, android) applications. In a very simplistic summary, Android Applications making use of layered architecture will have a UI and Data Layer. Though, additional layers depending on the app’s needs are not uncommon. This setup works very well with the concept of Illo, and will be very helpful during the development of our application.

When looking at the simplified version of layered architecture, it’s easy to picture exactly how our app will fit into the mold. In a very basic sense, the user will interact with the UI layer. This layer will feature a heavily abstracted version of our application’s various systems. The UI will feature three common screens that the user will swap between in order to access different features. Things such as statistics, the timer, and settings. The UI layer will simply house abstract glimpses into the systems at play under the hood. The statistics menu will borrow from the data layer, many settings will simply change behavior, and could fall under the business/domain layer, while all settings will have to be remembered and as such, the statuses of these settings will be remembered through files read by the data layer. The timer will be displayed as a string to the user, but under the hood will be constructing this string based off of some other method not displayed on the UI (such as counting each individual frame to divide the number into chunks for different units of time).

Each system of our app will require multiple classes and entities in order to work. Each layer will house its own major concepts which will be handled by a few abstract classes, with, potentially, a few smaller subclasses that will determine the behavior of the app. On the UI layer, since this is mostly used for menu navigation, that is the major system that will define this layer, with an abstract class of menu nodes defining the basic outline of how the menu will operate. From this will come different subclasses, these will be the different types of menu nodes the user will interact with, things such as buttons, switches and sliders. Each of them will determine the basic functionality of their specific subclass, but not assign any particular behaviors to themselves. That will be for individual entities to define for themselves.

The domain layer will be the messenger between the UI and data layers. Essentially, the UI layer is made to work in a way that is understandable to the user, while the data layer is made to work in a way that is more understandable to the device. The domain layer is what will assist in joining these two systems, while also carrying much of the logic for them. This layer will include a very general class as a translator between the layers, which will be extended by two subclasses meant for each direction, UI 🡪 data, and vice versa. From here, individual entities will be used to define where to receive information, how to translate it, and where to send it. The other major component of this layer is the logic it will handle. Since this layer has direct access to both of the other layers, this is the best place for the logic of each system to be handled. This part of the layer will be mostly determined by the individual entities within it, defined as functions within them to be called when necessary. These functions may or may not be called as part of the translation process, depending on how each system is set up. In this way, both aspects of this layer are very closely tied together.

The data layer, as mentioned before, will be mostly written in a way for the device to understand and store data for use in the two other systems. Classes for data storage, including individual data points, alongside ways of storing said data for easy access will need to be created. A subclass will need to be created whose specific purpose is to read from/write to files. That way data can be remembered between sessions. Once all three layers are put together, you can see how they would all interact. For example, when the user presses the play button on the timer screen, the play button entity will inform the ticker entity on the domain layer, which will begin the timer’s counting function, telling the data layer to add 1 for every tick. This number in the data layer will be taken by a translator on the domain layer, translated into a readable string for the user, and displayed on the UI’s timer entity.

