GoPirate Multi-player game in Python Language

Marco Ponce, Brysen Pfingsten

Friday 9th May, 2025

Dr. Shajina Anand, CSAS 2124, Seton Hall University

Contents

1 Introduction

The focus of this project was to explore how object-oriented design patterns can be used to build a GUI interface that has a combination of a chatbot, player chat, and a jjk-battle game interface. It can be dynamic and its interaction with the users can be easily extensible for future updates. The chatbot is designed to handle 40 predefined queries, with 3 additional flexible queries, the player chat is created to handle text chats between users and list from who the message is from either by oneself or the name they put when logging into the GoPirate system. The jjk-game is a GUI text-based game where players will take turns battling each other until there is one person left standing who is declared the winner, they can either attack, defend, or use their special move that has a cooldown included.

To achieve this, I applied object-oriented principles and design patterns to create a full-stack software application. The frontend allows users to interact with each other by chatting, asking the bot for help throughout the battle, and battle the other players out through the command line. The backend is responsible for retrieving data from JSON files and generating responses when using the bot, using Sqlite3 to store the messages the users have amongst each other during the session for any censor improvements and it gives the chatbot the feel of a real large language model (LLM) utilizing retrieval-augmented generation (RAG). To control the behavior of the game getting out of control and not easily raided, we only allowed the first player that joined, also will be known as the “host” to start the game when everyone is ready, giving them more permissions in control of the start game button or when to restart from the beginning if people joined late.

2 OOP Principles

This project leverages fundamental object-oriented programming (OOP) principles to create a robust, maintainable, and extensible software application. Each component in the GoPirate Multi-player game is structured to promote modularity and scalability, along with security.

2.1 Encapsulation

Encapsulation involved bundling related data and behavior into classes while hiding internal implementation details. In GoPirate:

- **LiveAgentService** encapsulates the logic for agent availability, exposing only essential methods such as **get_available_agent()** and **QueryHandler** encapsulates the validation and processing logic unique to its specific query type.

- **InputName** is another use of encapsulation since they incorporate getter/setter methods and detailing whose turn it is and what character each player has chosen and is stored.

2.2 Abstraction

Abstraction simplifies complex interactions by exposing high-level interfaces while concealing intricate details. For instance:

- The abstract class **QueryHandler** defines a unified interface **handle()** for processing queries, allowing clients to remain unaware of underlying complexities and any unknown responses get stored appropriately for future updates.

- **UnifiedGUI** is defined to have all separate GUIs combined into one to show organization and more facility for the user to use the interface. This includes all functionality beneath the GameGUI, ChatGUI, and ChatBotGUI.

2.3 Inheritance

Inheritance allows classes to reuse common functionality, establishing clear hierarchical relationships from another class and avoiding duplicate code:

- The **LiveAgentService** inherits from a common **Singleton** superclass, reusing logic to enforce a single instance.

- **Characters** uses a lazy walking approach when creating the characters for availability for the users to choose from, along with **UnifiedGUI** where it orders the GUIs from the centralized GUI where it grabs the necessary information from each separate one to create common behavior from the abstract **UnifiedGUI** class.

2.4 Polymorphism

Polymorphism enables interchangeable use of classes through shared interfaces, enhancing system flexibility and enable the usage of method overriding:

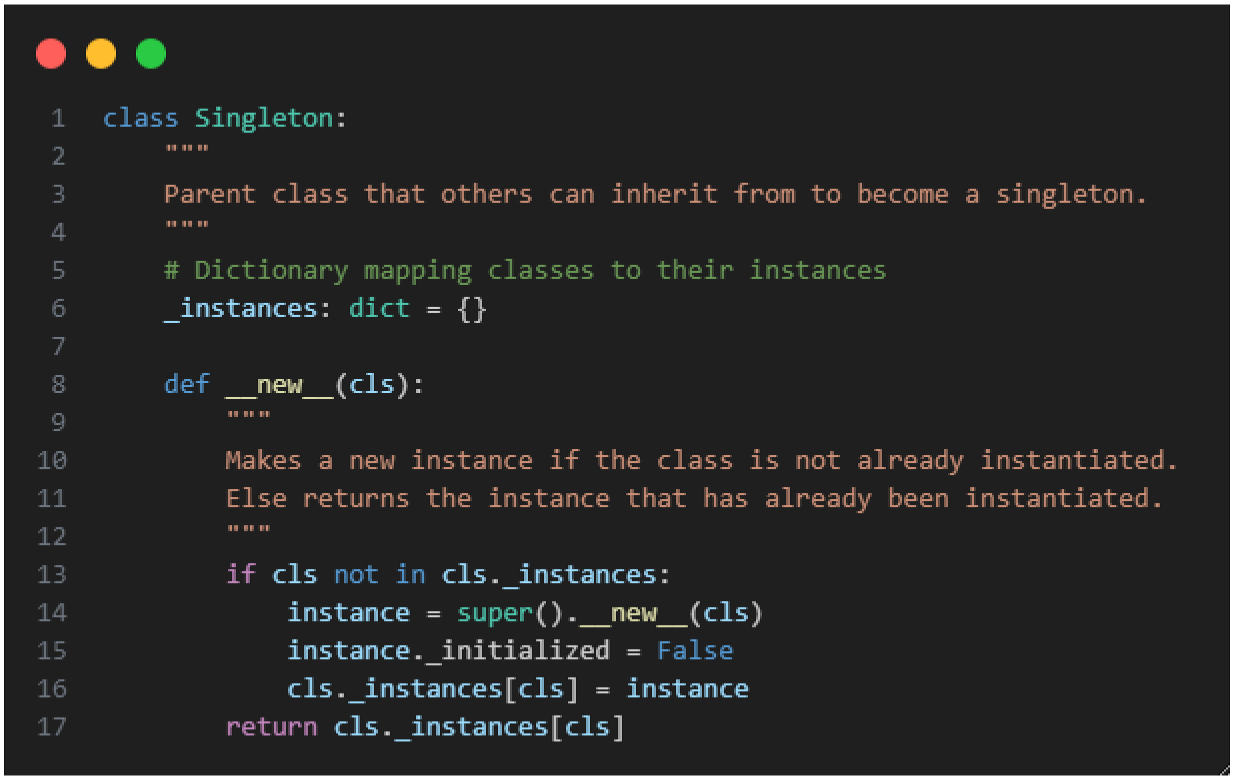
- Backend services accessed through the **BackendManager** use polymorphism to process diverse query types through a consistent interface.

- The **NetworkManager** is in control of processing different types of clients, the first client being considered the “host” and control of most of the games changes, meanwhile the rest are controlled as “players” which can play and interact with each other.

3 Creational Design Patterns

3.1 Singleton

The **Singleton** pattern is used extensively throughout this project to ensure that only one instance of a server exists at all times to avoid vulnerability in the client and server connections. To streamline this process, we created a base Singleton class that other classes could inherit from. This class overrides the __new__ method and stores instances in a shared dictionary, allowing for efficient and consistent access to a single software instance across the application.

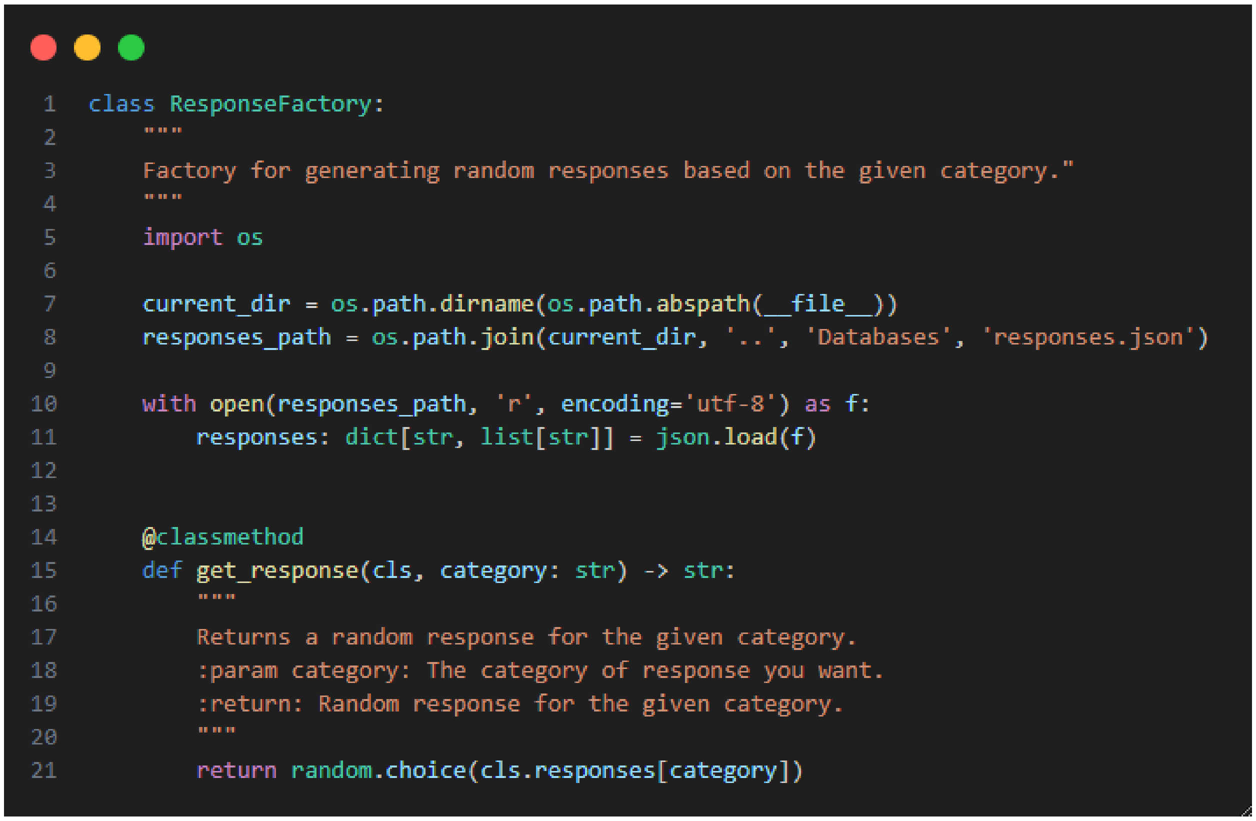


```python
class Singleton:
    """
    Parent class that others can inherit from to become a singleton.
    """
    # Dictionary mapping classes to their instances
    _instances: dict = {}

    def __new__(cls):
        """
        Makes a new instance if the class is not already instantiated.
        Else returns the instance that has already been instantiated.
        """
        if cls not in cls._instances:
            instance = super().__new__(cls)
            instance._initialized = False
            cls._instances[cls] = instance
        return cls._instances[cls]
```

When applied to backend services, it ensures consistent and synchronized data as the user interacts with GoPirate. Loading the JSON data from a file and structure it into a list of **Order** objects. Preventing more than one server to run at a time for this project allows correct behavior and predictable assumptions to be made with outcomes.

3.2 Factory

Another creational pattern used in this project is the **Factory** pattern. This pattern encapsulates the object creation logic, allowing for greater flexibility and decoupling in how new instances are constructed. A key example is the **ResponseFactory** where dynamic responses are generated for the specified query category.

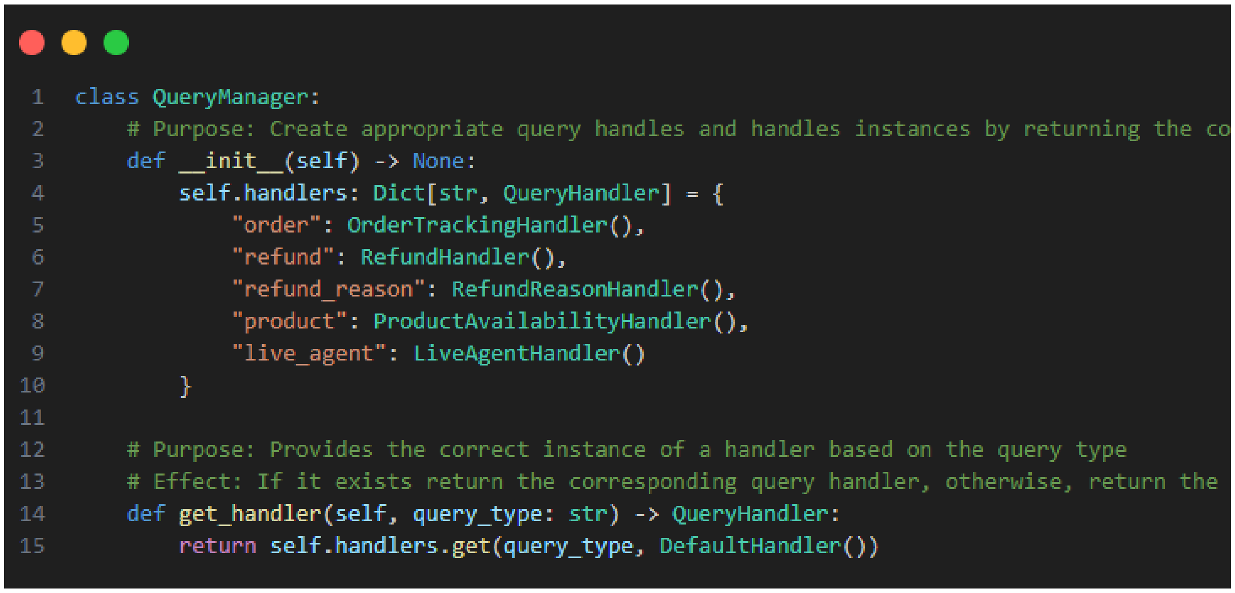


```python
class ResponseFactory:
    """
    Factory for generating random responses based on the given category."
    """
    import os

    current_dir = os.path.dirname(os.path.abspath(__file__))
    responses_path = os.path.join(current_dir, '..', 'Databases', 'responses.json')

    with open(responses_path, 'r', encoding='utf-8') as f:
        responses: dict[str, list[str]] = json.load(f)


    @classmethod
    def get_response(cls, category: str) -> str:
        """
        Returns a random response for the given category.
        :param category: The category of response you want.
        :return: Random response for the given category.
        """
        return random.choice(cls.responses[category])
```

It includes predefined response templates in order to reduce runtime variable substitution, enabling more natural and context-aware replies.

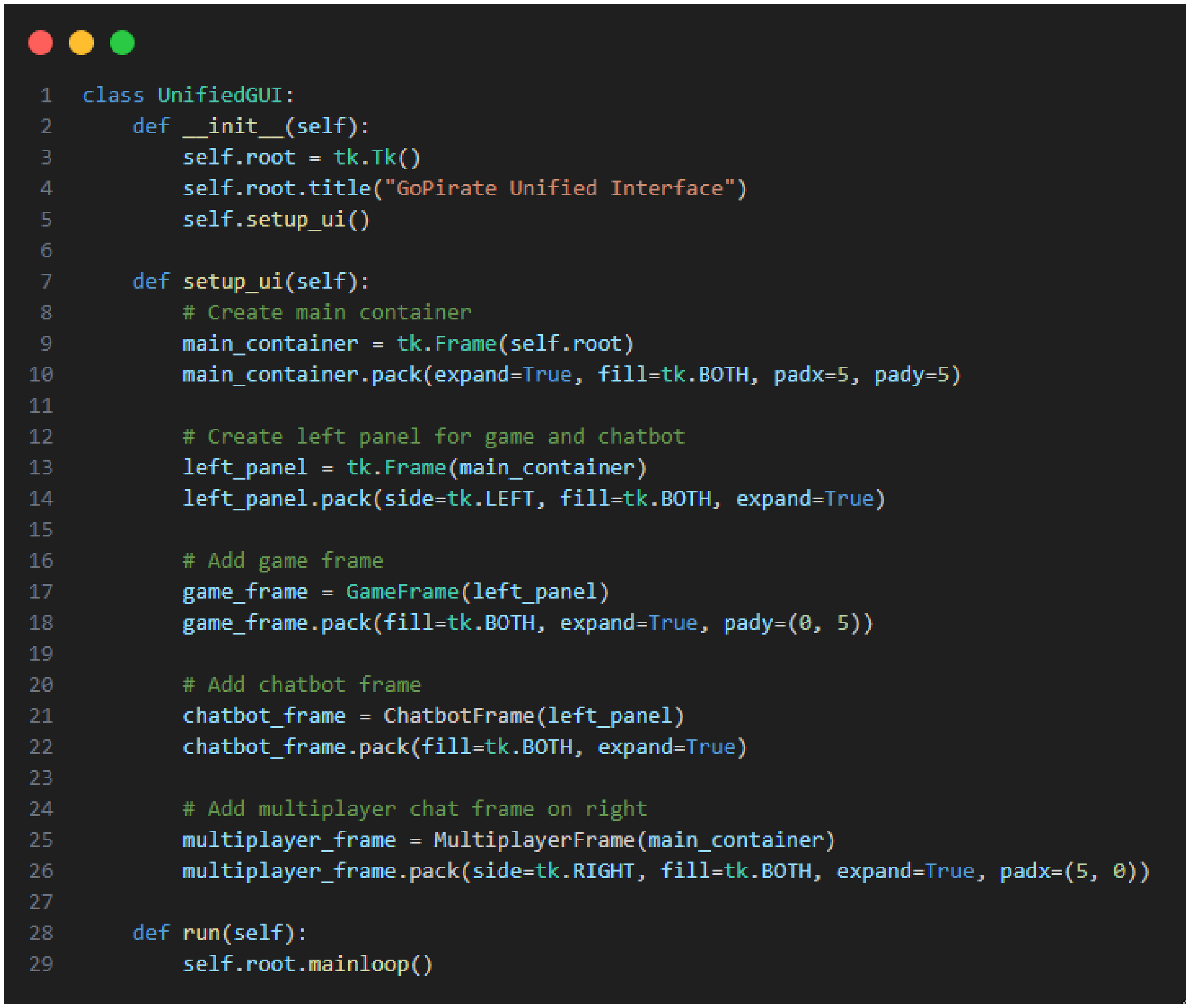
3.3 Abstract Factory

For this design pattern we wanted to make sure to include relatively similar characteristics easy and facilitate future updates with more inclusive characters and be able to update frequently according to user's feedback. Using the **QueryManager** to handle behaviors among functionalities. As well as creation of characters.



```python
class QueryManager:
    # Purpose: Create appropriate query handles and handles instances by returning the co
    def __init__(self) -> None:
        self.handlers: Dict[str, QueryHandler] = {
            "order": OrderTrackingHandler(),
            "refund": RefundHandler(),
            "refund_reason": RefundReasonHandler(),
            "product": ProductAvailabilityHandler(),
            "live_agent": LiveAgentHandler()
        }

    # Purpose: Provides the correct instance of a handler based on the query type
    # Effect: If it exists return the corresponding query handler, otherwise, return the
    def get_handler(self, query_type: str) -> QueryHandler:
        return self.handlers.get(query_type, DefaultHandler())
```

The usage of this class allows for future updates to become easily manipulated as well as the **UnifiedGUI** allows for GUIs to be updated with ease using the more general GUI factory that creates GUIs and exports them to be unified alongside the other interfaces.

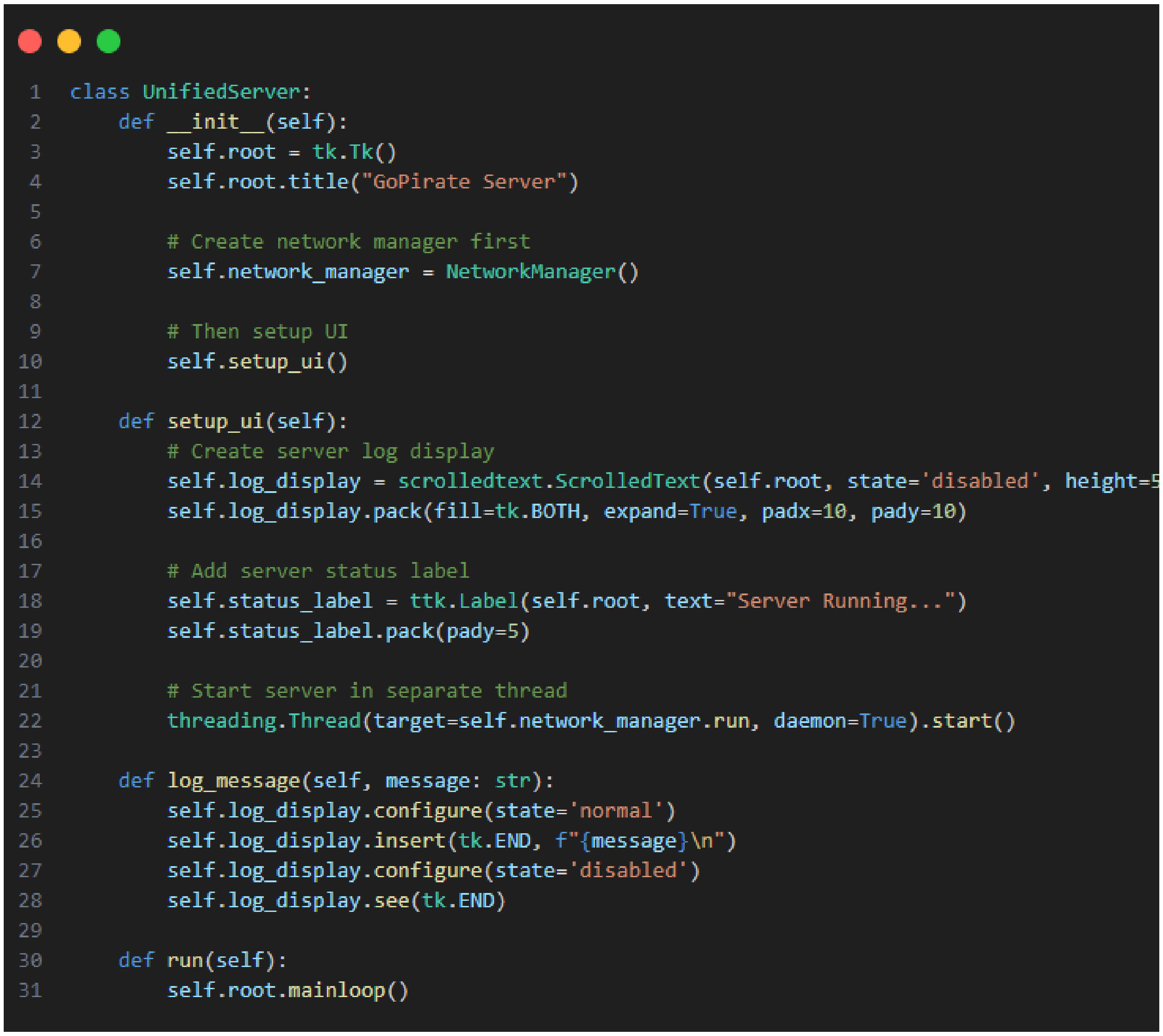


```python
class UnifiedGUI:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("GoPirate Unified Interface")
        self.setup_ui()

    def setup_ui(self):
        # Create main container
        main_container = tk.Frame(self.root)
        main_container.pack(expand=True, fill=tk.BOTH, padx=5, pady=5)

        # Create left panel for game and chatbot
        left_panel = tk.Frame(main_container)
        left_panel.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # Add game frame
        game_frame = GameFrame(left_panel)
        game_frame.pack(fill=tk.BOTH, expand=True, pady=(0, 5))

        # Add chatbot frame
        chatbot_frame = ChatbotFrame(left_panel)
        chatbot_frame.pack(fill=tk.BOTH, expand=True)

        # Add multiplayer chat frame on right
        multiplayer_frame = MultiplayerFrame(main_container)
        multiplayer_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=(5, 0))

    def run(self):
        self.root.mainloop()
```

4 Structural Design Patterns

4.1 Adapter

The Adapter design pattern was used to create an instance of a unified interface for the different services. The multiple services like the chat bot, player chat, and jjk-battle game all have different interfaces and communication protocols. This enables compatibility with the rest of our game and chat system by wrapping them into a standard interface that allows organization and easy responses for the user to have to look at one interface, rather than three.

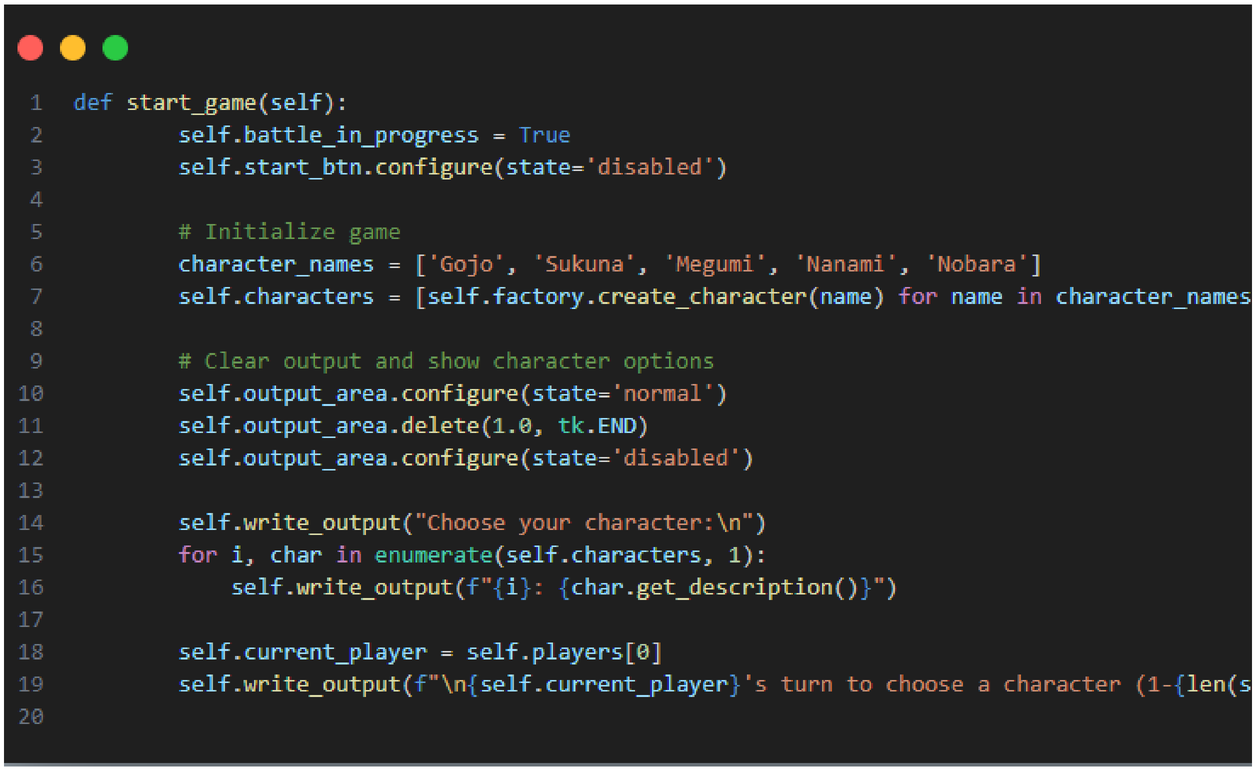


```python
class UnifiedServer:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("GoPirate Server")

        # Create network manager first
        self.network_manager = NetworkManager()

        # Then setup UI
        self.setup_ui()

    def setup_ui(self):
        # Create server log display
        self.log_display = scrolledtext.ScrolledText(self.root, state='disabled', height=5
        self.log_display.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # Add server status label
        self.status_label = ttk.Label(self.root, text="Server Running...")
        self.status_label.pack(pady=5)

        # Start server in separate thread
        threading.Thread(target=self.network_manager.run, daemon=True).start()

    def log_message(self, message: str):
        self.log_display.configure(state='normal')
        self.log_display.insert(tk.END, f"{message}\n")
        self.log_display.configure(state='disabled')
        self.log_display.see(tk.END)

    def run(self):
        self.root.mainloop()
```

Here we can see the **UnifiedServer** where it combines all the networks and connections and separates them to its own UI, to avoid functionality breaking once a GUI is in progress. This allows interaction amongst all the combined interfaces and runs the root in a loop for processing.

4.2 Proxy

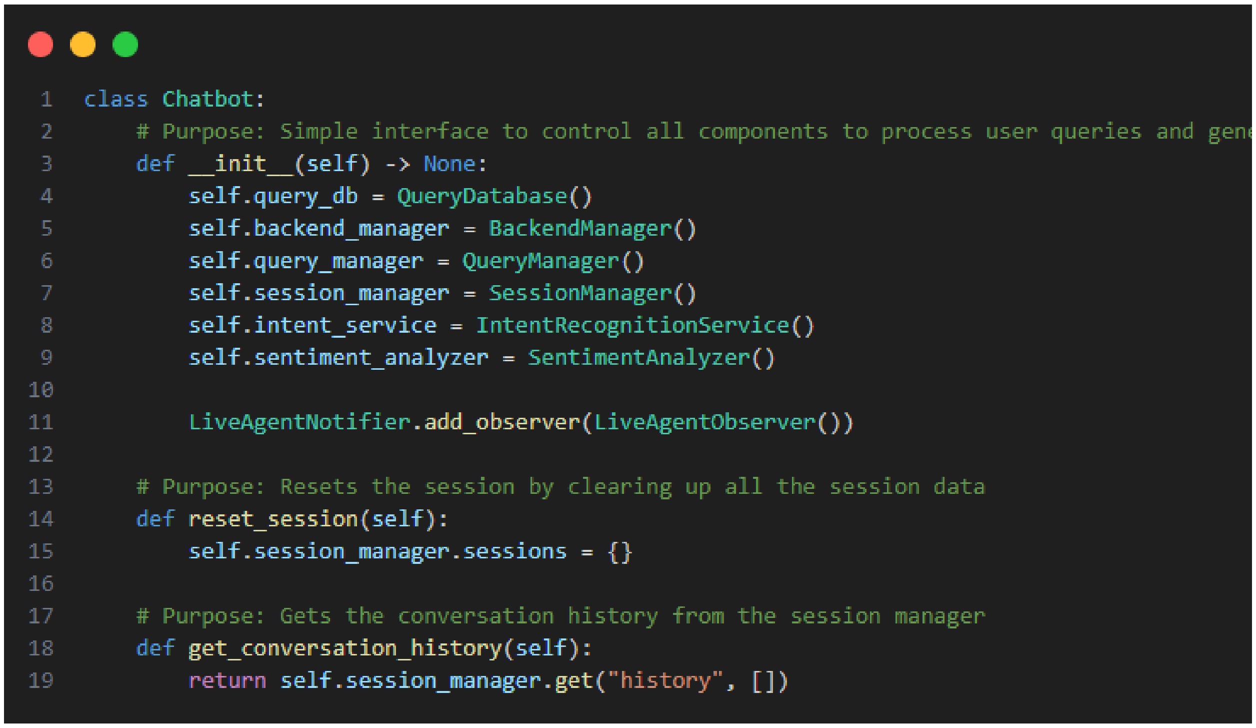
The usage of the proxy design pattern is used as a placeholder and surrogate for another object to control access to it. Like for instance, it is used for throughout the process of creating the characters, using lazy loading since there are only 5 characters which does not make it time consuming, instead it logs each character being created and caches them into storage and acknowledges which character has been used and popped out, and the remaining characters to remain functional within the cache storage.



```python
def start_game(self):
        self.battle_in_progress = True
        self.start_btn.configure(state='disabled')

        # Initialize game
        character_names = ['Gojo', 'Sukuna', 'Megumi', 'Nanami', 'Nobara']
        self.characters = [self.factory.create_character(name) for name in character_names

        # Clear output and show character options
        self.output_area.configure(state='normal')
        self.output_area.delete(1.0, tk.END)
        self.output_area.configure(state='disabled')

        self.write_output("Choose your character:\n")
        for i, char in enumerate(self.characters, 1):
            self.write_output(f"{i}: {char.get_description()}")

        self.current_player = self.players[0]
        self.write_output(f"\n{self.current_player}'s turn to choose a character (1-{len(s

```

4.3 Flyweight

Minimizing memory usage by sharing as much data as possible with similar objects is the main functionality of the flyweight design pattern. Created in the **ChatResponseFactory** to enhance the ability of quick responses and not consume as much memory to generate responses is included in the 40-query handler.



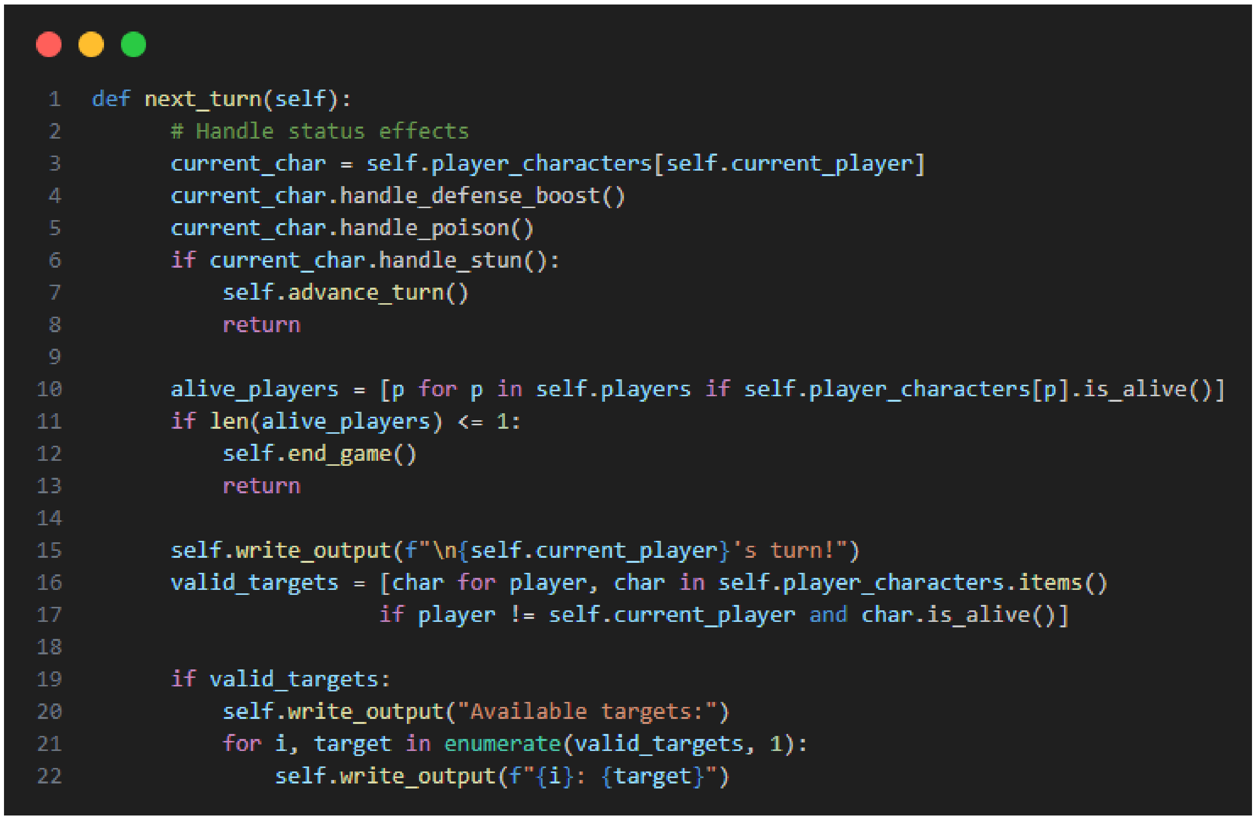
```python
class Chatbot:
    # Purpose: Simple interface to control all components to process user queries and gene
    def __init__(self) -> None:
        self.query_db = QueryDatabase()
        self.backend_manager = BackendManager()
        self.query_manager = QueryManager()
        self.session_manager = SessionManager()
        self.intent_service = IntentRecognitionService()
        self.sentiment_analyzer = SentimentAnalyzer()

        LiveAgentNotifier.add_observer(LiveAgentObserver())

    # Purpose: Resets the session by clearing up all the session data
    def reset_session(self):
        self.session_manager.sessions = {}

    # Purpose: Gets the conversation history from the session manager
    def get_conversation_history(self):
        return self.session_manager.get("history", [])
```

The usage of storage and memory efficiency allows for memory to be allocated correctly and be stored appropriately and sharing the data allows for faster manipulation and storage interaction amongst classes and objects. Instead of creating new players, new messages, or new objects in general, reusing the same instance reduces memory consumption.

5 Behavioral Design Patterns

5.1 Observer

The observer design pattern lets objects subscribe to and receive updates when another object changes state. This is used in the GUI/event-driven system alerting the change of the user interaction to enhance the handler system that the user is currently using.

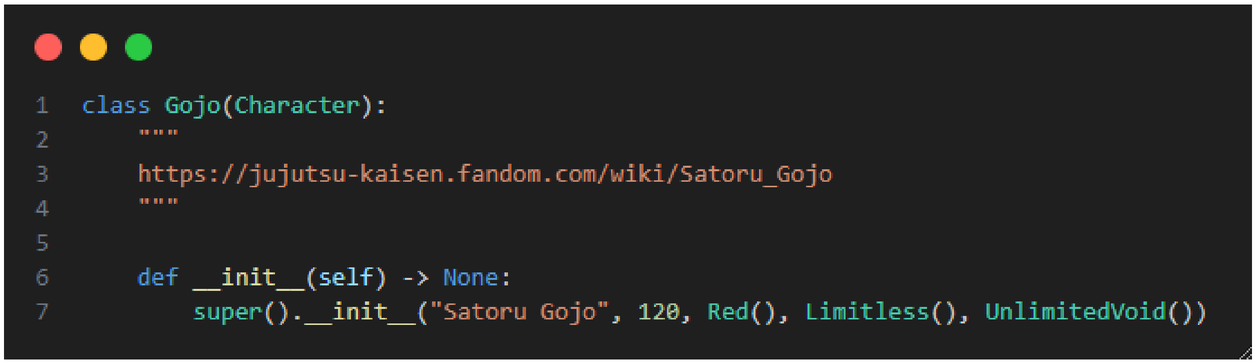


```python
def next_turn(self):
        # Handle status effects
        current_char = self.player_characters[self.current_player]
        current_char.handle_defense_boost()
        current_char.handle_poison()
        if current_char.handle_stun():
            self.advance_turn()
            return

        alive_players = [p for p in self.players if self.player_characters[p].is_alive()]
        if len(alive_players) <= 1:
            self.end_game()
            return

        self.write_output(f"\n{self.current_player}'s turn!")
        valid_targets = [char for player, char in self.player_characters.items()
                        if player != self.current_player and char.is_alive()]

        if valid_targets:
            self.write_output("Available targets:")
            for i, target in enumerate(valid_targets, 1):
                self.write_output(f"{i}: {target}")
```

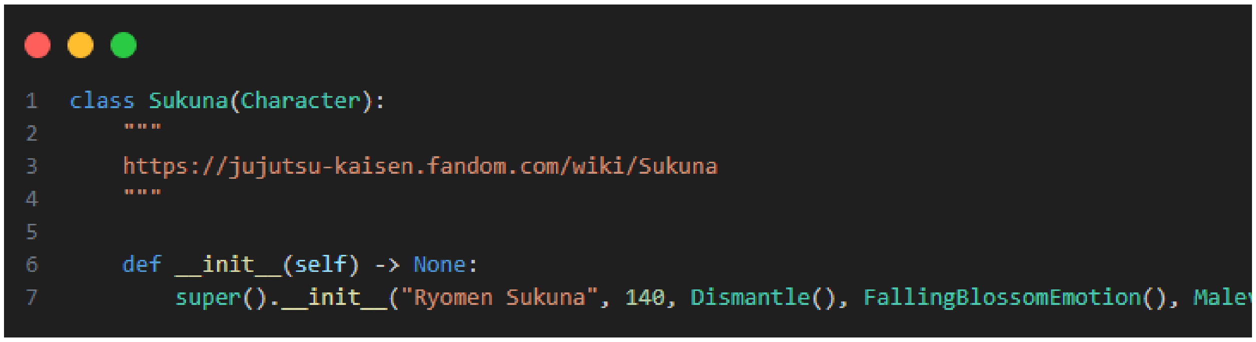
Seeing also a start game button for the ”host” user is to notify all listeners which are the clients in the same server as the host and the functionality beneath the GUIs and updates the system to alert the game state transition and load in the functionality of the jjk-battle game GUI in the top-left. This alert will be consistent allowing for all future interfaces to be alerted when it is the next person’s turn and then continue the game state until there is one player left and will be the deciding winner.

5.2 Chain of Responsibility

This pattern passes a request along a chain of handlers. Each handler decides whether to process the request or pass it to the next. Such action is taken when handling the queries of the question and expected answers for that question. The **CharacterInfoHandler** tries each one and which character handles the query the best is the information given to the user, the system will first try a sample handle then be passed onto the correct handler for instance, **Gojo** or **Sukuna** to return to necessary information.



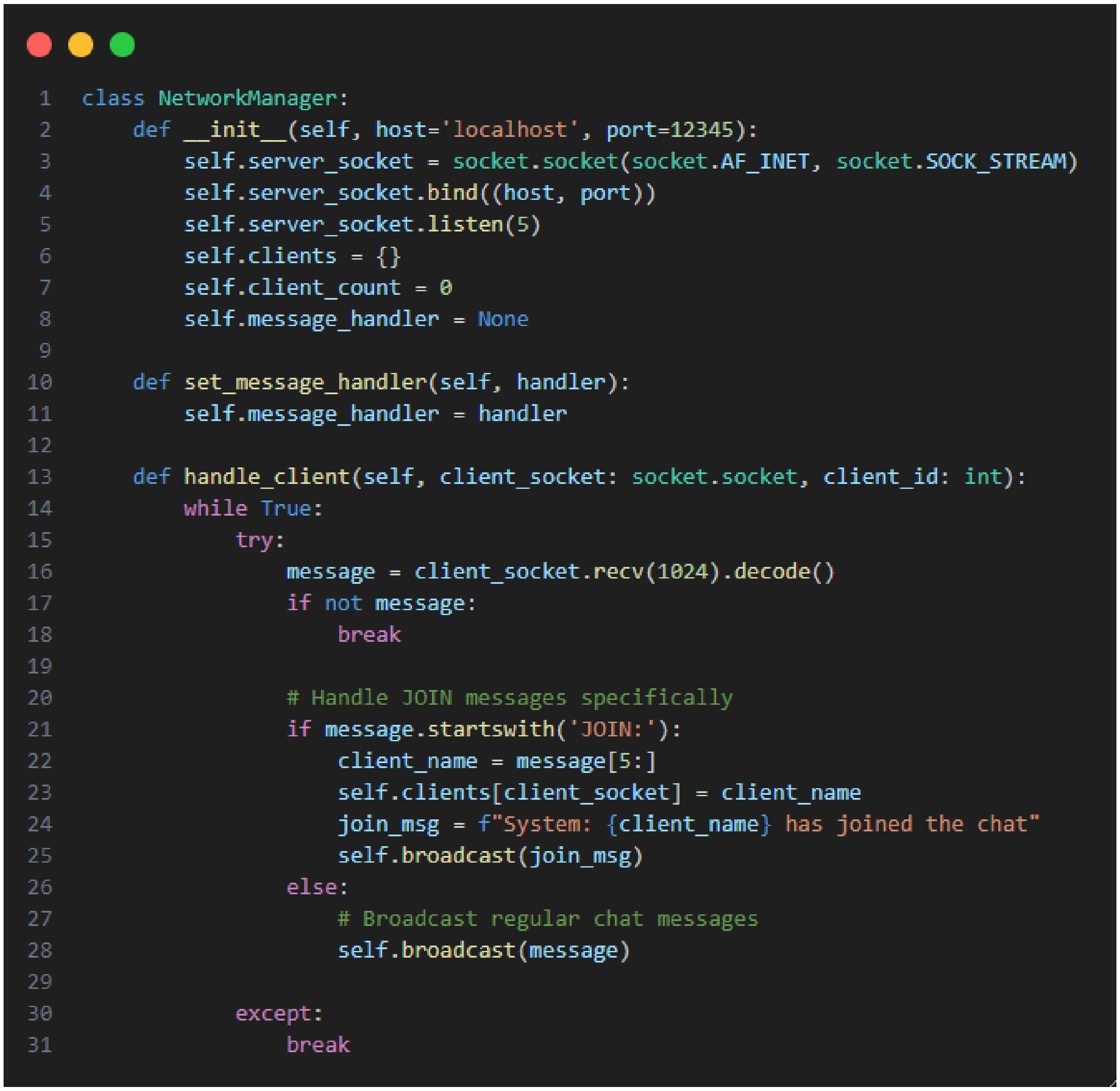
```python
class Gojo(Character):
    """
    https://jujutsu-kaisen.fandom.com/wiki/Satoru_Gojo
    """

    def __init__(self) -> None:
        super().__init__("Satoru Gojo", 120, Red(), Limitless(), UnlimitedVoid())
```



```python
class Sukuna(Character):
    """
    https://jujutsu-kaisen.fandom.com/wiki/Sukuna
    """

    def __init__(self) -> None:
        super().__init__("Ryomen Sukuna", 140, Dismantle(), FallingBlossomEmotion(), Male
```

5.3 Command

Encapsulates a request as an object, allowing us to parameterize clients with commands and support to undo or retry functionality. If a client entered an invalid character index and is unable to attack or use their special move, they receive a message alerting them to choose again and is prompted the initial message again not skipping their turn.

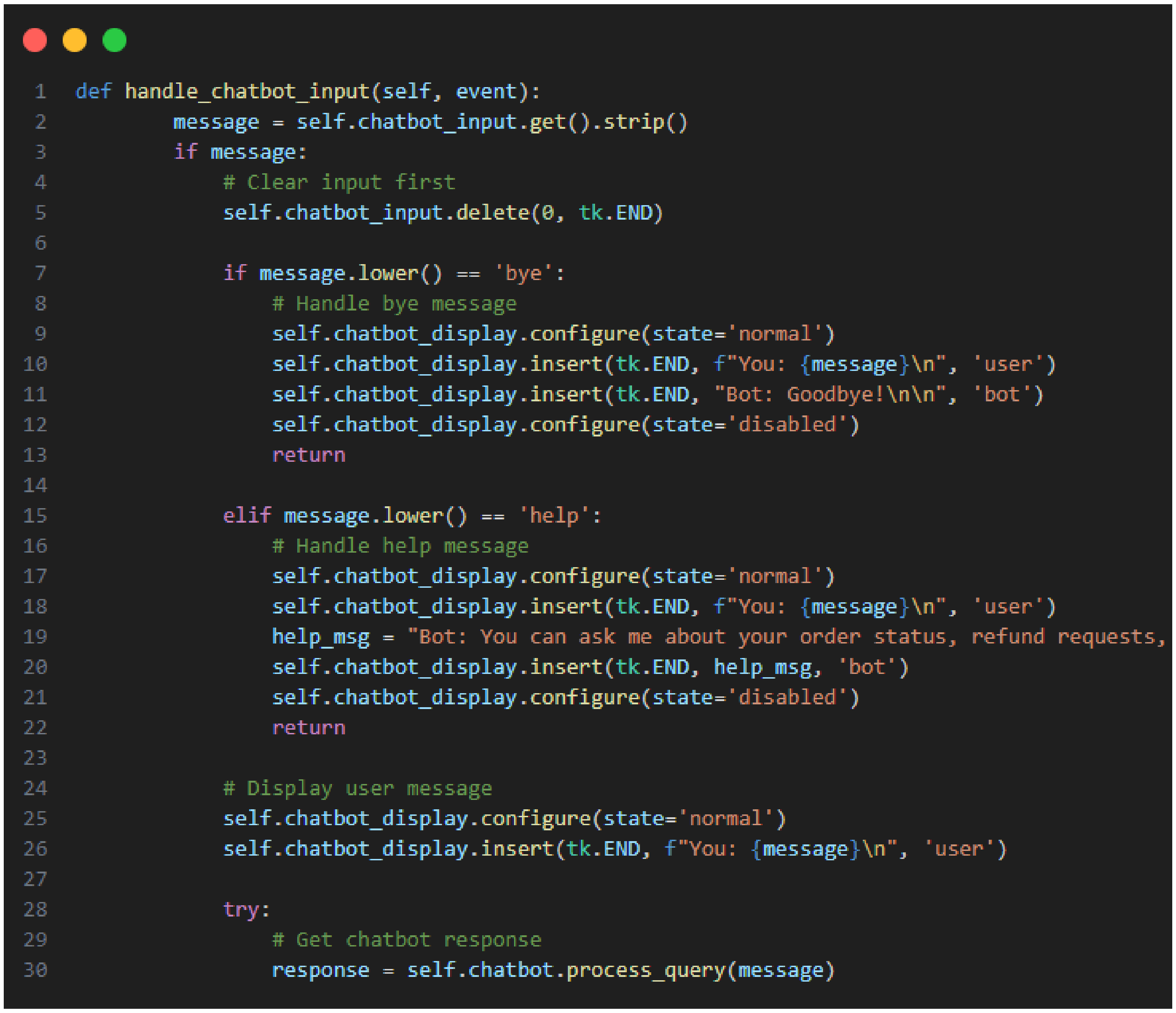


```python
class NetworkManager:
    def __init__(self, host='localhost', port=12345):
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.bind((host, port))
        self.server_socket.listen(5)
        self.clients = {}
        self.client_count = 0
        self.message_handler = None

    def set_message_handler(self, handler):
        self.message_handler = handler

    def handle_client(self, client_socket: socket.socket, client_id: int):
        while True:
            try:
                message = client_socket.recv(1024).decode()
                if not message:
                    break

                # Handle JOIN messages specifically
                if message.startswith('JOIN:'):
                    client_name = message[5:]
                    self.clients[client_socket] = client_name
                    join_msg = f"System: {client_name} has joined the chat"
                    self.broadcast(join_msg)
                else:
                    # Broadcast regular chat messages
                    self.broadcast(message)

            except:
                break
```

This handles all functionality and a player’s input triggers a **SpecialMove** or **Attack** or **Defend** class that is then used to execute and perform an action based on the character and does different effects to the other players.

5.4 Strategy

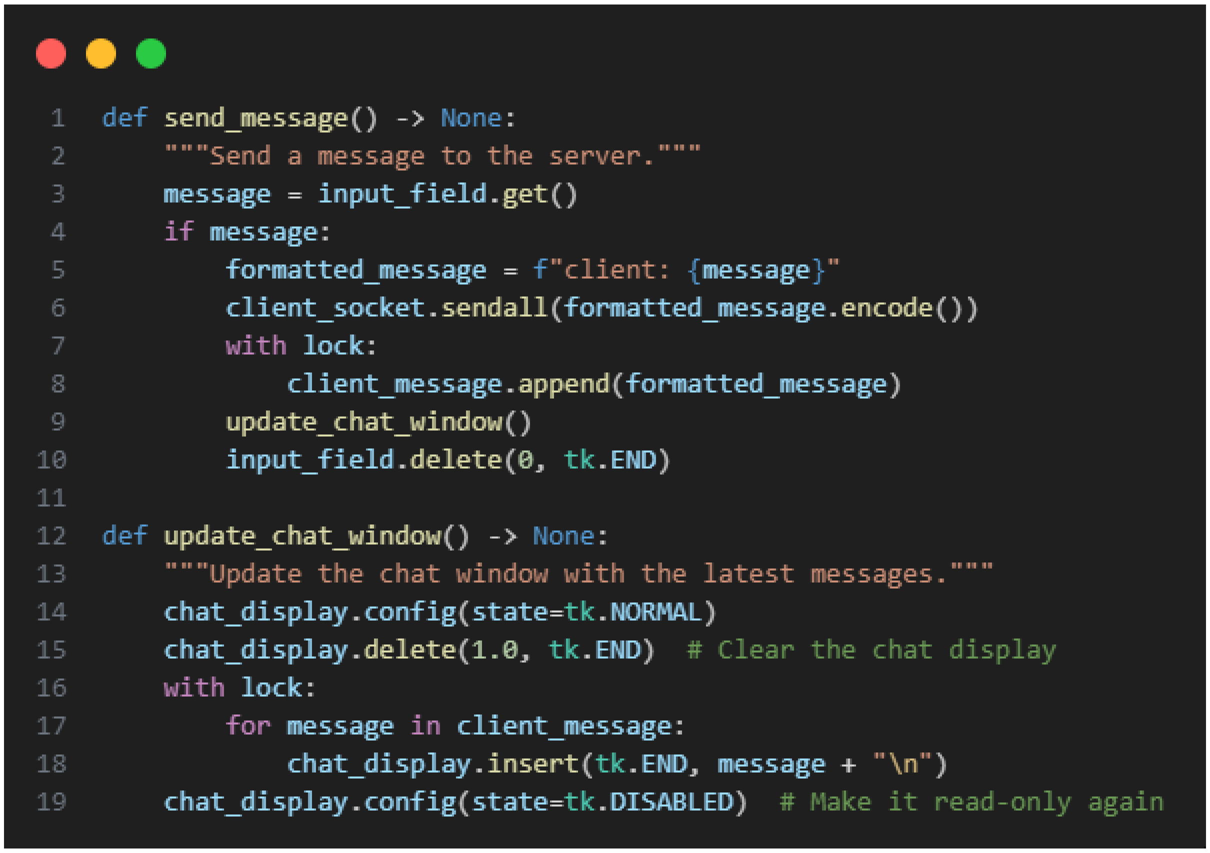
Is considered a family of computations, encapsulates each one, and makes them interchangeable and used for different query handlers in the chat bot system where each type or query is handled for generating a response that is most appropriate for the question at runtime. If the question is more of a casual response, then the **CasualChat** class will be able to handle it, likewise for other types of queries each handling the appropriate response type. Also, the chat bot is used for helping the user get assistance throughout the game, this will create a system of computations and possible moves to ensure victory over the other players.



```python
def handle_chatbot_input(self, event):
        message = self.chatbot_input.get().strip()
        if message:
            # Clear input first
            self.chatbot_input.delete(0, tk.END)

            if message.lower() == 'bye':
                # Handle bye message
                self.chatbot_display.configure(state='normal')
                self.chatbot_display.insert(tk.END, f"You: {message}\n", 'user')
                self.chatbot_display.insert(tk.END, "Bot: Goodbye!\n\n", 'bot')
                self.chatbot_display.configure(state='disabled')
                return

            elif message.lower() == 'help':
                # Handle help message
                self.chatbot_display.configure(state='normal')
                self.chatbot_display.insert(tk.END, f"You: {message}\n", 'user')
                help_msg = "Bot: You can ask me about your order status, refund requests,
                self.chatbot_display.insert(tk.END, help_msg, 'bot')
                self.chatbot_display.configure(state='disabled')
                return

            # Display user message
            self.chatbot_display.configure(state='normal')
            self.chatbot_display.insert(tk.END, f"You: {message}\n", 'user')

            try:
                # Get chatbot response
                response = self.chatbot.process_query(message)
```

5.5 State

Allows an object to change its behavior when its internal state changes, as if the object changed its class. This is the functionality that the **UnifiedGUI** follows beneath where it handles the user's interactions but also is able to understand and handle which GUI the user is currently using to not disable or malfunction the other GUIs not being used.

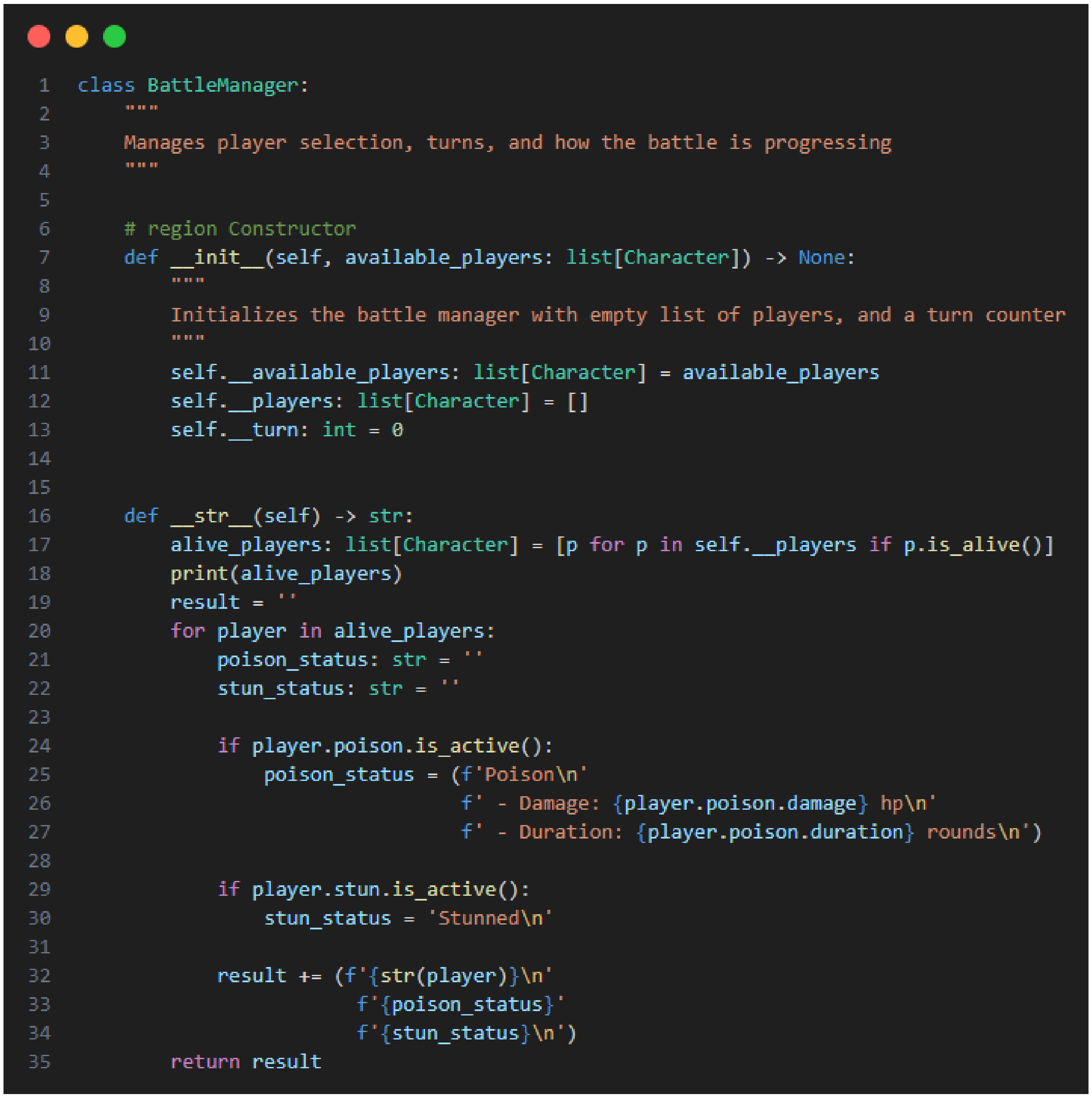


```python
def send_message() -> None:
    """Send a message to the server."""
    message = input_field.get()
    if message:
        formatted_message = f"client: {message}"
        client_socket.sendall(formatted_message.encode())
        with lock:
            client_message.append(formatted_message)
        update_chat_window()
        input_field.delete(0, tk.END)

def update_chat_window() -> None:
    """Update the chat window with the latest messages."""
    chat_display.config(state=tk.NORMAL)
    chat_display.delete(1.0, tk.END)  # Clear the chat display
    with lock:
        for message in client_message:
            chat_display.insert(tk.END, message + "\n")
    chat_display.config(state=tk.DISABLED)  # Make it read-only again
```

Here is where the client handles the client to others and changes the state of the message. For instance, if the user is sending a message to another client, on their end it would say “You: {message}” but for the other user it will be “{name}: {message}” this follows the same constraint of functionality and format as other popular games.

5.6 Template Method

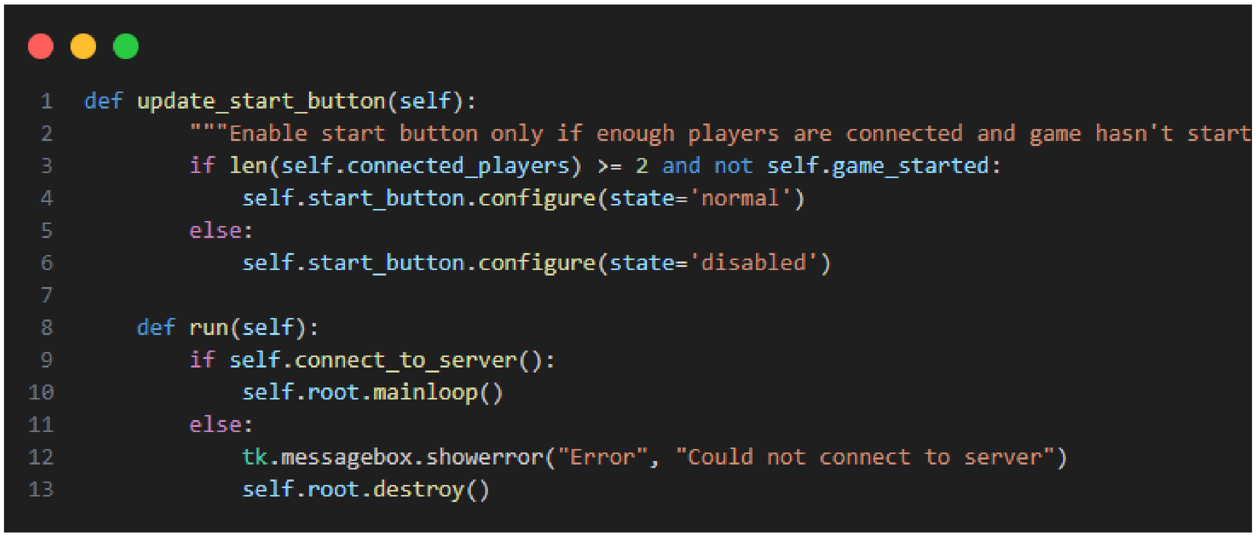
The skeleton of a system in a class, following with methods deferring some steps to subclasses and accepting all functionality. This is used for the **CharacterFactory** base class where it defines the blueprint of how characters should behave, **turn()** and **apply()** are methods that are used to create this blueprint of functionality and depending on how many users there are and what character they have chosen will have different functionality but similar aspects.



```python
class BattleManager:
    """
    Manages player selection, turns, and how the battle is progressing
    """

    # region Constructor
    def __init__(self, available_players: list[Character]) -> None:
        """
        Initializes the battle manager with empty list of players, and a turn counter
        """
        self.__available_players: list[Character] = available_players
        self.__players: list[Character] = []
        self.__turn: int = 0


    def __str__(self) -> str:
        alive_players: list[Character] = [p for p in self.__players if p.is_alive()]
        print(alive_players)
        result = ''
        for player in alive_players:
            poison_status: str = ''
            stun_status: str = ''

            if player.poison.is_active():
                poison_status = (f'Poison\n'
                                 f' - Damage: {player.poison.damage} hp\n'
                                 f' - Duration: {player.poison.duration} rounds\n')

            if player.stun.is_active():
                stun_status = 'Stunned\n'

            result += (f'{str(player)}\n'
                       f'{poison_status}'
                       f'{stun_status}\n')
        return result
```

5.7 Memento

Capturing and externalizing an object’s internal state so that in the future it can be restored without violating encapsulation. This is how we implemented the functionality of the game. Once the game ends and decides the winners, it will store the users and winner in a database, for future updates can be used for tournaments being added and other fun aspects. However, once the game ends, the start button becomes enabled again and the other buttons become disabled to avoid malfunctions and misinputs, then the “host” client can then decide to restart the game or wait for more users to join in and be able to participate alongside the already existing players.

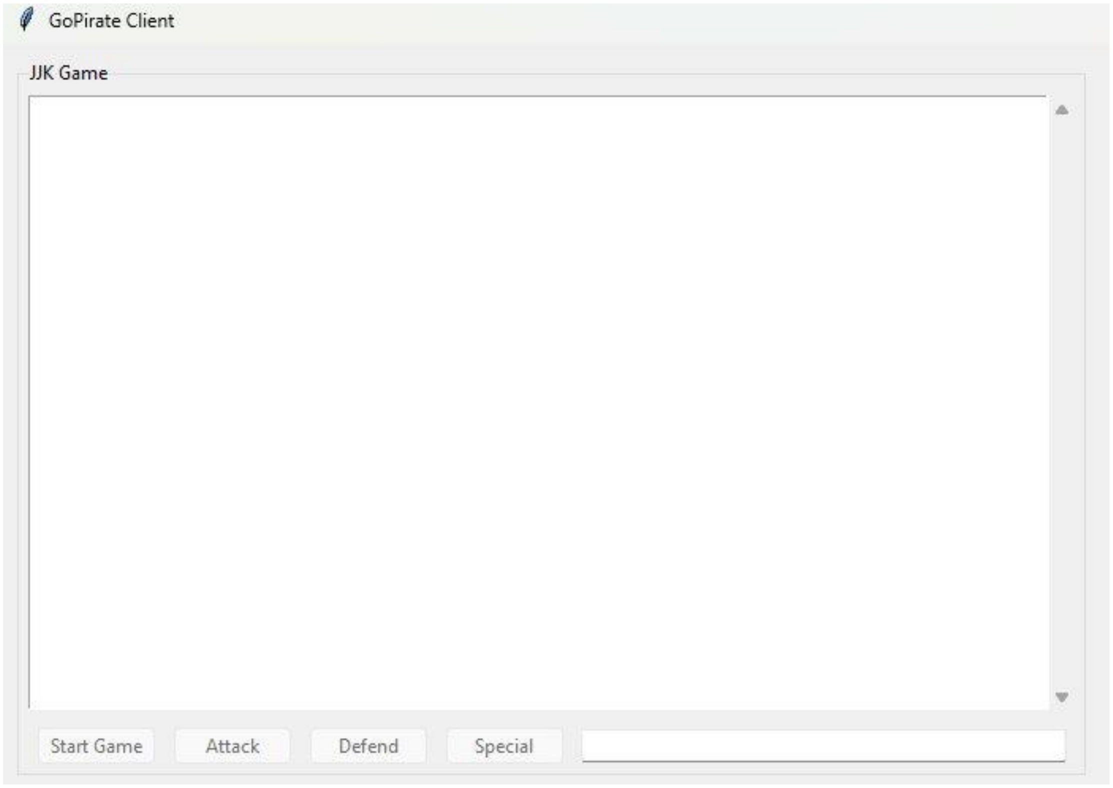


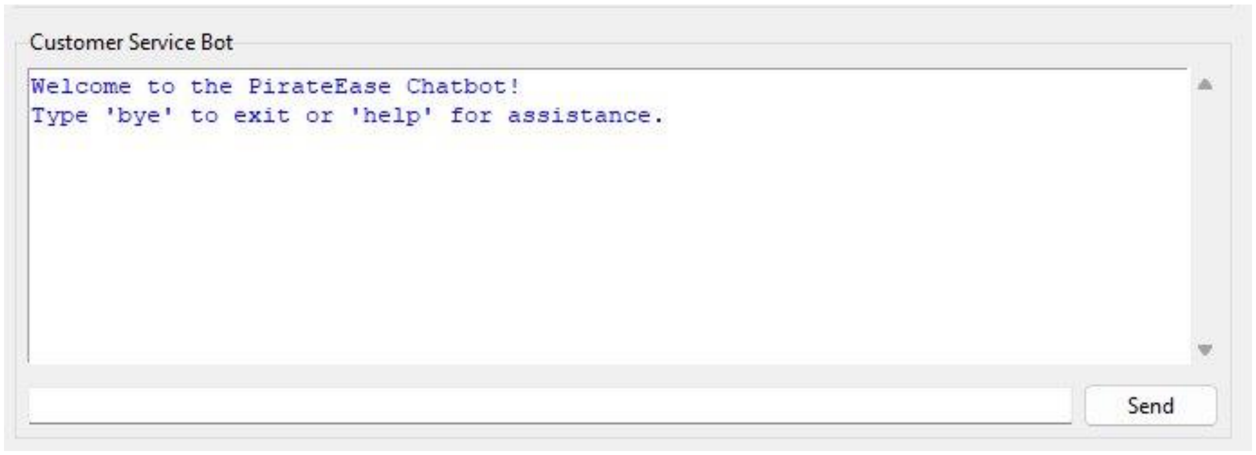
```
def update_start_button(self):
        """Enable start button only if enough players are connected and game hasn't start
        if len(self.connected_players) >= 2 and not self.game_started:
            self.start_button.configure(state='normal')
        else:
            self.start_button.configure(state='disabled')

    def run(self):
        if self.connect_to_server():
            self.root.mainloop()
        else:
            tk.messagebox.showerror("Error", "Could not connect to server")
            self.root.destroy()
```

This also includes the functionality of the button being enabled to begin with, if there is only the host connected, there is no point in having the button enabled, so it waits till at least 2 people are in the server connected to then enable the button and avoid complications with just battling with oneself.

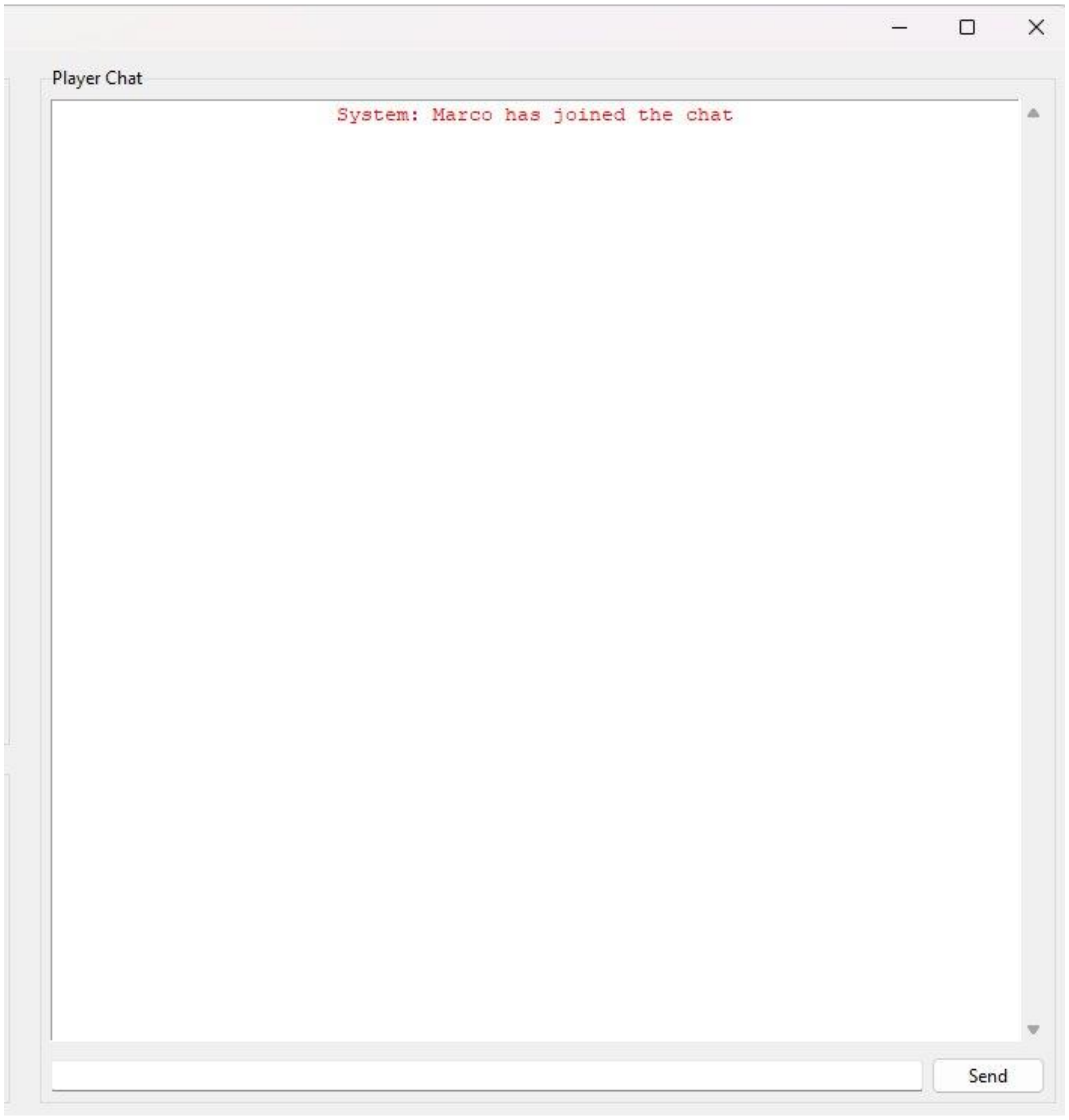
6 GUI Implementation

Each GUI design has its own separate interface; this allows for each to hold their functionality even if the client uses another interface. The JJK Game was designated to be in the top left corner of the main interface, the chatbot is right beneath it, and since most of the interacting is in the player chat, we made it the biggest on the right side of the interface, each interface having the ability to scroll and is protected from having players interfering with the GUI and typing over it.

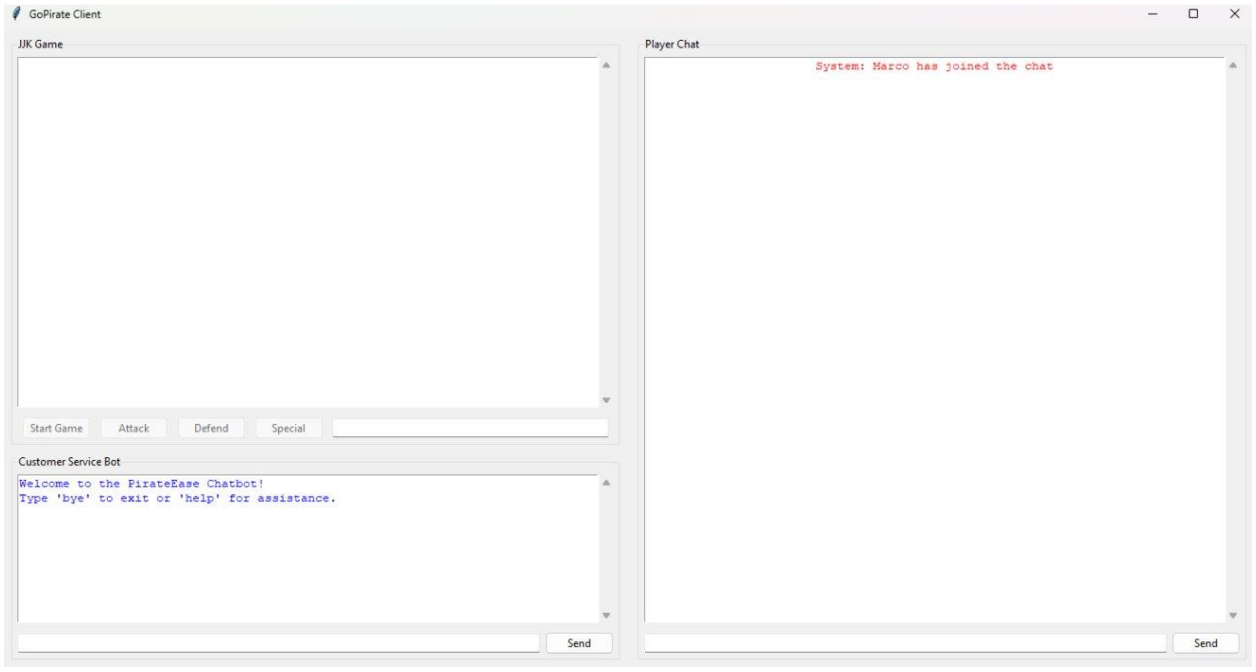




```
Welcome to the PirateEase Chatbot!
Type 'bye' to exit or 'help' for assistance.
```



```
System: Marco has joined the chat
```



```
System: Marco has joined the chat
```

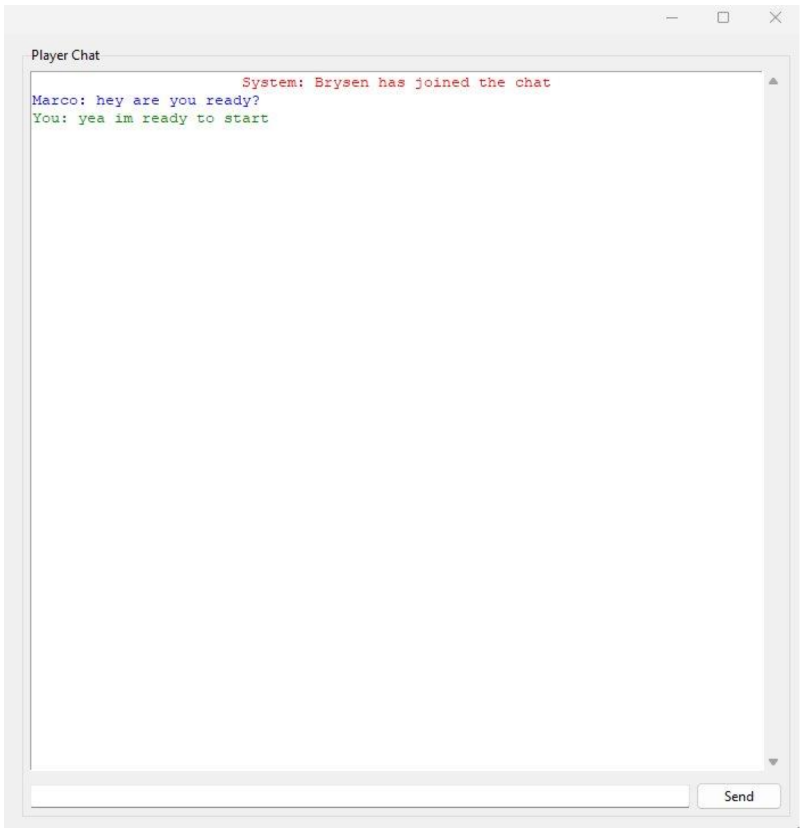
```
Welcome to the PirateEase Chatbot!
Type 'bye' to exit or 'help' for assistance.
```

7 Gameplay Action

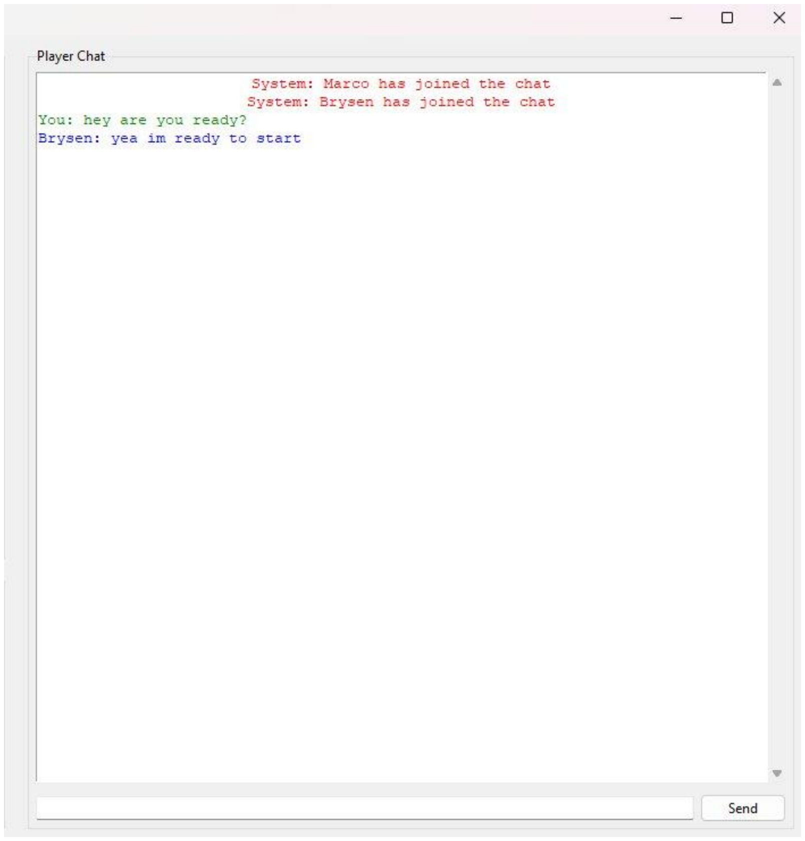
Once all the players have joined the server and entered their name, the first person who joined the server will be considered the "host". The host is the only one with permission to start the game. Once the game starts each player will take turns picking which character they would like to play and battle it out between each other until one remaining player is left standing, declared the winner.

8 Player Chat

Once a player joins, they will either their name or a message will be broadcasted from the system letting everyone know someone joined. The player-to-player chat will let new players come in and chat with others during mid-game and once a player leaves the system will broadcast it allowing everyone to know that a specific person has left, if the clients fail to connect to the server a "Connection to server lost" message will appear.



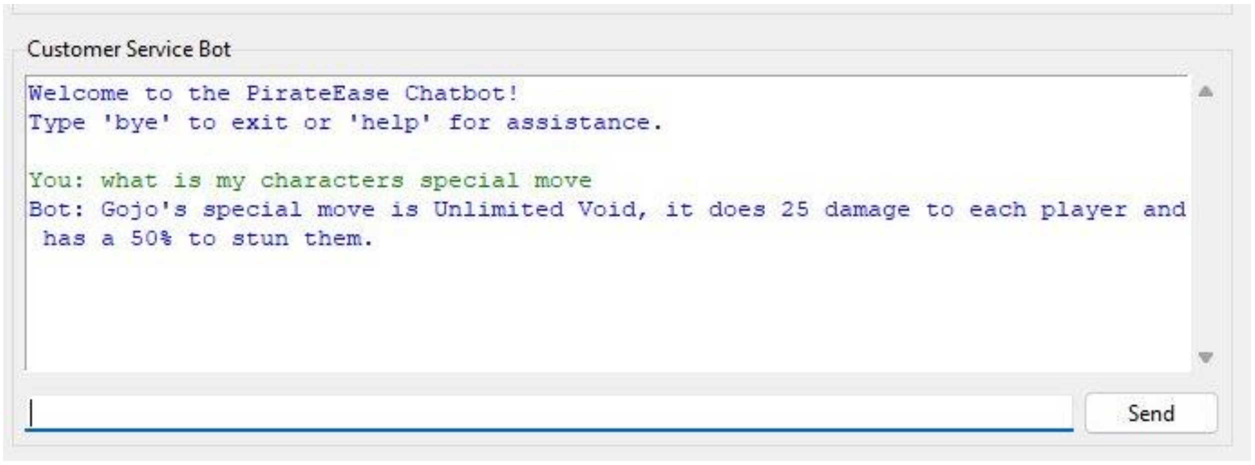
```
System: Brysen has joined the chat
Marco: hey are you ready?
You: yea im ready to start
```



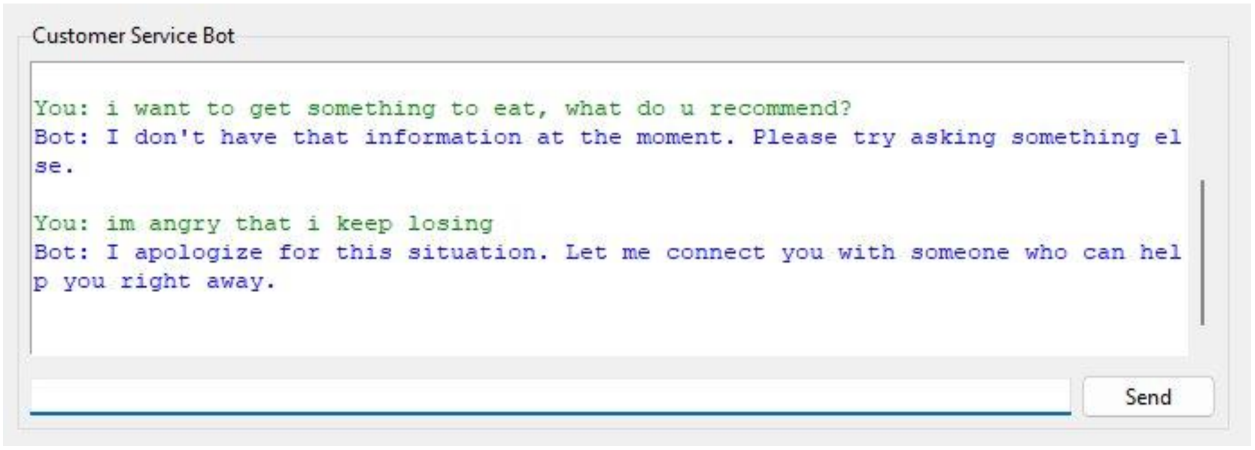
```
System: Marco has joined the chat
System: Brysen has joined the chat
You: hey are you ready?
Brysen: yea im ready to start
```

9 Chatbot Interaction

The smart chatbot handles each player's questions accordingly and has separate storages for each player, this will make it much faster and enhance the ability for the chatbot to be able to recognize what character that specific player chose to then directly connect with questions and answers for that specific character. If the chatbot is unable to produce an answer, it will simulate connecting with a live agent, as well as storing the unknown question into a json query file to be used for future updates.



```
Welcome to the PirateEase Chatbot!
Type 'bye' to exit or 'help' for assistance.

You: what is my characters special move
Bot: Gojo's special move is Unlimited Void, it does 25 damage to each player and
 has a 50% to stun them.
```



```
You: i want to get something to eat, what do u recommend?
Bot: I don't have that information at the moment. Please try asking something el
se.

You: im angry that i keep losing
Bot: I apologize for this situation. Let me connect you with someone who can hel
p you right away.
```

10 UML Diagram

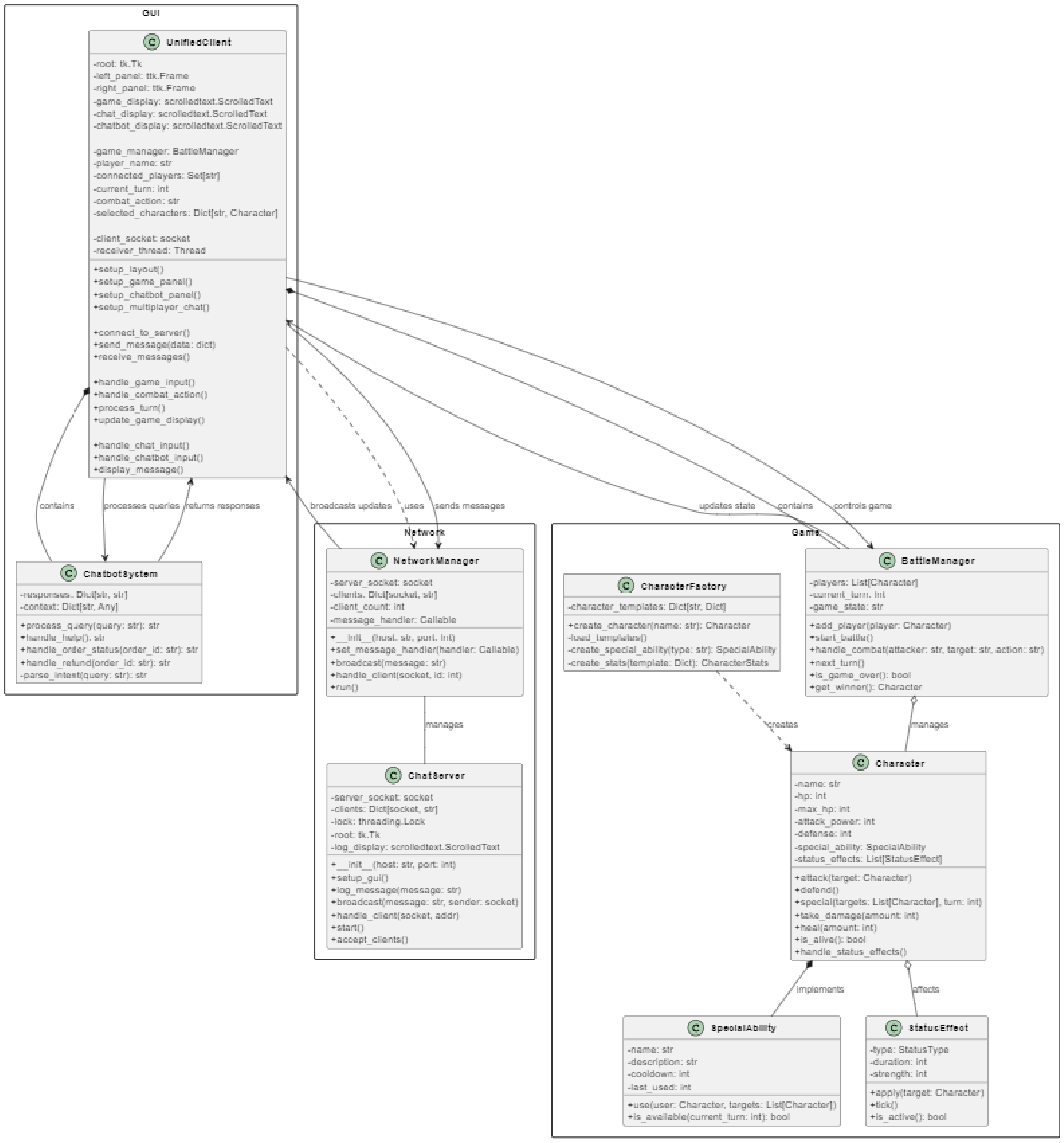
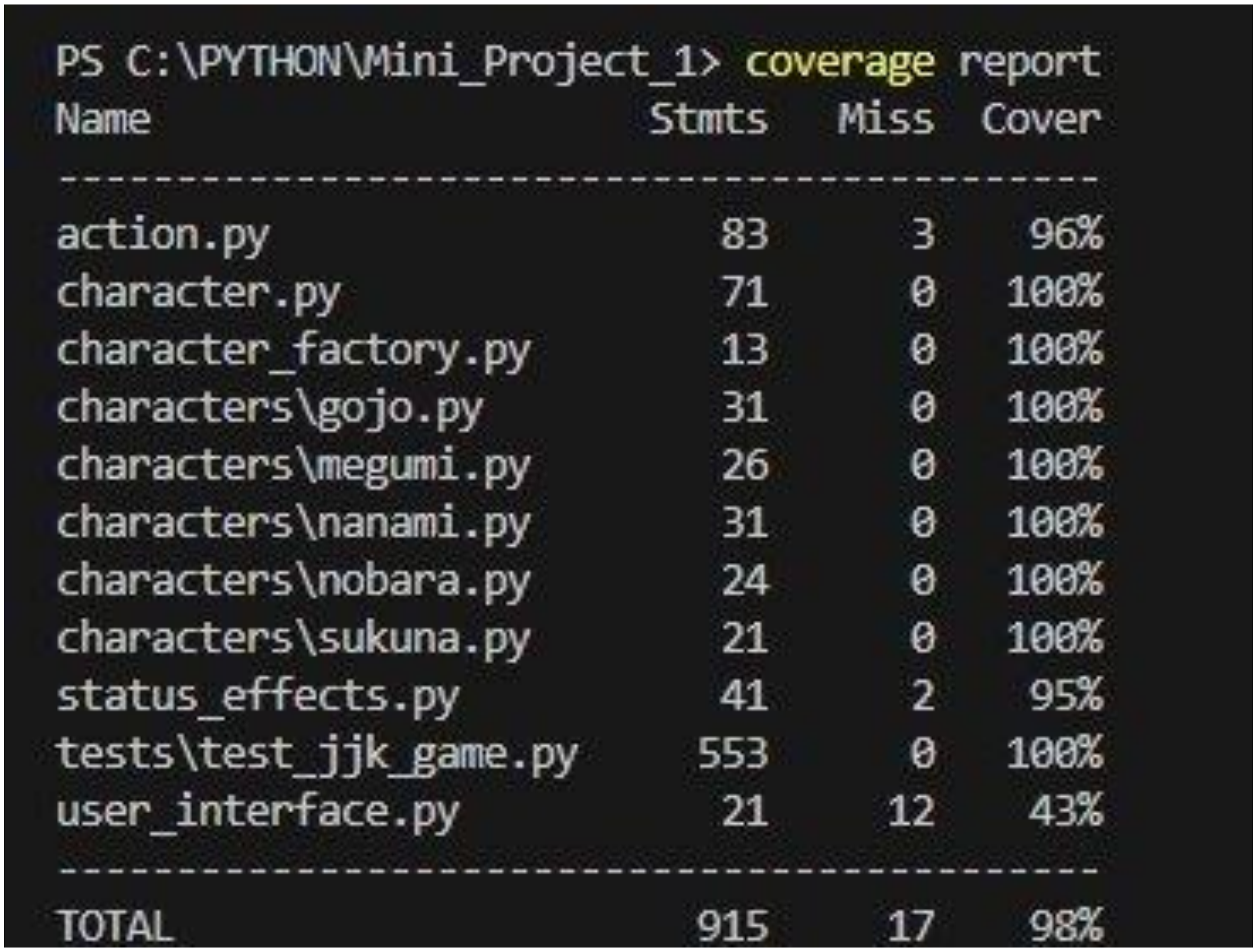

GUI
UnifiedClient
-root: tk.Tk
-left_panel: ttk.Frame
-right_panel: ttk.Frame
-game_display: scrolledtext.ScrolledText
-chat_display: scrolledtext.ScrolledText
-chatbot_display: scrolledtext.ScrolledText
-game_manager: BattleManager
-player_name: str
-connected_players: Set[str]
-current_turn: int
-combat_action: str
-selected_characters: Dict[str, Character]
-client_socket: socket
-receiver_thread: Thread
+setup_layout()
+setup_game_panel()
+setup_chatbot_panel()
+setup_multiplayer_chat()
+connect_to_server()
+send_message(data: dict)
+receive_messages()
+handle_game_input()
+handle_combat_action()
+process_turn()
+update_game_display()
+handle_chat_input()
+handle_chatbot_input()
+display_message()
contains
processes queries
returns responses
broadcasts updates
uses
sends messages
updates state
contains
controls game
ChatbotSystem
-responses: Dict[str, str]
-context: Dict[str, Any]
+process_query(query: str): str
+handle_help(): str
+handle_order_status(order_id: str): str
+handle_refund(order_id: str): str
-parse_intent(query: str): str
Network
NetworkManager
-server_socket: socket
-clients: Dict[socket, str]
-client_count: int
-message_handler: Callable
+__init__(host: str, port: int)
+set_message_handler(handler: Callable)
+broadcast(message: str)
+handle_client(socket, id: int)
+run()
manages
ChatServer
-server_socket: socket
-clients: Dict[socket, str]
-lock: threading.Lock
-root: tk.Tk
-log_display: scrolledtext.ScrolledText
+__init__(host: str, port: int)
+setup_gui()
+log_message(message: str)
+broadcast(message: str, sender: socket)
+handle_client(socket, addr)
+start()
+accept_clients()
Game
CharacterFactory
-character_templates: Dict[str, Dict]
+create_character(name: str): Character
-load_templates()
-create_special_ability(type: str): SpecialAbility
-create_stats(template: Dict): CharacterStats
BattleManager
-players: List[Character]
-current_turn: int
-game_state: str
+add_player(player: Character)
+start_battle()
+handle_combat(attacker: str, target: str, action: str)
+next_turn()
+is_game_over(): bool
+get_winner(): Character
creates
manages
Character
-name: str
-hp: int
-max_hp: int
-attack_power: int
-defense: int
-special_ability: SpecialAbility
-status_effects: List[StatusEffect]
+attack(target: Character)
+defend()
+special(targets: List[Character], turn: int)
+take_damage(amount: int)
+heal(amount: int)
+is_alive(): bool
+handle_status_effects()
implements
affects
SpecialAbility
-name: str
-description: str
-cooldown: int
-last_used: int
+use(user: Character, targets: List[Character])
+is_available(current_turn: int): bool
StatusEffect
-type: StatusType
-duration: int
-strength: int
+apply(target: Character)
+tick()
+is_active(): bool

Figure 1: UML Class Diagram

11 Test Coverage

The unit tests that we created were to ensure correct behavior was happening throughout our software system. Most of the methods were covered in the majority of the classes, although some were showing promising behavior, it was just difficult to write out the unittest in code, we ran multiple sub tests to ensure predictable outcomes with our methods built. We ensure that our code would behave 98% accurately and as intended with its functionality.



```
PS C:\PYTHON\Mini_Project_1> coverage report
Name                       Stmts   Miss  Cover
-----------------------------------------------
action.py                     83      3    96%
character.py                  71      0   100%
character_factory.py          13      0   100%
characters\gojo.py            31      0   100%
characters\megumi.py          26      0   100%
characters\nanami.py          31      0   100%
characters\nobara.py          24      0   100%
characters\sukuna.py          21      0   100%
status_effects.py             41      2    95%
tests\test_jjk_game.py       553      0   100%
user_interface.py             21     12    43%
-----------------------------------------------
TOTAL                        915     17    98%
```

Figure 2: Test Coverage Report

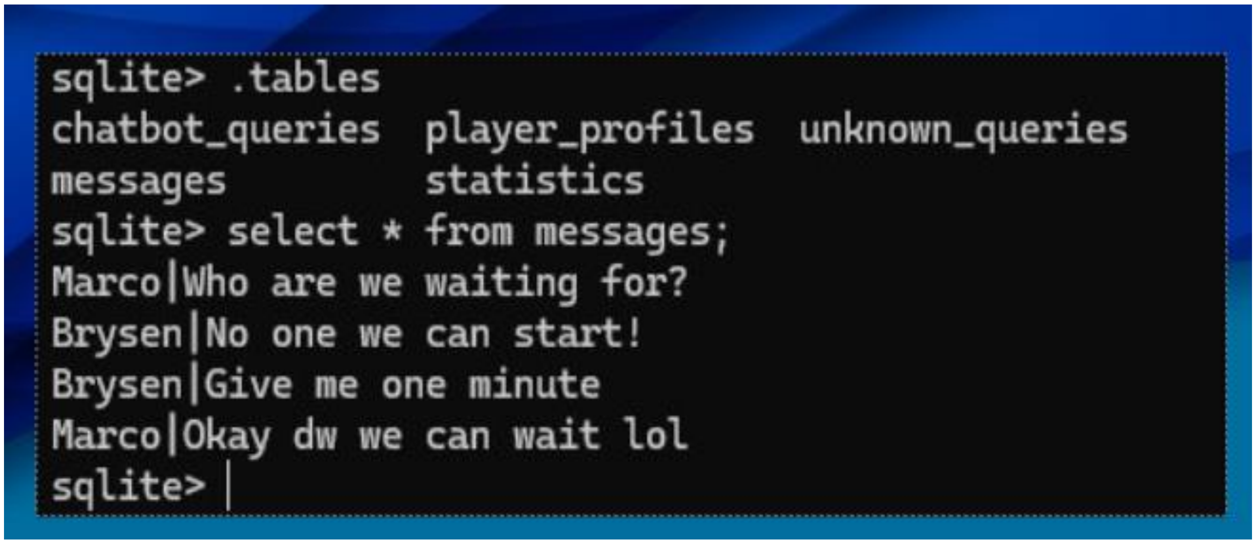
12 Team Contribution

Marco: Chat Bot GUI, JJK_Game GUI, Player Chat GUI, UnifiedGUI, Client, Server, Network Manager, Game Frame Design, UML Diagram, Presentation, Report

Brysen: JJK_Game functionality, Tests, Threading and Socket functionality, Chat Bot functionality

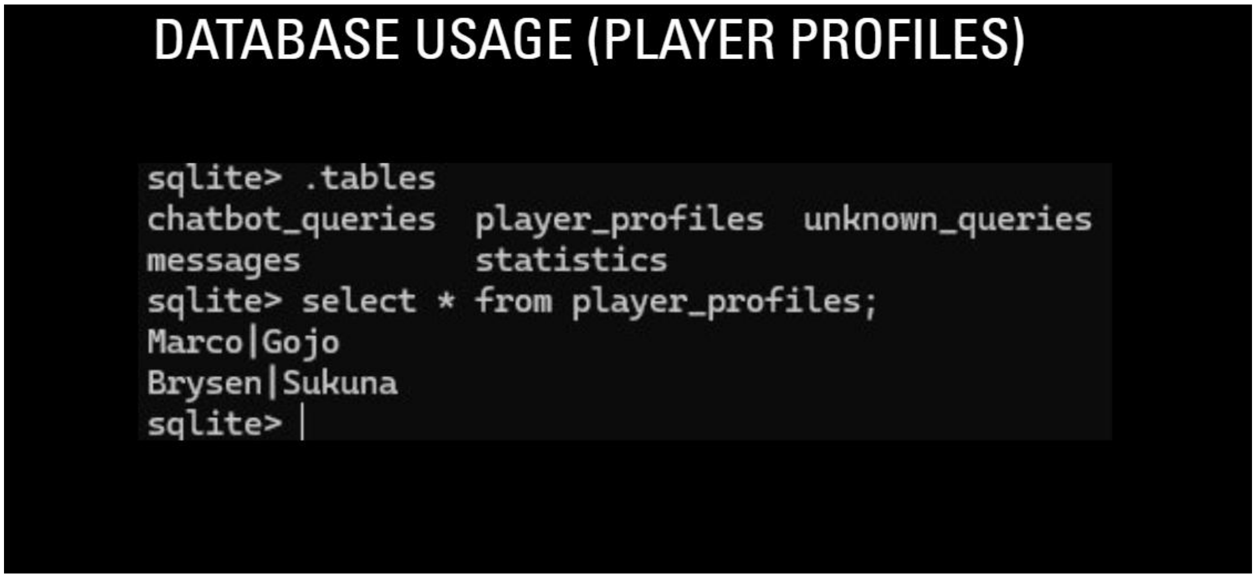
13 Screenshots

The way we used sqlite3 to create a database that incorporates tables was by breaking down the tables we needed to create to represent all the necessary information. We have one main database that is for player interactions, that includes tables like (messages), (player_profiles), (chatbot_queries), (unknown_queries), (statistics). Each table has its own properties and columns that will return information appropriate for each table and have important information stored within the database.



```
sqlite> .tables
chatbot_queries  player_profiles  unknown_queries
messages         statistics
sqlite> select * from messages;
Marco|Who are we waiting for?
Brysen|No one we can start!
Brysen|Give me one minute
Marco|Okay dw we can wait lol
sqlite> |
```

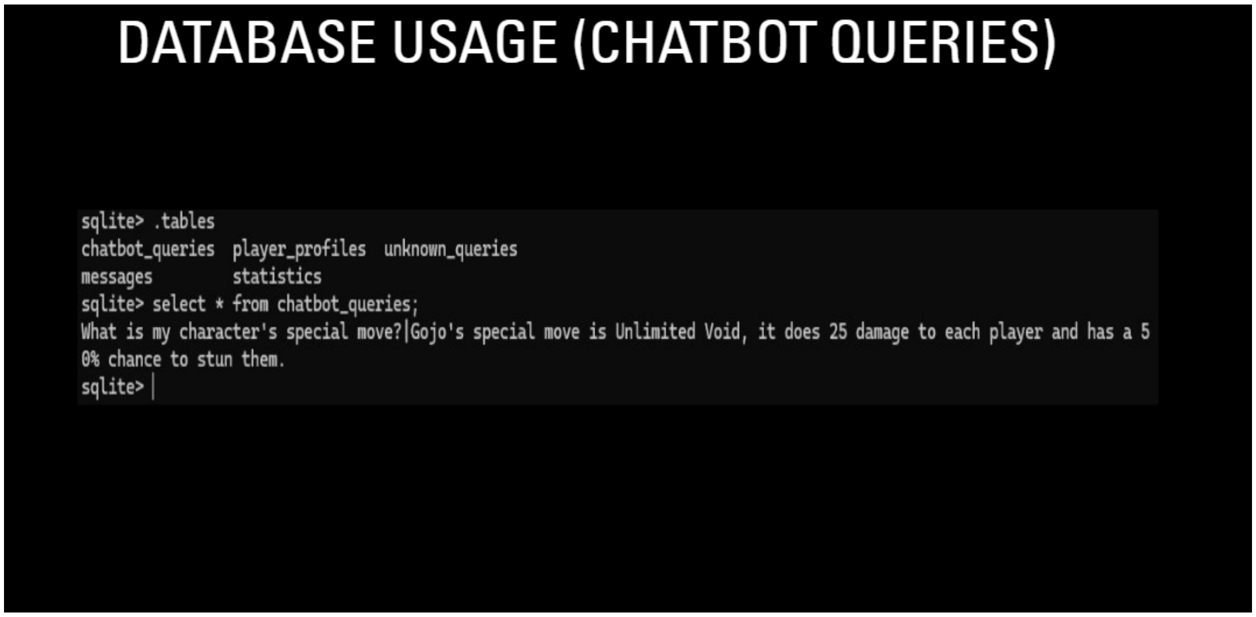
Figure 3: Messages Table



DATABASE USAGE (PLAYER PROFILES)

```
sqlite> .tables
chatbot_queries  player_profiles  unknown_queries
messages         statistics
sqlite> select * from player_profiles;
Marco|Gojo
Brysen|Sukuna
sqlite> |
```

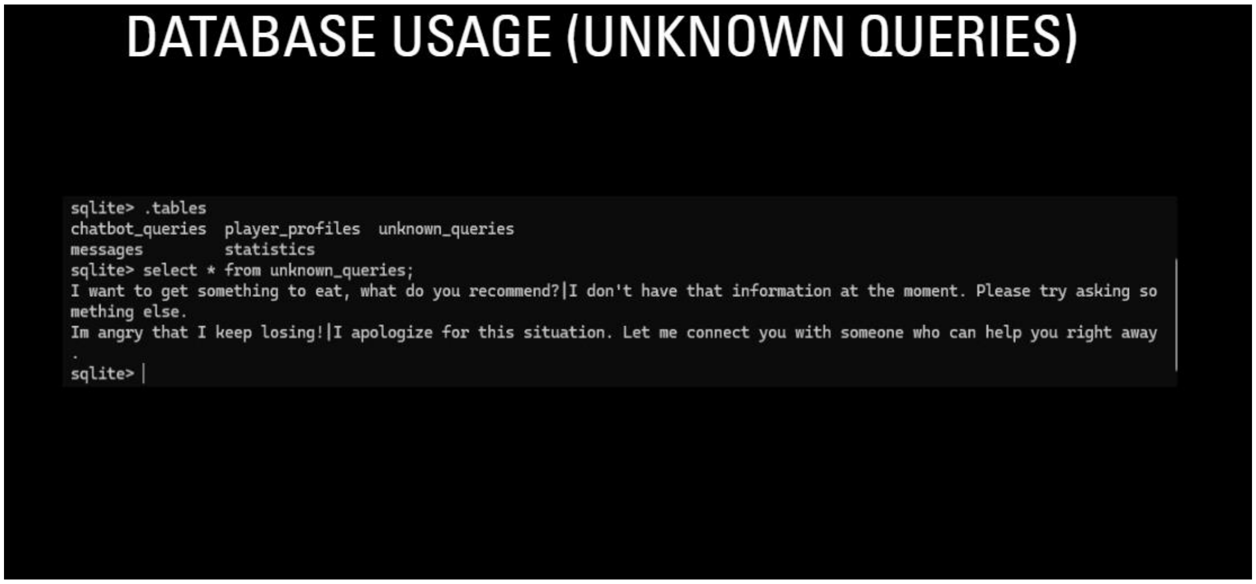
Figure 4: Player Profiles Table



DATABASE USAGE (CHATBOT QUERIES)

```
sqlite> .tables
chatbot_queries  player_profiles  unknown_queries
messages         statistics
sqlite> select * from chatbot_queries;
What is my character's special move?|Gojo's special move is Unlimited Void, it does 25 damage to each player and has a 5
0% chance to stun them.
sqlite> |
```

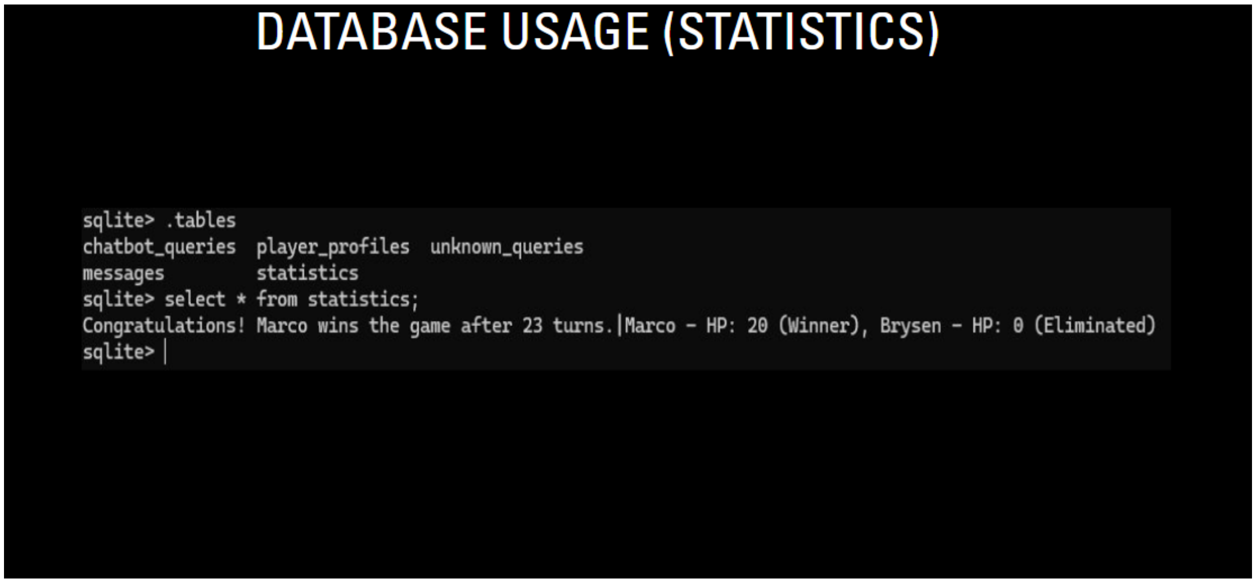
Figure 5: Chatbot Queries Table



DATABASE USAGE (UNKNOWN QUERIES)

```
sqlite> .tables
chatbot_queries  player_profiles  unknown_queries
messages         statistics
sqlite> select * from unknown_queries;
I want to get something to eat, what do you recommend?|I don't have that information at the moment. Please try asking so
mething else.
Im angry that I keep losing!|I apologize for this situation. Let me connect you with someone who can help you right away
.
sqlite> |
```

Figure 6: Unknown Queries Table



DATABASE USAGE (STATISTICS)

```
sqlite> .tables
chatbot_queries  player_profiles  unknown_queries
messages         statistics
sqlite> select * from statistics;
Congratulations! Marco wins the game after 23 turns.|Marco - HP: 20 (Winner), Brysen - HP: 0 (Eliminated)
sqlite> |
```

Figure 7: Statistics Table