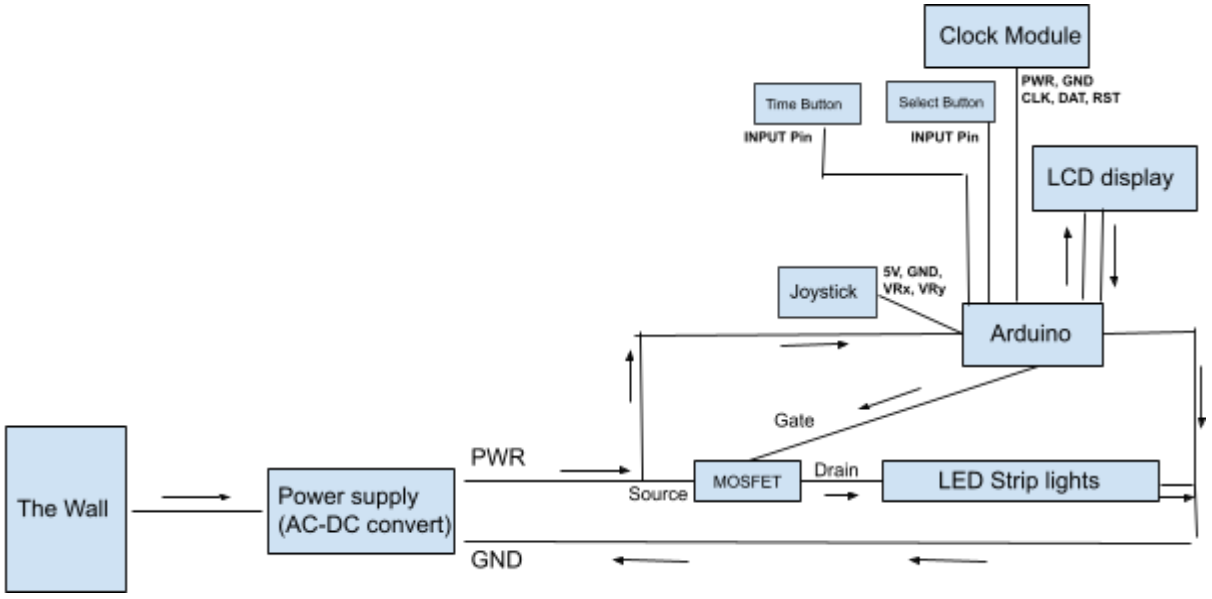
Breanne Chan

Project Description: Sunrise Alarm Clock

The Sunrise Alarm Clock was a personal project that I designed and built from scratch, involving designing and soldering an electronic circuit and components, developing C++ code for an Arduino Uno, and designing a 3D-printed case using a CAD program.

Electronics

Circuit Diagram



There were several technical challenges involved in the design of this circuit. First, I had to create a system where a light source would gradually increase in brightness. Although I initially tried to use an array of small LED lights, lighting up one row at a time, I found that they were not bright enough to serve as a wakeup system. Ultimately, I realized that I needed brighter lights, which required more power than the Arduino could output. Changing the light source meant I needed to carefully consider the appropriate power source for my project, and also find a separate device to modulate the brightness of the light source. Eventually, I settled on using 12V LED strip lights, since the Arduino Uno could also be powered with 12 volts, wired the Uno's lower-voltage digital PWM pins to an external MOSFET module serving as the brightness controller, and used a wall outlet for consistent, reliable power.

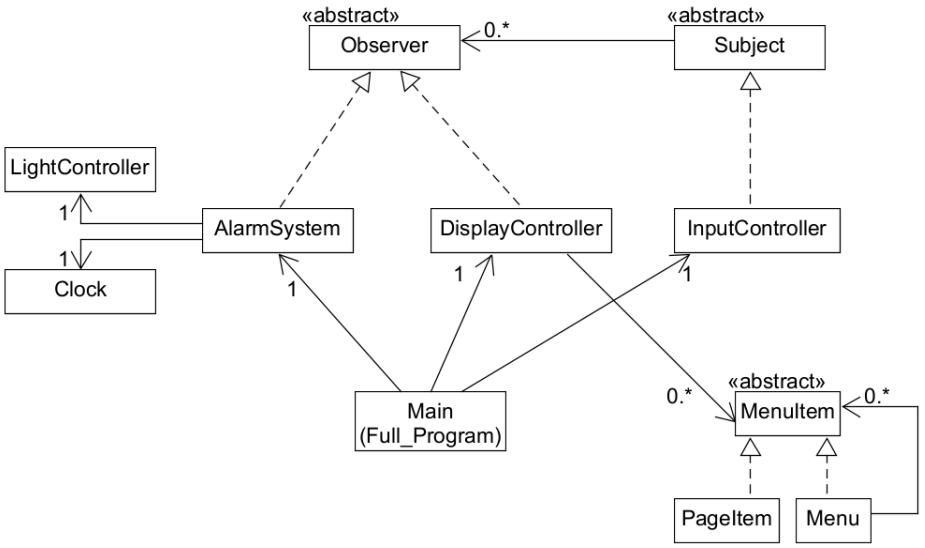
In the final circuit, an Arduino Uno serves as the "brain" for the project, controlling the light and user input/output system. The circuit makes use of a DS1302 Real-Time Clock module to keep accurate track of time, and an N-channel MOSFET to control the brightness of the LED strip light. Two momentary push buttons and a joystick are used to set or modify the alarm or current time, while a 16x2 liquid crystal display shows information to the user. A 12V DC power supply converts the power provided from a wall outlet into the appropriate voltage and current levels for the Arduino and LED strip lights.

Code

In my first iteration of the code, my primary aim was simply to build a full, working system. To do so, I broke down the large goal - creating a sunrise alarm clock - into several smaller chunks, such as getting the light system working, creating a user input/output system, implementing functionality to modify and track time accurately, etc. and wrote programs to solve each of those smaller problems independently. While making these programs, I was constantly researching and reading documentation to understand the capabilities of external software libraries that I used as well as the inputs/outputs of my hardware. Finally, I used what I'd learnt in the process of creating each small program to recombine them and create the complete system.

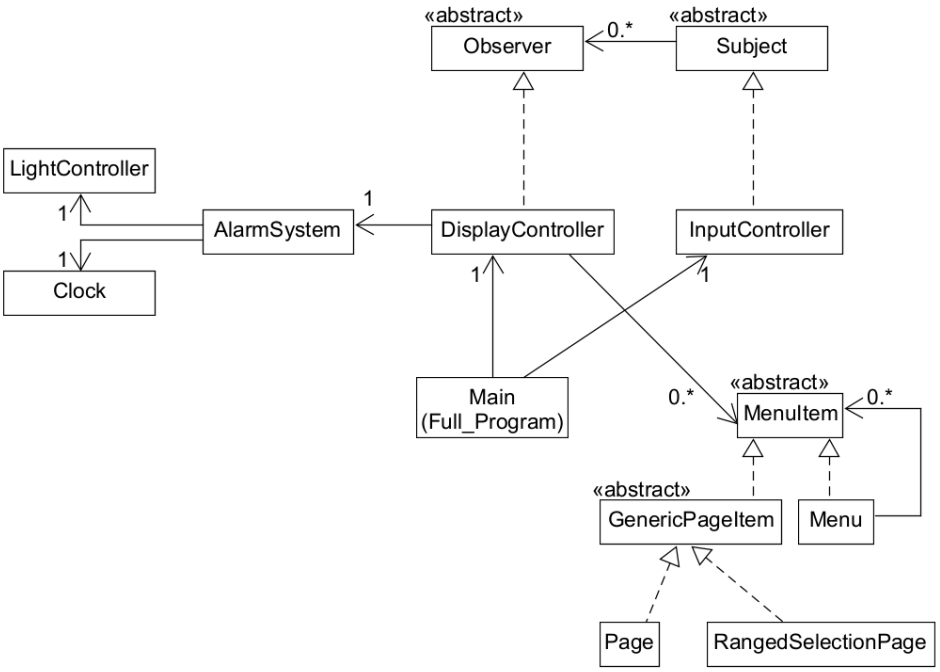
However, the original program I created was not without flaws. For instance, it was written in a messy procedural pattern and was neither easy to read nor simple to extend. Furthermore, the logic I used sometimes induced unwanted blocking, where the microprocessor was waiting for user input and could not check whether it was time for the light system to start. Currently, I am developing a refactored version of my original code to address all these problems, where I am applying object-oriented design concepts to create a class hierarchy which increases cohesiveness, creates modularity, and decouples separate functionality from each other. Additionally, I plan to use finite-state machine models with a set number of states to represent a tick-based update loop, preventing unwanted blocking.

UML Class Diagram V1



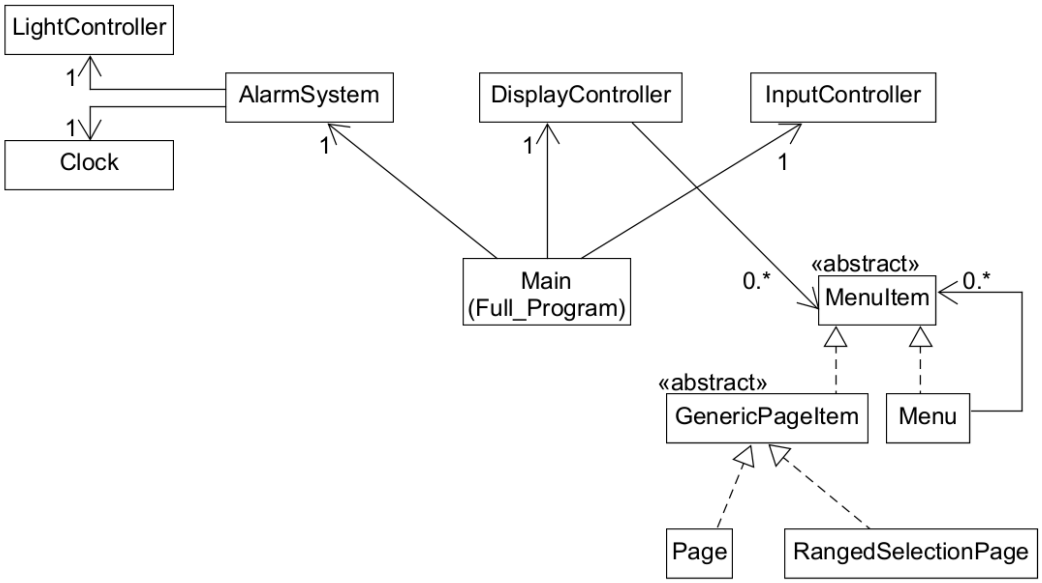
- Initial ideas for class hierarchy
- Main is the .ino file from which the program starts
- InputController listens for input from the user, notifies AlarmSystem and DisplayController to act
- Pages/Menus in the user display are represented by the composite MenuItem pattern

V2



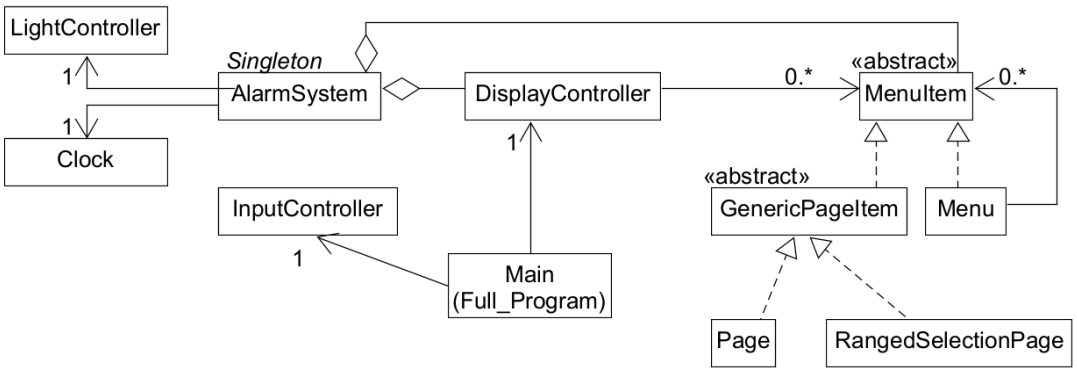
- DisplayController needs to react to user input, for that it needs some way to access AlarmSystem; I decided the easiest way to do that was to have DisplayController directly have a member AlarmSystem
- The Subject-Observer system is maintained for flexibility in adding future classes which respond to user input
- Added subclasses of Page; converted PageItem into the GenericPageItem abstract class

V3



- Realized that since Main instantiates and can access a member DisplayController and InputController, the Subject-Observer pattern only increases coupling of the program
- Removed the Subject-Observer system entirely, DisplayController will be the class whose responsibility is to process all user input

V4



- Refactored AlarmSystem into a singleton class - since there really only should be one AlarmSystem
    - Note that the notation for singleton is not typical; I just wanted to explicitly write it since I have not included fields and methods in the diagrams
    - Note that I also have not included the association for AlarmSystem to itself (where it contains a field for an instance of itself)

<u>Potential Future Improvements</u>

- Refactor DisplayController, InputController, potentially also LightController and Clock to be singleton classes
    - They each correspond to particular hardware modules
    - However, doing this may increase complexity of code without much benefit
- May add a RangedSelectionMenu as a child of Menu
    - Alternatively, may add RangedSelectionBehaviour class and implement a composition structure with both Page and Menu
- Both DisplayController and MenuItem contain associations with AlarmSystem; however, DisplayController also contains a collection of MenuItem. Can I remove the extra reference from DisplayController to reduce coupling?

Computer-Aided Design

The CAD design for this project included a base with snap-fit plug mounts to hold the PCB modules and a lid with similar mounts for a 16x2 LCD, joystick, and two panel mount buttons.

I started the design by creating mounts for each of the PCB modules (Arduino, RTC, MOSFET, etc). I initially started with a design that partially covered each side of the module. However, while researching alternative solutions, I came across a Medium article experimenting with snap-fit plugs,

and consequently decided to try and create my own in Onshape, based off of William Andrews' design. I tested several different sizes and tolerances before settling on a final size and creating prototype mounts for each module.

After that, I started experimenting with lid design. I tried a few versions of a spring-loaded lid, but ultimately pivoted to a much simpler brim design which was simple and clean-looking. However, the new lid design was causing a problem; because the lid was hollow on the inside, but the snap-fit plugs stuck out of the top, each print required a lot of support, no matter how I oriented the design. I wasn't sure how to resolve this problem until a family member suggested that I recess the plugs into the lid, so that the top of each sat flush with the surface of the lid. After some trial and error, I was able to create a final lid design that reduced my material use by 15% and decreased print time by 40 minutes!

Note: I also wrote an in-depth Instructables tutorial for this project, which was featured by the editors of the site and garnered over 100 views in its first two weeks after publishing.