High-level Design

High-level Design: Event-Driven Architecture (EDA)

Justification

Responsiveness to Events: EDA is ideal for applications like Discord bots that need to respond to specific events (user inputs). In this case, the events are the commands issued by the users. The bot listens for these events and triggers specific functionalities based on the commands received.

Decoupling of Components: EDA allows for the decoupling of event producers (users issuing commands) and event consumers (the bot's logic responding to the commands). This separation enhances flexibility and scalability, as the bot's reaction to different commands can be modified independently without affecting the overall system.

Scalability and Flexibility: As the bot grows to include more commands and functionalities, EDA makes it easier to add new event handlers without major modifications to the existing codebase. Each command can be treated as a separate event with its own unique handler.

Asynchronous Processing: EDA supports asynchronous processing, which is beneficial for a Discord bot operating in a potentially high-latency environment like networked communication. The bot can remain responsive to other events while processing a command.

Maintainability and Testability: The modular nature of EDA, where different events are handled by distinct components, aids in maintaining and testing the bot. Specific functionalities can be tested in isolation, ensuring that each event handler works as expected.

Application in Discord Bot

Event Handlers for Commands: Implement distinct event handlers for each command (!joke, !debug, !step, etc.). These handlers process the respective commands and interact with the Discord API to send appropriate responses.

Integration with Discord API: Use the event-driven model to listen for messages on Discord channels and trigger the corresponding event handlers.

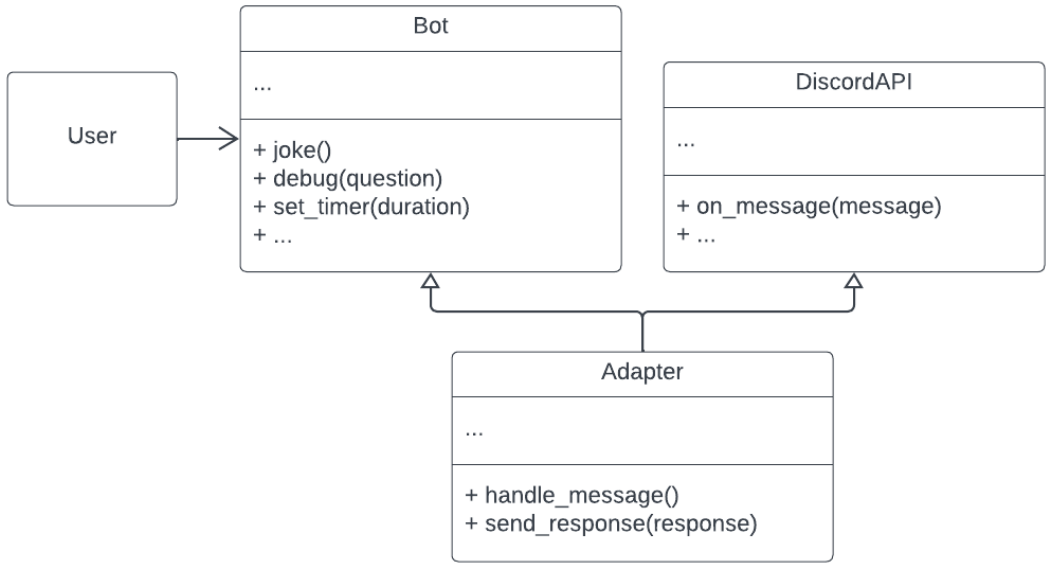
Modular Functionality: Organize the bot's features (Joke System, Rubber Duck Debugging, Break Period Timer) as separate modules that are invoked by their respective event handlers, promoting a clean and organized code structure.

By adopting an Event-Driven Architecture, the Discord bot will be well-equipped to handle user commands efficiently, be scalable for future enhancements, and maintain high responsiveness and reliability.

Low-level Design

The structural design pattern family may be helpful in implementing our discord bot. One of the requirements for our project is to use the Discord API to allow us to respond to user events. The challenge is determining the best way to interface with the API such that it is easy to add new functionality to our bot without rewriting the code each time. One potential approach would be to use an Adapter class. This allows our bot to interact more seamlessly with the DiscordAPI, by indirectly accessing it through the Adapter class. The Adapter class can act as a middleman to abstract only the relevant functionality from the DiscordAPI. Below is some pseudo code and a simple class diagram representing a potential approach for our implementation:

Class Diagram:



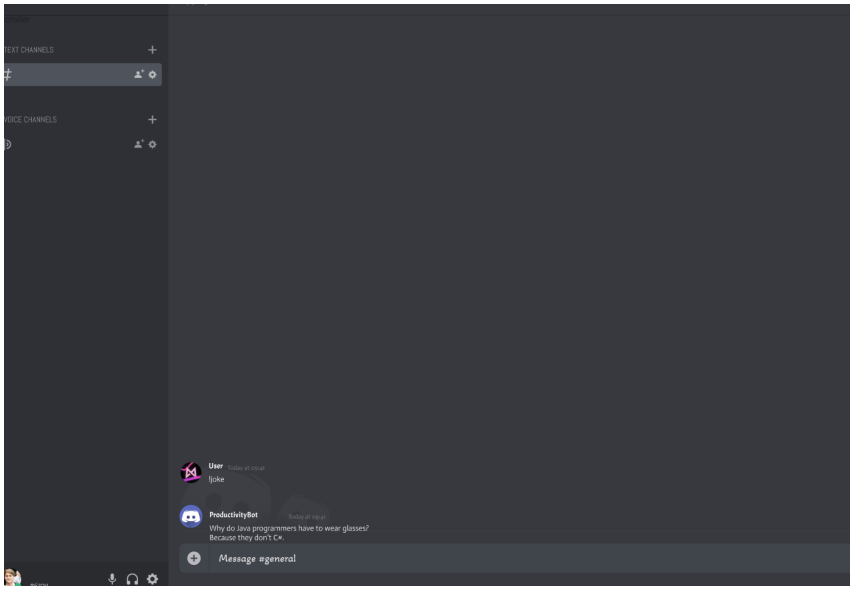
Pseudocode:

```
class Adapter:
    # This class will interact with Discord API directly

    def handle_message():
        # access discord API
        # do all of message handling here

    def send_response(response):
        # access discord API
        # handle responses to user

class Bot:
    adapter = Adapter()

    # interface with instance of adapter
    # rather than directly with discord API

    def joke():
        adapter.send_reponse(joke)
```

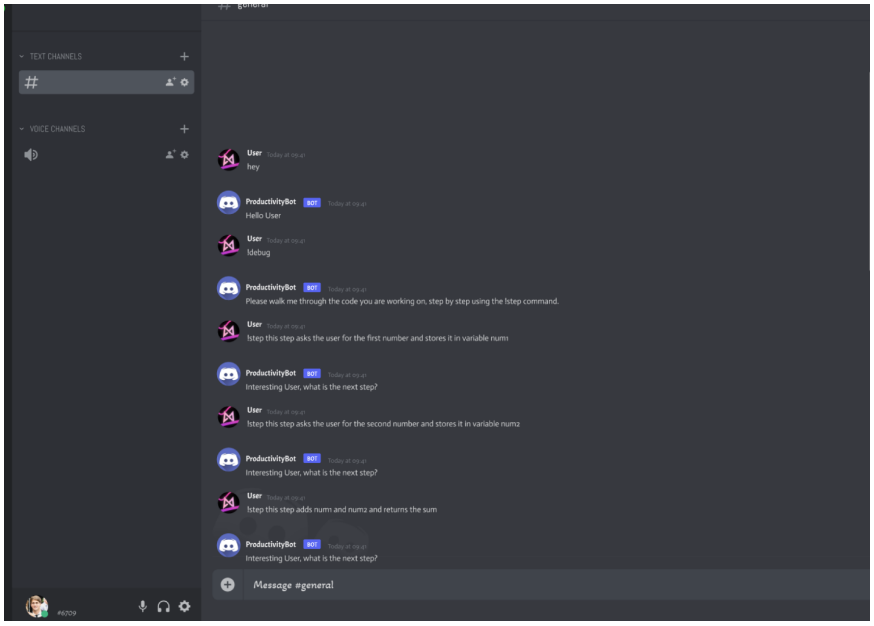
Design Sketch

We created three wireframes to simulate three distinct functions of the discord productivity bot. The first function, as shown in W1, simulates a user asking the bot for a joe by using the !joke command. The bot then replies with a light hearted, programming joke intended to lighten the mood of the software engineer who asked for the joke. THe next function is shown in W2, and simulates the user using the rubber duck method of debugging with the bot. In this case, the user uses the command !debug to prompt the bot, and the bot prompts the user to discuss their code by steps using the !step command so the bot knows when to comment on it by telling the user to move on to discuss the next step. Finally in W3, the bot creates a timer when the user needs to take a break, and notifies the user that there is going to be a thirty minute break, and it starts to count down to zero. There is a button present for the user to have the ability to stop the timer to finish their break early, but if they do not elect to use the button then after thirty minutes the bot sends another notification that lets the user know that their break is over.

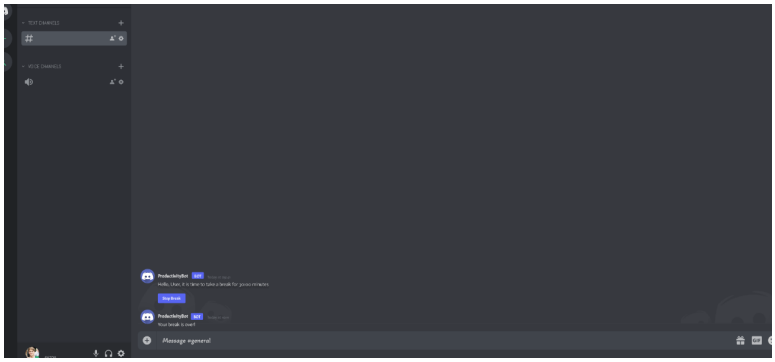
Joke System (W1)



Rubber Duck Debugging (W2)



Break period (W3)



Project Check-In

Aaron completed the survey for our group and used email aaronlambert@vt.edu for his response.

Process Deliverable (3%)

- Prototyping: submit a prototype of your system

For Milestone 3, we created a new iteration of our PM2 prototype that includes a basic rubber duck debugger function. Below is an example of the new functionality being used. The code itself is located at https://github.com/Sonny24/ProductivityBot.