**Design Justification**

Our main goal while developing this implementation was to deliver a solution that allowed for both simple testing, as well as the flexibility to adapt to changes in client requirements or the hardware. As a group, we settled on a solution that would take advantage of the fact that the vending machine configuration closely follows the observer design pattern, since the subject-observer relationship between the vending machine and its components is already established by hardware. The benefit of the observer design pattern is that changes to one component of the hardware do not have a large effect on other components, which keeps the relationships between different vending machine components simple and easier to deal with.

A point of emphasis for our code design was modularization, which allows for simplicity and for code re-use. The benefits of having all our methods be so short and simple can be seen in a few places. The separation of toggle methods for all the lights avoids large amounts of potential redundant code repetition, because there are multiple sections of the code where the functionality that these methods implement is required. Another advantage this design tactic provides is that it allows for easier adaptation to change. If a change were to occur, regardless of whether it is a hardware change or a change in the requirements of the system, and we had written large complex methods, it is likely that we would have had to rewrite most of these methods in order to adapt to this change. Instead, with small modular methods, we are more easily able to adapt to changes and support alternative hardware.

The ease of extensibility to other payment methods, such as credit cards, was also something we considered in our design. The credit in the vending machine is accessible globally so that we can add a method to increment it via credit card (or other methods of payment) when required. This will allow for the introduction of mixed modes of payment (which we are aware will likely be a requirement from the client in the future) without a major overhaul of the logic that is in place for the vending of pop, since the methods for calculating change are all separated.

In terms of ease of adapting to changes to the communication with the user, we took the same approach to separate all the processes that involve communication with the user, such as the timer for the display and the toggles for all the lights. If there is a change in the message that we want to display to the user, it would require very little rewriting of the code to implement the change. In respect to the event log, any changes the user wants to make are also quite simple to implement. For example, the user may want multiple log files, a limit on the size of the file, or even a different file type for the log. These listed changes can all be made with ease through arguments in the file handler constructor.

Lastly, the way that the constructor for our logic class also allows for flexibility. The constructor is takes in an instance of a vending machine as a parameter, which means it will work on a variety of vending machine configurations without having to change anything major. As none of the values for coin-rack size or any of the other properties of the vending machine are hard-coded into the constructor, our code can be adapted to the specific vending machine that is passed into it via the constructor. This design allows us to avoid having to change anything for major for vending machines of the same configuration with different presets and initialization factor values.