CSAS 2124 – Final project

"GoPirate" Multi-player game - can you build a smart game?

Due Date: 05/9/2025

Points: 100

Project description

In this extended project, your task is to design and develop a complete software system that combines two different applications into a single integrated program. You will build an extended version of Mini-Project 1's multiplayer turn-based battle game, while also integrating Mini-Project 2's smart chatbot support system within the same environment. Both parts of the system must work together smoothly, following good software engineering practices.

The multiplayer battle game will involve three players who select their own unique characters at the beginning of the game. Each character will have different strengths and abilities, such as health points, attack power, defense skills, and special moves. Players will take turns one after another, choosing actions like attacking an opponent, defending to reduce damage, or using a special move. The battle will continue until only one player remains standing, and that player will be declared the winner.

Along with battling, the players must also be able to chat in a distributed environment with each other during the game. A chat window will be available in the interface where players can type and send real-time messages to each other. This will make the game feel more interactive and livelier.

In addition to chatting with each other, players should also can interact with a smart chatbot that is available throughout the game. This chatbot is like a built-in game assistant. Players can ask the chatbot for help at any time if they want to understand a game rule, learn more about their character's special move, or get general advice. The chatbot must be intelligent, meaning it should understand the meaning of the question even if players phrase it differently. The chatbot should not rely on simple if-else chains for its responses. Instead, it must use a structured query handling system and be designed in a scalable way so that new types of questions can easily be added later without rewriting the chatbot's logic.

If the chatbot cannot properly handle a player's query, it should automatically escalate the request to a live agent simulation inside the system. This shows that the chatbot knows its limits and can hand off the conversation when needed.

The entire program must be built following OOP principles. Such as encapsulation, inheritance, polymorphism, and abstraction. You must organize your code into clear classes and modules, each with a specific responsibility. You should apply multiple design patterns throughout the system. These design patterns will make your program easier to expand, easier to understand, and more professional. You must use creational patterns (like Factory and Singleton), structural patterns (like Adapter and Facade), and behavioral patterns (like Strategy, Observer, and Command).

Your program must have a Graphical User Interface (GUI) so that players can interact with the game easily. The GUI must be designed to allow players to select their characters, take their turns, chat with each other, and chat with the chatbot - all in a user-friendly windowed application.

You are also required to use an SQLite3 database in this project. The database must store important information such as:

- Player profiles and the characters they selected,
- The full chat history between players,
- Chatbot queries and chatbot answers,
- Battle session details including who won the game, how many turns were taken, and player statistics,
- Any unknown questions asked to the chatbot, so that they can be reviewed, and new responses can be added later.

The database will ensure that important data is saved permanently and can be reviewed even after the game ends.

This project is designed to challenge you to combine two complex systems - a multiplayer battle engine and a scalable chatbot - into a single, fully functional, structured, and maintainable program. It will help you practice how to manage multiple modules working together, how to apply OOP and design patterns correctly, and how to build software systems that are ready for real-world expansion and improvements.

By completing this project, you will gain practical experience in software architecture, modular development, GUI programming, database integration, smart response handling, and professional software design - all of which are critical skills for your future career as a computer scientist / software engineer.

Game Functionality

At the beginning of the game, each player will select a character from a list that you provide. Each character should have different abilities, such as different health points, attack strength, defense power, and a special move that makes them unique. Some characters might be stronger in attack, while others might be better at defending or using smart tricks. You should show the players some information about each character before they make their choice, so they know what they are picking.
You can try designing a simple character panel where the health, attack, defense, and special move details are displayed when players hover over or select a character. You may store character data using a dictionary, a JSON file, or create a Character class structure - or you can use your own idea if you find a better way.

It's important that no two players pick the same character. Once a character is selected by a player, it should no longer be available for the others. If someone tries to pick a character that has already been taken, the game should politely ask them to choose a different one. You must make sure that at least three players have chosen their characters before the battle starts. If fewer than three players are ready, the game should not move forward.
You can try maintaining a list or dictionary to track selected characters, or you can dynamically disable buttons in the GUI after a character is picked. Think about how you can design a clean selection system that prevents duplicates without confusing the players.

Once character selection is done, the game will enter the battle phase. This part of the game must be turn-based, meaning players take turns one after another - not all at the same time. On a player's turn, they should be able to choose between three options:

- Attack another player to reduce their health points,
- Defend to take less damage if someone attacks them,
- Use their character's special move, which could have a powerful or tricky effect.

During a player's turn, you can try providing a simple menu with buttons or a pop-up selection window where the player can pick their action. You may also create a "BattleManager" class that handles player turns cleanly - or you can design your own approach based on what you feel works best. You should also keep track of turn order, possibly using a list or a queue that cycles through the players after every move.

After each move, the game should immediately update everything - like adjusting health points, showing if someone is poisoned or stunned, and making sure cooldowns (for special moves) are handled properly. For example, if a player attacks someone, the game should calculate how much damage they do based on their attack power and the other player's defense.
You can try writing a basic calculation like “Damage = Attacker’s Attack - Defender’s Defense” (making sure that damage is not negative), but you can also design a more advanced formula if you like.
If you have special moves that poison or stun, you may create a StatusEffect system to track effects like “Poisoned (2 turns left)” or “Stunned (skip next turn).” You can manage these effects with a dictionary under each player’s data or a separate class - whichever design you prefer.

Special moves might also apply effects like poisoning an opponent (making them lose a little health over time), stunning them (making them lose a turn), or giving the player a shield to block damage for a while. Your system needs to keep track of all these effects carefully and update them at the right time.
You may design special move effects as separate classes, or use tags inside your player status to control ongoing effects. Think about cooldowns too - if a special move needs two or three turns before it can be used again, you can add a “cooldown counter” that reduces after each turn.

While the battle is happening, you should clearly show important information on the screen, such as:

- Each player’s current health points,
- Any active status effects (like poison or stun),
- Cooldowns for special moves (if a move can’t be used right away),
- Whose turn it is right now.

You can try designing a live “Stats Panel” showing all players' names, health bars, active effects, and turn indicator in a clean way. You may use a table, grid layout, or your own creative GUI design to make the battle easy for players to follow.

The battle should continue with players taking turns until only one player still has health left. When only one player is left standing, the game should end and announce the winner clearly.
You can try adding a simple winner announcement screen or pop-up that congratulates the winning player and shows final stats like “Total Turns: 24.” Optionally, you can also save this session summary into the database if you want to keep battle history records.

In addition to the battle itself, players should also be able to chat with each other during the game. You need to create a simple chat window where players can type and send short messages at any time - not just on their turn. This will make the game more fun and lively. The messages should appear for veryone right away, and the full chat history should be saved into the database so that it can be reviewed later if needed.
You can try designing a simple chat box with a message display area and an input field. You can display messages along with the player name and a timestamp. You can also design a "ChatManager" class to handle sending and receiving messages if you want a more structured system.

It is important to make sure that battle actions and player chats work smoothly together, without confusing the players. Players should be able to take their turn and chat freely at the same time without any problems. You can try separating the battle system and chat system into two different parts of your code. If you know about multithreading, you can also use threads to keep chat and gameplay responsive at the same time - but it is fine to find a simpler way too if you prefer.

Your goal is to build a full multiplayer experience - not just a battle, but a real, interactive game where players can fight, talk, plan, and react to what's happening, all inside the same program.

Smart Chatbot integration

Along with the player-to-player chat, the game will also include a smart chatbot built directly into the system. Players should be able to interact with the chatbot at any time during the game whenever they need help or guidance. The chatbot will act like an assistant inside the game, allowing players to ask questions about how the game works, their character's abilities, strategies to win battles, special move details, and more.
A good design is to have a separate chatbot input panel or a special button in the GUI where players can type their questions. Chatbot responses may appear in a different color inside the chat area, or you can design a dedicated chatbot window for interaction.

- The chatbot must be built using OOP principles and must follow design patterns. Long if-else chains must be avoided. Instead, the system should be structured so that each type of question is handled separately by a specific handler class.
    - One possible design is to create a QueryHandler abstract class with specific child classes for categories like "Character Abilities," "Battle Help," "Special Move Explanation," and

"Game Rules." A QueryManager can then be used to route incoming questions to the correct handler based on detected keywords or intents.

- The chatbot must be able to recognize *different ways of asking the same question*. For example, "What does my special move do?" and "Explain my special attack" should both be understood as asking about a character's special ability.
    - To handle this, implement a synonym mapping system that groups different phrasings under a common intent /keyword. This could be a simple dictionary that matches various user inputs to a standard internal command like "special_move_info," or you can create an IntentRecognizer class that scans the input and returns the matched intent.
- Your chatbot must handle at least 40 predefined queries. These queries and their corresponding answers should not be hardcoded into the program logic. Instead, store them in an external data source such as a JSON file, CSV file, or SQLite database table.
    - Hint - A QueryDatabase class can be created to load all the predefined questions at startup and provide efficient lookup when needed. Organizing queries externally will make it easier to maintain and update the chatbot's knowledge base later without changing the source code.
- When the chatbot encounters a question it cannot answer, it must not ignore it. Instead, unknown or unrecognized queries should be logged into the database. This helps for future analysis and improvement.
    - One approach is to create a simple "UnrecognizedQueries" table where each failed query is saved along with the player's ID and timestamp. A fallback handler can automatically record the input if no known match is found.
- In cases where the chatbot cannot confidently answer - such as completely new topics or very complicated questions - the system must escalate the request to a *simulated live agent*.
    - Using the Observer Pattern is one way to handle this, where the chatbot acts as a publisher and notifies a registered live agent system when escalation is needed. Alternatively, a LiveAgentManager class could be triggered manually when an escalation condition is detected.
- Escalations must be handled cleanly, with clear feedback to the players. For example, displaying a message like "Your request has been forwarded to a live agent" ensures that players are not confused. Simulated agents can either respond with predefined supportive messages or simply acknowledge the escalation.

It is important to treat the chatbot as an important and independent module inside the overall system. The chatbot must integrate smoothly with the battle engine, chat system, database, and GUI without slowing down gameplay. Keeping the chatbot code separate from other game logic will help make the system more modular and maintainable. For GUI integration, the chatbot can be accessed by a dedicated button ("Ask the Bot") or by typing special commands like "/bot explain special move" in the general chat window. Whichever method you choose, make sure the chatbot is easy to find, simple to use, and fits naturally into the game's user interface.

Database Usage (SQLite3)

For this project, you must create and properly use an *SQLite3 database* to store all the important information that happens during the game. The database will act as a *storage center* that keeps track of everything - players, battles, chat messages, chatbot conversations, and unknown queries - even after the game is over. A well-designed database will make your system more reliable, organized, and easy to maintain later.

You are required to store important information such as:

- *Player details* - player names, the character each player selects, and any starting stats.
- *Game session information* - who won the game, how many turns were taken, and when the session started and ended.
- *Chat messages* - that players send to each other during the game.
- *Chatbot queries and responses* - where both the player's question and the chatbot's reply are saved.
- *Unknown or unrecognized queries* - that the chatbot could not answer, for future review and system improvement.

You should design your database with a clear and proper structure, so that the information is organized and easy to retrieve when needed. One approach is to create *separate tables* for different types of data. For example:

- A Players table to store player names, chosen characters, and initial stats.
- A Sessions table to record session start time, end time, winner name, and total number of turns.

- A Chats table to store all chat messages along with the sender's name and timestamp.
- A ChatbotQueries table for successful chatbot interactions, saving both the player's question and the chatbot's response.
- An UnknownQueries table to log any player questions that the chatbot failed to answer.

Each table should include a *primary key* (such as an auto-incremented ID) to uniquely identify every entry. Adding *timestamps* is strongly encouraged(mandatory), especially for actions like when a player selects a character, sends a message, asks a question, or when a session starts or ends. Timestamps allow for better tracking and can help in future features like session analysis or player performance tracking.

If you want to design a more relational and structured database, you may also choose to include *foreign keys* between tables. For eg, chat messages in the Chats table could reference the Player ID from the Players table, and chatbot queries could reference the Player ID as well. Using foreign keys can help maintain consistency and avoid orphaned data. But, for this project, foreign keys are optional. You are free to keep the tables independent if you prefer a simpler design, especially if you want to focus more on gameplay and core functionality rather than complex database relationships.

The database must be *updated automatically during gameplay*, without needing players to manually save anything.
Some examples of database updates include:

- When a player selects a character, insert a new record into the Players table.
- When a player sends a chat message, insert a new entry into the Chats table immediately.
- When a player asks the chatbot a known question, log both the question and the chatbot's answer into the ChatbotQueries table.
- When the chatbot cannot answer a query, insert the unknown question into the UnknownQueries table.
- When the game ends, insert a session record into the Sessions table, storing the winner's information and total number of turns.

You may consider designing a *DatabaseManager* class to handle all database operations in an organized way. This class could include simple methods such as insert_player(), save_chat(), log_query(), and

record_session(). Having a centralized DatabaseManager will help separate database logic from gameplay logic, making your code cleaner, easier to maintain, and more modular.
If you prefer a different structure for handling the database - such as direct connection management within each module - that is acceptable too, as long as database updates happen correctly at the right times.

By the end of the project, your database should provide a complete and well-organized record of all game activities - including player actions, battles, chat conversations, chatbot help, and improvement areas for the chatbot. A clean database structure will not only make your project look more professional, but it will also allow you to expand your system easily in the future, such as adding features like full game replays, detailed player statistics, leaderboard tracking, or adaptive chatbot learning.

Design patterns requirement

You must apply multiple design patterns carefully throughout your entire project. Using design patterns will help you build a clean, scalable, and modular system that is easier to maintain and expand in the future. For this project, you must apply at least three design patterns from each major category. This means:

- At least three different Creational patterns,
- At least three different Structural patterns,
- At least 3 different Behavioral patterns.

For Creational patterns, examples include using the Factory pattern to create different characters based on player selection, the Singleton pattern for database management, and the Builder pattern if you need to build complex objects like player profiles or chat sessions step-by-step.

For Structural patterns, you can use the Adapter pattern to help the chatbot connect to different backend services, the Facade pattern to simplify backend service access, and the Composite pattern to manage both player chat messages and chatbot responses in a unified way.

For Behavioral patterns, you must apply patterns such as the Strategy pattern for managing different battle actions (attack, defend, special move), the Observer pattern for notifying live agents during chatbot escalations, and the Command pattern for cleanly managing player chat commands and actions.

You are free to use more than three patterns per category if you like, especially if your system complexity increases naturally. However, using fewer than three per category will not meet the minimum

requirement. When selecting patterns, think carefully about where they fit naturally. Do not force patterns into your code if they do not make sense - instead, use them in a way that genuinely improves the structure and flow of your project.

You are encouraged to plan your system architecture carefully before writing detailed code. Sketching out a simple UML Class Diagram can help you decide where to apply patterns meaningfully. The best projects will show how patterns contribute to a stronger, more flexible design - not just satisfy the checklist.

Finally, your project should demonstrate:

- Clear separation of concerns between major systems (battle, chat, chatbot, database),
- Flexibility to add new features like new character classes, new chatbot capabilities, or new chat handling commands,
- Clean modular code that follows professional software design standards.

GUI requirements

Your project must use a GUI for all player interactions. All major parts of the game, including character selection, battle actions, chatting between players, and interacting with the chatbot, must happen inside the GUI. The game should not rely on typing everything into a console window. Instead, players should be able to easily click, select, and type inside different screens.

- At the beginning of the game, you must provide a character selection screen. This screen should clearly show all the available characters along with their basic details, such as health points, attack power, defense, and special move description. Players must be able to select their characters through the GUI. Once a character is selected by a player, it must no longer appear as available to the others.
- After character selection, the game must move to the *battle screen*. This screen should display the health and status of all players. It should clearly show which player's turn it is and what options are available - such as attack, defend, or use a special move. Players must be able to easily pick an action on their turn, and the results of the action (such as damage dealt, special effects, or eliminations) must be displayed immediately in a clear way.
- Your GUI must also include a *chat window* where players can send messages to each other during the game. Players should be able to type a message and see it displayed instantly to all players.

The chat system should work smoothly alongside the battle actions without interfering with the game's flow.

- In addition to player chat, you must create a *chatbot interaction window*. This will allow players to ask the smart chatbot questions about the game at any time. The chatbot's answers should be shown in the same window or in a connected panel, without disturbing the ongoing battle.

Your game must provide a *dynamic and responsive GUI* experience.

Full animated movements or sprite-based fighting animations are **not required** for this project. Instead, you must ensure that important game actions - such as health reductions, battle logs, chat messages, special move activations, and status effects - are updated *smoothly and immediately* on the screen after each event. The GUI should feel lively and interactive, similar to a turn-based board game app, where players can easily see the results of their actions and conversations without delay. Focus on clean, timely updates, such as health bars reducing on attack, status effects appearing beside player names, and chat messages appearing live in the chat window.

Finally, your GUI must display important battle messages clearly. For example:

- If a player attacks another player, a message like "Player 1 attacked Player 2 for 15 damage" should appear in the battle log area.
- When a special move is used, the special action should be announced.
- When a player's health reaches zero and they are eliminated, this elimination should also be clearly displayed on the screen.

You can use a simple GUI library *Tkinter*, which is easy to set up and we discussed in class as well. But, the focus must be on keeping the GUI simple, clean, and easy for players to use, without making the game overly complicated or hard to follow.

Sample output

Character selection phase:

- Jacob selects the character Warrior.
- Matt selects the character Mage.
- Charles selects the character Archer.

(The GUI updates immediately: Warrior, Mage, and Archer become “Selected” and are no longer available for others.)

Battle phase begins:

Turn 1:

Jacob (Warrior) chooses to Attack Charles (Archer).

- Jacob’s attack power = 30.
- Charles’s defense = 5.
- Damage dealt = 30 - 5 = 25 damage.
- Charles’s HP reduces from 70 to 45.
- Charles also becomes Poisoned (for 2 turns)

Battle log:

[BATTLE] Jacob attacked Charles for 25 damage.

[BATTLE] Charles is now poisoned (2 turns).

Player status panel:

Jacob (Warrior) - HP: 100 - Status: None

Matt (Mage) - HP: 100 - Status: None

Charles (Archer)- HP: 45 - Status: Poisoned (2T)

Turn 2:

Matt (Mage) uses Special Move: Magic Shield.

- Matt gains a Shield (next attack damage reduced by 50%).

Battle log updates:

[BATTLE] Matt used Special Move: Magic Shield.

Player Status Panel:

Jacob (Warrior) - HP: 100 - Status: None

Matt (Mage) - HP: 100 - Status: Shielded

Charles (Archer)- HP: 45 - Status: Poisoned (2T)

Player chat during battle:

Chat Window messages:

[Jacob]: That was a strong hit!

[Matt]: Thanks, I'm ready to defend now!

[Charles]: phhh, I'm poisoned... what do I do?!

Chatbot help interaction:

- Charles clicks the "Ask Bot" button and types:

How do I cure poison?

Chatbot reply appears in chat:

[Bot]: You can heal by using your special move or by defending for two turns.

Sample GUI Window layout (during gameplay)

GoPirate Multiplayer battle game

Player status

Jacob (Warrior) - HP: 100 - Status: None

Matt (Mage) - HP: 100 - Status: Shielded

Charles (Archer)- HP: 45 - Status: Poisoned (2T)

Battle log

[Battle] Jacob attacked Charles for 25 damage

[Battle] Charles is now poisoned (2 turns)

[Battle] Matt used Magic Shield

Player chat and Chatbot messages

[Jacob]: That was a strong hit!

[Matt]: Thanks, I'm ready to defend now!

[Charles]: Ugh, I'm poisoned... what do I do?!

[Bot]: You can heal by using your special move.

Type message: [_____] (Send) (Ask Bot)

Health bars, status effects, battle logs, player chat, and chatbot replies update live. No fancy animation needed - just clean, dynamic updates.

Next turns:

- Charles's turn: He chooses Defend to reduce incoming damage.
- Jacob's next turn: He attacks Matt (Mage), but Matt's Shield reduces the damage by 50%.
- Battle continues until only one player is left standing.

End of the session (Eg):

Once two players are eliminated:

Congratulations! Jacob wins the game after 18 turns!

Final Stats:

- Jacob (Warrior) – HP: 35 (Winner)
- Matt (Mage) – HP: 0 (Eliminated)
- Charles (Archer) – HP: 0 (Eliminated)

A simple popup or final screen should display these results.

Instructions

Before you submit your project, please ensure your system includes the following components:

- A multiplayer battle game where at least three players select different characters and take turns battling each other.
- A player-to-player chat system that works smoothly during the game.
- A smart chatbot integrated into the game that can answer player questions and escalate unknown queries to a simulated live agent.
- GUI for character selection, gameplay actions, player chat, and chatbot access through a clean, user-friendly design.
    - A properly designed SQLite3 database that stores necessary details, eg Player profiles and selected characters, game session details, chat history, chatbot queries and responses, unknown queries for future improvement, etc.
    - Don't forget to apply OOP principles: Encapsulation, Inheritance, Polymorphism, Abstraction
    - Use proper design patterns: At least three Creational patterns, three Structural patterns, three Behavioral patterns.
    - Apply good software design practices: Clear modular code organization, proper class responsibility separation, readable and well-commented code.
    - Write simple unit testing: Eg, write tests to verify important functionality such as battle calculations, database insertions, or chatbot responses.
- **Documentation:**
    - A short README file explaining: How to run your project, Any special libraries you used, Extra features you added, etc.
    - A clear UML class diagram showing the structure of your important classes.

Presentation Instruction:

Each team must also give a short presentation during the final exam session on May 9, between 8:00am

and 10:00am. Each team will be given 10 to 15 minutes to present their project. You must briefly explain your system structure, demonstrate your project running, and highlight the main features such as OOP use, design patterns, and the chatbot system. All the team members must participate as it is a team submission.

During your presentation, you must clearly cover the following points:

1. Project overview
    - Briefly explain what your project is about and what two systems you combined.
2. Character selection and Battle system
    - Show how players select their characters and how turn-based battling works.
3. Player-to-player chat
    - Show the chat system working during the game.
4. Smart Chatbot integration
    - Show how players interact with the chatbot,
    - Explain how known and unknown queries are handled.
5. GUI design
    - Walk through your GUI screens (character selection, battle screen, chat window, chatbot access).
6. Database usage (SQLite3)
    - Explain what information is stored in the database,
    - Mention the main tables used.
7. Application of OOP principles
    - Explain how you applied Encapsulation, Inheritance, Polymorphism, and Abstraction.
8. Design patterns used
    - List the design patterns you applied and where you used them.

9. Unit testing
    - Mention any unit tests you created and what functionality they tested.
10. Challenges, learning, and team contributions
    - Briefly mention any difficulties faced, what you learned
    - Clearly describe the contributions made by each team member, including who worked on which parts of the system (for eg: battle system, chatbot integration, database setup, GUI design, etc.).
11. Conclusion
    - Summarize your work and mention any extra features or ideas for future improvements.

Important:

- A live demonstration of your working program is expected.
- Keep explanations simple, clear, and organized.
- Practice ahead of time to stay within the time limit

Submission Guidelines

You must submit:

- Your complete project folder in .zip format, including:
    - All Python files, Your SQLite3 database file (.db), Any data files used (such as csv/JASON or images).
- A README file inside the project folder.
- A short final report explaining:
    - The UML diagram, How OOP principles were applied, Which design patterns were used and why, GUI implementation, Screenshots showing your GUI during - Character selection, Gameplay action, Player chat, Chatbot interaction. Your team contributions, etc.

Grading rubric

Category	Points
Game Functionality (battle engine, turn logic, special moves)	20 points
Player Chat System	15 points
Smart Chatbot System (correct structure, escalation handling)	15 points
GUI Design and Responsiveness	10 points
SQLite3 Database Integration	10 points
Application of OOP Principles	5 points
Design Patterns	7 points
Basic unit testing	3 points
Presentation and team contribution	10 points
Documentation (README and UML diagram)	5 points
Total	**100 points**

Attendance policy

Attendance for the final exam session and project presentation is mandatory. If you do not attend the final exam and do not present your project, you will receive a zero for the final project, even if you submit your code, documentation, or other materials. No grading will be done for student who are absent during the final exam presentation. Please make sure to be present, prepared, and ready to present your work during the scheduled time.

Best Project Selection

All teams are encouraged to do their best and show creativity, teamwork, and strong design in their projects. At the end of all presentations, I will select one project as the Best Project, based on all the factors described above, including system completeness, good use of OOP and design patterns, functionality, creativity, and overall presentation quality. The Best Project will be announced after all teams have finished presenting.

Good luck, and I am looking forward to seeing your great work!