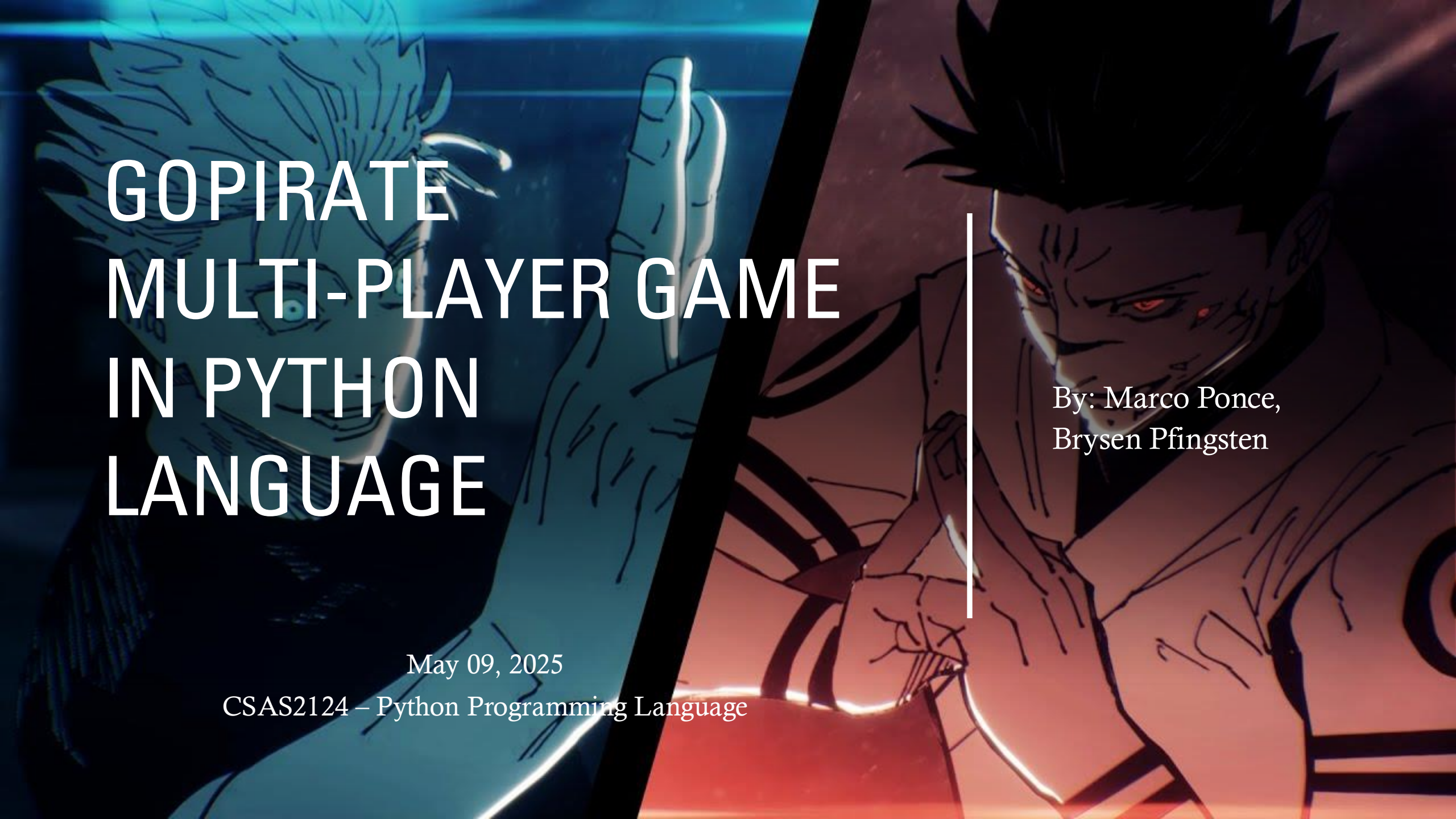


GOPIRATE MULTI-PLAYER GAME IN PYTHON LANGUAGE

By: Marco Ponce,
Brysen Pfingsten

May 09, 2025

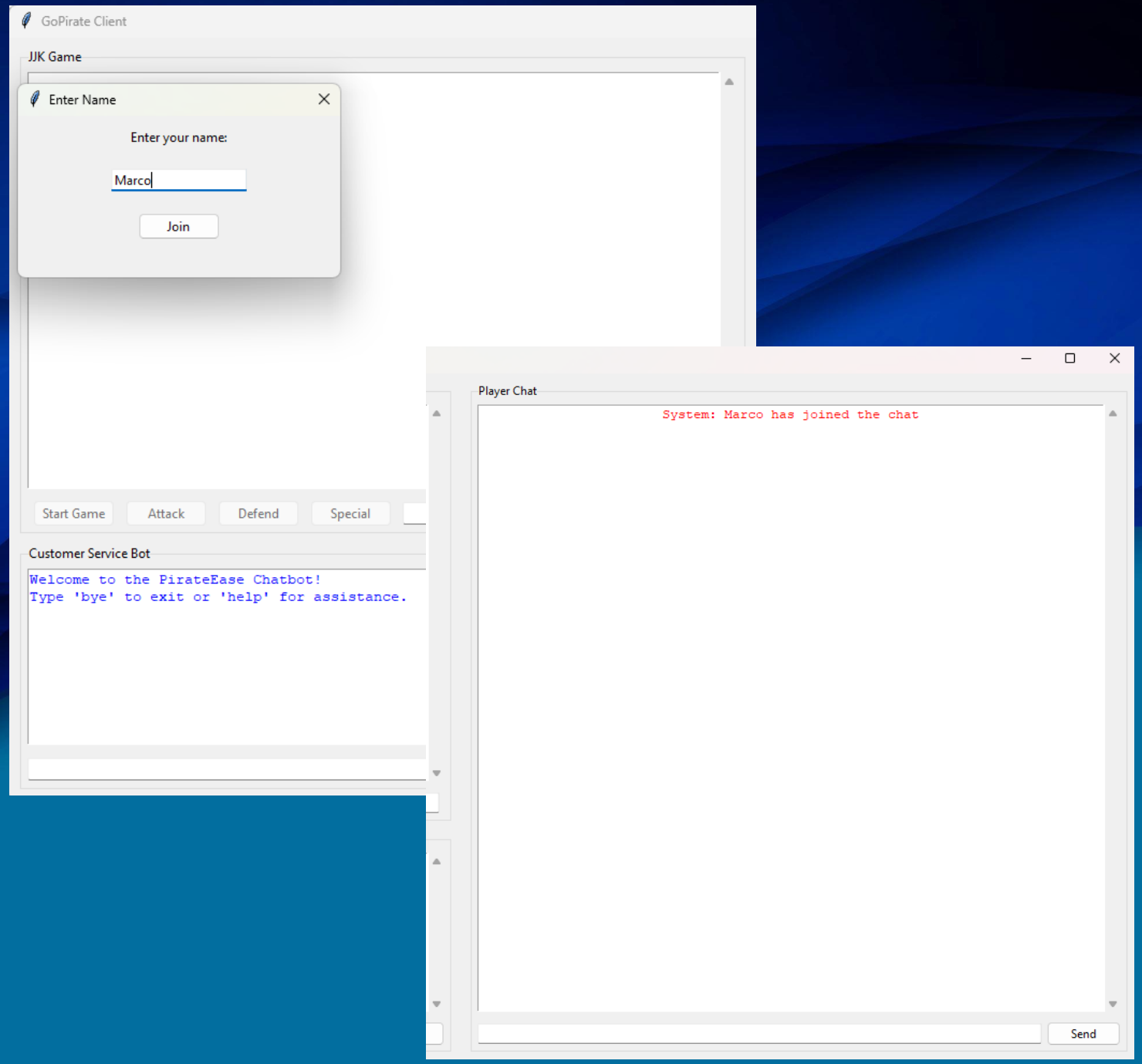
CSAS2124 – Python Programming Language

PROJECT OVERVIEW:

The GoPirate project consists of designing and developing a complete software system that combines two different applications into a single integrated program.

This includes are previously built JJK turn-based battle game, Chatbot support system within the same environment, and client-text communication across devices.

Key components include understand distributive programming, design patterns, OOP principles, and GUI modules.



INITIAL IDEAS:

Battle Engine:

- Game logic and interaction amongst clients connected to the server

Chat System:

- Player to Player text communication, as well see ones' own text as "You:"

Smart ChatBot:

- AI assistant able to store which character each player chose to get more insight

Database Integration:

- Store message conversations, player names, chosen characters, chatbot queries

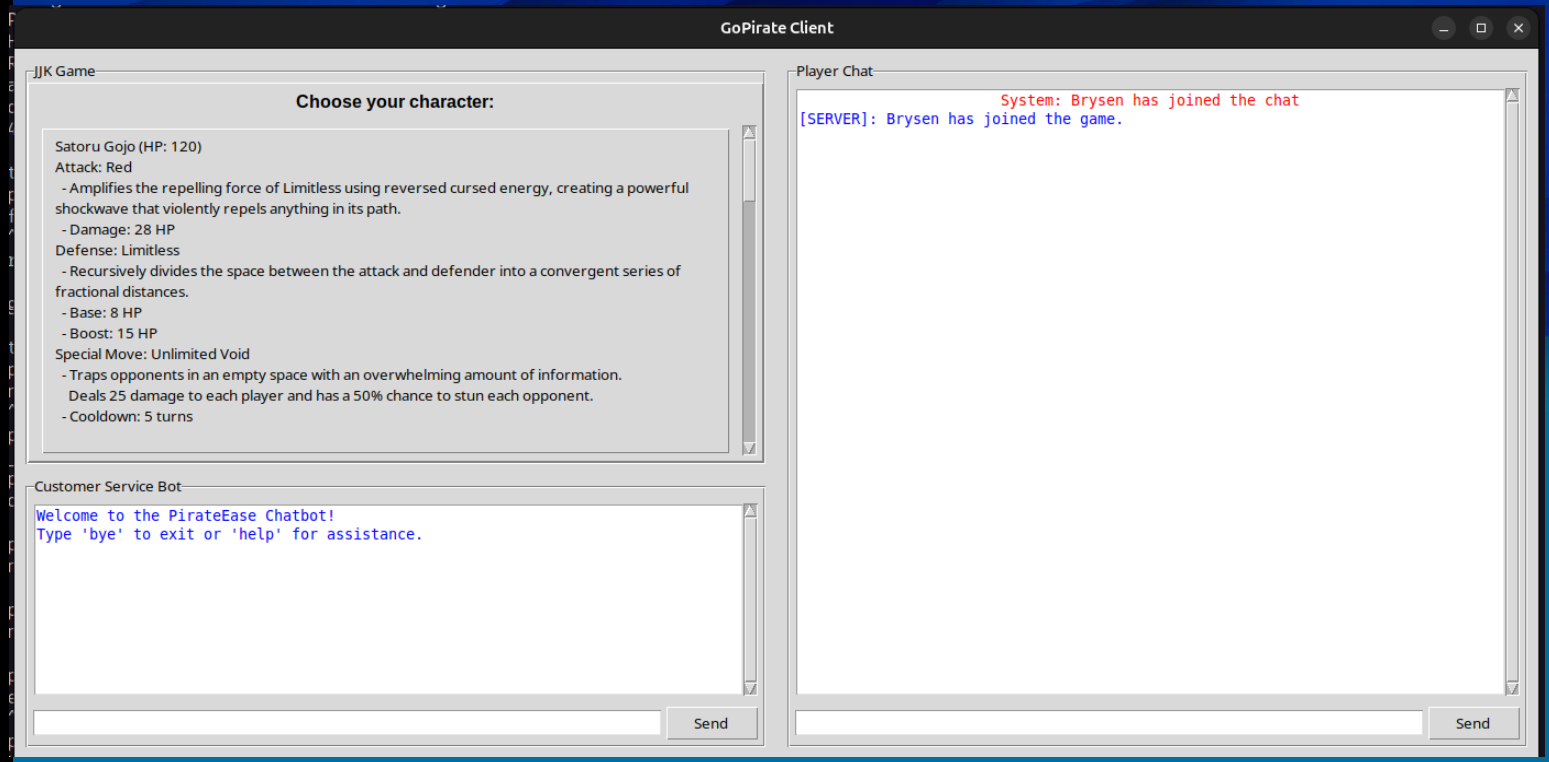
GUI Separation:

- How to handle each GUI separately but combine them together to one interface

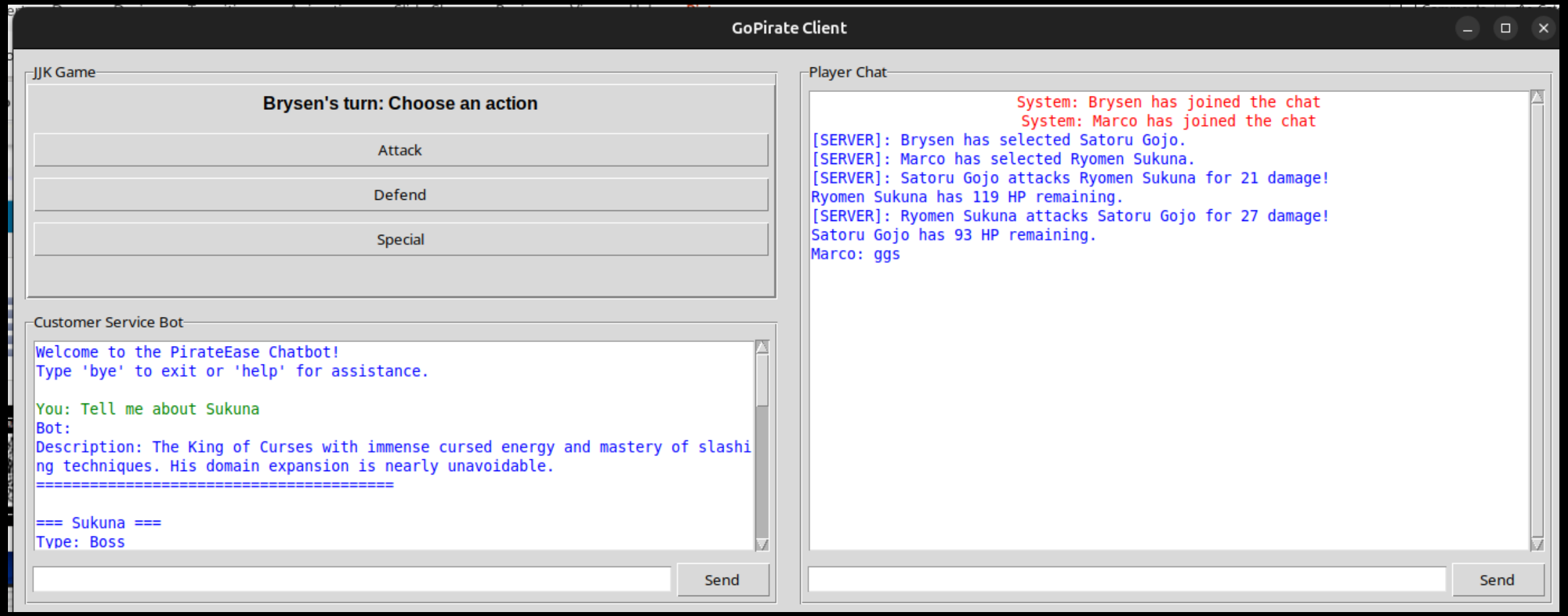
UML DIAGRAM

CHARACTER SELECTION AND BATTLE SYSTEM:

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 as and battle it out between each other until one remaining player is left standing, declared the winner.



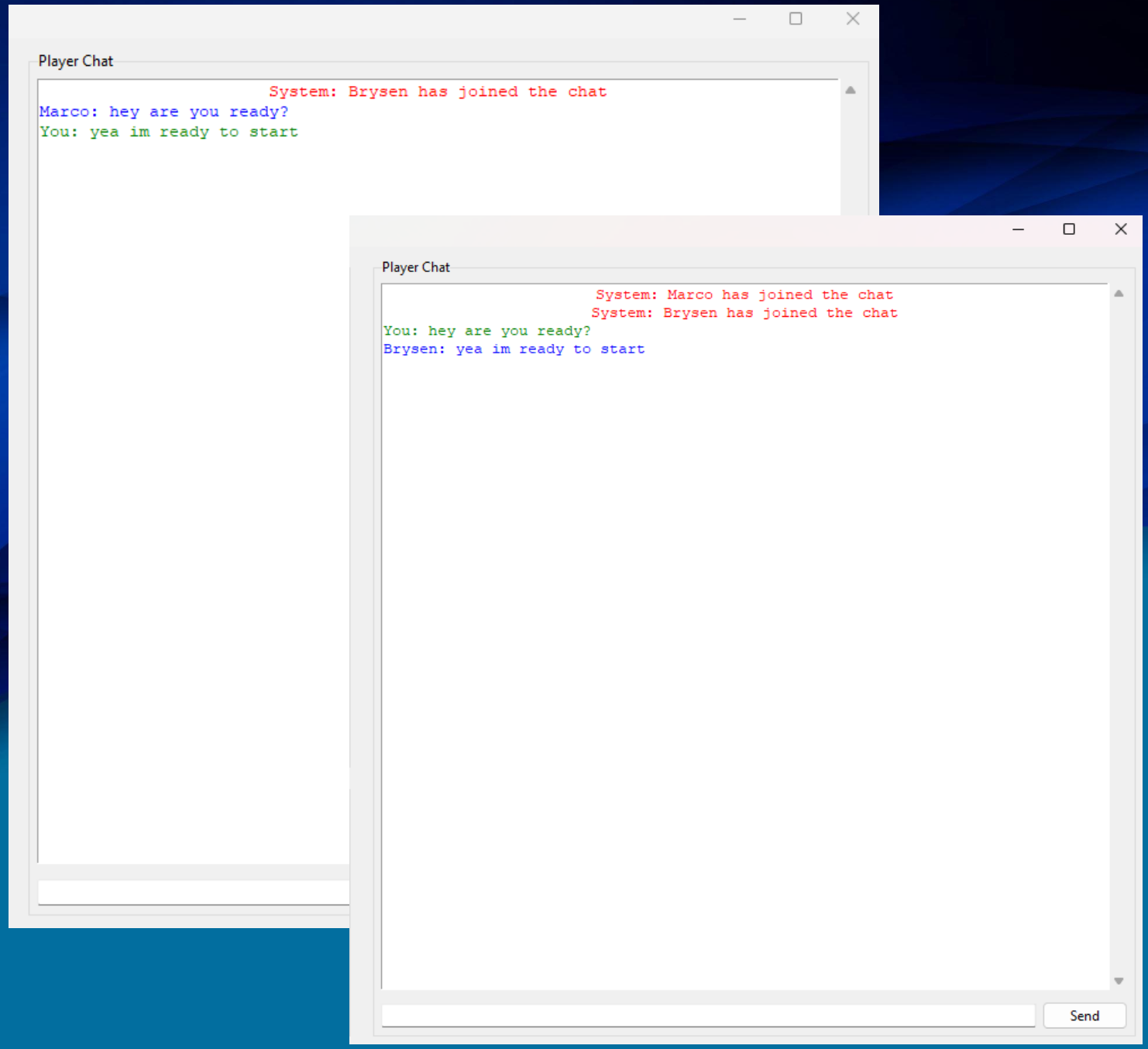
CHARACTER SELECTION AND BATTLE SYSTEM



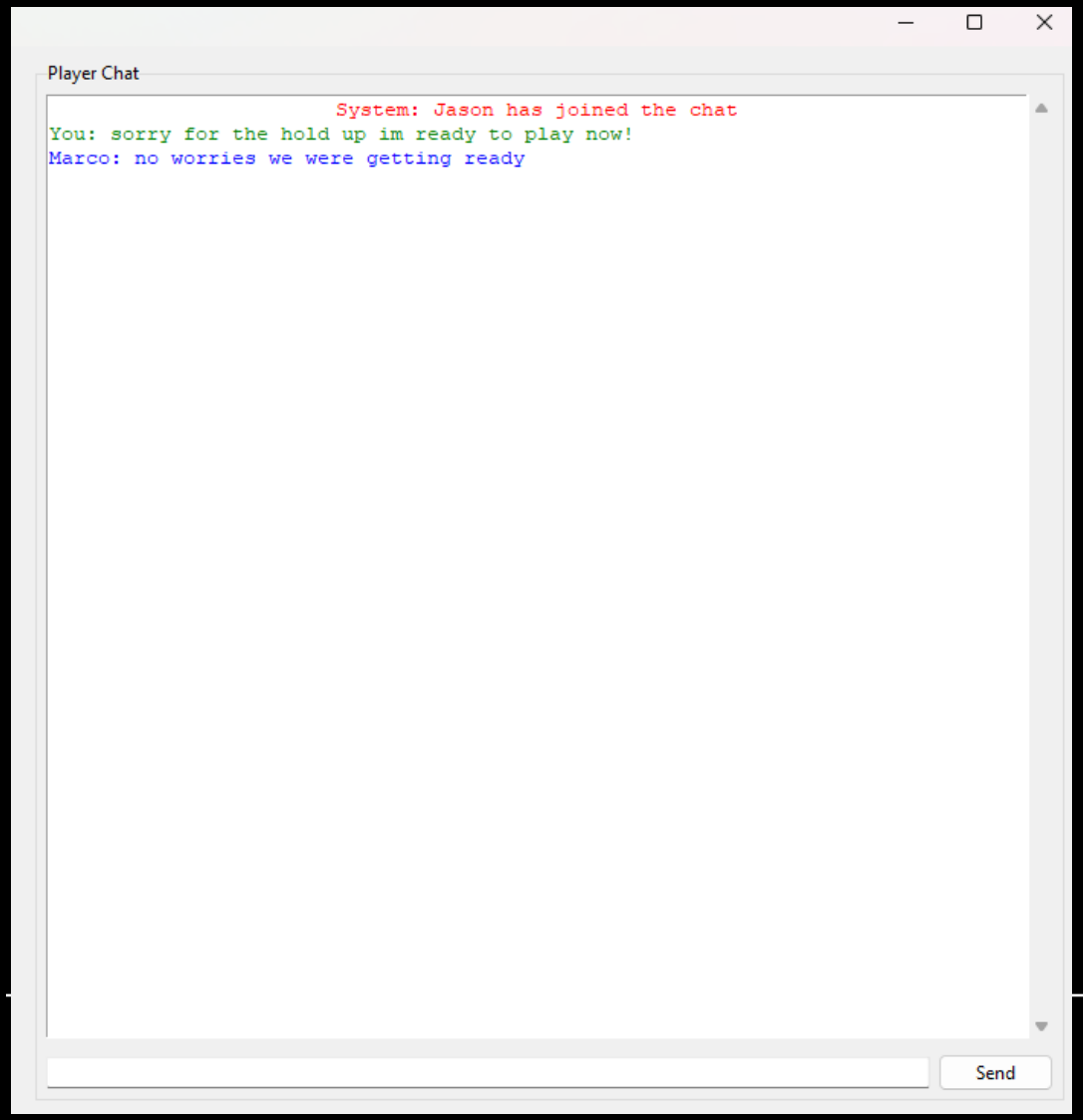
PLAYER-TO-PLAYER CHAT:

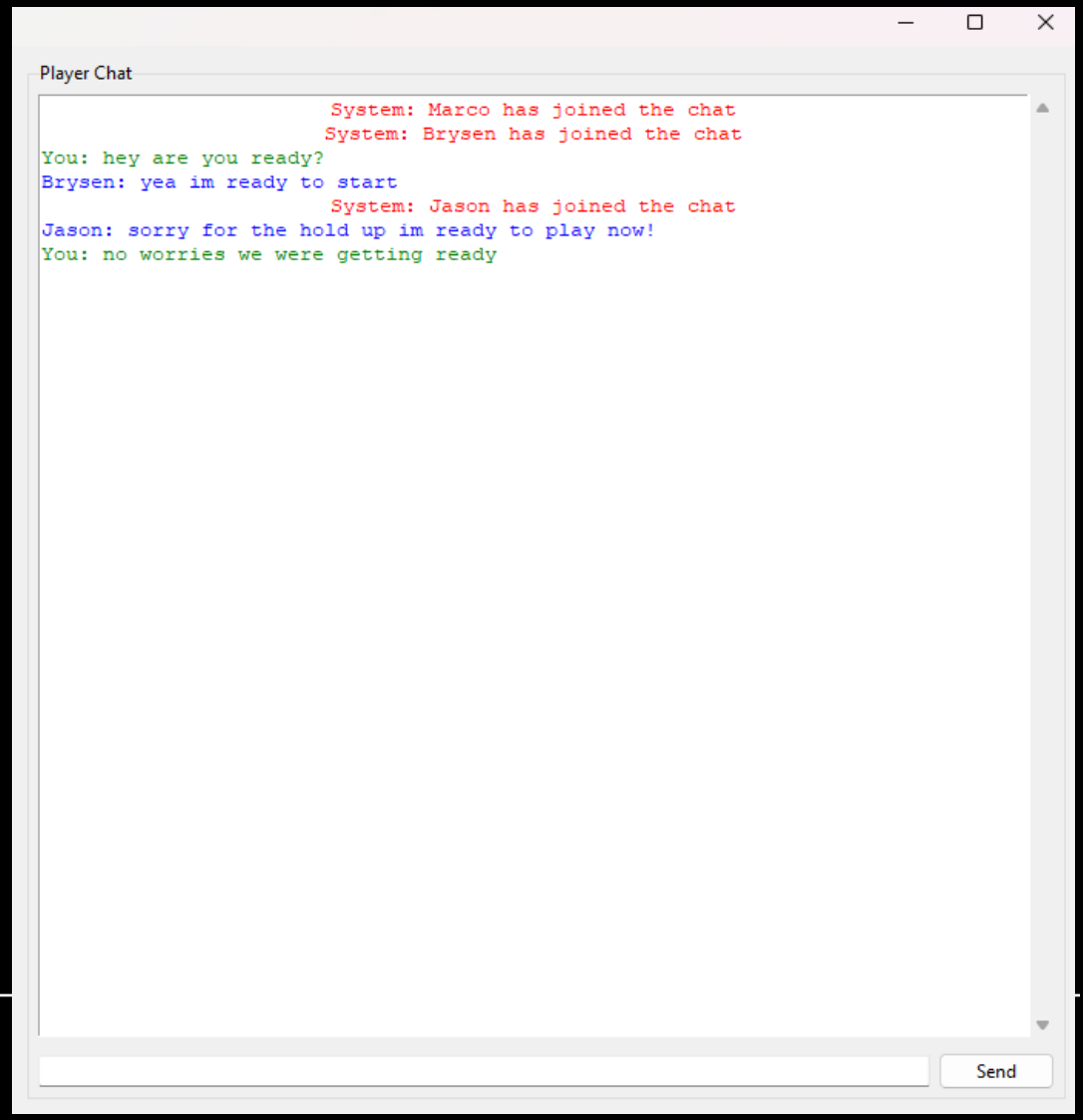
Once a player joins, they will either their name and a message will be broadcasted from the system letting everyone know someone joined.

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.

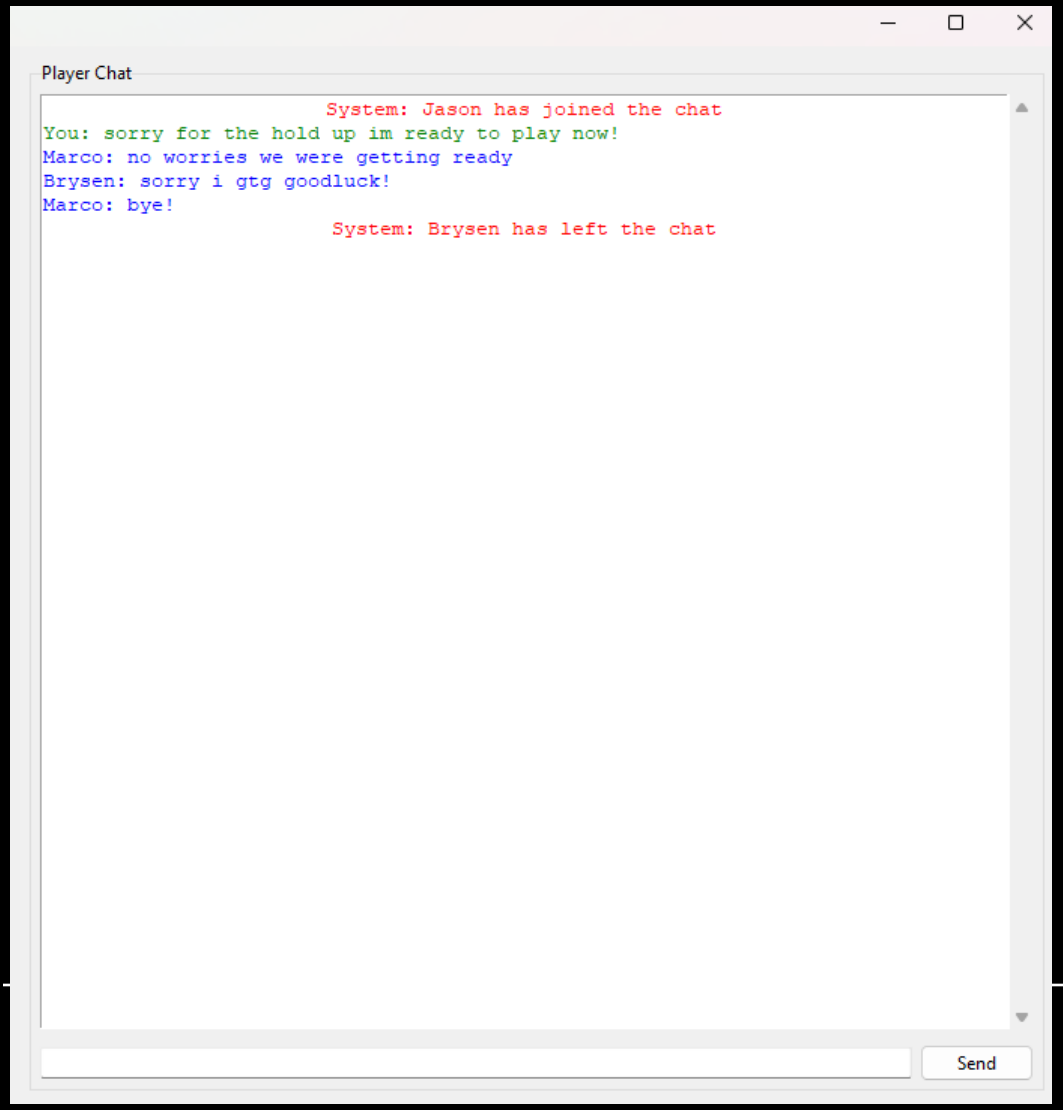


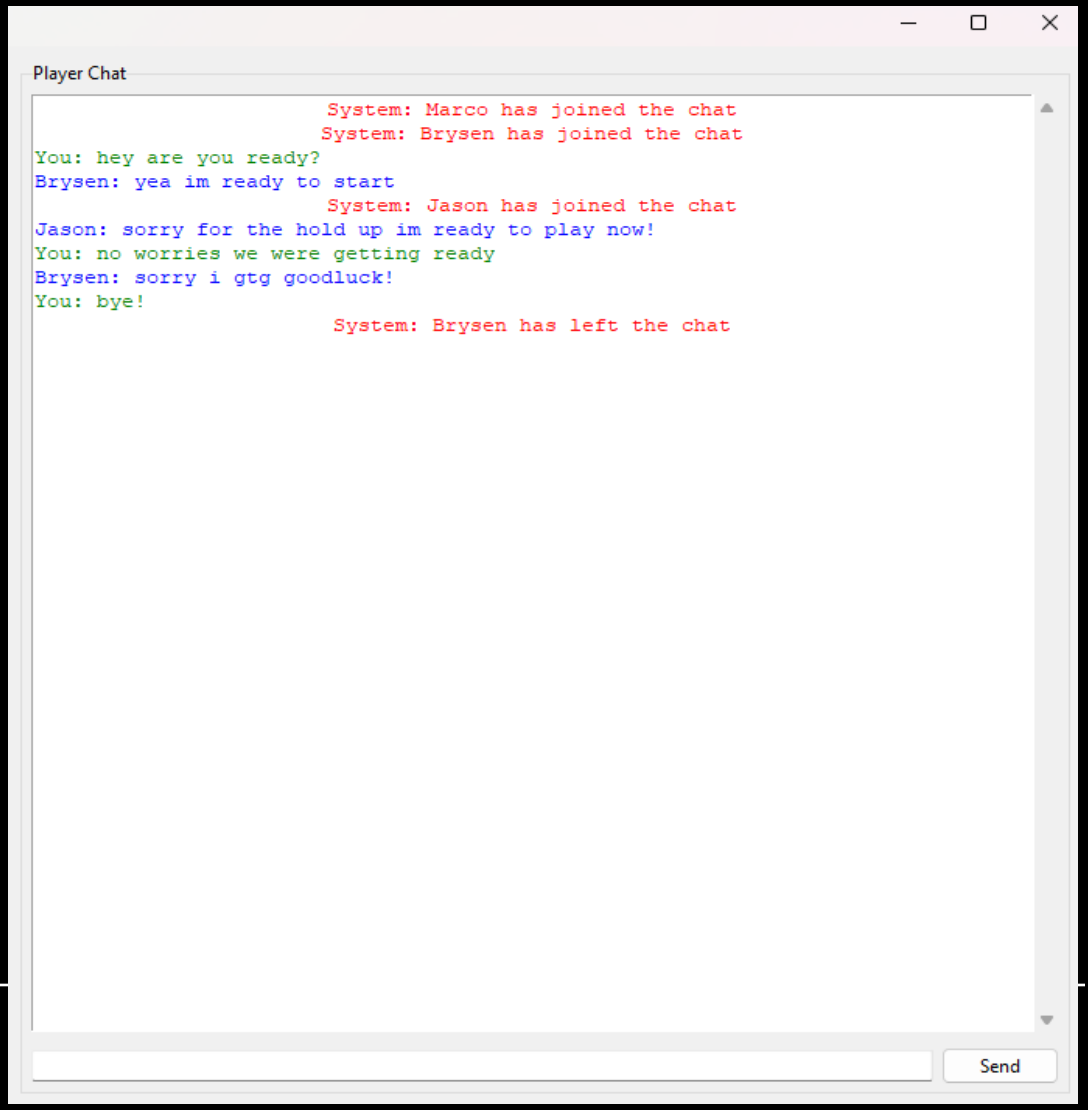
PLAYER-TO-PLAYER CHAT (LATE JOIN)





PLAYER-TO-PLAYER CHAT (PLAYER LEFT)

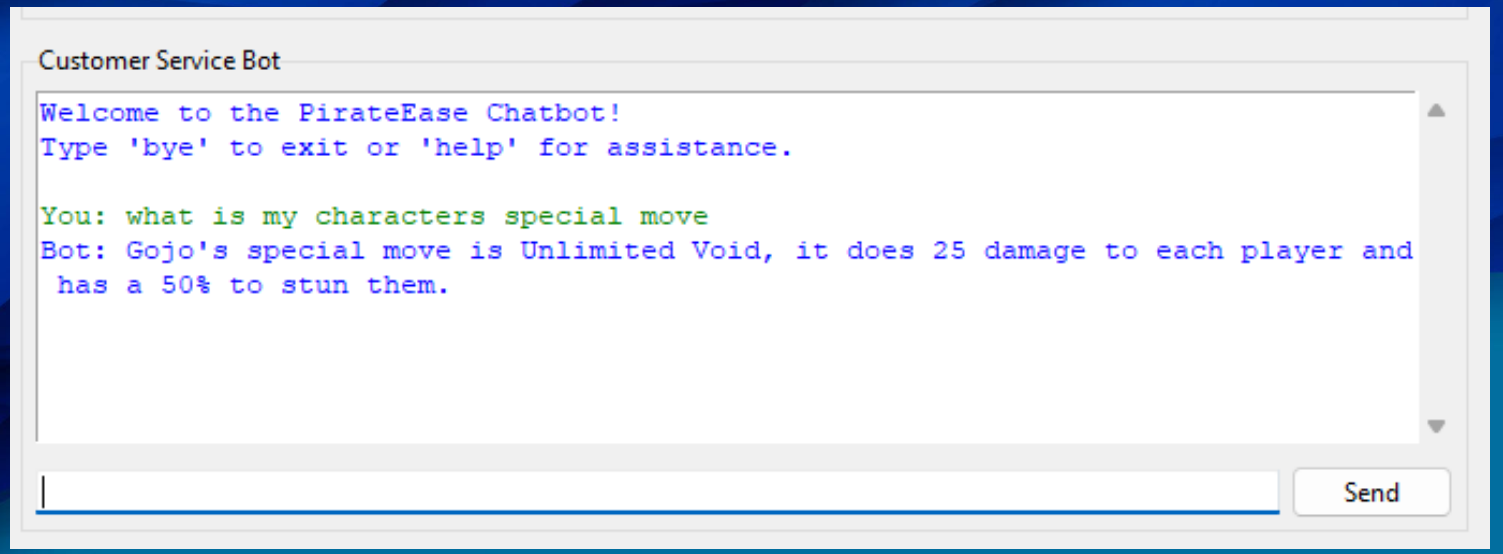




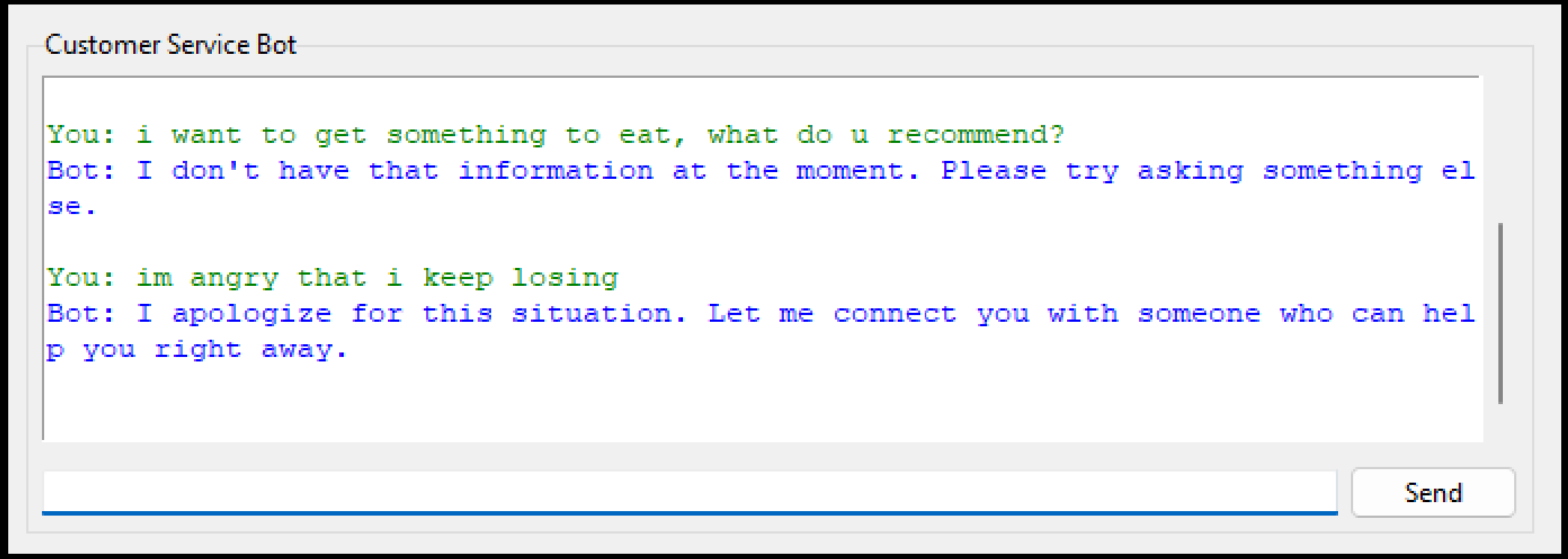
SMART CHATBOT INTEGRATION:

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.



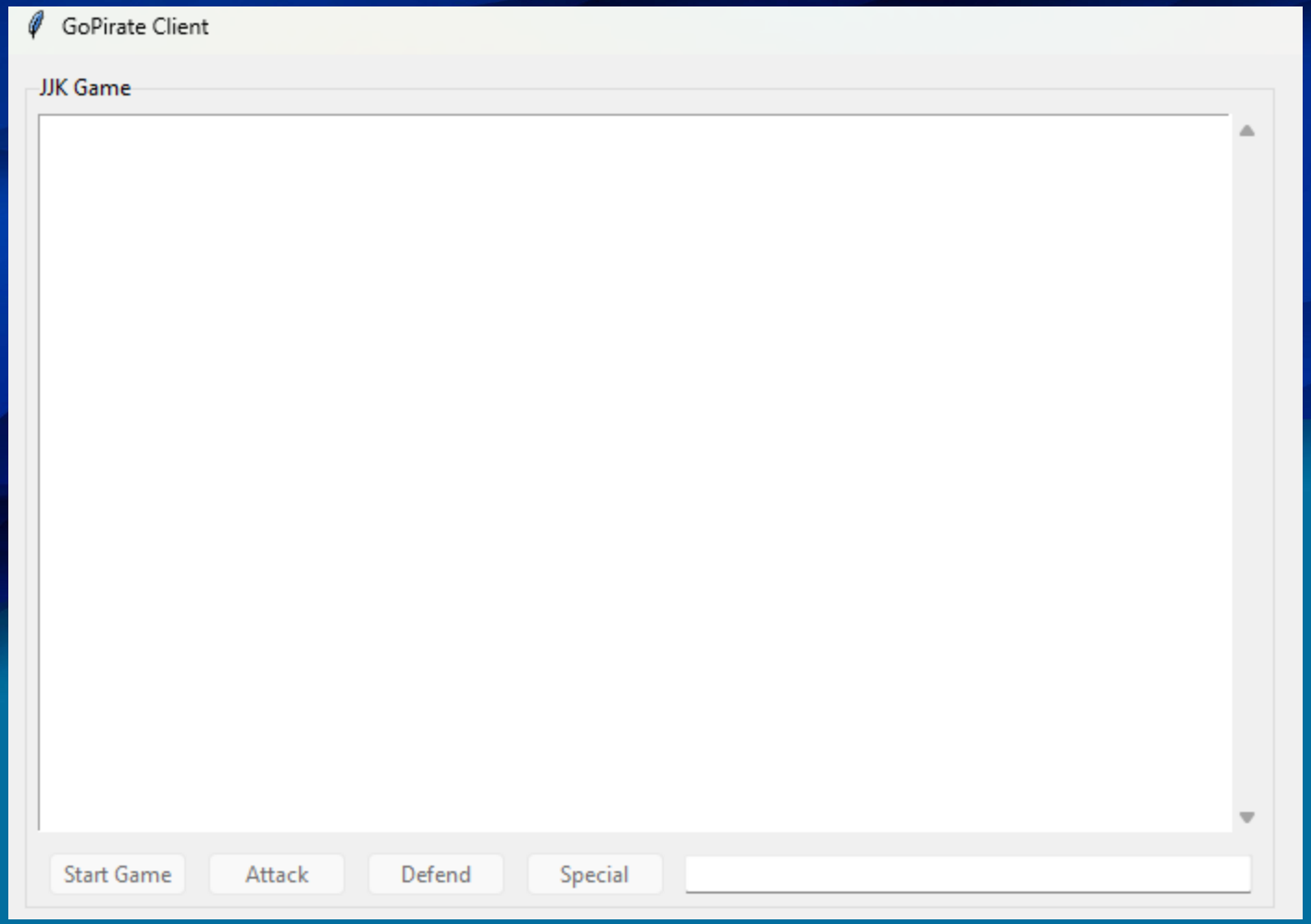
SMART CHATBOT INTEGRATION (UNKNOWN QUERY)



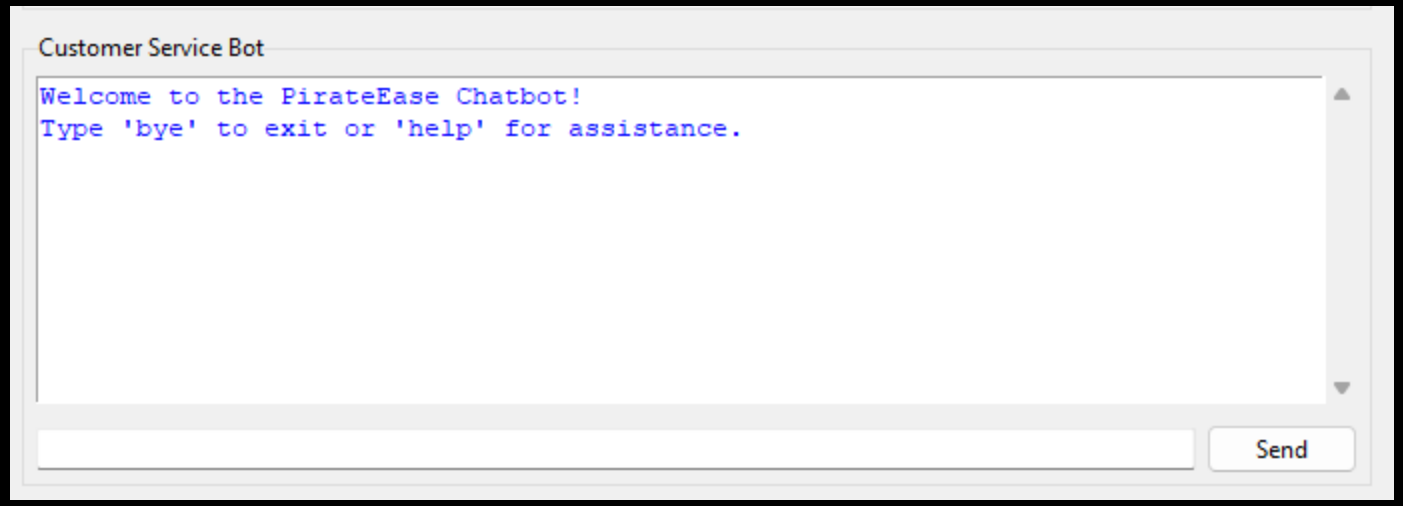
GUI DESIGN:

Each GUI design has its own separate interface; this allows for each to hold their functionality even if the client uses another interface.

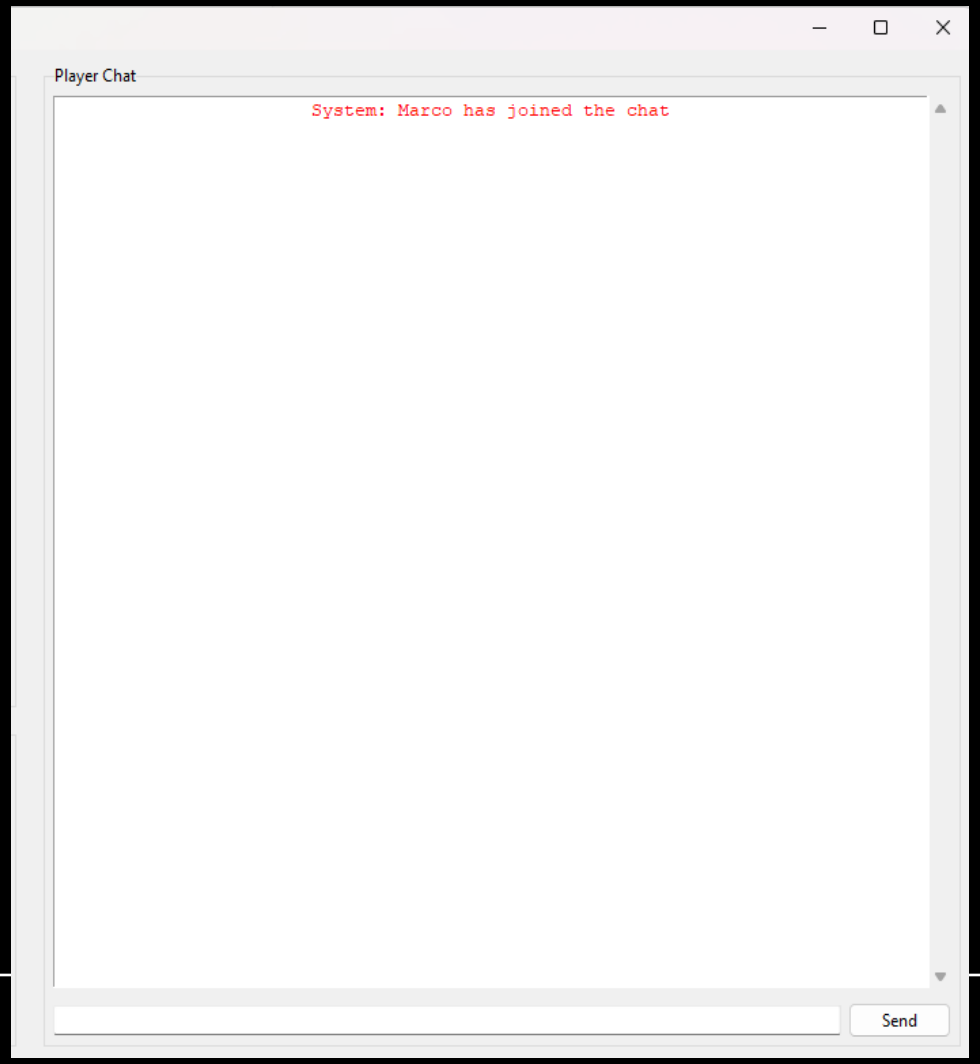
The JJK Game was designated to be in the top left corner of the main interface, the chatbot is right beneath it, and since most of the interacting is in the player chat, we made it the biggest on the right side of the interface, each interface having the ability to scroll and is protected from having players interfering with the GUI and typing over it.



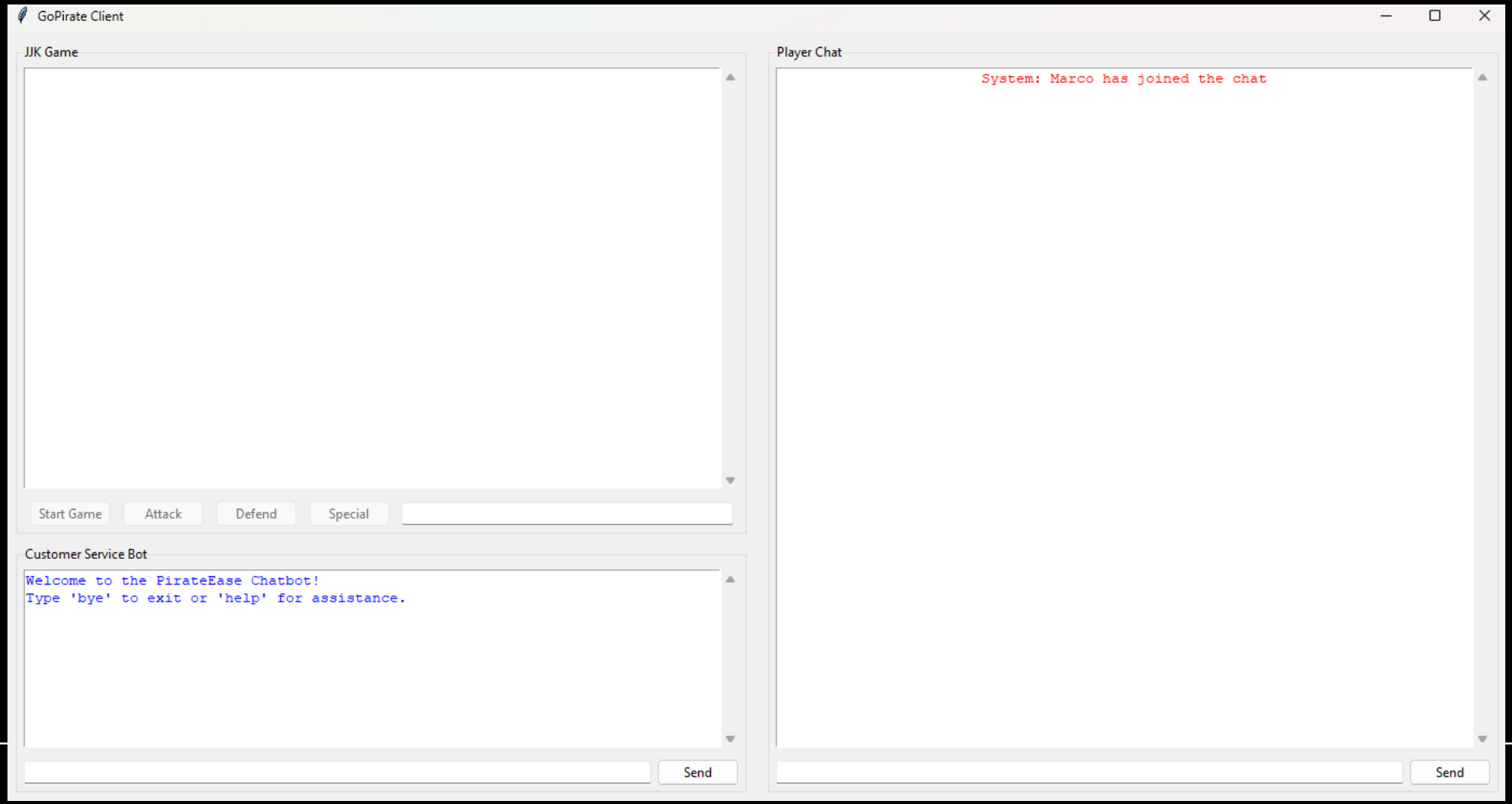
GUI DESIGN (CHATBOT)



GUI DESIGN (PLAYER CHAT)



GUI DESIGN (ALTOGETHER)



DATABASE USAGE:

The way we used sqlite3 to create a database that incorporate 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 their 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>
```

DATABASE USAGE (PLAYER PROFILES)

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

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>
```

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>
```

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>
```

APPLICATION OF OOP PRINCIPLES:

Encapsulation: We used encapsulation to bundle data and methods that operate on that data into a class, restricting direct access to some of its components (getter/setter methods)

- Ex: input name, player hp, player character, turns

Inheritance: Allows a class to inherit properties from another class and avoiding duplicate code

- Ex: StatusEffect, Attack, Defense, SpecialMove, Characters, UnifiedGUI

Polymorphism: Enables objects of classes to be treated as an object of a superclass, able to use method overriding

- Ex: apply for SpecialMove based on character, handle for StatusEffect based on poison/stun object

Abstraction: Hides the complexity of the code by only showing the essential features and behavior while hiding the internal implementation

- Ex: QueryHandler, Action, StatusEffect, GameGUI, ChatGUI, ChatBotGUI, UnifiedGUI

DESIGN PATTERNS USED:

Observer Pattern: Used in the event handler system for GUI components
Command Pattern: Used for handling game actions (Attack, Defend, SpecialMove)
Factory Pattern: Used for creating different character types (Gojo, Sukuna, Nanami, Megumi, Nobara)
Strategy Pattern: Used for different query handlers in the chat system
Singleton Pattern: Used for managing shared resources like the chat server and only one exists
State Pattern: Used to manage game states (character selection, battle, game over)
Template Method Pattern: Used for the character base class to define the blueprint for specific character behaviors
Chain of Responsibility Pattern: Used in the chat system's query handling (handler1/handler2...etc)
Flyweight Pattern: Used in the common responses to save memory and similar response pools
Adapter Pattern: Used to provide a unified interface for different services
Proxy Pattern: Used for the lazy loading of game resources and creating characters when loaded
Memento Pattern: Used to maintain game state history and player chat history

UNIT TESTING:

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, 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%
```

BUGS FACED:

- Maintaining functionality between unified GUI
- Players could choose a character that has already been chosen
- Players could sometimes take turns out of order
- Status effects would continue to happen even when the player is dead

SKILLS LEARNED:

OOP principles and design patterns (singleton/factory)

Polymorphism

Inheritance

Turn-Based System

Abstraction

Getter/Setters -> @property/@attribute_name.setter

Tkinter (GUI), Socket/Threading

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

POTENTIAL ADD-INS/IMPROVEMENTS:

- Enhance the player chat to also have a voice chat option for those who prefer that option
- Counter-attack mechanics having a change to dodge an incoming attack
- Experience points and leveling up system
- Unlockable abilities
- Character customization option

- Visual enchantments (pictures for each character)
- Animations and movements of characters
- Tournament mode
- Story mode
- Friending system
- Spectating mode for on-going battles
- Sound effects when an attack is done, or when the game starts/ends

THANK YOU ☺