CS3704 - PM3
Team Octo

Process Deliverable II
Retrospective

- We started this project with requirements elicitation, for which we used a survey to determine the need for an application like ours as well as the general knowledge and perspective people have about Kanban. In summary, our 11 responses said that Kanban is generally underutilized and that with an application like ours, they would want to see features such as checklists per card, the ability to assign cards to certain team members, linking related cards together, and being able to archive completed cards.
- With requirements analysis, we determined nonfunctional and functional requirements. Nonfunctional includes loading within 2 seconds, proper encryption and backing up of data, being accessible to both website and mobile users, uses a relational database, and documentation on each feature. Functional requirements include being able to load Kanban boards and display progress, manage and apply client-side changes, assign attributes to task cards, and handle communication between users.
- Then we focused on design based on our requirements analysis and specification. We brainstormed both the low and high level designs of our app, namely system structure and UI design, through a series of methods, such as storyboarding, which is described below in the next section.
- As novice developers, we learned that this is an incredibly crucial step that requires much attention. It took us no time to realize the importance of being thorough with our design, which reflected in our efforts: we made several storyboards and mockups of the interface design, taking our time to consider how the end user would react. This process started out slow and clunky, something that we intend to refine on future development iterations at this stage (whether it be on this project or future endeavors).
- We feared reaching design conflicts that would require concessions due to team members having different ideas for the final product. But, the team shared a very similar mental image of the end product, meaning there was little conflict when determining the top-level structure of the codebase and the interface design as a whole.

Sprint planning

- With a reasonably steady design in place, the overarching goal for our next sprint is to begin implementation. This will manifest itself in one or more prototypes that translate our vision into a concrete codebase.

- With each prototype, we'll expose the build to relevant playtesters to gather their thoughts. Ideally, we'll iterate and refine each prototype through this feedback, but it'll depend on the volume and quality of opinions we get.

High-level Design

- The high-level design pattern we find most appropriate is **event-based architecture**. Event-based approaches are extremely common for web applications, which our product just happens to be (regardless of whether it's wrapped in an executable environment like Electron). With the real possibility of multiple users performing actions on a Kanban board at once, we must be able to support asynchronous calls without stalling the main thread: event-driven programming is literally made for this. Moreover, working within an existing and flourishing environment of event-based frameworks, such as React, will allow us to develop at lightning fast speeds. We also make use of **client-server architecture** since our product relies on clients connecting to a central server that tracks changes to their Kanban board, but this is barely worth mentioning due to its triviality–of course a web app requires a server with clients.

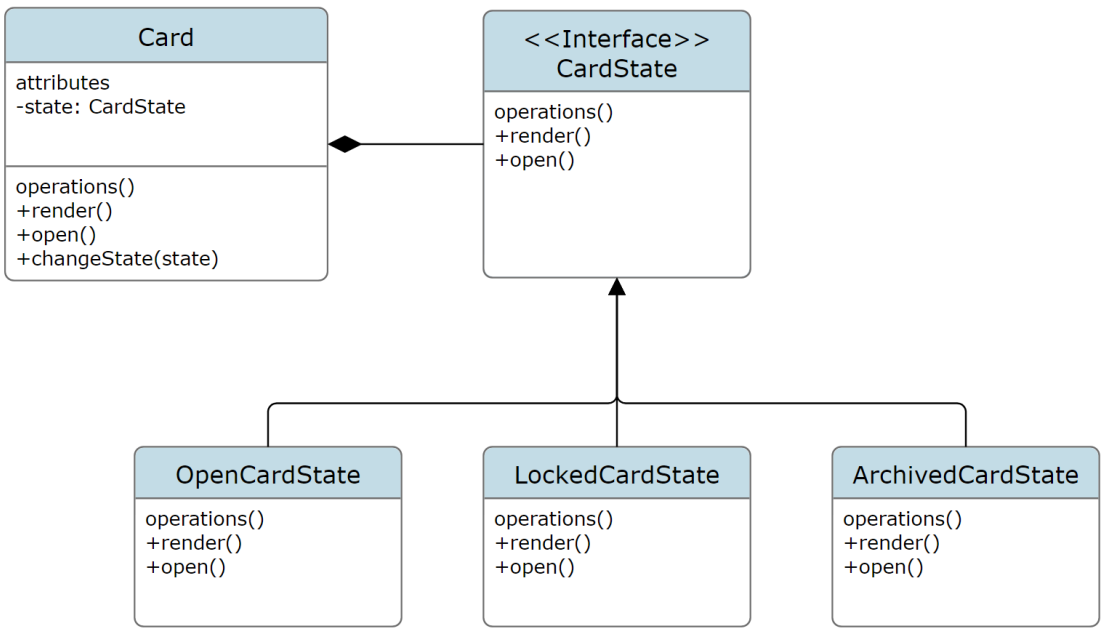
Low-level Design

- One of the most obvious low-level design patterns to use is **state**, a behavioral pattern used for changing how an object behaves depending on some given internal state: this would be extremely helpful for the card objects in our implementation. One idea is to give each card a state field that can either be *Open*, *Locked*, or *Archived*. The behavior of the card then depends on this state: open cards are visible and can be freely edited by any user, locked cards are visible but are read-only, while archived cards are hidden from the main board display.

Pseudocode

```
interface CardState {
    public virtual void render();
    public virtual void open();
}

class OpenCardState implements CardState {
    public override void render() {
        // Display card.
        ...
    }

    public override void open() {
        // Open card in read-write mode.
        ...
    }
}

class LockedCardState implements CardState {
    public override void render() {
        // Display card.
```

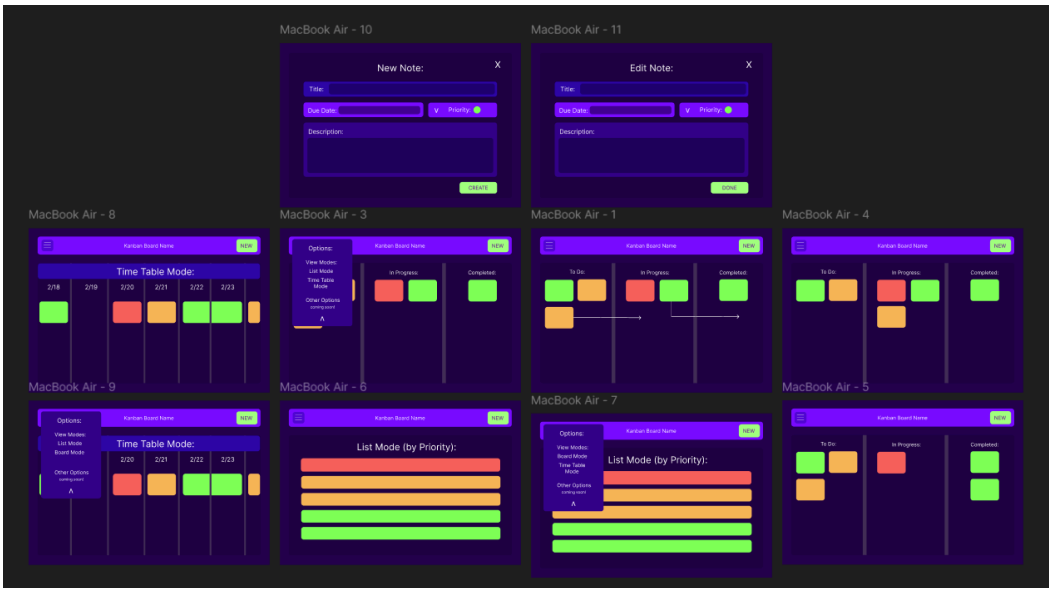
```
        ...
    }

    public override void open() {
        // Open card in read-only mode.
        ...
    }
}

class ArchivedCardState implements CardState {
    public override void render() {
        // Do not display card.
        ...
    }

    public override void open() {
        // Open card in read-only mode.
        ...
    }
}

class Card {
    private CardState state;

    public void render() {
        state.render();
    }

    public void open() {
        state.open();
    }

    public void setState(state) {
        this.state = state;
    }
}
```



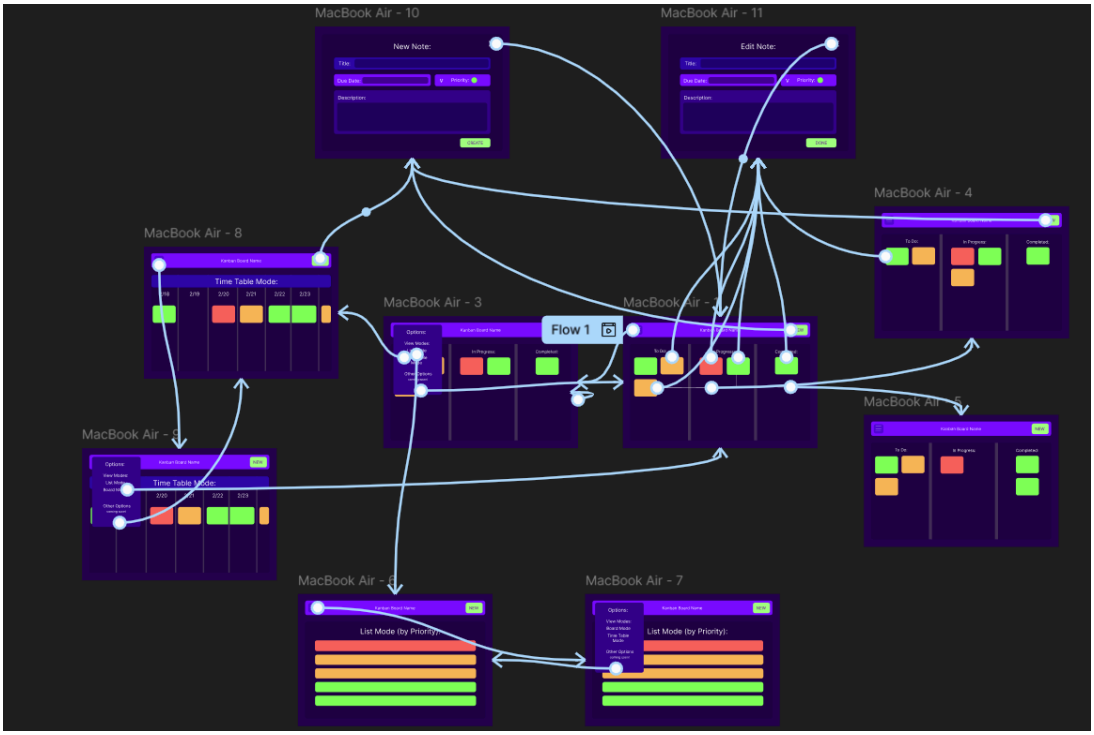
Design Sketch

For a design sketch, our team decided to create a wireframe board using Figma's sketch and flows functionality. The wireframe was created with two major steps, creating all possible boards, and connecting them using the flow functionality.

Base board images



After flow connections



Using Figma we were also able to produce a live demo using the boards and their connections, you can preview one here.