Project Milestone 3

By
Ethan Surber
Menase Yirdaw
Ian Yoon
Yasser Algburi

Process Deliverable II (3%)

The submission for this deliverable will depend on the specific SE process model your team plans to use to complete the group project (as described in your project proposal). Example submissions for different processes include:

Prototyping: A prototype/design sketch is already required for this milestone. Should be more formal/advanced than the previous milestone prototype.

Scrum: submit notes (include each teammate) from at least weekly scrum meetings.

Kanban: submit a depiction (i.e., screenshot, link, etc.) of completed and prioritized tasks in your team's Kanban board

Agile: submit a summary of retrospective (retrospective) and prioritized tasks for the next milestone (sprint planning)

Extreme programming: submit calculation of project velocity and estimate velocity for next project milestone

Incremental: submit documentation explaining how design maps to prioritized requirements

Waterfall: submit updated SRS document with high-level, low-level, and UI design contraints

High-level Design (4%)

Describe which architectural pattern you would use to structure the system. Justify your answer.

For the focus bot project using a layered architecture with a model view controller pattern aligns well with the design principles of being maintainable, scalable and modular. The MVC architecture will structure the system by first separating core logic and data (model), user

interface (view), and the processing of user inputs (controller). This separation is perfect as it allows each layer to handle specific responsibilities. The model will manage essential data and dependencies such as user information, task lists, and pomodoro cycles. The view will enable user interactions, letting users create tasks, set timers and view analytics. The controller will be the bridge for the model and view ensuring both have seamless data flow and responsive interactions. This architecture will also simplify the management of dependencies and data flow among features, which supports FoucsBot's goal of productivity, real-time notifications, and user engagement.

The MVC pattern offers practical benefits, such as higher abstraction levels, modular design and reusability, which allows for efficient testing and better maintenance. For FoucsBot's agile approach this means that foundational features like registration and task management could be built and expanded upon iteratively. The structure of MVC also supports extensibility which makes it easier to add future features and improvements such as real-time notifications or productivity analytics without having to disrupt the existing system. Hierarchical communication and event-driven flows also perfectly align with FoucsBot's planned interactive features such as pomodoro timer, enabling responsive and user-friendly functionality while keeping the codebase organized and adaptable for future iterations.

Low-level Design (4%)

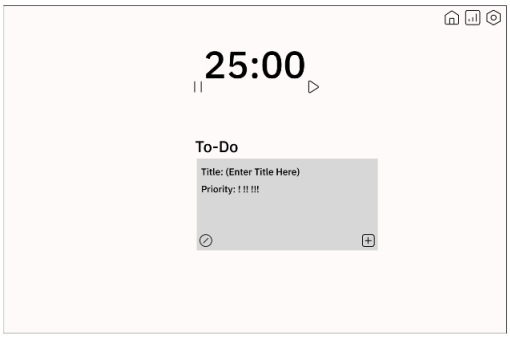
Discuss which design pattern family might be helpful for implementing a specific subtask for this project. Justify your answer, providing a code or pseudocode representation and an informal class diagram.

For the FocusBot project, the Behavioral Design family, or more specifically the Observer pattern, would be helpful for implementing the real-time notifications subtask of our project. This subtask needs to notify users about tasks, pomodoro session completions, and/or other time-sensitive updates. The Observer Pattern and Behavioral Design Family would be helpful because it is event-driven, which allows a certain subject to notify multiple observers when a specific event occurs. The behavioral design family also ensures real time updates, as it ensures that all relevant components are updated immediately when the subject's state changes.

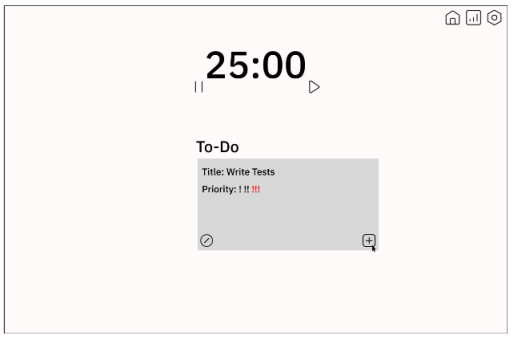
Design Sketch

Storyboard
Scenario: A user wants to add a task to a do to list
Persona: Software Developer, James

James is on the FocusBot dashboard then he clicks on the plus button to add a task.

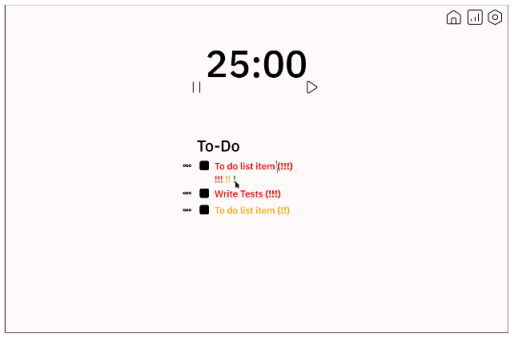


A popup window then appears prompting for the task name and priority.



James then names the task and sets it at the highest priority.

The task appears, but then James realizes a task is actually a lower priority than what he thought, so he clicks the three dots to edit it.

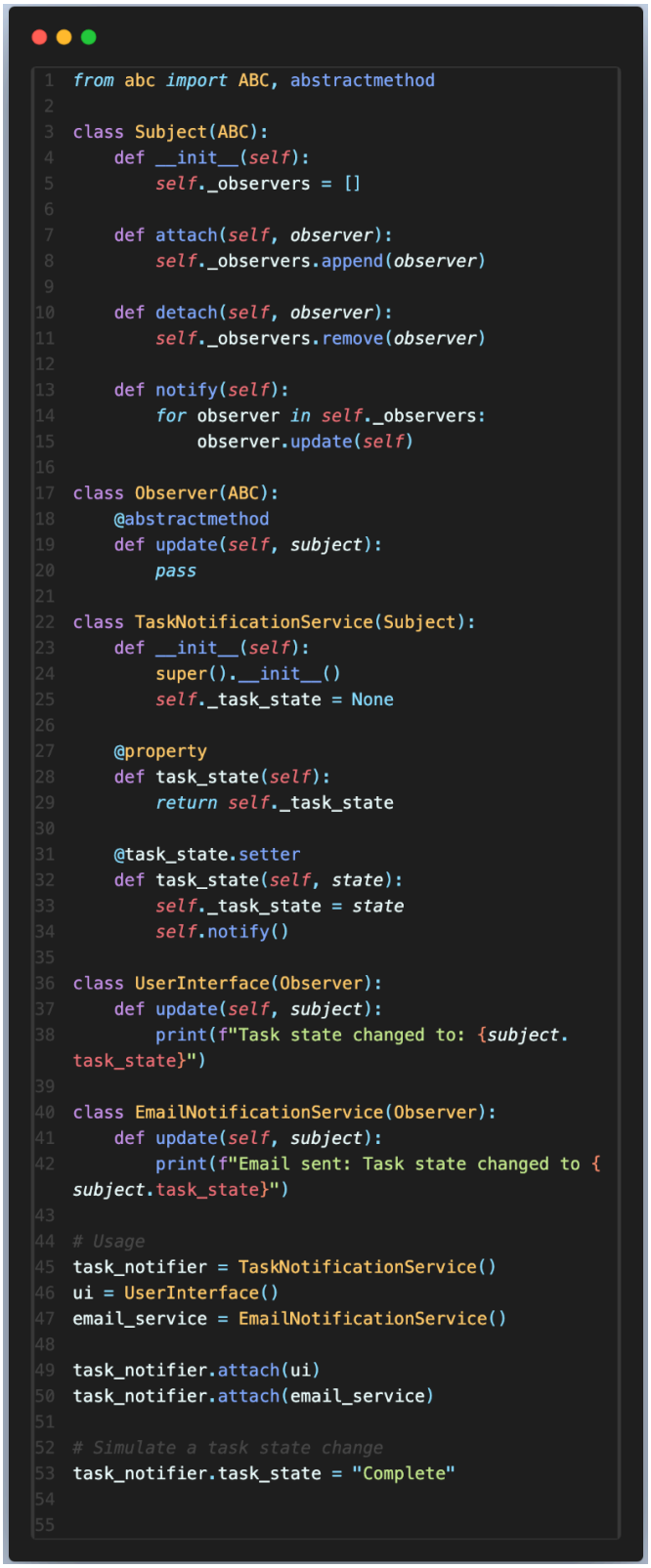


Priority markers drop down from the task, and the title is now allowed to be changed.

James then sets it to the lowest priority and the to do list updates accordingly.

For the storyboard, a minimalist design was chosen to adhere to the design principles in class. In addition, the red, yellow, and green priority color scheme, and the various icons across the dashboard were chosen to match real-life. This makes it easier for the user to distinguish what certain buttons on the dashboard do. The design of the dashboard and interactions were chosen to align with the usability heuristics.

Pseudocode



```python
from abc import ABC, abstractmethod

class Subject(ABC):
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self)

class Observer(ABC):
    @abstractmethod
    def update(self, subject):
        pass

class TaskNotificationService(Subject):
    def __init__(self):
        super().__init__()
        self._task_state = None

    @property
    def task_state(self):
        return self._task_state

    @task_state.setter
    def task_state(self, state):
        self._task_state = state
        self.notify()

class UserInterface(Observer):
    def update(self, subject):
        print(f"Task state changed to: {subject.
task_state}")

class EmailNotificationService(Observer):
    def update(self, subject):
        print(f"Email sent: Task state changed to {
subject.task_state}")

# Usage
task_notifier = TaskNotificationService()
ui = UserInterface()
email_service = EmailNotificationService()

task_notifier.attach(ui)
task_notifier.attach(email_service)

# Simulate a task state change
task_notifier.task_state = "Complete"
```

Subject Class: This abstract class manages the list of observers and provides methods to attach, detach, and notify them. When its state changes, it notifies all registered observers.

Observer Class: This abstract base class requires concrete observers to implement the update() method, which is called when the subject's state changes.

TaskNotificationService Class: Inherits from Subject and holds the task state. When the state is updated, it notifies all attached observers.

UserInterface Class: A concrete observer that updates the user interface when notified of state changes.

EmailNotificationService Class: Another observer that, when notified, simulates sending an email about the task state change.

Project Check-In

Complete this survey to provide an update on your team progress on the project for this semester. Only one team member needs to complete this for the group.

Due: November 22 at 11:59pm

Process II deliverable

High-level design

Low-level design

Design sketch

Check-In survey