Neha Bangari, Vrishank Bangari, Jake Frohlich, Ethan Werner
Intermediate Software Design

Project Milestone 3

Process Deliverable II

Agile Process Model

Name	Jake	Ethan	Neha	Vrishank
What went well?	The code for the low-level design worked as expected.	I was able to complete my work on time and without too much trouble.	Our group was able to decide on a general idea for the application design	Initial ideas for design mockup
What didn't go well?	There were multiple initial ideas that were too difficult to implement into the low-level design.	I had to spend a significant amount of time researching to refresh my knowledge of the topics I needed for my section.	Deciding how much detail to include in the wireframe/how much to actually implement in the future	Converting those ideas into Figma
What will we change next time?	Deciding which low-level designs should be implemented considering the timeline for the project.	I will try to better understand the important concepts to the assignment before starting.	Refactor design based on actual implementation goals	Learn figma better

Prioritized Task List for PM4

1. Discuss the measures of success for this system.
2. Determine what aspects of the system are most important to test.
3. Create a list of test cases to use.
4. Split up test cases and write out the details for each black box test plan.

Neha Bangari, Vrishank Bangari, Jake Frohlich, Ethan Werner
Intermediate Software Design

High-Level Design

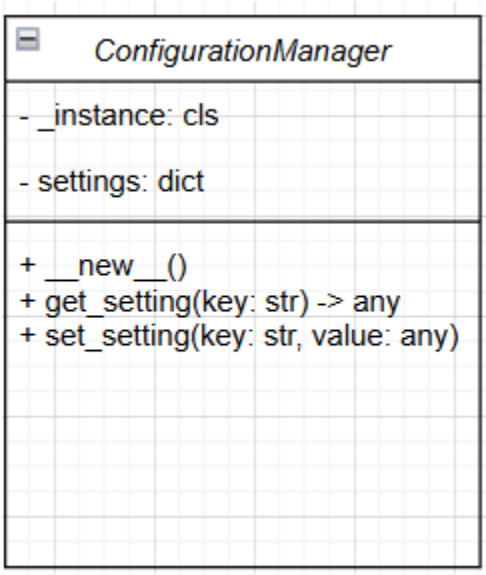
To implement our system, we would use an event-based architectural pattern. This would make sense for a few reasons. First of all, the system should have a fast response time when adding, removing, or updating tasks and recalculating the optimal schedule each time. By using an event-based approach, we can utilize event queues and asynchronous processing to allow the system to respond much faster to such operations. Additionally, all processes in this system would be triggered by the user, so an event-based system would make sense. Finally, an event-based system would allow us to implement a notification system later. The event that would trigger this could be a task or meeting nearing its start or completion time.

The other architecture options are layered and pipe-and-filter. A layered approach would not work as well because of its lack of scalability. It would also be less flexible for the dynamic changes we would want and would not be suited for asynchronous workflows needed to speed up the system's processes. As for a pipe-and-filter architecture, this would be very inefficient when recalculating the user's schedule on an event trigger. Pipe-and-filter architecture would also have much slower processing times because of its linear nature.

Low-Level Design

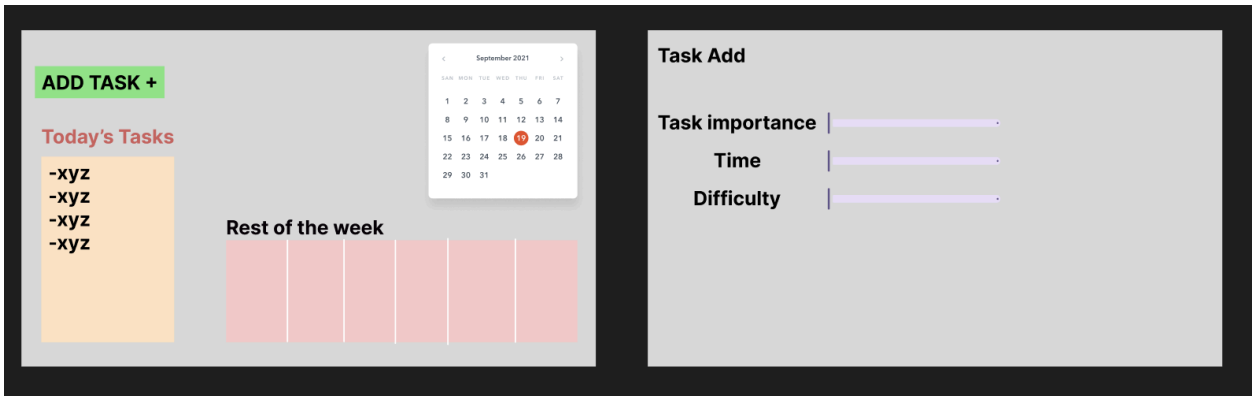
For the Dynamic Scheduler Project, the Creational Design Pattern family, specifically the Singleton Pattern, would be helpful for implementing the global configuration manager. This manager could handle system-wide settings such as default prioritization strategy, notification preferences, or theme configurations (such as light mode, dark mode or otherwise for the UI).

```python
class ConfigurationManager:
    _instance = None

    def __new__(cls):
        # Ensure only one instance is created
        if cls._instance is None:
            cls._instance = super(ConfigurationManager, cls).__new__(cls)
            # Initialize default settings
            cls._instance.settings = {
                "default_strategy": "DifficultyBasedStrategy",
                "notification_enabled": True,
                "theme": "light",
            }
        return cls._instance

    def get_setting(self, key):
        return self.settings.get(key)

    def set_setting(self, key, value):
        self.settings[key] = value



# Running Example:


config1 = ConfigurationManager()

# Output: light
print(config1.get_setting("theme"))

config2 = ConfigurationManager()
config2.set_setting("theme", "dark")

# Verify singleton behavior

# Output: dark (same instance as config2)
print(config1.get_setting("theme"))

```



The Singleton Pattern (as shown in the class diagram and the Python code) ensures that the ConfigurationManager class has only one instance throughout the system. This is particularly useful in the scheduler project we are doing for maintaining consistent global settings, such as default prioritization strategy or notification configurations, without the risk of accidental duplication or conflicting instances throughout the program. The Singleton Pattern helps keep the code clear and efficient for settings that do not require multiple instances.

Design Sketch



Our program supports a desktop view as it is meant to be used in the workplace amongst software development teams. The initial wireframe was designed with several user goals in mind, such as maintaining a clean and simple interface. The design is structured so that users can clearly view current tasks, a calendar display, as well as a weekly view of tasks. Additionally, when users go to add a new task, they are able to input fields such as task importance, time, and difficulty. The system will utilize these values to determine the task's priority relative to existing tasks within the team and project. One main usability heuristic we wanted to address through this design is to maintain an aesthetic and minimal design that would allow users to quickly input and view tasks. Additionally, we wanted to have a flexible and efficient design so that developers can quickly use this tool to help them complete project assignments. Another factor to consider with the design is the color palette. In order to preserve a visual flow, we made sure that important features of the program were highlighted (such as the "Add Task" button).