**Pair Development Journey of the Project**

As a six-person team, we used pair programming principles for all our project work. Overall, we felt that it helped us write better code, pre-emptively identify and rectify mistakes, helped us learn from each other, and become more productive. Although it was tricky to allocate mutually exclusive and collectively exhaustive project responsibilities, we managed to share the work load by using various collaborative tools and regular communication.

Our first task as a team was to put together the state machine diagram. We did the initial brainstorming discussion during the class, and then drew out versions based on our understanding. Since there was no collaborative tool to work on the diagrams together, we came up with our own diagrams. Then we communicated over emails and group texts, iterated a few times, and then came to a consensus on the final version of our diagram. The next task was to identify the unit tests needed for the project. This was a relatively straightforward exercise where we finished our brainstorming in class and then finalized the list of tests over emails.

For our main task, we imported the code into bitbucket. However, the tool limits sharing to 5 people so we switched to GitHub. We encouraged each other to push changes regularly so that our collaboration was more effective and we could learn from each other. We also communicated over group texts and emails to avoid repetition and clarify any questions or concerns. All design decisions were discussed through this process, and then a decision was reached through a consensus opinion.

Our starting point for the project was the stopwatch example, which was then modified to meet our project’s functional and non-functional requirements. We also borrowed elements from the clickcounter example and resources mentioned on the class website. Refactoring was very useful in helping us assimilate different code examples and resources to develop our code. We had to be careful while naming/renaming classes, methods and variables to avoid any conflicts with existing states, events, methods and classes.

Using the principles of Test Driven Development, we first wrote the tests for our project and then started to write codes for the individual features of the timer. Finally, we all tested the code individually to identify any bugs and ensure that our code met all the functional and non-functional requirements.

**Relation between state model diagram and project**

We have followed the model to write our code, so at the big picture level, there are no major differences between the model and code. So the states identified in the model are part of the code. We have added DefaultTimerStateMachine in addition to those states. Since a model is a more high-level depiction of the actual program, it doesn’t include code elements such as interfaces, classes and tests that make up the actual code.

The model is essentially a planning tool so we think it is more effective to model the application first before we code it. The model helps us plan and visualize the code first, how individual states interact with each other, and how the process flow happens. Having a good perspective on that before writing a single line of code, makes the overall process more efficient, helps us avoid redundancies and mistakes and helps us code better because we know the big picture, i.e., how this portion of the code (e.g. class, interface, variables, methods, test) is going to interact with others. Moreover, as designs grow more complex, it gets too labor intensive to code first before thinking of how it fits in with the overall scheme of things.

We think that our model was an excellent depiction of the actual code we wrote, so we wouldn’t like to make any changes to it.