My implementation of the People Counter app uses the MVVM (Model-View-ViewModel) architecture, where I specifically used Jetpack Compose for the UI and SharedPreferences to ensure data persistence. I decided to use MVVM for several reasons, one of which being that to make sure that the UI and counts are able to persist across app sessions as well as configuration changes. The ViewModel efficiently handles that, and the Model/Repository handles data persistence. On the other hand, for example, in MVC, I would have to manually handle data persistence through configuration changes. Additionally, with MVVM, I was able to use LiveData, which allowed me to automatically have the UI reflect changes in counts or text color, whereas with MVC, I would have to call UI update methods for those data changes. Furthemore, with an architecture using a ViewModel, I was able to manage and calculate different state dependences given in the spec, such as as the '-' button visibility (if the current number of people is greater than 0), or the changing text color (purple and red depending on if the current number of people is greater than 15). In addition, it was more efficient to separate the logic in a ViewModel file than in MVC, where the View directly manipulates the Model, as my MVVM implementation made my code easier to test and debug.

Throughout the building of this application, I created three explicit packages (view, model, and viewmodel), as seen in my codebase, to distinctly separate each of my layers.

Looking at each layer:

Model:
My model layer can be seen in the *PeopleCounterRepository.kt* file, where I chose to implement a repository rather than using SharedPreferences within the ViewModel itself, as I emphasized abstraction throughout my implementation, as I had the model handle the data and its related operations, and abstracted the storage implementation details. This is seen through my design decision to only allow the ViewModel to access save and load operations, and the Model interacts with SharedPreferences. This accounts for future scalability of this application, as if we were to need to change it to a different database (Room, for example), then we would only need to change the Model file, and everything else would still be functional as is.
Additionally, while choosing between SharedPreferences and DataStore for data persistence, I chose to use SharedPreferences because this application has limited data complexity (only storing two values) and has a more straightforward implementation with fewer dependencies. I also referenced this article to help me make a more informed decision:
https://medium.com/mobile-innovation-network/sharedpreferences-vs-datastore-choosing-the-right-one-72f0789c956c
However, if I were to change to DataStore, I would only need to change the Model layer, which again is efficient for future scalability and ensures abstraction.

ViewModel:
My ViewModel layer can be seen in the *PeopleCounterViewModel.kt* file, where I chose to use AndroidViewModel instead of ViewModel. For my implementation, I needed Context to use SharedPreferences, and I decided to use Application Context instead of Activity Context from a regular ViewModel to avoid potential memory management issues since Application Context

lives for the app's entire lifetime. Throughout the file, some other implementation decisions I made were to ensure encapsulation by using private data variables for the current and total count (using LiveData), with additional read-only public variables, which allowed me to prevent external classes from modifying the data (preventing having negative values, for example). I also made sure to have the data be saved in the model/repository to prevent any data loss and to have data persistence through multiple app sessions. Overall, I consolidated all the logic in the ViewModel layer through the MVVM architecture.

View:
I used Jetpack Compose to implement the View layer, as seen in *MainActivity.kt*. I chose to use Compose instead of a Traditional View System implementation with XML layouts, as I don't need to manually update views and explicitly tell the UI what to change with data changes. Compose allows me to instead declare what the UI should be for each state, and it can handle any updates automatically. This makes it easier for me to use LiveData through my implementation and have a reactive UI when LiveData changes.
Additionally, if you take a look at my View layer implementation, you can see that I separated each UI section into its own function (ex, Top, Center, Bottom), so that they are reusable and that changes to one section don't impact the other. Also, as mentioned previously, the UI is updated from the ViewModel layer data, ensuring consistency and automatic updates.

Note: I started off implementing the UI initially with XML files using a Traditional View System; you can take a look at my earlier commits as well if you would like to see how I started off before I switched over to Jetpack Compose.