PM3 FocusBot Application
Michael Pang, Matthew Topping, Elijah Tynes, Ajay Seethana

Process Deliverable II

1. Prototype Overview

Our prototype focuses on key features, now enhanced with event-based architecture to improve responsiveness and scalability:

- Break Reminders: Notifications triggered in real-time based on productivity data, reducing fatigue by encouraging timely breaks.
- Hydration Tracking: Alerts for staying hydrated throughout the workday, monitored and delivered asynchronously for efficiency.
- Movement Suggestions: Real-time prompts to stretch or walk, leveraging event-driven insights for optimal timing.

2. Feature Mockups

The prototype now includes mockups reflecting event-based and fault-tolerant designs:

- Smart Watch Integration: Real-time notification screens that adapt to productivity drops or extended work periods.
- User Dashboard: A simplified, responsive layout tracking hydration levels, activity, and break statuses.
- Break Activity Suggestions: Fault-tolerant notifications providing actionable suggestions for breaks, even if other components fail.

3. User Flow Demonstration

The user flow incorporates event-driven processing to ensure responsiveness:

- Detection of Inactivity: Events captured by the productivity tracker initiate a notification.
- Real-Time Break Management: Follow-up notifications are issued if no action is taken.
- Hydration Tracking: Regular event-based prompts for water consumption with a status update for completion.

4. Requirements Prioritization

- Health Reminders: Minimizing burnout with accurate, actionable notifications.
- Fault Tolerance and Responsiveness: Ensuring reliability and real-time responses to user needs.
- Usability: Maintaining a user-friendly and intuitive interface.

5. Feedback Collection Plan

After presenting the prototype, we will gather feedback on:

- Effectiveness and timing of notifications, considering event-based triggers.
- Usability of the fault-tolerant notification system and user interface design.
- Suggestions for improved integration of productivity data for retrospectives.

This feedback will guide iterative improvements, leveraging Agile development principles for refinement.

High-Level Design

Our project would likely utilize an event-based architecture. Event-based architecture holds the advantage of real-time responsiveness, which would greatly benefit our application, as it would be able to collect the data about the user's productivity (rate at which tickets are completed, how active their IDE, etc.) and respond in real time when the application notices any drops in productivity or prolonged periods of working. Therefore, the real-time nature of event-based architecture would allow the application to effectively collect and analyze user data, allowing it to react immediately when the application deems a break to be necessary for the user's well-being.

Additionally, the event-based architecture also provides fault tolerance, which can be pretty helpful for our application. For example, when a user is notified about an incoming break, they will also be notified about a potential break activity they can do during that time that will take them away from their seats to avoid being too stagnant during the workday. However, if this component fails or becomes unresponsive, we wouldn't want the break notification system, which would be handled by a separate entity, to also go down. Therefore, by using the event-driven architecture, the application would be able to remain operational even when one of the components is not working properly.

Event-based architecture also enables the application to take advantage of the asynchronous processing of events and have the benefit of scalability. Asynchronous processing of events allows events to be supervised at the same time, improving the overall efficiency of the application. Scalability would allow our application to be utilized by a wider audience of employees at the same time. These would both be particularly important, since we don't want the application to run into issues collecting data/triggering events at the same time for several company's worth of employees.

Another reason to apply event-based architecture would be that it allows for the integration of external services that would expand the functionality of the application. For example, companies may want to store the data collected by the application in a data storage for use in meetings like sprint retrospective meetings, where managers may want to reflect on the past sprints and the performance of each of the employees/teams.

Low-Level Design

Discussion -
One of our project's main subtasks is a productivity tracker. The subtask's main functionalities involve collecting data, calculating a productivity rating to determine how active the user is on their work device, and notifying the user about the beginning/conclusion of breaks.

This sub-task would most suit the Behavioral design pattern family, because the Behavioral design pattern family takes into account how classes or objects interact and distribute responsibility, like how algorithms assign responsibilities between objects. This is important for our project, especially for this sub-task, since the software application will be required to take in information based on how the user interacts with their work laptop, specifically the progress they are making with their tickets and the amount of continuous activity performed by the user. Therefore, having a productivity data collector and a productivity manager to monitor whether or not a user is slowly losing focus would be instrumental to the success of the application. The productivity data collector class would prioritize gathering data, including the rate at which JIRA tickets are being completed and the rate at which they perform actions on their work device. The productivity manager class would take the calculation made by the productivity data collector and dynamically update the user's productivity level. Once the user's productivity levels drop below a specified level, the manager will signal to the Notification System class that it should start a break. The manager class will also have the ability to lock certain features of the work device, like editing on certain IDEs, as a way of helping the user take the time they need to rest and stretch before continuing the rest of their work.

Additionally, our application would involve another class tasked with being in charge of the notification system, including the break notification message/alarm. Once the manager finds that the user has already worked for an extended period/is gradually losing focus based on the productivity calculation we will provide, the manager will communicate to the notification system that the user needs a break to avoid stagnating progress. The notification system will take this signal and start the break event, notifying the user with an alarm and a message to start a break. Additionally, the notification system class will signal to the manager class to block access to the work device, allowing the user to focus on taking a well-earned break. Once the break ends, the notification system will notify the productivity manager, who will tell the productivity data collector to start collecting data again.

Another potential class that would be helpful for the application would be a break class that handles everything related to each break event. It would be in charge of responsibilities such as keeping track of the break start time and sending a message to the Notification System class once the break officially ends, allowing the notification system to notify other classes that are still waiting for the break to end before continuing certain processes, like the productivity manager and data collection classes.

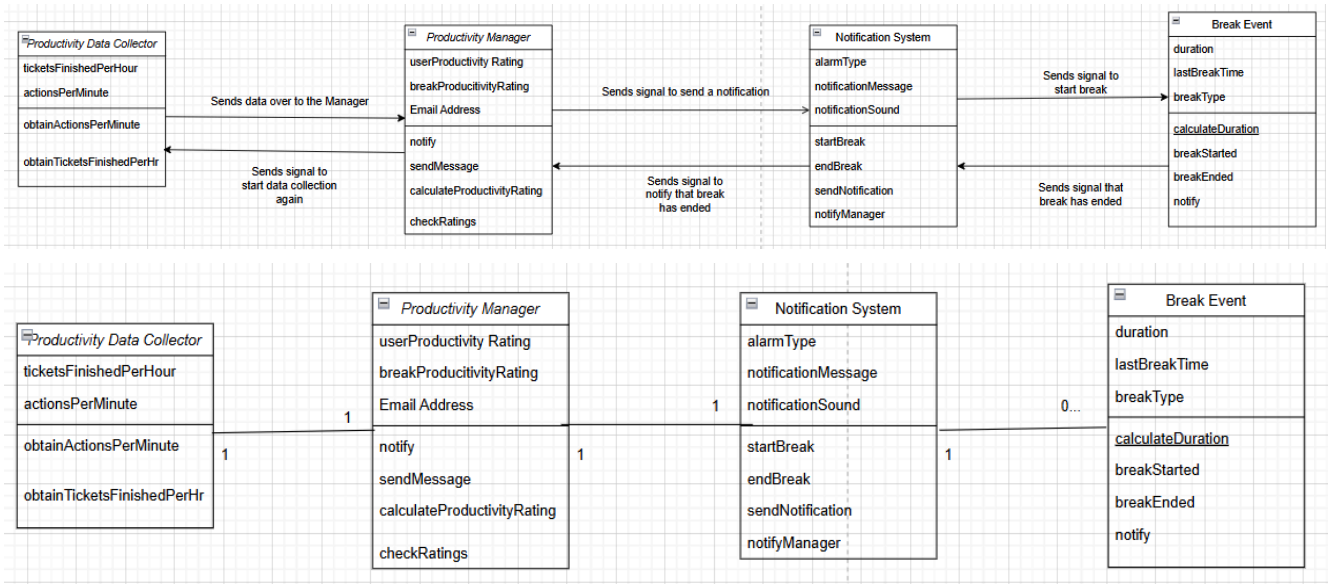
Pseudocode - (get/set methods assumed to be present, currently just shows necessary functionalities)

```
public class ProductivityDataCollector() {
	private int actionsPerMinute;
	private int ticketsFinishedPerHour;

	public int obtainActionsPerMinute() {
		// Calculate actions per minute, to be used in ProductivityManager
	}

	public int obtainTicketsFinishedPerHour() {
		// Calculate tickets finished per hour, to be used in ProductivityManager
	}

}

public class ProductivityManager() {
	private int userProductivityRating;
	private int breakProductivityRating;
	private String emailAddress;

	public void notify() {
		// Logic to notify the Notification System
	}

	public void sendMessage() {
		// Sends a message to a user
	}

	public void calculateProductivityRating() {
		// Determines a productivity rating using the metrics from
ProductivityDataCollector
	}

	public void checkRatings() {
		// Evaluates productivity ratings and takes necessary actions (calls
calculateProductivityRating and checks the value with the breakProductivityRating)
		If userProductivityRating < breakProductivityRating
			Notify the Notification System
		// Otherwise, do nothing
	}
}
```

```
public class NotificationSystem() {
	private String alarmType;
	private String notificationMessage;
	private String notificationSound;

	public void startBreak() {
		// Logic to initialize a break and notify relevant systems
	}

	public void endBreak() {
		// Logic to end a break and notify relevant systems

	}

	public void sendNotification() {
		// Notifies the user with a message containing relevant information
		// Uses the notificationSound and notificationMessage to alert the user
	}

	public void notifyManager() {
		// Alerts the productivityManager of specific events
	}
}

public class BreakEvent() {
	private int duration;
	private int lastBreakTime;
	private String breakType;

	public void calculateDuration() {
		// Determines the length of a break
		// Uses the time from the last break to determine the amount of time
recommended for the user to take as a break
	}

	public void breakStarted() {
		// Logic for the application while break is started
	}

	public void breakEnded() {
		// Logic for the application while the break is ending, uses the duration to
determine when the break is supposed to end
		// Application will wait for the user to come back and interact with it before
completing the current break fully
```

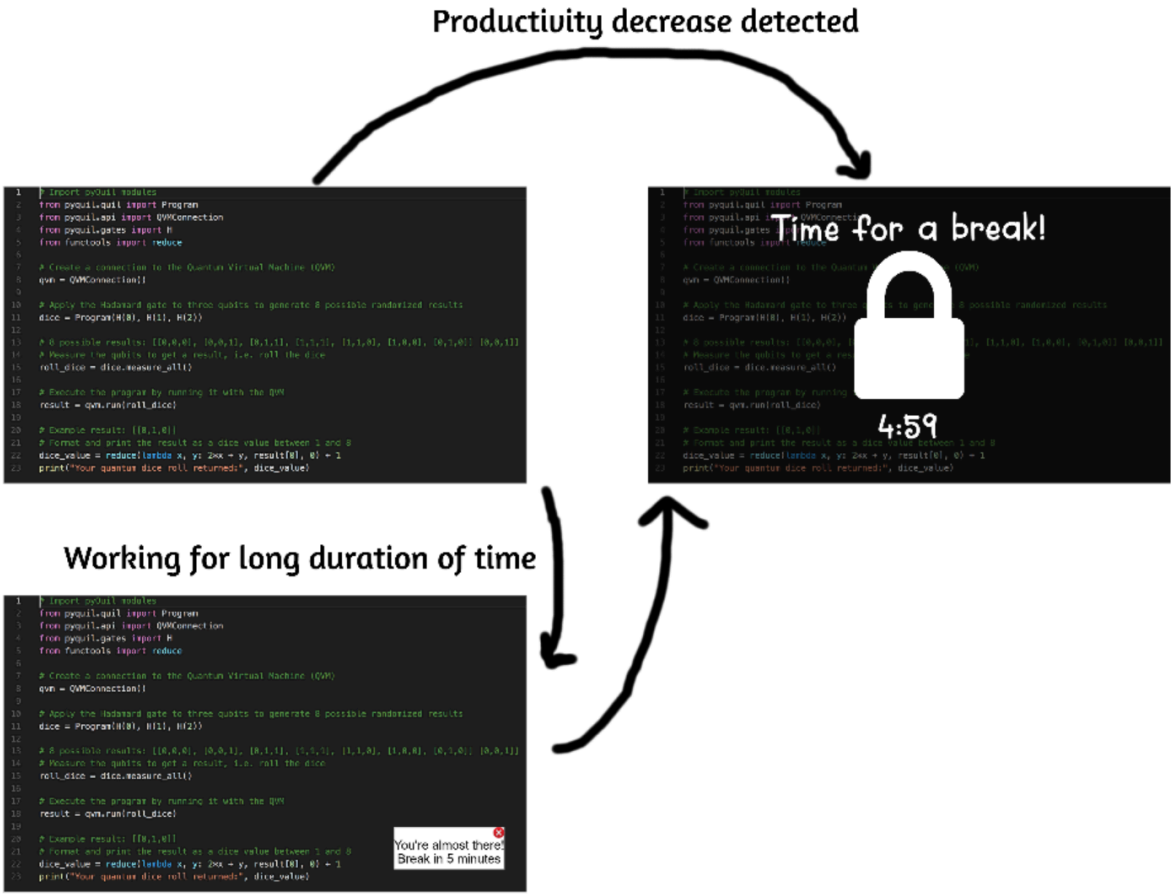
```
        }

        public void notify_end() {
                // Logic to send a notification to the Notification System class that a break event
has ended
        }
}
```

Class Diagram -



Design Sketch

Sketch -



Rationale -

The design goal of this sketch was to represent the relationship between the events (such as productivity decrease) and functionalities of our application, aligning with usability principles discussed in class. Aspects of usability such as learnability, efficiency, visibility, and satisfaction with no errors were prioritized to create a user-friendly interface. For instance, ensuring that the state of the application is clear (visibility) while minimizing intrusiveness allows for the engineers to stay focused on their work. This behavior is exhibited via the upcoming break notifications, which are designed to occupy minimal screen space and inform the user of the state of the application.