We first met as a team on Zoom to discuss how everyone thought we should approach the project. After reviewing the project requirements and examining both sample projects (StopWatch and ClickCounter), we documented which programs had similarities to the program we needed to build. StopWatch had more in common with our Timer than ClickCounter, so we decided to use StopWatch as our base program and modify it as needed. StopWatch has 4 states, handles rotation, and already had a working clock model that could be used for recording ticks and adjusting the display time. We would then just use ClickCounter as a reference for other needed functionality. We started at the interfaces and worked our way down to the classes that implemented them, eliminating unneeded code, renaming variables and methods, and added comments as needed. We used comments as reminders for what needed to be changed, why we changed it, and if we were unsure of something and needed to revisit. We at first focused on core functionality, such as transitioning between states and getting the clock and display time to work as desired. We wanted to take a surgical approach so that we didn’t create runtime and compile errors since this would make the program untestable with the emulator. Our process was normally as follows, plan a change and see if we could determine how it would affect the program, make any additional changes to keep the program functional if the original change broke anything, rebuild, and test. Once the core functionality was complete, we added secondary features, such as audible notifications to the increment and alarm states. We pushed to the repository often so that if something did break it was easy to roll back the program and not have to reimplement changes that were working. When we would refactor, we saw a lot of value in renaming methods, classes, and variables globally since this normally led to errors being thrown. We then traced those errors and addressed them in each of the files. Often while refactoring we would discuss where the change needed to happen and why it was beneficial to have it implemented in one place instead of others.

Our group did the vast amount of programing on Zoom together. We were very successfully since we had multiple sets of eyes examining code and making suggestions. Normally one person would code while sharing their screen, while the others gave input and referenced resources if needed. It normally wouldn’t take long to discover a solution to an issue since everyone would approach it differently. This diversity in thought was crucial. Also, all teammates had technical issues at different times, having everyone code together created great redundancy since anyone could share their screen and open their IDE.

Our extended state machine model reflects our code but at a very basic level. We were able to implement all of the transitions and conditionals that we illustrated. How to implement though can be viewed in the code since it is too complex/overly descriptive for a diagram. We felt it is more effective to model before coding. Modeling gave us a very basic and bird’s eye view of the program. It made sure we all understood how it should function before attempting to implement. This saved us a great deal of time since there was less confusion. A change to our program we would have liked to implement was to have the text on the multi-button change depending on what state it was in. We would also have liked to have different notifications play in the increment state versus the alarm state.