onlinecards.uk

Matti Armston

March 19, 2025

For my first pet - whose name I can no longer remember.

Contents

1 Analysis **4**
- 1.1 Stakeholders . . . 4
  - 1.1.1 Surveys . . . 4
  - 1.1.2 Interviews . . . 6
- 1.2 Justification . . . 7
  - 1.2.1 Abstraction . . . 7
  - 1.2.2 Thinking Ahead . . . 7
  - 1.2.3 Decomposition . . . 8
  - 1.2.4 Thinking logically . . . 8
  - 1.2.5 Thinking concurrently . . . 9
- 1.3 Research . . . 9
  - 1.3.1 online-spades.com . . . 10
  - 1.3.2 cardsjd.com . . . 14
  - 1.3.3 pokerpatio.com . . . 17
- 1.4 Features . . . 23
- 1.5 Requirements . . . 25
  - 1.5.1 System Requirements . . . 26
- 1.6 Limitations . . . 26
- 1.7 Success criteria . . . 26

2 Design **28**
- 2.1 Design principles . . . 29
- 2.2 Games and Variations . . . 29
- 2.3 Networking . . . 30
- 2.4 Front End . . . 32
- 2.5 Lessons . . . 33
  - 2.5.1 Games and Variations . . . 33
  - 2.5.2 Networking . . . 39
  - 2.5.3 Front End . . . 42
  - 2.5.4 Algorithms . . . 45
  - 2.5.5 Testing . . . 46
  - 2.5.6 Further development . . . 47

3 Implementation **48**
3.1 Overview . . . 48
3.2 Version 1 . . . 49
3.3 Version 2 . . . 51
3.4 Version 3 . . . 53
3.5 Version 4 . . . 54
3.6 Version 5 . . . 59
3.7 Version 6 . . . 60
3.8 Version 7 . . . 63
3.9 Version 8 . . . 65
3.10 Version 9 . . . 67
3.11 Prototypes . . . 69
3.12 Testing . . . 77

4 Evaluation **81**
4.1 Final Testing . . . 81
4.2 Usability Features . . . 83
4.3 Reflection . . . 83

5 Appendix **86**
5.1 Figures . . . 87
5.1.1 Surveys . . . 87
5.1.2 online-spades.com . . . 89
5.1.3 cardsjd.com . . . 92
5.1.4 pokerpatio.com . . . 96
5.1.5 Design . . . 101
5.1.6 Implementation . . . 110
5.2 Interviews . . . 153
5.2.1 Jamie . . . 153
5.2.2 Duncan . . . 153
5.3 Development log . . . 154

I

Analysis

"Look on my Works, ye Mighty, and despair!"
- *Ozymandias* Percy Bysshe Shelly

I want to make a program that would allow users to play card games with their friends over the internet. One person would host the game, selecting the rules and variations (e.g. Spades with the Bauer trumps variation) and they would then invite their friends to join. The host would then connect to their friends computers creating a client-server architecture. The host would track the state of the game and update the clients when it changes. When the players need to make a decision; the host would send a packet to the client and await their response, the client would respond with their decision and the host would update the game state and notify the clients.

1.1 Stakeholders

Stakeholders will be people who enjoy card games but, for whatever reason, cannot play in person. This could include the elderly, people who are ill, or friends who live far apart. 'I know your deeds, your hard work and your perseverance' Revelation 2:2 and I wish to provide people with an escape and relaxation. The website could also be used to teach and introduce people to card games and new rules they haven't tried before.

1.1.1 Surveys

I sent a survey to potential stakeholders to see what I should focus on. As Figure 1.1 shows, my stakeholders would be more likely to play cards if they can play with friends online. This shows that there is a market for my program.

There are 3 different ways that I can imagine to release the program: as a desktop application, a mobile app or as a website. In my opinion the most important factors are how easy the app is to use and how likely a new person is to download the app or visit the website. In my survey, I identified that a website is the best option because the majority of people voted it the easiest to explain and most likely to use (Figure 1.2).

I also asked which features my stakeholders considered most important (Figure 1.3). Based on the results, I will focus on making a clear and minimalist interface, fast networking and having multiple different card games for users to choose from. For the interface, I will use HTML forms and buttons they can click since most people are familiar with these. Most stakeholders also preferred an interface 'dark like the tents of Kedar' Song of Songs 1:5. I will also use animations to show visually what is happening. The networking code will be the focus of most my optimisation.

One of the advantages of card games is that complex games, such as Contract Bridge, are often descendants from simpler games, such as Whist, so I can build more complex features on top of the simpler games to create their more descendants. For example, Spades is very similar to Whist but it has a bidding round, this means that after writing the code for Whist it is not much more complex to expand it for Spades. This will allow me to create lots of different games and variations quickly, making the project more feasible. However, I don't want all the games to be too similar, so I will aim to create games from different genres.

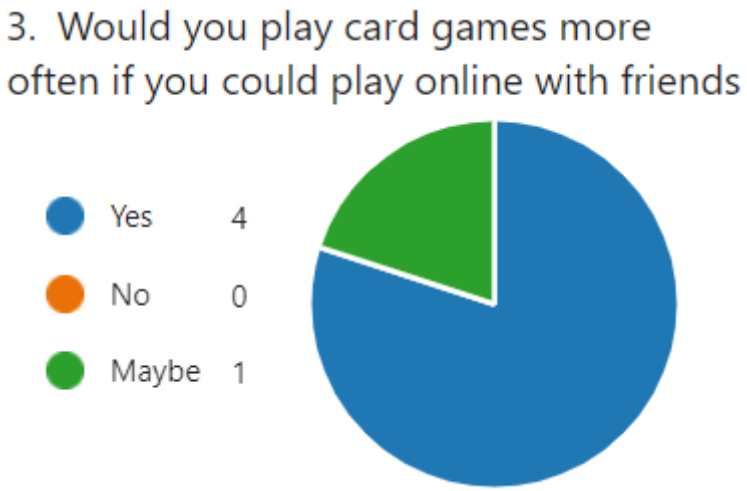


Figure 1.1: Stakeholders are more interested in playing cards online

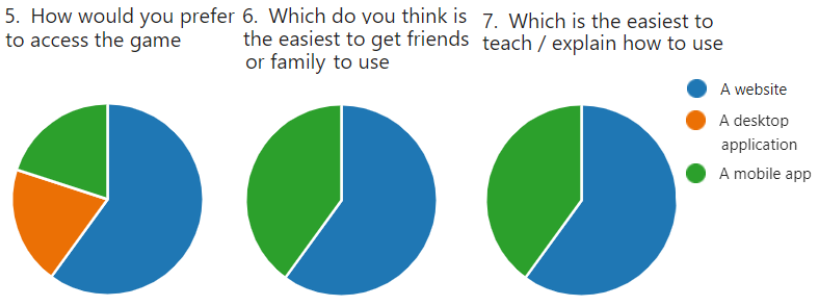


Figure 1.2: A website is the most popular format

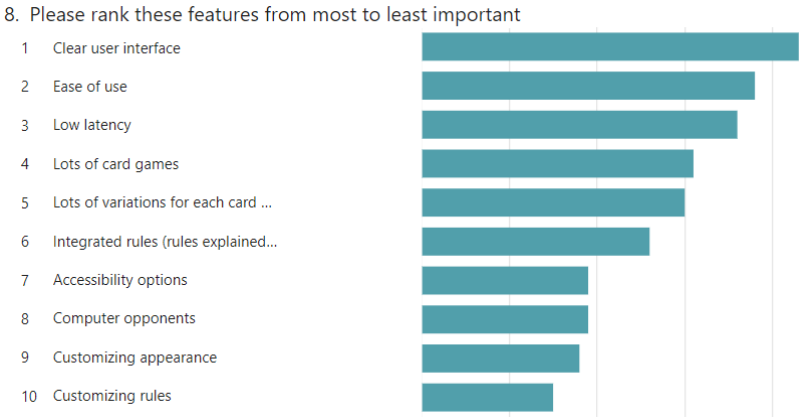


Figure 1.3: Ranking of features

1.1.2 Interviews

I decide to conduct interviews with a couple of my stakeholders. I wanted to get more in depth information about their situations and how to tailor the program to fit their needs. One of my main aims is to make my program convenient so I focused on how I could achieve this. I also wanted to get in depth answers to more open ended questions than I had asked in my survey. My interviewees may also have ideas that I had not thought of which could improve my program.

One of my stakeholders is Jamie Butler-Eales who regularly plays card games every Thursday morning, but he would like to play more often. I interviewed him (see 5.2.1) to see why he plays card games, what he dislikes about them and how my program can improve his experience of them. He enjoys card games because they 'require a level of skill, problem solving and on the spot thinking' but he dislikes 'silly rules that you cannot edit' and my variations would allow him to change

them. I would also offer games that 'cannot be found online such as Promise'. Online he could play 'at any time, with anyone' where he would like the program to performant and make 'finding your friends' easy. Jamie would also like the ability to play with 'random players' if his friends are not available.

Another one of my stakeholders is Duncan Moynihan who has an app on his phone that allows him to play Gin Rummy and Spades against computer opponents. However, he also wants to play with his friends and dislikes the copious number of advertisements present in this app. He likes the 'high level of skill to master' a card game but can't always 'be physically with someone to play a card game' (see 5.2.2) It would be more convenient for him to play 'from the confines of my abode'. Playing online would also 'increase the speed of play' and 'reduces the chance of rigging the deck'. Duncan would also like more customisation.

1.2 Justification

I am writing this as a computer program to allow users to play cards with their friends even if they cannot arrange to meet in person. This could be particularly good for people that have demanding jobs, live far apart or cannot meet due to other reasons e.g. health concerns. 'Come to me, all you who are weary and burdened, and I will give you rest.' Matthew 11:28 exemplifies how my project would allow them to stay connected with their friends more easily and play games they enjoy.

1.2.1 Abstraction

I will use abstraction by reducing most parts of the games into objects. Cards will have attributes including suit and rank, Players will have names, scores, a hand (a list of Cards), etc. This will help me to focus on only the necessary details for the implementation. I will also try to reduce the number of actions a player can take, to reduce complexity and the ability for a user to cheat.

1.2.2 Thinking Ahead

One of the major issues I will face is networking. I am planning to have a user create a game and then become the host computer, who will track the game state, player attributes, etc. When it is a user's turn, the host will inform them of the decision they have to make, the client will receive the message, the user makes their decision and the client informs the host of this. Next, the host updates the game state and sends the updated game state to all the clients.

In order to do this I will use web sockets to establish a connection, and then communicate using JSON. For this I will create an API, a

format for messages and various fail safes and defaults if connection is lost etc.

The website may not work if a user is using a VPN since the web server will not be able to get the IP address of the actual host machine (only the VPN server). This means that clients would not be able to connect to the host, and would instead attempt to connect to the VPN server. In order to prevent this, users will have to use a Peer-to-Peer VPN server.

1.2.3 Decomposition

I will use decomposition by splitting the project into several large parts: the user interface, the game method (and its creator) and the network code. Each will have a interface that they use to communicate with each other, without exposing the inner details of their implementation.

The user interface will be a website since most people are familiar with them and it is easy to encourage a new user to visit a website (See Figure 5.2). One part of the website will be quite simple, hosting rules and variations for each game and allowing users to create and join games. When a user creates a game, the website will create a corresponding database entry, and when a user wants to join a game, the database will be queried for a corresponding game.

During the game, the user interface will communicate what is happening to the user as well as what decisions they have to make. This will have to be dynamic and change based on what the game is. For example, the decisions that a user has to make in Spades is very different to those in Poker and a user needs to always be able to follow what is happening.

For each game, the rules will be coded so that the users cannot cheat. I will try to create reusable sections of code that can be used for multiple games and variations. I will decompose each of these games into smaller parts such as bidding, deciding which cards can be played and deciding the trick winner. Different variations would each be different functions and these would be combined by the game method creator the build the game method. This would allow for lots of flexibility as the user could simply change rules that they do not like by choosing a different variation.

1.2.4 Thinking logically

There are multiple different times when users will have to make decisions. My aim is that the game 'will make straight your paths' Proverbs 3:6 and provide a easy user experience. For example, when creating a game, they will have to complete a HTML form to decide which rules, variations and how many players they wish to play with (if the game allows for it) to tailor the game to their tastes.

During the game, users will also have to make decisions within the context of the game rules. This could be deciding how many tricks they wish to bid, or choosing which card they wish to play. The client will be informed that they have a decision to make as well as any additional, necessary information (e.g. the options they can choose from). Once the user has made a decision, the client will inform the server of the decision and the game state will be updated. This system would allow for lots of flexibility since the choices a users has to make in Poker and Spades are very different.

However, due to the unpredictability of internet connections, real life distractions, etc. a user may take a very long time to make a decision which would disrupt the game. In order to prevent this, I would add an optional feature where, after a set period of time, the player makes a predetermined, default decision. For some games, this would be simple (e.g. check/fold in poker) but for others (e.g. Spades) this would be more complex. It would also be important to communicate this to the players. Overall, I think this would improve the user experience since it prevents the game from becoming stuck if a user looses connection and ensures that the game has a consistent pace.

1.2.5 Thinking concurrently

Most card games are turn based so very little concurrency is required. However, I will need to ensure that all clients and the host have the same game state. I will do this by the host sending JSON packets to the clients informing them of the new game state, which the clients will then confirm they have received. If a client does not receive a packet, and the game updates again, I will have to think of a method of ensuring that they stay up to date. A system that numbers each update (for example) would allow the clients to ensure that their game is up to date and they haven't missed any updates. This would prevent a scenario where 2 different users have different ideas about what the game state is.

1.3 Research

In order to improve my project, I found some examples of existing websites, where users can play cards. I focused on the card game Spades, since it is a well known game with many online adaptations. I found 2 different but similar websites that I could take inspiration from.

1.3.1 online-spades.com

online-spades.com is a website that allows users to play the card game Spades in their web browser. They team up with a computer controlled partner to play against 2 computer opponents.

What I Like	**What I Dislike**
Minimalist interface, reduces confusion (Figure 1.4)	No score board after each round so difficult to track scores
Players sit at cardinal directions	Separate buttons for each bid, I would use a select or number input instead.
Playable cards are highlighted (Figure 1.6)	No variations
Animations show who has played which card and who wins each trick	Only 1 card game
Users can see which bids are allowed	
Users can see each player's bid at all times	
Users can't break the rules of the game	
Can change appearance (Figure 1.5)	

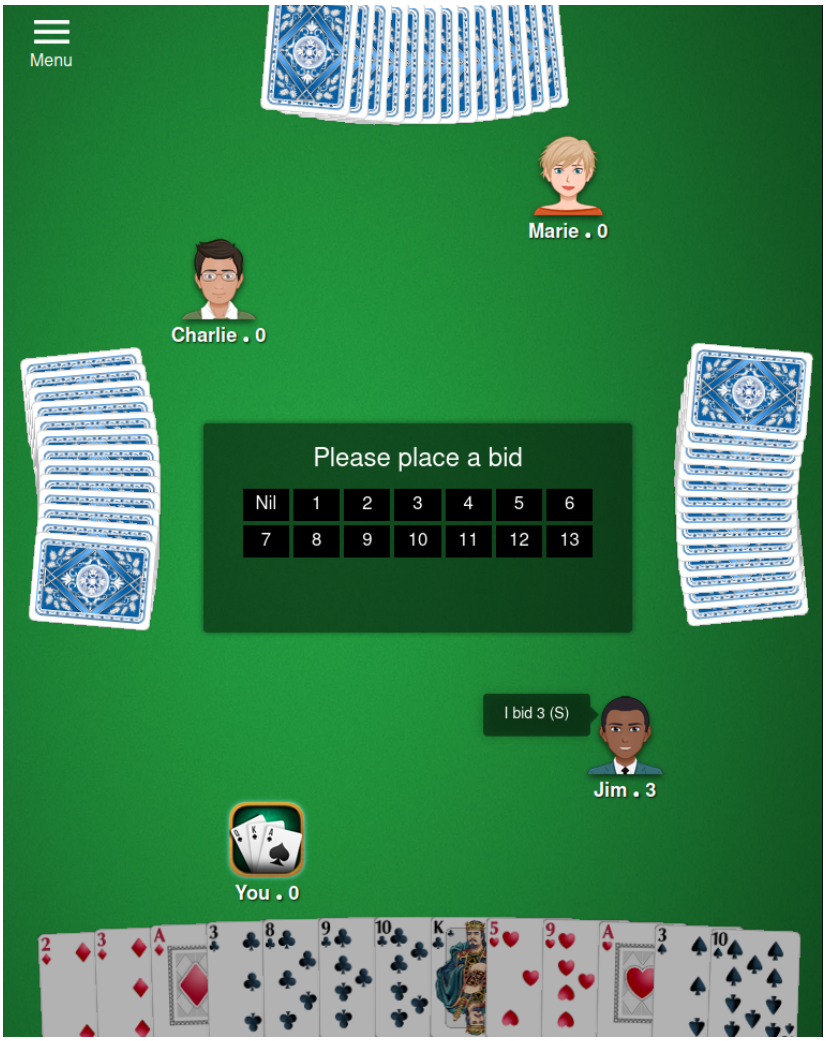


Figure 1.4: online_spades.com allows users to play spades in their browser.

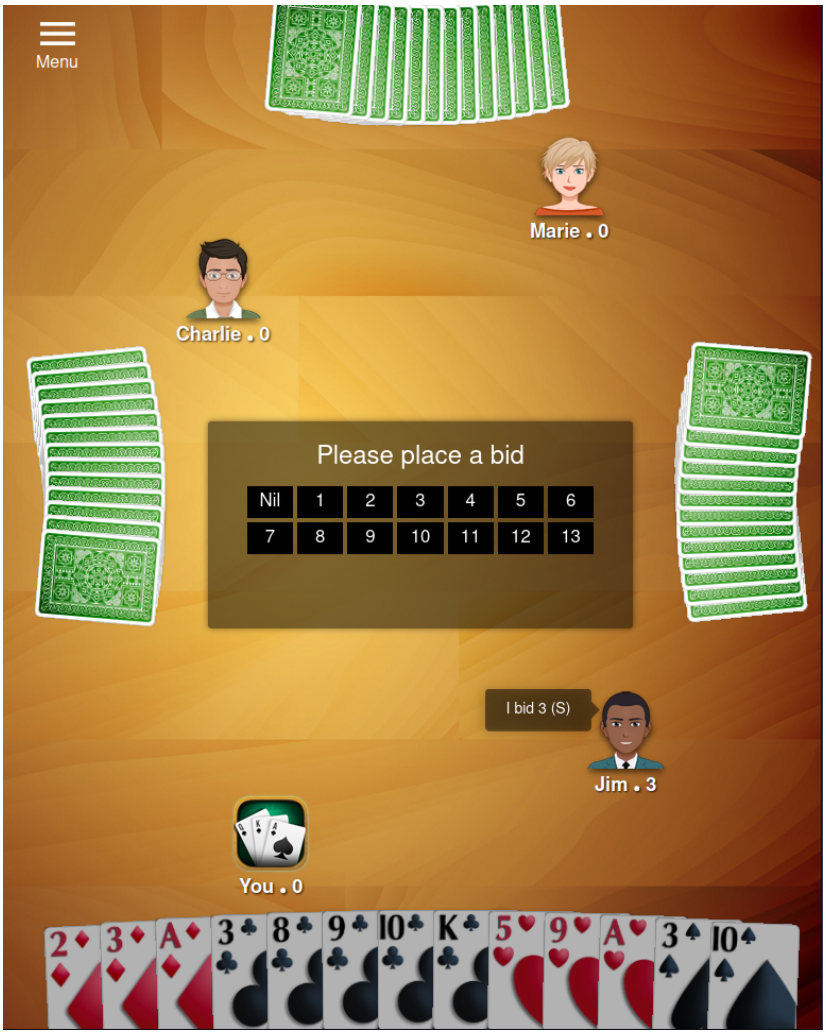


Figure 1.5: Alternate themes are available.

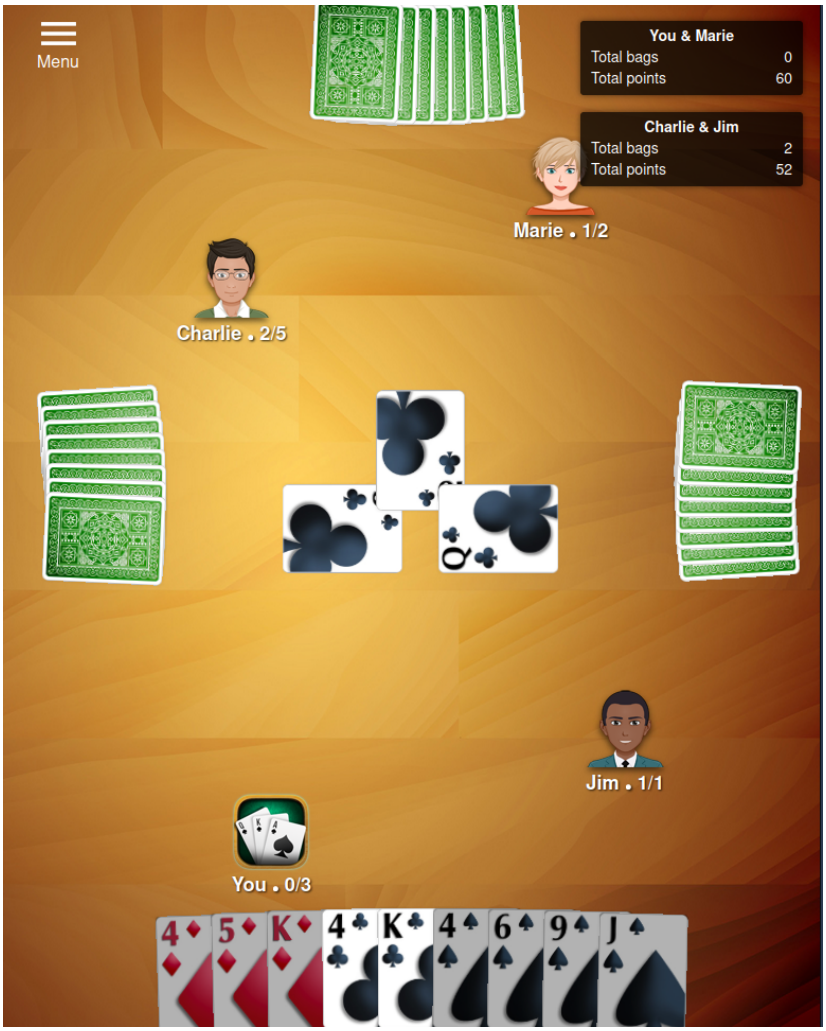


Figure 1.6: Playable cards are highlighted.

1.3.2 cardsjd.com

cardsjd.com is another website that allows users to play spades against the computer. It is rather similar to online-spades.com but it does have some differences. For example, it offers different games such as Gin Rummy and Hearts as well as multiple difficulty levels.

What I Like	What I Dislike
Score board shown after every round (Figure 1.7)	Animations play too fast
Multiple games (Figure 1.8)	Can't change appearance (Figure 1.10)
Multiple difficulty levels (Figure 1.9)	No variations

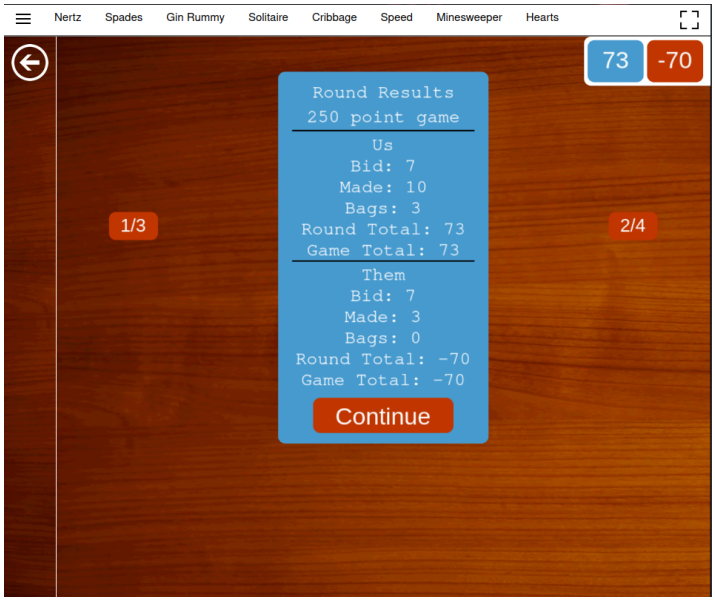


Figure 1.7: The scoreboard in cardsjd.com/spades

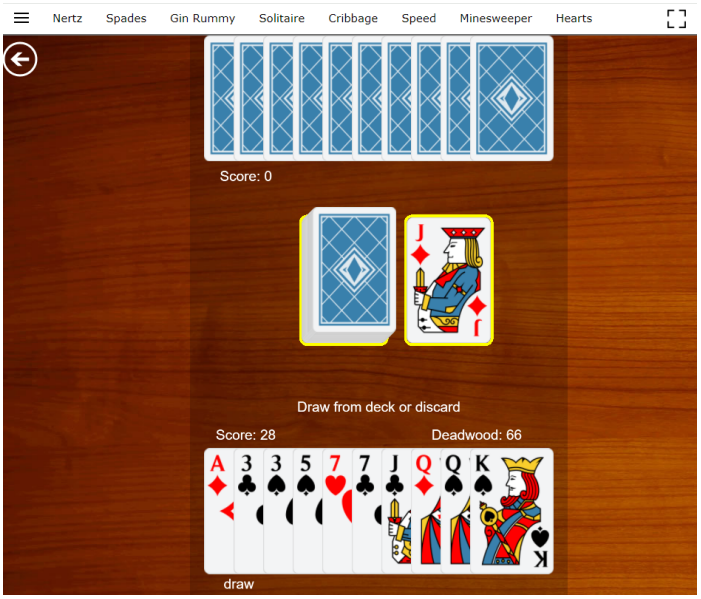


Figure 1.8: cardsjd.com offers other games such as Gin Rummy

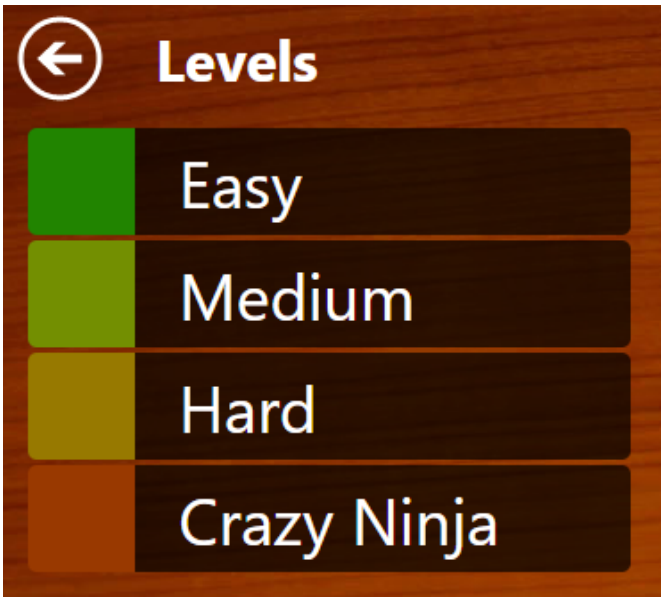


Figure 1.9: cardsjd.com allows users to change the difficulty for some games

Figure 1.10: cardsjd.com/spades has similar visuals to online-spades.com

1.3.3 pokerpatio.com

pokerpatio.com (Figure 1.11) is a website that allows users to play poker against bots or their friends (unlike online-spades.com and cardsjd.com) for play money. This is important since 'Wealth gained hastily will dwindle' Proverbs 13:11. It also provides rules and strategy pages to allow new players to quickly learn and improve.

What I Like	**What I Dislike**
Users can play against friends	I encountered a game breaking bug while testing
Players are arranged around a table (Figure 1.12)	Users can't tell the program to make an automatic action (if possible) on their turn. For example, when another player has made a large bet, automatically folding, regardless of what the other players do.
The host can decide which players can join the table (Figure 1.12)	Little customisation of the visuals/UI
The host can decide which players can join the table (Figure 1.12)	
Clear and simple UI (Figure 1.13)	
Can customise game options (Figure 1.14), including: • Blinds • Decision time • Maximum number of players • etc.	
Players default to check/folding if their decision time runs out	
Convenient bet sizing options	
Built in chat function (Figure 1.15)	
Stats page (Figure 1.16)	
Rules and strategy guides to help new players (Figure 1.17)	

Handles resizing well (Figure 1.18)	

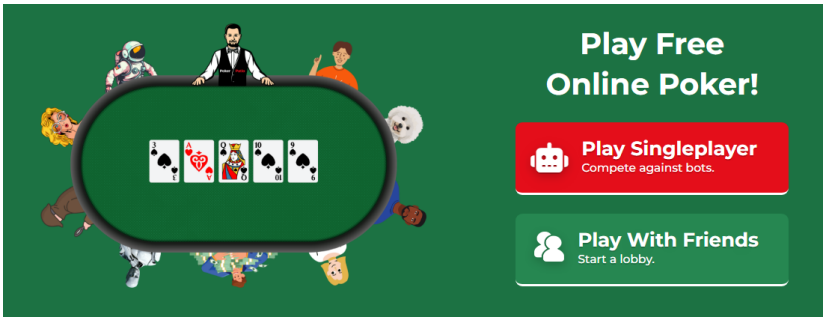


Figure 1.11: The homepage of pokerpatio.com

Figure 1.12: The host approves each player

Figure 1.13: Clear and simple UI

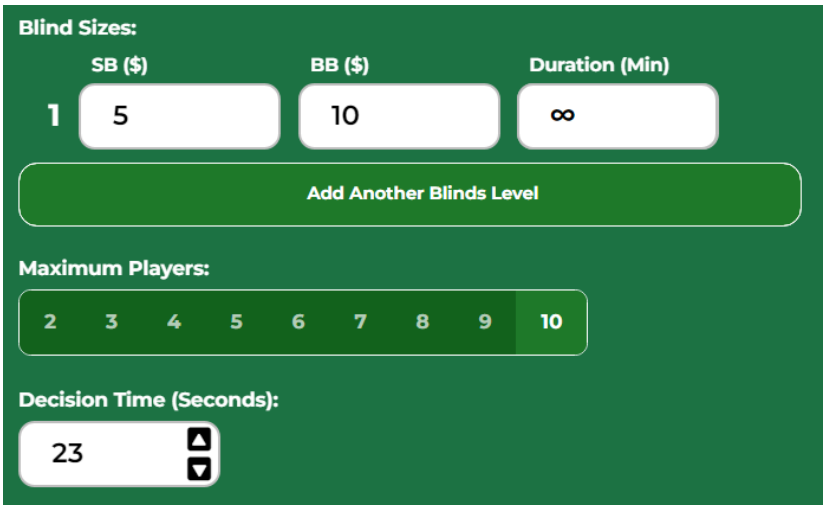


Figure 1.14: Game settings the host can change

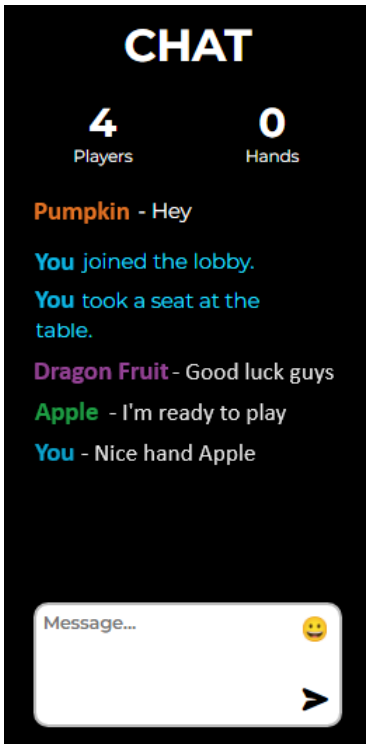


Figure 1.15: Built-in chat function

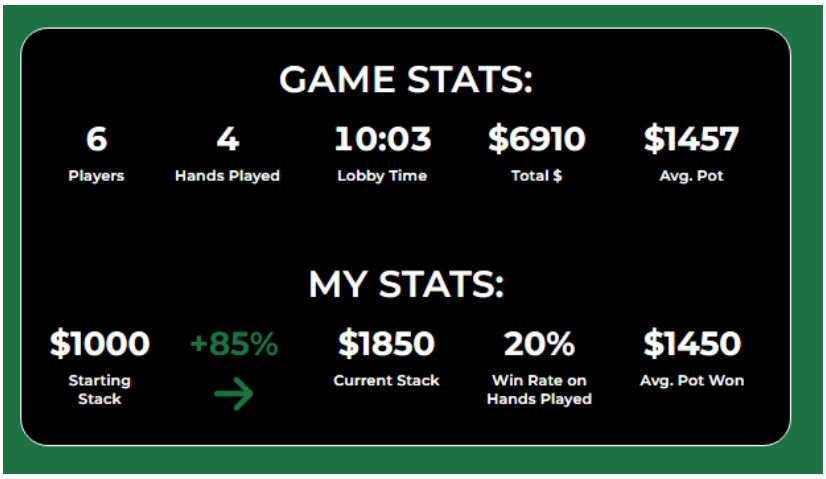


Figure 1.16: Stats page for the session

Beginner's Guide to Playing Poker:

Poker is one of the most beloved card games in the world, known for its depth, variety, and social aspect. As such, it can be played in various formats, three of which we will explore today – **Texas Hold'em, Seven Card Stud, and Omaha**.

These games share some common elements, such as the standard 52-card deck and the fundamental hand rankings, but they each bring their own distinct set of rules and dynamics. For someone interested in learning poker, understanding these three popular versions is a great place to start.

This comprehensive guide provides an introduction into the basics of poker, explores the rules of popular variants, and beginner strategies.

Poker Hands

Poker is played with a standard deck of 52 cards, with cards ranked from 2 (lowest) to Ace (highest). The deck has four suits: Hearts, Diamonds, Clubs, and Spades. No suit is higher than another.

Before we start with the game types and different rules, let's understand the possible combinations of cards or 'hands' you can get in poker, ranked from highest to lowest:

- **Royal Flush**: The best possible hand in poker. It's an ace high straight flush.
- **Straight Flush**: A five-card straight, all in the same suit.
- **Four of a Kind**: Four cards of equal value.
- **Full House**: A three of a kind of one value and a pair of another value.
- **Flush**: Any 5 cards, all of the same suit.
- **Straight**: Any 5 cards of sequential value.
- **Three of a Kind**: Three cards of the same value.
- **Two Pair**: A pair of one value and a pair of another value.
- **Pair**: Two cards of the same rank.
- **High Card**: When you don't have any of the above, your highest card plays. If two or more players hold the highest card, a kicker comes into play.

Figure 1.17: The website has guides for beginners

Figure 1.18: The game can also be played vertically

From these websites, I want to copy the minimalist design that reduces confusion, but I want to change the visual style. I want to have simpler card faces, card backs and a smaller scale. My website should look unique and be easy to identify from its competitors. I also want to mimic the animations, highlighted cards and mouse controls. However, I also want to add keyboard shortcuts and other convenience features if I have time.

However, these examples are all single-player games where the player plays against computer opponents rather than with their friends. My project would focus on allowing friends to play together. These websites offer the well-known game of spades, whilst I am hoping to offer many

different games, including a game called 'Promise'. I have not seen English rules for this game online and this could be a unique selling point.

1.4 Features

These are the essential features I will need to code in order for the website to be functional. There are many other optional features that I would like to include, but I must first implement these main features. Once I have implemented them, I will have a minimum viable product that I can then expand upon.

- When a user clicks on the website they will see 3 options
  - New Game, so they can create a new game
  - Join Game, so they can join a friend's game using a game code
  - Rules, so they can learn the rules and variations for all of the games offered
- When creating a New Game, a user will be able to choose
  - The game rules (e.g. Promise or Spades)
  - Game variations (e.g. Solo Spades)
  - No. Players (if allowed by game rules)
  - Game Length (e.g. starting hand of 10 in Promise)

  to tailor the experience to their tastes.
- When a New Game has been created
  - The user who created the game becomes the host
  - A database entry is made for the New Game
  - The host is given a link and game code to send to their friends
  - Clicking the link or entering the game code allows a friend to join the game
  - Clicking the link checks the database for a corresponding entry and gets the IP address of the host
  - The friend's computer now connects to the IP address and initialises the game based on rules, variations etc.
  - When the required number of players has joined the game begins

This reduces the overhead and cost of the web server. This is because the web server only needs to handle web requests and does not need to track the state of the game or communicate with players.

- When playing a game
  - The host keeps track of the game state
  - When a player has to make a choice, the host sends a packet to the client informing them of their options
  - The client updates the screen, the user makes a decision and the client sends a packet back to the host telling them which option was chosen
  - The host updates the game state based on the decision made
  - The host informs clients of the new game state and the clients update the screen

  This ensures that there is a single source of truth on the game state (the server) and it keeps all players in sync.

- When a user wants to join a game
  - They click the link their friend has given them
  - Or they click Join Game and enter the game code their friend has given them
  - The link would use the same game code as the Join Game page to avoid confusion

  This ensures that it is easy for new users to join a game.

- When a user clicks on the rules page
  - They see a list of all the supported games
  - Each game is a link that takes them to a rules page for that game
  - Each rules page lists the basic rules for each game
  - Below the rules, all supported variations are listed and explained

  This will allow new users to easily learn the rules and variations for each game.

If I complete the project before schedule additional features could include are:

- A chat box

- Public games that anyone can join
- A 'lobby list' for users to browse public games
- Customisation of visuals
- Rankings for players

1.5 Requirements

In order for my project to succeed these are the minimum requirements.

- Users can create a game and choose:
  - The rules
  - The variations
  - No. of Players (if allowed by rules)
  - Game length
  - Whether the game is online or local

  to tailor the game to their tastes.
- Users can join a game
  - By clicking on a link
  - By visiting the Join Game page and entering a game code

  Both of these would be given by a friend who created the game and would be used to fetch the corresponding database entry for the game.
- Users can select a nickname for the game, in case they don't want to use their real name
- Game creator and their friends would connect directly to each other (using client-server architecture) to reduce the overhead and cost of the web server.
- Players cannot break the rules of the game to reduce the chance someone cheats
- Minimalist interface and clear animations so they can be easily understood by new users

1.5.1 System Requirements

In terms of software, my program only requires a web browser and a stable internet connection. However, the hardware required varies depending on whether a user is hosting a game or joining a friend's game. If joining a game, a user only requires 512MB of RAM and a 1GHz CPU. However, the host would require at least 1GB of RAM and a 1.5GHz CPU.

1.6 Limitations

There are some features that I do not want to include in my project and will be limitations of it. These include:

- Computer opponents to play against: I want to focus on giving users lots of different games and variations to choose from to tailor their experience towards something classic, competitive, unique or best for their personal taste. This means that each player must account for the variations chosen and adjust their strategy accordingly. Programming a computer opponent to do this would be very difficult and would massively increase the scope of the project.
- Chat features: One of the main things that I like about card games is the ability to socialise and talk to friends. However my program would not have a chat box at launch. It is a feature that I would like to add, but otherwise users would have to use a separate program such as Discord or WhatsApp to chat with their friends.
- Public games: My main focus would be allowing users to play with their friends rather than strangers online. One reason for this is that public lobbies would require moderation, to ensure that users are not trolling, being toxic, or hacking. Within friend groups this moderation is likely to already take place since people would rather be invited back to play again, than win by cheating.
- Rankings: Rather than focusing on competitive play, I would rather focus on a casual audience who likes to play for fun. I believe that this a larger audience and one that is less likely to cause issues with toxic behaviour or hacking.

1.7 Success criteria

In order to declare my project a success I ensure the following:

- Users can fill out a HTML form which automatically generates a corresponding database entry representing the game. Users are then given a link and a game code to send to their friends. This will allow them to easily create the game with the rules of their liking and have their friends join.

- Clicking on a join link (or entering a game code) checks the database for the corresponding entry. If it exists, it will connect with the host computer. This will allow for a peer-to-peer system which will reduce overhead and costs for the web server.

- The host and client computers can connect and communicate to sync the game state and inform each other of any decisions that have been made. This will be done using a JSON API since browsers already have good JSON support. It would also offer lots of flexibility for passing different messages depending on the game.

- The host computer is given a game method that will mutate a list of players into a scoreboard. This will be generated dynamically depending on the game and variations chosen. This will be done by changing the methods used for each task. For example, replacing the default bidding method with the 'sum of bids' variation (see Wikipedia), where the sum of all the players bids cannot equal the number of tricks. This will allow for greater variety than other card game websites.

- A user interface that can change based on the game to better display the necessary information. However, it will almost always need to show the player's hand of cards, how many cards their opponents have and also their opponents' score. This will allow users to more easily track the state of the game.

II

Design

"Failing to prepare is preparing to fail"

- Sam McDonnell

My project can be separated into 3 major parts: the web based front end, the code for each card game and its variations, and the network code. I will approach these 3 parts as individual problems and use different techniques to solve each one. Each problem has different limitations and details that make them better suited to different approaches. Separating each problem allows me to use different techniques for each which will lead to a better overall solution.

This can be done so long as I design each part to use a standard interface. In the case of combining the networking and front end, the front end will read the JSON received by the network code and generate the graphics based only on this. An advantage of this is that the front end is encapsulated so changes to the network code (assuming the output is the same) has no impact on the front end. Similarly the game method will use an abstracted interface to communicate with the network so that it is unaffected by the network implementation.

Card games are well suited to using object oriented techniques to represent cards and players since many details can be abstracted away and there is very little necessary information (see Figures 2.6 and 2.7). However, I do not want to use solely object oriented techniques for the game rules and I will actually use a procedural approach instead. Whilst players and cards will be objects, they will not have methods and so will behave more like structs in C. This may provide more versatility than pure object oriented if I can more easily and predictably replace functions with their variations. The game method creator will be programmed with a similar approach to increase continuity between the two but will likely use fewer object oriented techniques. However, the front end may use more object oriented techniques since it mostly

uses templates to show different data for each player in the same format. These templates could be modelled as objects with the data being attributes so different players have different information in the same format. Many element will also be similar across different parts, for example text boxes will have the same formatting and so should share some common attributes.

2.1 Design principles

The Design Game

In order to quickly prototype various ideas for the design, adhering to the mantra of 'fail fast' I have created a series of party games that help quickly test and identify issues with various parts of the design. Players will attempt to design solutions to the major problems that I will face during development and I will use their successes and failures to guide my own solutions for these problems. These games are described below along with guidance for player numbers, time and ages - though these are recommendations and each game can be adapted to accommodate a different number of players. These games are perfect for any occasion and good-humoured group of people - due in part to their short, but adaptable, play time.

2.2 Games and Variations

The Variation Game

- 2 - 8 players
- 1 - 2 seconds
- 10+ years old

Objective

Players will be asked to write custom rules for people so that they can play their card games with their favourite variations and house rules. At the beginning of each round, players will draw cards to randomly decide the base set of rules and variations they will have to incorporate. The timer will begin and each player will try to write down the rules as quickly and accurately as they can. Players will score points based on accuracy and completeness.

Game play

Before the game begins players should decide on a set of base card games, variations for each and their rules. Once these have been de-

cided, one deck of cards, with the names of each base game should be made, along with decks for each game's variations. Wikipedia is an excellent source for this.

Once setup is complete the first round can begin. Each round begins with the base game deck being shuffled and the top card being drawn. Next the appropriate variation deck is located and two or three (as decided by group consensus) variations are drawn. The timer begins (1 to 2 seconds is recommended to reduce the website's latency) and each player attempts to write a corresponding set of rules. Once the timer runs out, the players hand in their rules and they are marked.

The marking guidelines are as follows:

- 1 mark for a complete set of rules
- 1 mark for an accurate set of rules for the base game
- 1 mark for accurate rules for each variation
- 1 mark for a fully correct set of rules

In order to maximise your accuracy and speed it is recommended to write rules for each game in advance. You should then separate the rules into smaller components (e.g. bidding in Spades) and then again into individual rules. Then you can create a template to build the full rules from each of these component parts. For each variation (including the default variation) you should write separately the rules that it is composed of. Then when you come to

2.3 Networking

The Communication Game

- 2-8 Players
- Less than 1 second
- 13+ years old

Objective

Players will establish a shared communication format and protocol, which they will use to establish a game state that will be kept consistent amongst all players. Players will make decisions that influence the game state and will notify others of what they have done.

Set up

The game's host is called the server and chooses the games rules and variations, then ensures all players are using the same protocol and version, through a 3 way handshake. They will then inform players of the initial game state, known as game state 0. All game states will be numbered so that players can easily tell if they have missed an update. Once all players have confirmed that they have received and parsed the initial game state, the game can begin.

Server

When playing as the server, it is this player's responsibility to ensure that they keep the authoritative copy of the game state, which all other players will refer to in case of mistakes or desynchronisation. Running the game method, as given by ¡prev game name¿, the server will be told to communicate information to players. When this happens, it will generate a JSON message that describes the changes that have happened and send this to each player.

When the server is told to have a player make a decision, it will send a similar JSON package that describes the decision that the player has to make as well as the options they have available. For example, in Spades, a player may have to play a club to follow the suit led. In this case, the server would check if the player has a club, if they do not they can play any card. Then the server would make a list of every card in the players hand that they can play. The message given to the player would be that they can play any one of these cards. One advantage of this is that if a variation changes which cards can be played, the server can change the list they give the player without the player client needing to be updated. This means that the client instructions can be smaller and simpler.

Client

When playing as a client, these players need to keep their own personal record of the game state the same as the servers. They do this by listening for updates; when they receive an update they apply the changes to their copy of the game state thus keeping it in sync with the server. Since the updates are numbered, clients must also keep track of which is the last update they received. If they notice that they have missed an update, they must send a packet to the server asking them to resend the update they missed. Some groups may also decide to implement a two way handshake where the client confirms that they have received each update.

Ending

The game ends once the server running the game method reaches an end state. This end state will differ based on the game being played, for example, Spades will end once a team has reached the winning points total (usually 500). Once this happens the server will calculate final scores and the scoreboard. This will be shared with each client as the games final update. Players can then choose if they want to save a copy of the scoreboard as a record of the game.

2.4 Front End

The Pretty Pictures Game

- 2 - 8 players (must be even)
- 0.1 - 0.2 seconds per round
- 5+ years old

Objective

Players should first form pairs that will compete against each other. Designed as an asymmetric cooperative game, the pairs of players will work together to communicate as effectively as possible. One player takes the role of the artist and the other the critic. As the artist, this player is given a description of the game state and must create a drawing that represents this. Then the critic will see the drawing, try to understand it and answer a few questions about the game state. The pair score a point for every question answered correctly. Then one critic will have to make a decision to update the game state (e.g. what card to play), and the pairs will play another round.

Game play

As the Artist, your are given a description of the game state (including what cards you have, the current bid, etc.) and you have to create a drawing based upon this. This description will vary based on the game and variations, so you must be able to adapt to this. The format of this message will be decided in the "The Communication Game" so you should ensure that you can read it before it is decided.

Since a website is the most popular format (see Figure 5.2), you will have to be aware of the constraints of a web page. At any point another player can challenge an artist's design; if they cannot describe how they would create the design using HTML, CSS and JavaScript their design is void and they do not score any points for the round. In order to meet the strict time restrictions (to ensure the interface is responsive)

it is advised that you create a template that is filled based on the data give. For example, you may decide to draw the player's hand of cards at the bottom of the screen; in this case you would take the data for the first card the player has and draw the corresponding image in the bottom left, then repeat this for the second offsetting it slightly. It is also recommended that you have drawn each 'sprite' in advance, so you can simply place or render each image in the desired location.

Stylistically artists should stick to a minimalist aesthetic in order to increase clarity and readability. Similarly each game should use a common deck of cards and colour scheme to help create a shared style and identity among the games on the website.

Scoring

Before the game begins questions about the game state are written on cards which are then shuffled before each round. Once the artists have drawn their pictures, four cards are drawn from the top of the deck. Each critic must write down their answers informed only be the image their partner has made. The pair score one point for each question the critic answers correctly. After each round the players update collect each pair's score and update their total.

2.5 Lessons

After designing and playing these games, I came to a number of realisations. Some of these recommendations I have written as advice as suggestions in the game rules. I found that these are almost necessary to level the playing field, since not using these strategies gives a player a significant disadvantage. This has very informative since it shows the general approach that I should take towards solving these problems.

2.5.1 Games and Variations

My initial plan for creating a single card game is to have a scoreboard, which is list of dictionaries that each represent one player (see Figure 2.1). The dictionary would contain each player's name, score, and their rank (e.g. '2' if they are in second place). The game would then be a series of functions that would mutate this list as the game progresses. Once the game has finished, this scoreboard could then be saved or outputted as a record of the game (Figure 2.2).

In order to include different variations I will write multiple versions of each function for each variation. For example, for Spades, I will write the default bidding function (Figure 2.3) as well as a function for the 'sum of bids' variation (see Wikipedia). Then to create a game with a particular variation I will replace the default function with the

respective function for that variation (e.g. replace the default bidding function with the 'sum of bids' bidding function).

In order to increase the number of variations, I will not manually place each of the relevant functions but I will write a 'game method creator' function which will choose the relevant variations automatically (see Figure 2.4). It will take a dictionary as an input, which will show which game and variations have been chosen, and then it will choose the relevant template to use (see Figure 2.5). It will iterate through each stage of the template either choosing the default function or the variation specified, and then it will link each of the functions together. Once it has finished it will have created the game method that uses all the specified variations that the user wanted.

Once the game method has been created, it can be given to the server to run. The server will execute the method, taking inputs from the clients whenever necessary, mutating the scoreboard, and once the method returns, it can output the scoreboard and inform the clients of the result.

This is known as the strategy design pattern and one of its advantages, is that it primarily uses encapsulation to reduce the complexity of code. The problem is heavily decomposed and each function deals with only a single part of the problem. These parts are isolated, clearly defined and encapsulated, which means that each different solution only has to be concerned with the necessary details for its implementation and can be changed at runtime to allow for different behaviours. In order for this to be successful each function should use a standard interface for communication. However, this could reduce the number of variations that can be implemented, or require that every variation is given redundant information that only 1 variation requires. I will also have to decide how much I will decompose each problem. Decomposing the problem into smaller parts will reduce the likelihood that code is repeated between variations and increase the number of potential variations but will increase the complexity of the 'game method creator' function.

Typically, the strategy design pattern is implemented within programs that use Object Oriented Programming methodologies. I wish to implement it within a functional program, but I do not foresee this being any more difficult. I do however wish to use some Object Oriented Programming techniques to further abstract and encapsulate parts of my solution. For example I will create a card class (see Figure 2.6) that contains the minimum necessary attributes. I will also create a player class (see Figure 2.7) that will change at runtime based on what data is necessary for each game and variation. For example, in Spades players are in 2 partnerships and a player may require a reference to their partner. However, I wish to use objects more like structs in C than classes in Java, by containing attributes but not methods. In my opinion this better matches the functional approach that I have used

for the game method rather than having two competing methodologies.

The host will have to send and receive packets to and from clients, which cannot be done using a web browser. IP addresses on the internet are distributed to routers not computers. This means that an individual computer cannot be identified by its (external) IP address since this points to a router rather than the computer on that LAN. This problem can be avoided by installing the program or browser extensions. However this will raise the barrier to entry since users will have to install software. Alternatively, the server can host the game and run the game method which will increase the load on the server but will increase ease of use. In prioritising a low barrier to entry, I will have the server be the host and run the game method.

Since the game method will be ran on the server, I will have to write it in the same language as the rest of the server. The game method creator will also have to be written in the same language as the game method that it is creating. I am most familiar with Python and Flask, having used it before for a web server with basic SQL queries, so that is the language and library I will use for this program.

```
scoreboard = [
    { 'name': string, 'score': int, 'rank': int }
]
```

Figure 2.1: Pseudo code and data types for the scoreboard

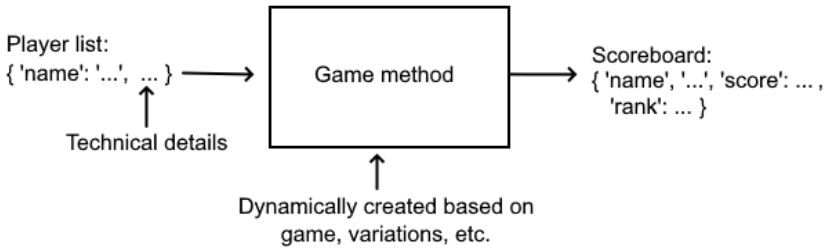


Figure 2.2: An abstract view of the game method

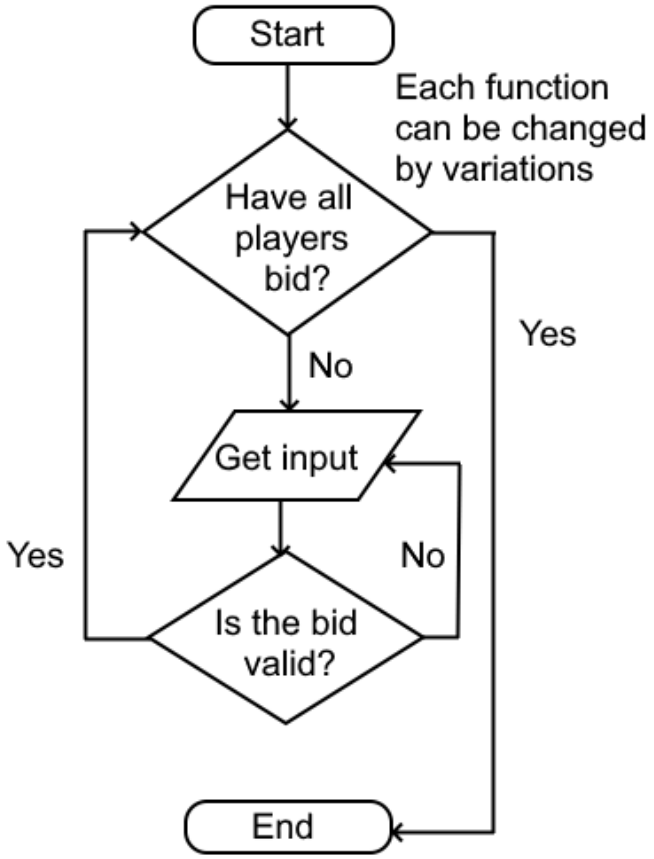


Figure 2.3: The default bid method

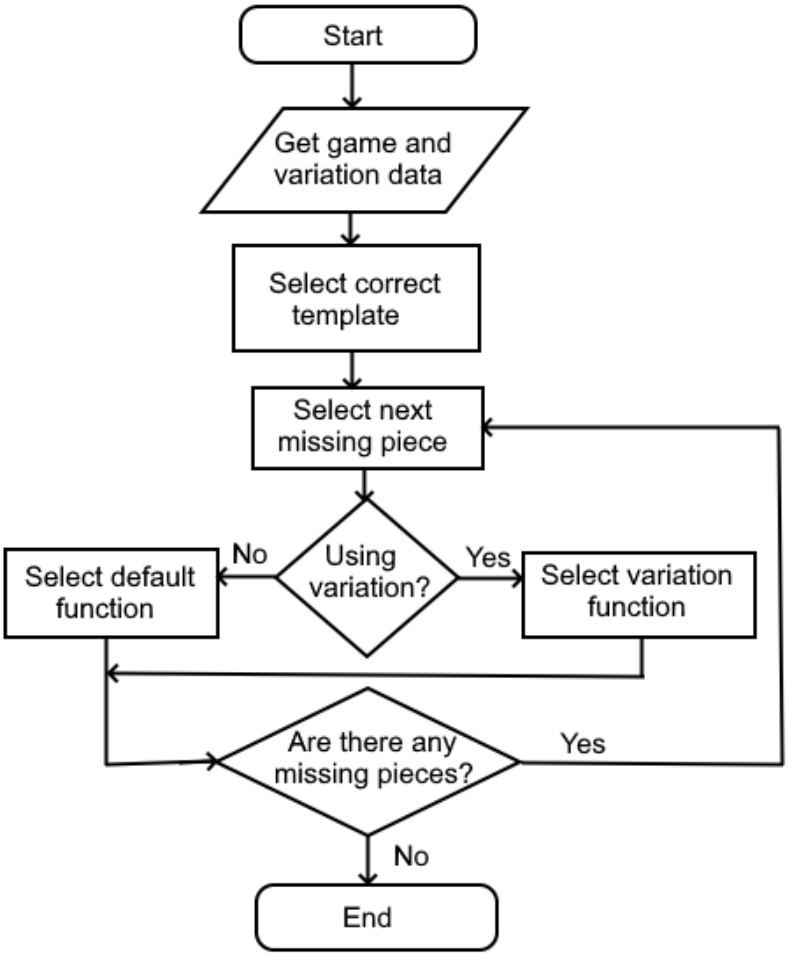


Figure 2.4: A flowchart for the game method creator function

```
game_start:
    create_scoreboard -> ()
rubber_start:
    pick_partners -> ()
round_start:
    pick_trumps -> ()
    deal_cards -> ()
playing_trick:
    get_valid_cards -> ()
    determine_winner -> ()
round_end:
    determine_winner -> ()
rubber_end:
    determine_winner -> ()
game_end:
    output_scoreboard -> ()
```

Figure 2.5: Pseudo code for a whist game method template

```
Card:
    rank: int
    suit: enum
    name: string
    image: image path
```

Figure 2.6: Pseudo code and data types for the card class

```
Player:
    name: string
    score: int
    rank: int
    ... # Other attrs vary
    # by game & variation
```

Figure 2.7: Pseudo code and data types for the player class

2.5.2 Networking

One of the main problems is that each client must have a separate copy of the game state which must all be kept in sync. In order to do this I will have the server keep an authoritative copy of the game state that can be referred to in case of any dispute.

To make hacking and cheating more difficult each player will only be given the information about the game state that a real player would know. This would likely include their own hand, the bids other players have placed, and any cards that are in the middle - though this would vary depending on the game and variations.

When the game begins, each player needs to agree on an initial game state, which will be dictated by the server. Thus the first message sent will establish what each client needs to know about the initial game state. For this to be possible the game state must be abstracted and serialised into a JSON file. I am using JSON because it is JavaScript which is well supported by web browsers. This process will vary based

on the game, but I have made an example for Whist (see Figure 2.8). After this each time the game state changes, the server will send an updated version of the game state to each client.

Each packet would need to include some metadata, such as its version, the game being played (and variations), the game code and potentially some networking information, such as the sender's IP address. It should also contain information as to its purpose, such as establishing an initial game state, testing for a connection, or an updated game state. Each time the game state is updated, it will be given an update number that will increment with each change. This will allow clients to ensure that they have not missed an update.

```
{
    "game": "whist",
    "variations": {
        "solo": false, "bid": false,
    },
    "updateNo": 13,
    "players": [
        {
            "id": 0,
            "name": "Mark",
            "partnerID": 3,
            "score": 3,
            "rank": 1,
            "tricksWon": 4,
            "hand": ["AS", "2S"]
        },
        {
            "id": 1,
            "name": "Luke",
            "partnerID": 2,
            "score": 2,
            "rank": 2,
            "tricksWon": 2,
            "hand": ["3H", "7D"]
        },
        {
            "id": 2,
            "name": "John",
            "partnerID": 1,
            "score": 2,
            "rank": 2,
            "tricksWon": 5,
            "hand": ["8C", "5H"]
        },
        {
            "id": 3,
            "name": "Judas",
            "partnerID": 0,
            "score": 3,
            "rank": 1,
            "tricksWon": 0,
            "hand": ["6D", "10C"]
        }
    ]
}
```


Figure 2.8: Pseudo code for a whist game state

2.5.3 Front End

In general, I want the screen to have a central ellipse that represents a table. Players will be seated around the table and represented as circles. They may also have profile pictures or initials to help recognise them. Each player will also have an information box that is draw next to their icon, which contains further information. This would include their username and other information important for the game. For example, when playing Whist, the information box would include the number of tricks that each player has won and Spades would also show each player's bid. A central information box would show information that is general to the whole game. It would be positioned on the table, above the logo, which may be removed if the box is too large. When playing poker, the value of the pot would be shown here, or the trump suit during Whist or Spades.

Images necessary for each game will be stored on the server where they can be fetched when needed. This will include pictures for playing cards, logos and other icons. The images will be stored as SVG files so they can be resized without distortion. The front end will have to take a JSON serialisation of the game state and will produce a HTML page to visually represent this. This will be done using JavaScript since the content can change but only along predefined rules. Depending on the variation

I am using HTML since it is common within the modern world and so I believe most people are familiar with it. In my stakeholder survey, stakeholders also stated it was their preferred interface (see Figure 5.2. They also said that a clear user interface was most important to them (see Figure 5.3), so I that will be the focus of my design. In order to achieve this I will try to ensure that most choices have a common interface. For example, most choices in Poker can use the same interface, where each choice is a separate button the player can click (see Figure 2.11). Whist and Spades can share the interface for choosing a card when playing a trick (see Figure 2.12). When a player has to make a choice, their computer will receive a request from the server to make that decision. The server will inform the player of the type of decision, what the options are and if any additional information is needed (see Figure 2.13). All this will inform the front end how it should render the choice.

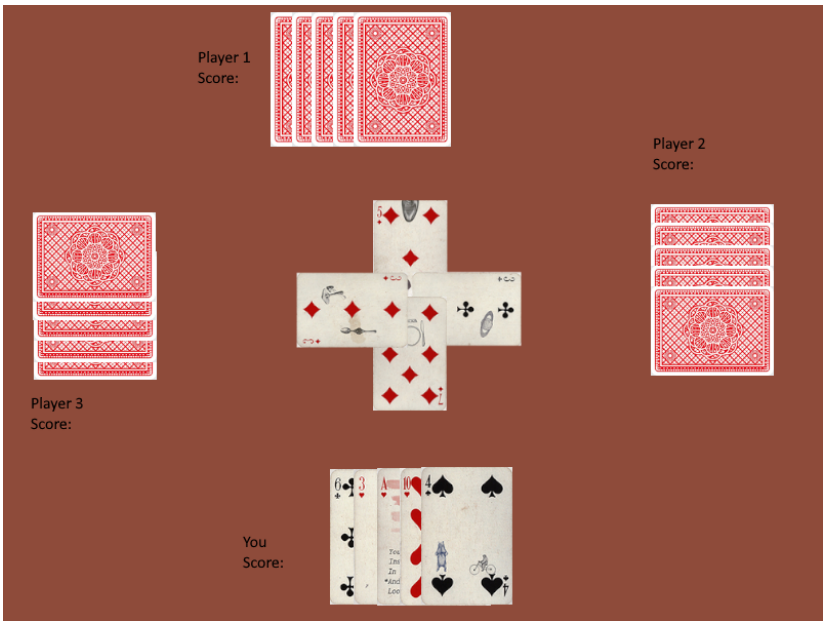


Figure 2.9: Example user interface for whist or spades

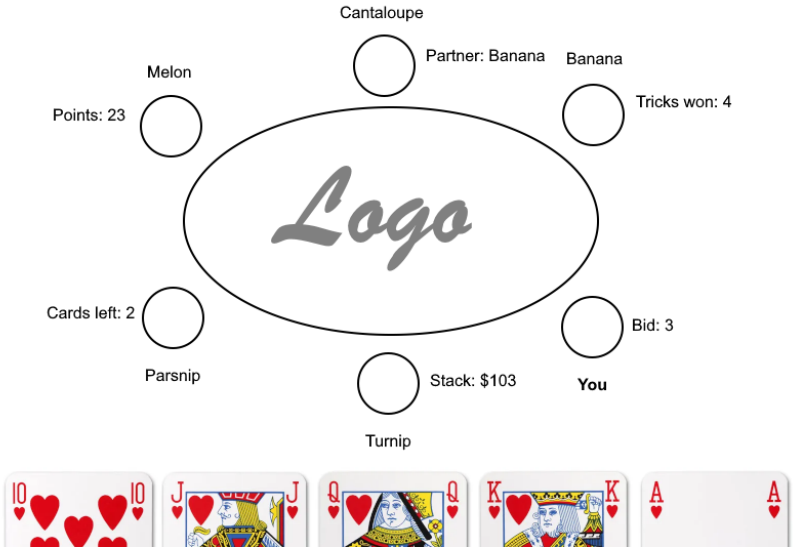


Figure 2.10: A revised, more general user interface

Figure 2.11: Example user interface for making a bet in Poker

Figure 2.12: Example user interface for player a trick in Whist or Spades

```
{
    "game": "whist",
    "variations": {...},
    "updateNo": 7,
    "updateType": "choice",
    "choiceType": "pick_card",
    "options": {
        "2H", "7H", "8H", "JH"
    }
}

```

Figure 2.13: Pseudo code for a choice in Whist

2.5.4 Algorithms

Within Computer Science there are many algorithms that have been developed and standardised to solve particular problems. Such algorithms are widely known, tested, optimised and their order is also well known. These include sorting and graph algorithms and many such algorithms come from maths and discrete mathematics in particular. However, my project has little relation with such fields of study and so very few algorithms are applicable to my program. Instead for most of the algorithms that I will use, I will have to design and write bespoke versions for my program.

Big O notation is a mathematical notation used to show how the time taken for the algorithm changes as the input increases. It can be used to decide which algorithms are more efficient and should be used for a particular purpose. Bubble sort and quick sort are both sorting algorithms, with Bubble sort having quadratic or n^2 order and quick sort with $n \log n$ order. This means that quick sort is a more efficient algorithm and will likely be quicker, particularly for large inputs which means values of n.

When shuffling a deck of cards, I could use the modern version of the Fisher-Yates shuffle, which has linear order n. Since a deck of cards always has 52 cards the order will actually be constant. Most of the algorithms used when playing the game will take the number of players as an input. Even if these algorithms have very poor order, n will always be small (very rarely > 10) and so this will not have a large impact on the speed of the algorithm. For this reason, I believe that I

should focus on optimising the networking code since I believe this will have a larger impact.

Key variables used in the program will include the scoreboard, the list of players and the deck. The purpose of the game method is to mutate the scoreboard to show the results of the game so the game method needs to update the scoreboard at the relevant times (for example after each hand in Poker). The structure of many card games includes each player having a turn, when they can act. This can be emulated by iterating through the player list and calling the same function with each player. This means that the list of players will be used often and changing it will have a big impact on the game (Poker strategies with 2 players and 9 are very different). A deck of cards is required for most card games and the program needs to ensure that it always references the same list otherwise multiple players may be dealt the same card.

2.5.5 Testing

Testing could be done using an automated test system, as in common in the industry, but this would increase the complexity of the project and I would rather spend more time on other parts of the program. Instead, I shall manually test each feature when they are added. Once the project is complete I shall organise a group test to find any other issues or bugs that occur. Once these issues have been fixed, I can release the project.

For the main testing of features, this will happen immediately after they have been developed. This is to increase the efficiency of fixing bugs, since I will still be familiar with the code, and its potential flaws, when I discover the bugs. Depending on the feature, different test data will be used. When receiving user input it must be validated, so these functions will be checked with valid, invalid and boundary data. For the networking code, this will be tested by checking that users can consistently send messages to each other, ensuring that the integrity of the message remains after it is sent and that the code is not excessively slow. However, the serialisation of the game state is more complex. I will test this by creating a series of test game states and manually checking the serialisation to ensure that all data is properly recorded. This may be slow, but I anticipate that it will be faster than developing an automated testing system for this. Similarly the front end will read the serialisation of the game state and translate it into graphics, which I will manually cross reference with the original game state.

Initially, all internal data will be considered valid, so for example the server will not check current game state before it is serialised and sent to the clients and the clients will not check it when they receive it. This will increase the speed of development, since validation and exception handling will not need to be coded. However this will also increase the likelihood of a crash. The justification I have for this is

that data corruption or bugs should be uncommon and fixed before the program is released and if data is corrupted, I believe it will be very difficult to recover it.

2.5.6 Further development

After development further improvements could be made by adding a chat function to allow users to send messages, public games that anyone can join, or by including more games and variations. Adding a chat function would have added complications, particularly in combination with public games, in that profane language may have to be censored and users could harass others. It would also have to be handled differently be the networking code since, when a users sends a message they want it to be received as soon as possible. Users can send messages at any time so these should be considered completely separate from game state updates. Thus they would have their own numbering system and format.

Public games would be relatively easy to add, since even for private games, a database of all games must exist. Making a game public would then mean that the corresponding attribute is set to true, and false for a private game. Then a browse page could be added where it queries every public game and renders the information. A user could then browse this and then choose a game to join.

The cost of adding more games or variations would mostly depend on how similar they are to previous ones that I have written. If the game shares many elements with existing ones, much of the code and infrastructure can be reused. For example Spades can reuse most of the code for playing and taking tricks from Whist and the rules for scoring are similar in both games. However, Poker shares very few similarities with games outside of it's own family of games. This means that new infrastructure will have to be built for specifically for it, significantly increasing the time and cost. This will likely be the case for most types or families of games.

III

Implementation

"Good artists borrow, great artists steal."

- Pablo Picasso

My implementation was written over several months, during which I tracked the development using `git` and wrote a devlog where I discussed the issues that I faced and how I planned to fix them. Earlier in development I didn't have a plan or task list which caused development to be rather unstructured. This culminated in a large issue when I attempted to develop a feature before realising that I hadn't the structure in place for it. I attempted to develop both the feature and structure in parallel, but this created 'spaghetti code' and I had to revert my changes before I could reimplement the feature. After this I made a 'todo' list to better plan and track development.

My focus during development was creating as much functionality as possible sacrificing error handling and robustness. This means that the program likely has bugs that I have not found or fixed and this is its largest flaw currently. However, I have been able to develop the structure for all the features that I wanted to implement. Whilst I was not able to complete the project, adding all of the games and variations I had hoped for, this would be a relatively simple task, since I have created all necessary structure for them.

3.1 Overview

This gives a general overview of the order in which I tackled the programming project. This details key stages of the development process and omits many details and smaller decisions made during development. I have 9 versions each of which represent key milestones in development. They are not complete rewrites or replacements (as is typical

with software versioning) but represent adding important new features. A version also does not represent the final state of the pictured code or file, and many are quite different in the final version.

A more complete history can be found at github.com, which has the current state of the code, and all previous commits, which show the full state of the code at a previous point in time.

3.2 Version 1

Initially I created the web server that would serve the supplementary webpages, which includes the homepage, rules, etc. that are not used when playing a game. I experimented with using a few languages for the project including PHP and Javascript, before settling on Python. I chose to use Python for 2 reasons: I am familiar with the language and have used it in previous projects and second Flask is a simple framework that I found easier to understand than larger MVC frameworks.

```python
from flask import Flask, render_template, request

def create_app():
    # Gives current path to flask
    app = Flask(__name__)

    @app.get("/")
    def index():
        return render_template("index.html")

    @app.get("/rules/")
    def rules():
        return render_template("rules.html")

    @app.get("/rules/whist")
    def rules_whist():
        return render_template("rules_whist.html")

    @app.get("/games/new")
    def games_new_get():
        return render_template("games_new.html")

    @app.post("/games/new")
    def games_new_post():
        game = request.form["game"]
        return render_template("games_new_post.html", game=game)

    @app.get("/games/join/")
    def games_join_get():
        return render_template("games_join.html")

    return app
```

Figure 3.1: A basic server written in Python, using Flask

This screenshot shows the routes that I have defined for the website. Most of these send HTML files (see Figures 5.34 to 5.38), which is necessary since this is what a browser expects and can render. These routes are almost identical except for `games_new_post` which is for a post request and accesses the form to add data to the HTML document.

Once I had finished this I moved onto creating an interactive page. In my design, I planned to use websockets but I was not familiar with these and decided that creating a chatroom would be an easier introduction than a card game.

Figure 3.2: A HTML form for customizing a game and picking variations

Above is the `games/new` page where users can create a new game. They can fill out a form which changes some settings and variations such as the scoring and length of the game. The number of players cannot be changed since Whist must be played with 4 players. This data is sent to the server in the form of a POST request which the server can then use to customise the rules and variations used when running the game. It does not have any styling since I had not created any CSS at this point - though this is something that I later add.

3.3 Version 2

```python

import websockets, asyncio, json

import games
from chatroom import handle_chatroom

async def handler(websocket):
    async for eventJSON in websocket:
        print(eventJSON)
        event = json.loads(eventJSON)
        match event["type"]:
            case "create":
                await create(websocket, event["gameID"])
            case "join":
                await join(websocket, event["gameID"])
            case _:
                config = json.loads(event["config"])
                match config["game"]:
                    case "chatroom":
                        await handle_chatroom(websocket, eventJSON)

async def create(websocket, gameID):
    games.GAMES[gameID] = set()
    print(f"Created Game {gameID}")

async def join(websocket, gameID_str):
    gameID = int(gameID_str)
    try:
        connected = games.GAMES[gameID]
        games.GAMES[gameID] = connected | { websocket }
        print(f"New player joined game {gameID}")
    except KeyError:
        print(f"Error could not find {gameID}")
        event = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(event))

async def main():
    async with websockets.serve(handler, "", 8001):
        # Wait for a promise that will never be fulfilled - run forever
        await asyncio.Future()

if __name__ == "__main__":
    asyncio.run(main())
```

Figure 3.3: A websocket server written in Python

This server must be ran alongside the Flask server and it handles all websocket connections. To create a new game, a user will visit the website and complete a HTML form deciding what game and variations they would like to play. When this is submitted, the Flask server will send a `"create"` message to the websocket server to create a new game. All events are handled by the `handler` function with `"create"` events

being handled by the `create` function. The user will then be redirected to another page to play the game (see Figure 5.42). Here Javascript code will be ran that will connect to the websocket server and send a `"join"` event (see Figures 5.43 to 5.45). These are the only responsibilities of the main `ws_server.py` file; handling of events for each game are in separate files.

```python
#!/usr/bin/env python

import json

import games

async def handle_chatroom(websocket, eventJSON):
    event = json.loads(eventJSON)
    match event["type"]:
        case "message":
            await send_message(websocket, eventJSON)
        case _:
            await error(websocket, eventJSON)

async def send_message(websocket, eventJSON) -> None:
    gameID = int(json.loads(eventJSON)["gameID"])
    try:
        connected = games.GAMES[gameID]
        for websocket in connected:
            await websocket.send(eventJSON)
    except KeyError:
        print(f"Error could not find {gameID}")
        event = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(event))
    return

async def error(websocket, eventJSON) -> None:
    print("Error handling event")
    return
```

Figure 3.4: Code that handles chatroom-specific events

Above is the handler for the chatroom, which handles events specific to this type of game. In this case it is only a single event type `"message"` so this example is quite simple. When a message is received it is rebroadcast to all connected users.

3.4 Version 3

```
12 async def handler(websocket):
13     async for eventJSON in websocket:
14         print(eventJSON)
15         event = json.loads(eventJSON)
16         match event["type"]:
17             case "create":
18                 await create(websocket, event)
19             case "join":
20                 await join(websocket, event)
21 
22         # Ensure the same DB connection is used rather than creating a new one
23         with server.app.app_context():
24             gameID = int(event["gameID"])
25             cursor = database.get_db().cursor()
26             result = cursor.execute(
27                 "SELECT config FROM games WHERE gameID = ?",
28                 [gameID]
29             ).fetchone()
30             config = json.loads(result["config"])
31         game_handler = get_game_handler(config["game"])
32         await game_handler(websocket, event)
33 
34 def get_game_handler(game_type):
35     async def error(*_):
36         print(f"Error could not find handler for '{game_type}'")
37 
38     map = {
39         "chatroom": handle_chatroom,
40         "whist": handle_whist,
41     }
42     try:
43         return map[game_type]
44     except KeyError:
45         return error
46 
```

Figure 3.5: Refactored `ws_server.py` that can handle both chatroom and Whist games

Next I began work on creating a card game. I chose Whist since it is a simple, historic game and the 'father' of many other games such as Spades and Bridge. I tried to reuse as much of the code between Whist and the chatroom as possible. This required that I rewrite some parts of the code and make some changes to the design of the system. Once example is that previously `ws_server.py` only would handle creating a game and users joining and would have other all other events handled only by individual 'handlers'. I have changed this so that each 'handler' has is given every event for that game. This allows them to run 'set-up' code when the game is created and users join, which is necessary particularly for more complex card games. However one benefit of creating a shared interface is that I can use these new events to improve the chatroom; all messages are stored so that when a new user joins they can see messages sent before they had joined.

3.5 Version 4

```python
import json

import games

WHIST = {}

async def handle_whist(websocket, event):
    match event["type"]:
        case "create":
            pass
        case "join":
            await join(websocket, event)
        case _:
            await error(websocket, event)

async def join(websocket, event):
    gameID = int(event["gameID"])
    # 2 players should not be able to join simultaneously so this should work
    if len(games.get_userIDs(gameID)) != 4:
        await waiting(websocket, event)
        return
    await test_game_state(websocket, event)

async def waiting(websocket, event):
    gameID = int(event["gameID"])
    try:
        connected = games.get_websockets(gameID)
        response = {
            "type": "waiting",
            "players": len(connected),
            "players_required": 4
        }
        responseJSON = json.dumps(response)
        for websocket in connected:
            await websocket.send(responseJSON)
    except KeyError:
        print(f"Error could not find {gameID}")
        response = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(response))

```

Figure 3.6: A basic handler for Whist games

For the Whist handler above I implemented features incrementally in the order that a user would see them. This meant that I started with a 'waiting' screen when users were waiting for their friends. After this I added a test state that gradually grew more complete as I added more features to the renderer. I developed the handler and renderer in parallel since both of them influence the design of each other. For example one decision that I had to make was whether I would send each user the entire game state and have the client display only what the player should see or if the server should send each player only what they can see. In each case the code for both the client and server would be

very different. Developing in parallel allowed me to better understand these decisions as I could view them from both sides.

```
response = {
    "type": "game_state",
    "players": [
        {
            "username": "matti",
            "bid": 4,
            "tricks_won": 2,
            "hand": [
                "", "", "", "",
            ],
        },
        {
            "username": "test1",
            "bid": 2,
            "tricks_won": 0,
            "hand": [
                "6H", "JD", "7H", "3S",
            ],
        },
        {
            "username": "test2",
            "bid": 5,
            "tricks_won": 3,
            "hand": [
                "KC", "2C", "8S", "TS",
            ],
        },
        {
            "username": "private",
            "bid": 2,
            "tricks_won": 1,
            "hand": [
                "8D", "9S", "QD", "5C",
            ],
        }
    ],
    "community_cards": [
        "AC", "3C", "KD",
    ],
    "trump_suit": "H",
}
```

Figure 3.7: A complete test game state, including community cards that are used in poker

Above is an example game state that I used to test and develop the renderer. The structure of which is that information relevant to all players is part of the top-level game state, such as the `"trump_suit"` and `"community"` cards. Then `"players"` is a list of each of the players with data specifc to them. In this case that would be their `"username"`, `"bid"`, etc.

```
118
119 function renderHands(event) {
120   const content = document.getElementById("content");
121   const linebreak = document.createElement("div");
122   linebreak.className += "flex_linebreak";
123   linebreak.style.marginBottom = "1em";
124   content.appendChild(linebreak);
125   for (let i in event.players) {
126     player = event.players[i];
127     const playerDiv = document.getElementById("player" + i)
128     const handWrapper = playerDiv.querySelector(".hand");
129     const linebreak = document.createElement("div");
130     linebreak.className += "flex_linebreak";
131     playerDiv.insertBefore(linebreak, handWrapper);
132     for (let card of player.hand) {
133       // Queries the server for the corresponding svg file
134       handWrapper.appendChild(getCardHTML(card));
135     }
136   }
137 }
138
139 function renderCommunityCards(event) {
140   const communityCards = document.querySelector(".community_cards");
141   for (let card of event.community_cards) {
142     communityCards.appendChild(getCardHTML(card));
143   }
144 }
145
146 function renderGameState(event) {
147   renderTable(event);
148   renderTableInfoBox(event);
149   renderPlayerInfoBoxes(event);
150   renderHands(event);
151   renderCommunityCards(event);
152 }
153
154 function renderWaiting(event) {
155   renderTable(event);
156   renderTableInfoBox(event);
157   renderPlayerInfoBoxes(event);
158 }
159
```

Figure 3.8: Javascript code to render a game state

When the client receives a `"game_state"` client-side Javascript is ran to generate HTML to render the view. This includes rendering the table and players (see Figure 5.55), info boxes showing the trump suit,

each player's bid, tricks won, etc. (see Figures 5.56 and 5.57) as well as the player's hands and any community cards (see above).

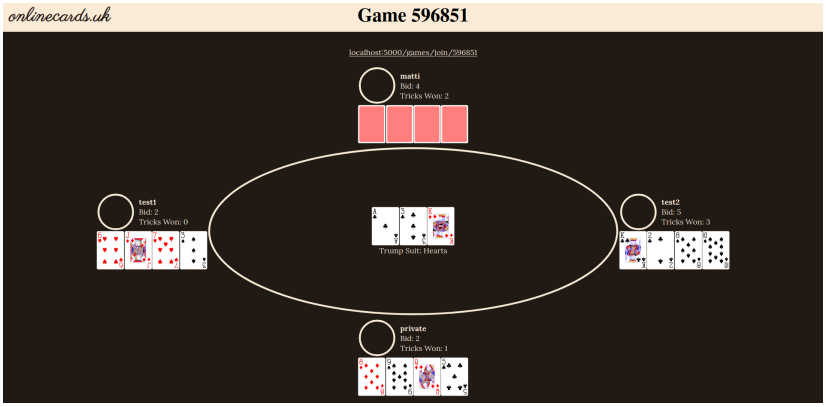


Figure 3.9: The test state rendered by the browser

Above is the game state as rendered by the browser. Users can be seen represented by a circle with their name, and other information in an 'info box' next to this. Below this is a player's hand which can be either face up, as with the bottom 3 players, or face down, as with the top player. In the middle the table has a row of 'community cards' that could be used either in Poker or other games to show cards that have been played onto the table.

3.6 Version 5

```
 97
 98 def create_deck_default(shuffle=True) -> list[str]:
 99     deck: list[str] = []
100     for suit in ["C", "D", "H", "S"]:
101         for i in range(1, 14):
102             if i in range(2, 10):
103                 card = str(i) + suit
104                 deck.append(card)
105                 continue
106             match(i):
107                 case 1:
108                     card = "A" + suit
109                 case 10:
110                     card = "T" + suit
111                 case 10:
112                     card = "T" + suit
113                 case 11:
114                     card = "J" + suit
115                 case 12:
116                     card = "Q" + suit
117                 case 13:
118                     card = "K" + suit
119                 case _:
120                     raise ValueError
121             deck.append(card)
122     if shuffle:
123         random.shuffle(deck)
124     return deck
125
126 def set_partners_default(players: list[dict[str, Any]]) -> None:
127     random.shuffle(players)
128     partners = [2, 3, 0, 1]
129     for i, player in enumerate(players):
130         player["partner"] = partners[i]
131
132 def deal_hand_default(game_state: dict[str, Any]) -> None:
133     deck = game_state["deck"]
134     players = game_state["players"]
135     while len(deck) != 0:
136         for player in players:
137             try:
138                 player["hand"].append(deck.pop())
139             except KeyError:
140                 player["hand"] = [ deck.pop() ]
141
```

Figure 3.10: An initial game state is created and broadcast

```python

def initialize_default(game_state: dict[str, Any]) -> None:
    game_state["deck"] = create_deck_default()
    set_partners_default(game_state["players"])
    deal_hand_default(game_state)

async def func_default(gameID: int, game_state: dict[str, Any]) -> None:
    initialize_default(game_state)
    await broadcast_game_state(gameID, game_state)
    print(game_state)

def get_whist_func() -> Callable:
    return func_default
```

Figure 3.11: An initial game state is created and broadcast

Once the renderer was complete I began work on the back end and so that users could play a real game. Rather than sending a dummy game state it needed to create, track and update a real game state as the game progresses. Above is an early version that initializes the game state and broadcasts it to all connected players. My plan was that the program would update the game state as far as it could without user input. Then it would request and wait for it (asynchronously so that other games could run). Once it had received the response it would then repeat this loop.

3.7 Version 6

```python

def create_main_func(
        censor: dict[str, Callable],
        add_funcs: list[Callable]
    ) -> Callable:

    def censor_main(game_state: dict[str, Any], userID: int) -> str:
        censored_state: dict[str, Any] = {}
        key: str
        for key in game_state:
            if key not in censor:
                # Copy the key/value pair into the `censored_state`
                default(censored_state, game_state, key, userID)
                continue
            # Otherwise mutate the `censored_state` by calling a replacement / variation function
            func = censor[key]
            func(censored_state, game_state, key, userID)
        for func in add_funcs:
            func(censored_state, game_state, userID)
        return json.dumps(censored_state)

    return censor_main

```

Figure 3.12: Python code to censor a game state for a user with support for variations

```python

def get_whist_censor_func(variation: str) -> Callable:
    match variation:
        case "first_trick":
            # Show the dealers last card which decides the trump suit
            censor: dict[str, Callable] = {
                "hand": censor_hands_first_trick,
                "userID": censor_userIDs,
            }
            add_funcs: list[Callable] = []
            censor_players = create_players_func(censor, add_funcs)
            censor: dict[str, Callable] = {
                "players": censor_players,
                "func": ignore,
                "deck": ignore,
            }
            add_funcs: list[Callable] = [set_event_type("game_state")]
            return create_main_func(censor, add_funcs)
        case _:
            censor: dict[str, Callable] = {
                "hand": censor_hands,
                "userID": censor_userIDs,
            }
            add_funcs: list[Callable] = []
            censor_players = create_players_func(censor, add_funcs)
            censor: dict[str, Callable] = {
                "players": censor_players,
                "func": ignore,
                "deck": ignore,
            }
            add_funcs: list[Callable] = [set_event_type("game_state")]
            return create_main_func(censor, add_funcs)
```

Figure 3.13: Python code to censor a game state for a user with support for variations

One of the key issues that I faced was that each player should see only part of the game state and this was different for each player. Whilst the server would track every player's hand each player should only see their own. What made this problem more difficult is that in Whist the dealer deals their final card face up to set the trump suit and picks it up on their turn. This means that different rules have to be used to determine what players can see based on the state of the game. Other variations and house rules may also change this, so a flexible system is required to support these. During development I designed and implemented 4 different versions of this system; part of the final version is above (see Figures 5.62 to 5.66).

My final design has a single interface that other code should use `get_whist_censor_func` which returns a function that will censor a game state. If given a variation this will modify the behaviour of the function returned. This function works by having `game_state` and `userID` as parameters and creating a blank 'copy' of the game state called the `censored_state`. Then it iterates through each key / value pair in the `game_state` state either copying it into the `censored_state`,

ignoring it or 'censoring' it. Censoring is done by calling a function to censor that attribute. For example `censor_userIDs` takes each player's `"userID"`, uses it to get their `"username"` and copies this into the `censored_state`. This means that the user can now see their friend's usernames whilst the server can use unique `userID`s to identify them. If any additional data needs to be added to the `censored_state` that is not present in the `game_state` (for example the `event_type`) a list of `add_funcs` can be given. These are functions which are ran after the `censored_state` has been constructed and can add or modify its data.

The reason that I chose this design is because it is easy to change the behaviour of one part of the system by changing the function assigned each key. For example above the `"first_trick"` variations and the default are identical apart from how they censor a player's `"hand"`. This gives me fine control over how each variation differs and allows me to reuse as much code as possible (both variations censor a player's `"userID"` with the same function, for example).

Additionally this functioned as a proof of concept for using a similar system to change the behaviour of a game by replacing functions with a variation. This would allow me to support many different variations and house rules by replacing only the code that needs to change and reusing other parts. This could also allow for having multiple variations simultaneously, as long as they don't override the same code.

3.8 Version 7

```
215 
216 def get_whist_state_handler() -> Callable:
217     async def state_handler_default(
218         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
219     ) -> None:
220         # Call setup and teardown functions after each event
221         match event["type"]:
222             case "waiting":
223                 return
224             case "start":
225                 play_trick_default(gameID, game_state)
226                 await ask_card_default(gameID, game_state)
227                 return
228             case "choice":
229                 match event["choice"]["type"]:
230                     case "play_card":
231                         check_trick_default(gameID, game_state, event)
232                         await broadcast_game_state(gameID, game_state)
233                         await ask_card_default(gameID, game_state)
234                     case _:
235                         return
236             case _:
237                 return
238 
239     return state_handler_default
240 
241 def get_whist_event_handler() -> Callable:
242     def error(event):
243         return lambda *_: print(f"Error could find handler for event {event}")
244 
245     def func_default(event: str) -> Callable:
246         try:
247             return events[event]
248         except KeyError:
249             return error(event)
250 
251     events: dict[str, Callable] = {
252         "start": initialize_default,
253         "waiting": waiting_default,
254         "choice": choice_default,
255     }
256 
257     return func_default
```

Figure 3.14: An 'event-based' system for playing card games

During development I discovered that my initial design for getting user inputs (as detailed above) would not be possible. This was because I had designed the web socket server so that all data would be sent to the main `ws_server` and would then 'trickle down' to the individual handlers for each `game_type`. However with the design that I described the user input would be sent straight to the handler that was waiting for it and to the main `ws_server`, but it is not possible to have both. I was also unsure if it was possible to have the handler wait asynchronously or if it would block execution of other code, meaning that only one game could run at a time. For this reason I chose to redesign the user input system. Now each card game would run the same way as a chatroom –

waiting for user input before processing it, broadcasting the new state, and then waiting again.

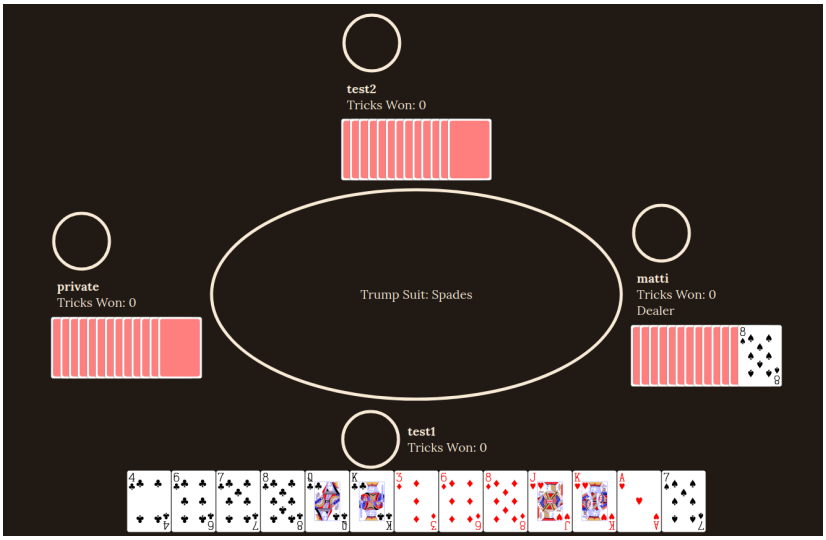


Figure 3.15: Users playing a trick in Whist

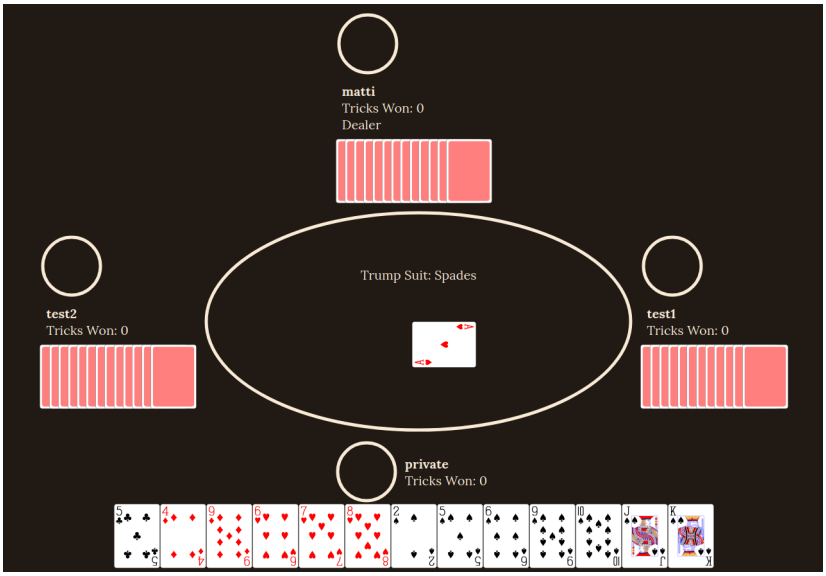


Figure 3.16: Users playing a trick in Whist

Above are screenshots of users playing a trick in Whist. The first screenshot shows the user `test1` after they have been sent a `"choice"`

event. And are deciding which card they would like to play. This is signalled to the player by spreading their hand of cards and a users can play any card by clicking it. The second screenshot shows the game after a new game state has been broadcast – showing `test1` having played the card they chose. The next player `private` is then sent a `"choice"` event where they have to decide which card they want to play.

Internally, the event-based system operates as follows: once the final player has joined a `"start"` event is broadcast after which the server does the initial set up such as picking partners, shuffling the deck and dealing cards. Then the first `"choice"` event is sent to the first player and the server then waits for a response. Once the user has chosen their card a response is sent to the server. When received: it is processed, the `game_state` is updated, rebroadcast and the next `"choice"` event is sent to the next player. How events such as a trick, hand or game ending are handled have not yet been coded. These are added and refined in later versions.

3.9 Version 8

```
304
305 def get_whist_state_handler() -> Callable:
306     async def state_handler_default(
307         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
308     ) -> None:
309         # Call setup and teardown functions after each event
310         match event["type"]:
311             case "waiting":
312                 return
313
314             case "start":
315                 play_trick_default(gameID, game_state)
316                 await ask_card_default(gameID, game_state)
317
318             case "choice":
319                 match event["choice"]["type"]:
320                     case "play_card":
321                         end = check_trick_end_default(gameID, game_state, event)
322                         await broadcast_game_state(gameID, game_state)
323                         if not end:
324                             await ask_card_default(gameID, game_state)
325                         else:
326                             event["type"] = "end_trick"
327                             print('event["type"]', event["type"])
328                             await state_handler_default(gameID, game_state, event)
329
330                     case _:
331                         return
```

Figure 3.17: A completed `state_handler` for Whist

```
332 
333             case "end_trick":
334                 end = check_hand_end_default(gameID, game_state)
335                 if not end:
336                     play_trick_default(gameID, game_state)
337                     await broadcast_game_state(gameID, game_state)
338                     await ask_card_default(gameID, game_state)
339                 else:
340                     event["type"] = "end_hand"
341                     print('event["type"]', event["type"])
342                     await state_handler_default(gameID, game_state, event)
343 
344             case "end_hand":
345                 update_points_default(gameID, game_state)
346                 await broadcast_scoreboard(gameID, game_state)
347                 end = check_game_end_default(gameID, game_state)
348                 # TODO remove logging once fully tested
349                 if not end:
350                     print("game not done")
351                     initialize_hand_default(gameID, game_state)
352                     play_trick_default(gameID, game_state)
353                     await broadcast_game_state(gameID, game_state)
354                     print("trick initialised")
355                     await ask_card_default(gameID, game_state)
356                     print("asked card")
357                 else:
358                     event["type"] = "end_game"
359                     print('event["type"]', event["type"])
360                     await state_handler_default(gameID, game_state, event)
361 
362             case "end_game":
363                 print('event["type"]', event["type"])
364                 await broadcast_scoreboard(
365                     gameID, game_state, "game_end"
366                 )
367 
368             case _:
369                 return
370 
371     return state_handler_default
372 
```

Figure 3.18: A completed `state_handler` for Whist

Redesigning the system did slow development as I had to approach the problem from a different perspective. Rather than having a list of players, iterating through it and asking each player which card they would like to play – I had to ask the first player what card they would like to play, and then each time I got a response: check if all players had played a card; if not, determine the next player and ask them to play a card. Above is a complete version of the state handler for Whist, where I believe the 'event-based' approach can be clearly seen. Each event is named and has corresponding actions taken including calling the `state_handler` recursively if another event has occurred. For example, if the last card in a trick has been played the `"end_trick"` event is processed, but if the players are also out of cards the `"end_hand"` event is raised and processed as well. This recursive design allows for good separation of concerns since the `"end_trick"` code does not have to

know or worry about how the `"end_hand"` code will process the end of hand. These can be completely separated and encapsulated.

3.10 Version 9

```python

def get_whist_state_handler(config: dict[str, Any]) -> Callable:
    async def default(
            *_
        ) -> None:
        return

    async def start(
            gameID, game_state, event
        ) -> None:
        play_trick_default(gameID, game_state)
        await ask_card_default(gameID, game_state)

    def get_choice_handler(config: dict[str, Any]) -> Callable:
        # This can recurse as many times as necessary for more granular control
        async def default(
                *_
            ) -> None:
            return

        async def play_card(
                gameID, game_state, event
            ) -> None:
            end = check_trick_end_default(gameID, game_state, event)
            await broadcast_game_state(gameID, game_state)
            if not end:
                await ask_card_default(gameID, game_state)
            else:
                event["type"] = "end_trick"
                print('event["type"]', event["type"])
                await state_handler_default(gameID, game_state, event)

        map: dict[str, Callable] = {
            "play_card": play_card,
            "_": default,
        }

        async def choice_handler(
                gameID: int, game_state: dict[str, Any], event: dict[str, Any]
            ):
            func = map.get(event["choice"]["type"], map["_"])
            await func(gameID, game_state, event)

        return choice_handler

```

Figure 3.19: The final version of the Whist `state_handler`

```python

    async def end_trick(
            gameID, game_state, event
        ) -> None:
        end = check_hand_end_default(gameID, game_state)
        if not end:
            play_trick_default(gameID, game_state)
            await broadcast_game_state(gameID, game_state)
            await ask_card_default(gameID, game_state)
        else:
            event["type"] = "end_hand"
            await state_handler_default(gameID, game_state, event)


    def get_end_hand(config: dict[str, Any]) -> Callable:
        # This structure can be copied to implement more variations
        match config["scoring"]:
            case "american_short_whist":
                game_end_func = check_game_end_american_short_whist
            case _:
                game_end_func = check_game_end_default

        async def end_hand(
                gameID, game_state, event
            ) -> None:
            update_points_default(gameID, game_state)
            await broadcast_scoreboard(gameID, game_state)
            end = game_end_func(gameID, game_state)
            if not end:
                initialize_hand_default(gameID, game_state)
                play_trick_default(gameID, game_state)
                await broadcast_game_state(gameID, game_state)
                await ask_card_default(gameID, game_state)
            else:
                event["type"] = "end_game"
                print('event["type"]', event["type"])
                await state_handler_default(gameID, game_state, event)

        return end_hand

    async def end_game(
            gameID, game_state, event
        ) -> None:
        print('event["type"]', event["type"])
        await broadcast_scoreboard(
            gameID, game_state, "game_end"
        )

```

Figure 3.20: The final version of the Whist `state_handler`

```python

    # Potentially each of these functions should have a standard interface
    # `get_X_handler(config)`, but currently functions that do not have
    # variations have their defaults hard-coded
    map: dict[str, Callable] = {
        "start": start,
        "choice": get_choice_handler(config),
        "end_trick": end_trick,
        "end_hand": get_end_hand(config),
        "end_game": end_game,
        "_": default,
    }

    async def state_handler_default(
        gameID: int, game_state: dict[str, Any], event: dict[str, Any]
    ) -> None:
        # Call setup and teardown functions after each event
        func = map.get(event["type"], map["_"])
        await func(gameID, game_state, event)

    return state_handler_default

```

Figure 3.21: The final version of the Whist `state_handler`

With the goal of encapsulation – and the greater goal of modularity – I made a final rewrite of the `"state_handler"` and the final version of my program. In my Success Criteria I described allowing users to decide on variations that they wanted and having the program change its behaviour based on the variations specified. As I concluded in my analysis, this would require replacing the functions that handle bidding, playing, determining winners etc., with variations. Thus, I took inspiration from how I had solved this problem with `censor_game_state` and copied the same structure into the `state_handler`. Rather than using `match` I used a dictionary to set what code would be ran for each event. I could change this at run time to reflect users picking different variations when creating a new game and this would modify the behaviour of the game. Currently the game has only a single variation 'American Short Whist' that changes the length of the game, however this structure would allow me to add as many as I desired.

3.11 Prototypes

I created various prototypes during development in particular for the `censor_func` that I did not include above. I focused on creating a working, but imperfect system and gradually and iteratively improving it. I did this through refactoring the system as new requirements emerged rather than attempting to design and implement the system in one go.

```
99
100 def game_state_for_user(game_state: dict[str, Any], userID: int) -> str:
101     copy: dict[str, Any] = {"type": "game_state"}
102     for key, value in game_state.items():
103         if key != "players":
104             copy[key] = value
105             continue
106         if key == "players":
107             copy["players"] = []
108             for player in game_state["players"]:
109                 copy_player = {}
110                 for k, v in player.items():
111                     if k not in ["hand", "userID"]:
112                         copy_player[k] = v
113                         continue
114                     if k == "userID":
115                         copy_player["username"] = games.get_username(v, server.app)
116                         continue
117                     if player["userID"] == userID:
118                         copy_player["hand"] = player["hand"]
119                         copy["current_user"] = game_state["players"].index(player)
120                     else:
121                         copy_player["hand"] = [ "" for _ in player["hand"] ]
122                     copy["players"].append(copy_player)
123     print(f"{userID} {copy}")
124     return json.dumps(copy)
125
```

Figure 3.22: The first version of what would become the Whist `censor_func`

Above is the first version; it is static, cannot be changed at run time and is designed only for a single type of game. It does not support variations and as such is not suitable for the final version of the program. However this was not the purpose of this first version, it was designed to specify an interface that the function could use (the parameters it has and what it returns) and to prove that it was possible for me to program this feature.

```
72
73 def censor_game_state(game_state: dict[str, Any], userID: int) -> str:
74     def default(
75         censored_state: dict[str, Any],
76         game_state: dict[str, Any],
77         key: str,
78         userID: int,
79     ) -> None:
80         censored_state[key] = game_state[key]
81
82     def censor_hand(
83         censored_player: dict[str, Any],
84         player: dict[str, Any],
85         hand: str,
86         userID: int,
87     ) -> None:
88         if player["userID"] == userID:
89             censored_player[hand] = player[hand]
90         else:
91             censored_player[hand] = [ "" for _ in player[hand] ]
92
93     def censor_userID(
94         censored_player: dict[str, Any],
95         player: dict[str, Any],
96         userID: str,
97         current_userID: int,
98     ) -> None:
99         censored_player["username"] = games.get_username(player[userID], server.app)
100
101     def censor_player(
102         censored_players: list[dict[str, Any]],
103         player: dict[str, Any],
104         _,
105         userID: int,
106     ) -> None:
107         censored_player: dict[str, Any] = {}
108         censor: dict[str, Callable] = {
109             "hand": censor_hand,
110             "userID": censor_userID,
111         }
112         key: str
113         for key in player:
114             if key not in censor:
115                 default(censored_player, player, key, userID)
116                 continue
117             func = censor[key]
118             func(censored_player, player, key, userID)
119         censored_players.append(censored_player)
120
```

Figure 3.23: A Whist `censor_func` that uses a `dict` that links each key to a `censor` function

```
120
121     def censor_players(
122         censored_state: dict[str, Any],
123         game_state: dict[str, Any],
124         players: str,
125         userID: int,
126     ) -> None:
127         censored_state["players"] = []
128         player: dict[str, Any]
129         for player in game_state[players]:
130             censor_player(censored_state["players"], player, "", userID)
131             if player["userID"] == userID:
132                 censored_state["current_user"] = game_state["players"].index(player)
133
134     censored_state: dict[str, Any] = {"type": "game_state"}
135     censor: dict[str, Callable] = {
136         "players": censor_players,
137     }
138     key: str
139     for key in game_state:
140         if key not in censor:
141             default(censored_state, game_state, key, userID)
142             continue
143         func = censor[key]
144         func(censored_state, game_state, key, userID)
145     print()
146     print(f"{userID} {censored_state}")
147     return json.dumps(censored_state)
148
```

Figure 3.24: A Whist `censor_func` that uses a `dict` that links each key to a `censor` function

The first rewrite was the most drastic moving from a hard-coded system that expected the game state to be of a very particular format to a much more dynamic one. This version used a dictionary that could be changed to add support for variations to determine how each key should be censored. The `game_state` is a python `dict` that is iterated over to create a `censored_state` that includes only the information that this player should see. This is done by taking each key in turn and checking if it is in the `censor` dictionary. If it is not, the default function is called (copying the key / value pair into the `censored_state`), otherwise the corresponding function is called. This provides greater modularity since the behaviour of the function can be changed simply by changing the `censor` dictionary. I believed that this design could provide an easy way to add variations and this was the reason I chose it.

```
13 
14 def censor_game_state(game_state: dict[str, Any], userID: int) -> str:
15     def default(
16         censored_state: dict[str, Any],
17         game_state: dict[str, Any],
18         key: str,
19         userID: int,
20     ) -> None:
21         """
22         Copy the key / value pair to censored_state from game_state.
23         """
24         censored_state[key] = game_state[key]
25 
26     def default_list(
27         censored_state: list[dict[str, Any]],
28         game_state: list[dict[str, Any]],
29         key: str,
30         userID: int,
31     ) -> None:
32         """
33         Given 2 lists, copy the key / value pair for every item.
34         """
35         for i, item in enumerate(game_state):
36             censored_item = censored_state[i]
37             censored_item[key] = item[key]
38 
39     def ignore(
40         censored_state: dict[str, Any],
41         game_state: dict[str, Any],
42         key: str,
43         userID: int,
44     ) -> None:
45         return
46 
47     def censor_hands(
48         censored_state: dict[str, Any],
49         game_state: dict[str, Any],
50         key: str,
51         userID: int,
52     ) -> None:
53         for i, player in enumerate(game_state["players"]):
54             censored_player = censored_state["players"][i]
55             if player["userID"] == userID:
56                 censored_player["hand"] = player["hand"]
57             else:
58                 censored_player["hand"] = [ "" for _ in player["hand"] ]
59 
```

Figure 3.25: A Whist `censor_func` that ensures a consistent interface for each different variation

```python

    def censor_userIDs(
        censored_state: dict[str, Any],
        game_state: dict[str, Any],
        key: str,
        userID: int,
    ) -> None:
        for i, player in enumerate(game_state["players"]):
            censored_player = censored_state["players"][i]
            username: str = utils.get_username(player["userID"], server.app)
            censored_player["username"] = username

    def censor_players(
        censored_state: dict[str, Any],
        game_state: dict[str, Any],
        _: str,
        userID: int,
    ) -> None:
        censored_state["players"] = [ {} for _ in game_state["players"] ]
        # Assuming that all players have identical keys
        for key in game_state["players"][0]:
            censor: dict[str, Callable] = {
                "hand": censor_hands,
                "userID": censor_userIDs,
            }
            if key not in censor:
                default_list(
                    censored_state["players"],
                    game_state["players"],
                    key,
                    userID
                )
                continue
            func = censor[key]
            func(censored_state, game_state, key, userID)
        # There may be a more efficient approach for this
        player: dict[str, Any]
        for player in game_state["players"]:
            if player["userID"] == userID:
                censored_state["current_user"] = game_state["players"].index(player)

```

Figure 3.26: A Whist `censor_func` that ensures a consistent interface for each different variation

```
99
100     censored_state: dict[str, Any] = {"type": "game_state"}
101     censor: dict[str, Callable] = {
102         "players": censor_players,
103         "func": ignore,
104         "deck": ignore,
105     }
106     key: str
107     for key in game_state:
108         if key not in censor:
109             default(censored_state, game_state, key, userID)
110             continue
111         func = censor[key]
112         func(censored_state, game_state, key, userID)
113     return json.dumps(censored_state)
114
```

Figure 3.27: A Whist `censor_func` that ensures a consistent interface for each different variation

This third prototype was created after I discovered an issue with the previous version that stemmed from not passing a complete game state to some of the censor functions (see 5.3). In Whist the final card is dealt to the dealer face up and this determines the trump suit for the hand. Once it is the dealer's turn to play, they pick up the card and add it to their hand and play continues as normal. To implement this, I would need to create a variation of the censor function where every player could see the dealer's last card and use this variation during the first trick and use the normal one otherwise. This was a good test case for the system since this was what I had designed it to do.

However the previous design could not cope with this because the `censor_player` function would only take a subset of the `game_state`, which had the data for the player that it was censoring. This meant that it could not read other parts of the `game_state`, such as who was the dealer, to change its behaviour. This significantly reduced the modularity of the system since it prevented any variation from using data that was not for that specific player, which went against my main principle during development.

To fix this issue I needed to have each function take the entire `game_state` so that it (or any other variations) could read any data that it might need. This was not possible with the current system as `censor_player` would need to be told which player it should censor which would require a change to the interface that would have to be replicated for all other variations. Instead I designed a function that would censor every player at once, one attribute at a time, removing this requirement. This is the version above.

```python

def create_main_func(
        censor: dict[str, Callable],
        add_funcs: list[Callable]
    ) -> Callable:

    def censor_main(game_state: dict[str, Any], userID: int) -> str:
        censored_state: dict[str, Any] = {}
        key: str
        for key in game_state:
            if key not in censor:
                # Copy the key/value pair into the `censored_state`
                default(censored_state, game_state, key, userID)
                continue
            # Otherwise mutate the `censored_state` by calling a replacement / variation function
            func = censor[key]
            func(censored_state, game_state, key, userID)
        for func in add_funcs:
            func(censored_state, game_state, userID)
        return json.dumps(censored_state)

    return censor_main

```

Figure 3.28: Python code to censor a game state for a user with support for variations

```python

def get_whist_censor_func(variation: str) -> Callable:
    match variation:
        case "first_trick":
            # Show the dealers last card which decides the trump suit
            censor: dict[str, Callable] = {
                "hand": censor_hands_first_trick,
                "userID": censor_userIDs,
            }
            add_funcs: list[Callable] = []
            censor_players = create_players_func(censor, add_funcs)
            censor: dict[str, Callable] = {
                "players": censor_players,
                "func": ignore,
                "deck": ignore,
            }
            add_funcs: list[Callable] = [set_event_type("game_state")]
            return create_main_func(censor, add_funcs)
        case _:
            censor: dict[str, Callable] = {
                "hand": censor_hands,
                "userID": censor_userIDs,
            }
            add_funcs: list[Callable] = []
            censor_players = create_players_func(censor, add_funcs)
            censor: dict[str, Callable] = {
                "players": censor_players,
                "func": ignore,
                "deck": ignore,
            }
            add_funcs: list[Callable] = [set_event_type("game_state")]
            return create_main_func(censor, add_funcs)
```

Figure 3.29: Python code to censor a game state for a user with support for variations

Whilst I had designed a system that could have variations, I had no

support for this. I had planned to use currying to achieve this but I was not familiar with how I could achieve this. However it seems that using a `dict` to link from keys to functions was a stroke of luck since this design allowed for the `dict` to be passed as an argument to the function. Since this `dict` is what determines the behaviour of the function, this is the only thing that has to be changed to add a variation. I was also able to use currying to ensure that this `dict` would only have to be passed to the function once, this would be used to create a function that has the desired behaviour which is then returned. This is the interface that I designed for the system `get_whist_censor_func(variation)` that is used to get the corresponding censor function for the variation provided.

3.12 Testing

During development I ran into a series of issues many of which occurred throughout:

Error	**Fix**
`SyntaxError`	I encountered many of these during development, with many different causes. Most were easy to fix and stemmed from forgetfulness, for example adding a colon after an `if` statement, closing a bracket after a function call or adding commas between each element in a list.
`NameError`	This occurred when I renamed a variable but forgotten to change one instance. Again this was an easy fix, just changing the reference to use the new name.
`KeyError`	Since I used a dictionary to store the `game_state` if I tried to access an attribute that I had not defined, Python would raise a `KeyError`. Static analysis tools were also quite bad at detecting this since I had type hinted `game_state` to be a `dict[str, Any]` which provided no type guarantees.
Live reloading	During development I worked on 2 servers (`server.py` and `ws_server.py`) which were HTTP and Web Socket servers respectively. I used Flask for the HTTP server and Python's `websockets` package for the second. Flask provides live reloading so the server restarts automatically when it detects changes, but `websockets` does not. Often I made changes to both servers and the Flask server would restart automatically but the `websocket` server would not, leaving the 2 out of sync and causing errors.

Design errors	The most difficult errors that I faced during development were caused when I discovered that my design was incomplete or mistaken. These issues were the most varied and difficult to fix. They required me to redesign a system with pen and paper first before I could begin implementing the new version. In these occasions I would describe the issue and my design in my development log (see 5.3).

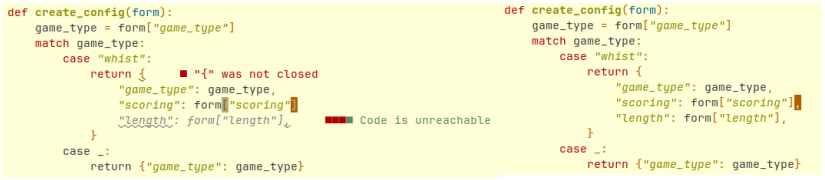


Figure 3.30: A syntax error caused by a missing comma

Above is a syntax error that occurred during development. One the left is the incorrect code and on the right is the corrected code. The additional text and highlighting on the left is produced by an LSP (Language Server Protocol) for Python. This is a program that is similar to the main interpreter and parses the code in a similar manner. However it does not run the code but looks for syntax, indentation, type errors and more. It can be used to increase development speed through spotting errors early. By using it I was able to identify this error before running my code which saved time.

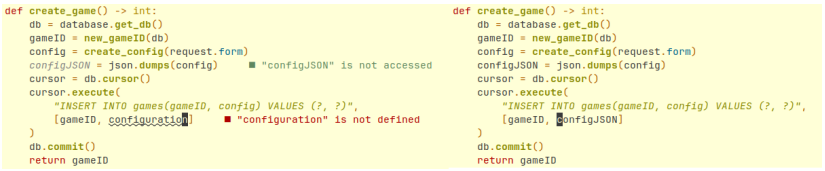


Figure 3.31: A name error caused by changing the name or variable in all but one instance

Above is a name error that occurred when I was changing the name of a variable from `configuration` to `configJSON`. I changed its name where it was defined but forgot to change the name when it was used. Again my LSP provided advance warning for this telling me that the old variable name was undefined and the new variable name was not used.

```
# This gives the whole game state
players = WHIST[gameID]
```

```
# rather than this which gives only the players (which is what I wanted)
players = WHIST[gameID]["players"]
```

Above is an error that I encountered during development, which is rather similar to `KeyError`s that I experienced during development (see 5.3). However, whilst I had used the wrong attribute, in this case it was defined which caused a logic rather than syntax error. I was able to catch this error through the logging that I had added. During development of a new feature I would add `print` statements to print intermediate values. This would allow me to manually check these and more easily discover where issue lay. If there was an error with one of these intermediate values I would then know that the bug was between the last correct and first incorrect print statement. Such logging also allowed me to quickly catch silly mistakes such as the one above.

```
The logging printed this:
{ ... 'players': {"players": [ {"userID": ... }, ... ]}, ... }
rather than the expected:
{ ... 'players': [ {"userID": ... }, ... ], ... }
```

Figure 3.32: An error caused by changing 1 file without changing a second that depends on it

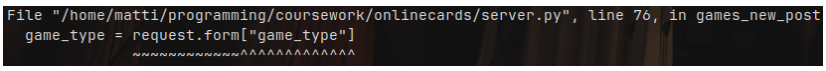


Figure 3.33: An error caused by changing 1 file without changing a second that depends on it

Above is an example of an error that occurred when I had misremembered the name of a variable and used different variables names in 2 different files. In the first image you can see the incorrect and corrected code on the left and right respectively. Below in the second image, you can see the error that was raised. The python code in `server.py` was expecting the HTML form entered to have an attribute for `"game_type"` but it one called `"game_name"` instead. This caused an error since the code could not identify the game name to continue processing the request. This error was more difficult to catch since my LSP did not provide any hint that this would be the case – since the error was due to variable names not being synced across multiple files, markup languages and programming languages.

One example of a design error that I experienced was during the development of the `censor` function. Once it became clear that my initial version would not be sufficient I had to redesign the system before I could implement it. I initially described the problem and discussed 2 different methods that I could use to complete the task (see 5.3). Once I had decided the method that I would use I wrote the second version of the function, discussed the design and flaws (see 5.3). During further development the flaw that I had noticed earlier (not passing the complete game state to each censor function) began to limit development and required another rewrite (see 5.3). Lastly the final update was a small rewrite that added the final feature and flexibility that I had envisioned for the program (see 5.3).

Despite being a large part of programming and safe input handling, my program does not have large amounts of validation because during development I focussed on adding features rather than security. I believe it is more important to have a complete and functioning - if fragile - program than a smaller but more robust one. In my development log I discussed how I would add validation (see 5.3) but I did not have the time to implement these ideas. Some parts of the code (mainly front-end Javascript) does have validation when it was added to fix bugs. The problem with not including validation server-side is the potential for users to send intentionally malicious data to cheat the game. Even accidentally incorrect or corrupt data could cause errors if improperly handled, for example attempting to access part of an event that is not defined would raise an error. For this reason such validation would have reduced the likelihood of a server crash and make cheating more difficult.

IV

Evaluation

"An unexamined life is not worth living"

- Socrates

4.1 Final Testing

As part of my final testing, I recorded playing the game against myself. I then uploaded the video to Youtube. Below I recorded timestamps where the features from the analysis (see Section 1.4) are demonstrated.

Time	Success	Feature
0:10	Partial	The website homepage has 4 options for creating a new game, joining an existing game, reading the rules for each game and viewing the program source code. However the rules page is not complete as it was a low priority.
0:12	Partial	A new game is created by filling out a HTML form. The user can choose the game (Whist, Promise, Poker, etc.) and the variations, though only British and American Short Whist are currently supported - no other games can be played.

0:22	Failure	When a new game is created, the user that created the game does not become the host. Previously I had designed the game so that this user would run the server code to manage the game, ask other players for input, and update the game state. However I discovered that this would not be possible since it would require the client's computer to act like a server, which is not possible with a web browser (without installing additional software). Instead the server manages the game state, asks for user input, and broadcasts the game state in the same manner that I described in my design.
0:25	Full	Another user navigates the website and joins the game by entering the game code.
0:41	Full	A third user joins the game by entering a link. This would be sent to them by a friend.
0:42	Full	A waiting screen is displayed to connected users while they wait for others to join.
0:56	Full	Once all 4 players join, the game begins with each player being dealt cards.
0:58	Full	The player left of the dealer is the first to play so they are sent a 'decision' event. This event asks the player which of their cards they want to play, which the browser displays by separating each card so that they do not overlap.
1:06	Full	Once the player has chosen which card they would like to play they can click it.
1:09	Full	The user interface then updates showing the card that has been played.
1:11	Full	The next player is then sent a decision event for which card they want to play. However they can only play cards of the correct suit and to communicate this, invalid cards are 'greyed out' and cannot be clicked.
1:19	Full	Once the last player in the trick has played a card a new `game_state` is sent to each user showing who won the trick by updating each player's 'Tricks Won'.
1:23	Full	Play continues like this until all players have exhausted their cards
4:46	Full	Once all cards have been played the hand has ended and scores can be calculated
4:47	Partial	A scoreboard can briefly be seen but the screen is redrawn too quickly to be useful (see 4).

14:30	Full	Once a partnership has scored enough points to win the game (determined by whether the scoring is American or British short Whist) a final scoreboard is shown along with a victory message for the winning partnership. Below this is a 'Play Again' button for if users wish to restart the game.

4.2 Usability Features

Since I was developing a website I mostly relied upon using standard HTML and JavaScript to create a user interface. I attempted to use semantic HTML as much as possible, which implemented many accessibility features for me. I used CSS to create a consistent visual style (a dark brown background with contrasting antique white text) and to ensure that buttons were clear and easy to use. However this was not always used and when playing a card game the user interface changes slightly. The colour scheme is kept consistent and each player is positioned around a 'table'. Information about each player such as their name, how many tricks they have won, etc. is placed in an 'info box' next to their seat. Information about the game is in a central 'info box' on the table. Each player's hand is displayed beneath their seat and info box (either face down or face up depending on the game and its state), and is 'squashed' by default so that their cards overlap. Once it is a player's turn to play a card their hand is redrawn so that their cards do not overlap. Cards that they cannot play are shown as 'greyed out' by reducing their opacity, but clicking on a valid card lets a user play that card by sending a response to the server, which then updates the game state.

4.3 Reflection

Whilst I was able to meet all of my success criteria the project did not quite reach the goals that I had. In particular I did not create the range of games and variations that I had planned - instead creating a single game and variation as a 'proof of concept'. This was due to the unexpected complexity of creating a website and my inexperience with web development. I had to spend large amounts of time learning about HTTP protocols, client-server architecture and websockets. This meant that I was not able to focus on the main problem - creating a system that could automatically collate the correct rules for any given variation - and this system is not as complete as I would like it to be.

Future variations could be added to Whist by updating the `games/whist.py` file and the `get_whist_state_handler_default` function in particular. It has `config` as a parameter which is a dictionary that specifies

which variation should be used for each part of the game which determines: which cards can be played, who wins each trick, how many points each player scores, etc. An example of this is the two scoring variations: 'American Short Whist' and 'British Short Whist' that change the function used to calculate whether the game has ended.

However particularly for more complex variations that affect multiple parts of the game, this system may not be well suited. One problem that I have not tackled is how variations should be combined. We can consider card games as being a list of rules and each variation as overwriting some of these rules. Then for any 2 variations we can see if they both overwrite the same rules or they do not. If they do we can say that a pair of variations are 'mutually exclusive' meaning that only 1 can be used at a time. The previous system is similar to this, categorising variations by which rules they change (e.g. bidding variations change how players place bids), but does not have the same flexibility. However using a more complicated system introduces more complexity, particularly in calculating which variations are mutually exclusive, which may want to be done client side to improve the menu performance for choosing variations. Apart from this issue there are likely more that I have not considered that would become apparent during development - so this feature may have a larger scope than I anticipate.

Adding different card games would vary in difficulty based on the card game and how much code can be reused from previous games. I began with implementing Whist because I wanted to reuse parts of the code (such as for playing tricks) for games such as Spades and Bridge. For this reason implementing either of these games would be easier than a game such as Poker, that would have to have lots of new code written for it. However many different versions of Poker could share the same code which would reduce the cost of developing new variations for it. This also raises questions of how to reduce code repetition and prevent different games such as Spades and Whist having their own copies of the same code whilst still providing flexibility for replacing functions and defaults. These new games could also share many of the existing systems such as a `game_state`, `event_handler`, etc.

If I were to continue development I would first add more variations to Whist until the game had 5 - 10 variations; then I would add Spades as a new game. Once I had added a similar number of variations to Spades I would then continue development either with another trick-taking game such as Bridge or Promise, or I would implement Poker as well as different variations such as Texas Hold 'em, Pot Limit Omaha, etc. I may also want to add some 'quality of life features' such as a chat box within each game or the possibility to add a profile picture / colour. During development I experimented with having each player's circle filled with a different colour and found that pastel colours greatly improved the visual style and variety so this would be a feature that I would add in future development.

For a future maintainer they would likely have to make some improvements to the code even if they did not want to add new features. One major issue is the lack of validation - which I discussed in my implementation (see 3.12) - and this would have to be rectified. This would increase the stability of the code making it less prone to accidental errors and less susceptible to malicious attacks. During development I added code comments to aid future maintainers and wrote documentation on how I had designed the systems and how they should be expanded or reused. Another developer may find this useful and even want to expand upon this for future maintenance.

V

Appendix

> *"The appendix ... is a finger-like, blind-ended tube connected to the cecum, from which it develops in the embryo ... The human appendix averages 9 cm (3.5 in) in length"*
>
> - Wikipedia

I would like to offer a great many thanks towards all the people that have helped in the creation of this project. Including but not limited to: my teachers, parents, mentors, friends, pets, fellow students, Linus Torvalds, stakeholders, acquaintances, interviewees and academic sponsors.

5.1 Figures

5.1.1 Surveys

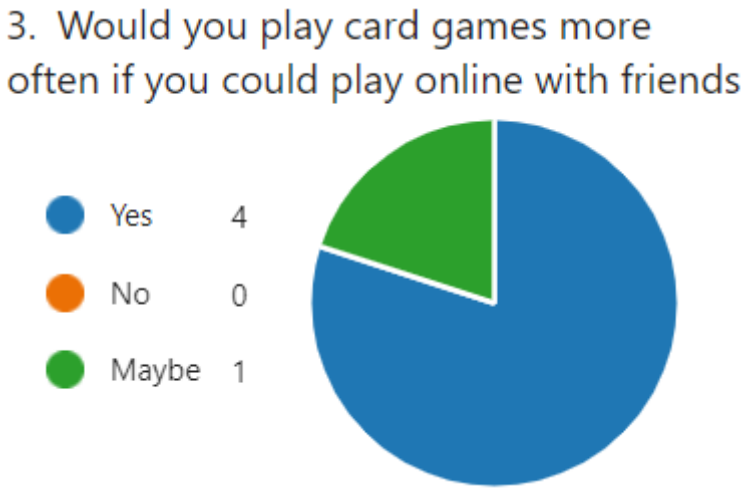


Figure 5.1: Stakeholders are more interested in playing cards online

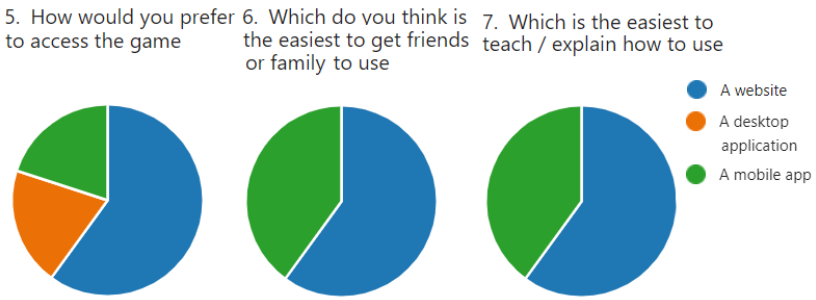


Figure 5.2: A website is the most popular format

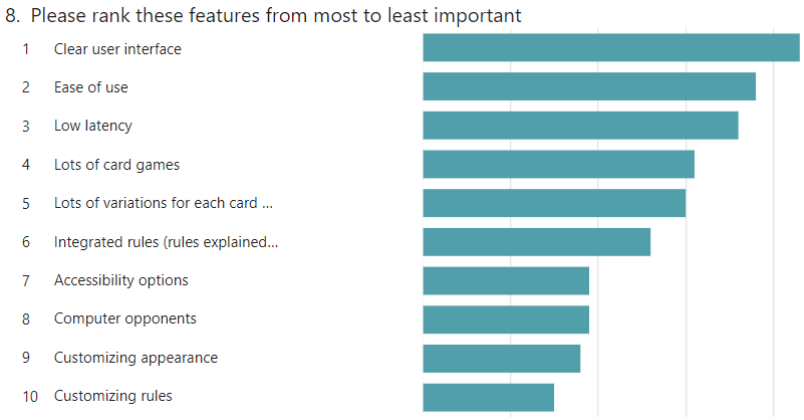


Figure 5.3: Ranking of features

5.1.2 online-spades.com

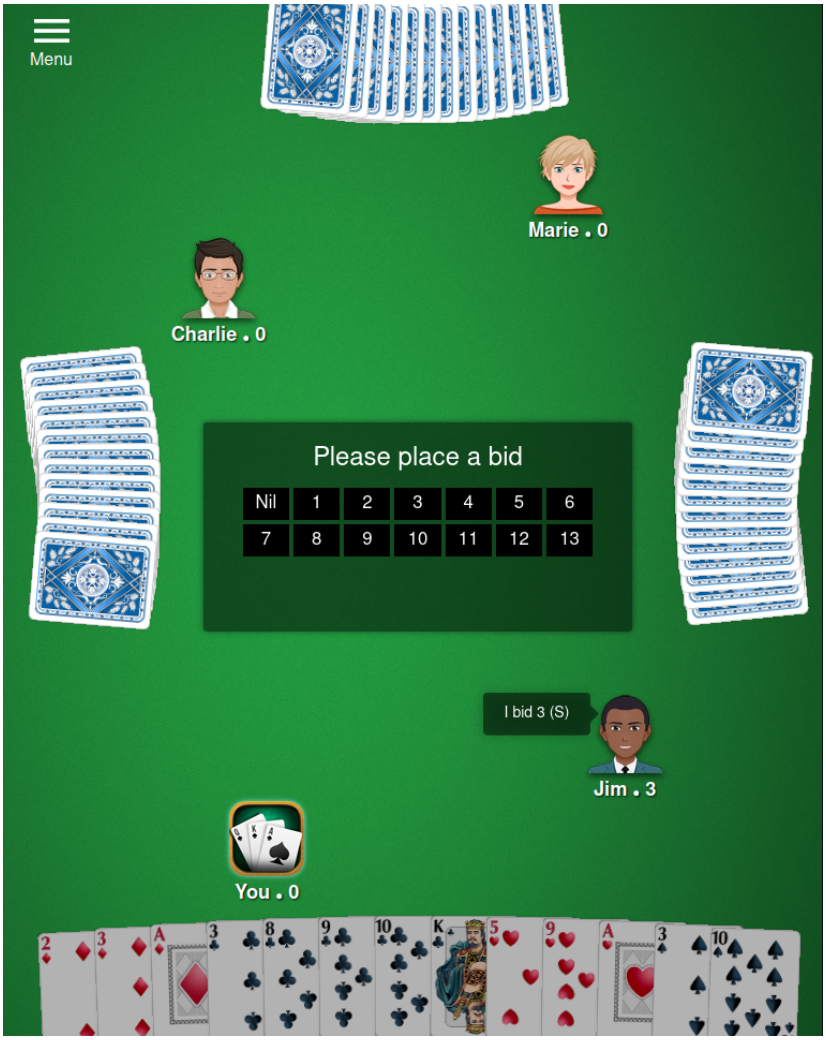


Figure 5.4: online_spades.com allows users to play spades in their browser.

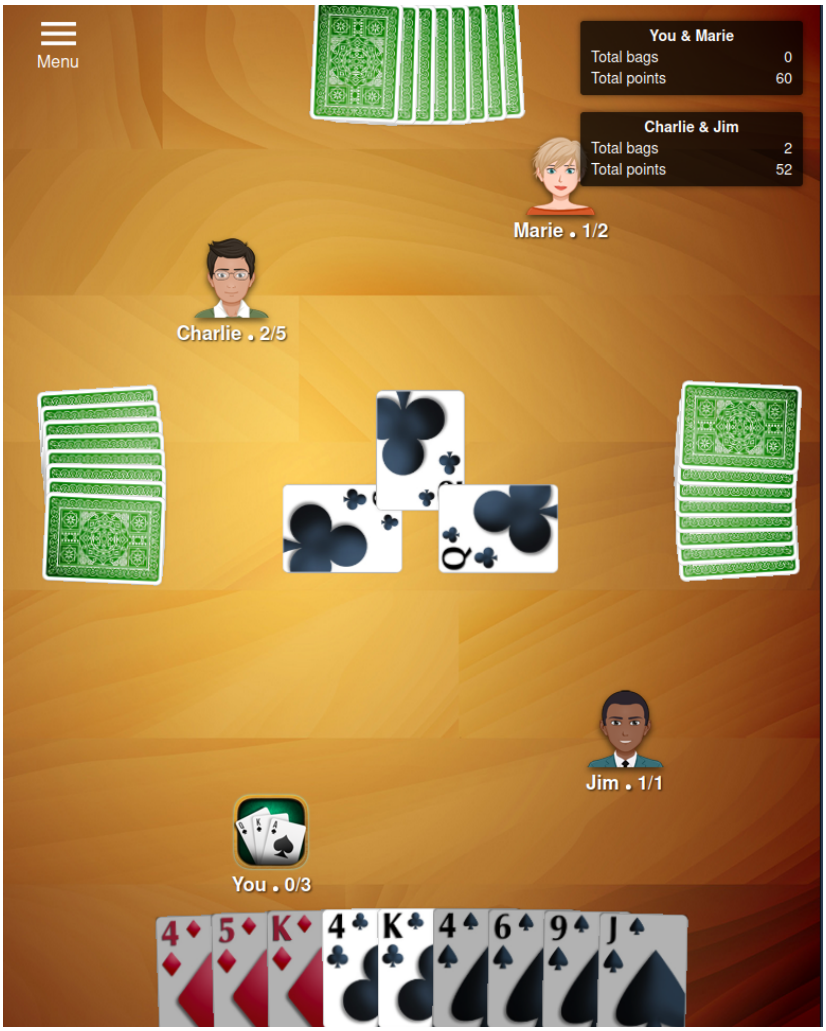


Figure 5.5: Playable cards are highlighted.

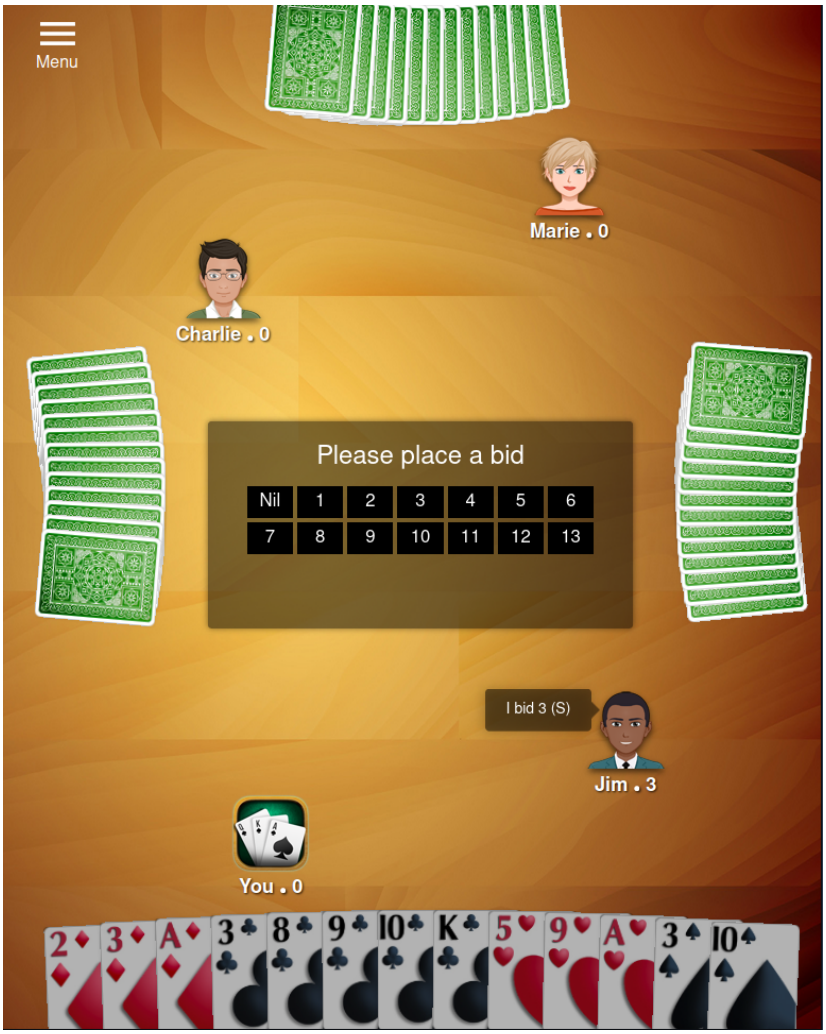


Figure 5.6: Alternate themes are available.

5.1.3 cardsjd.com

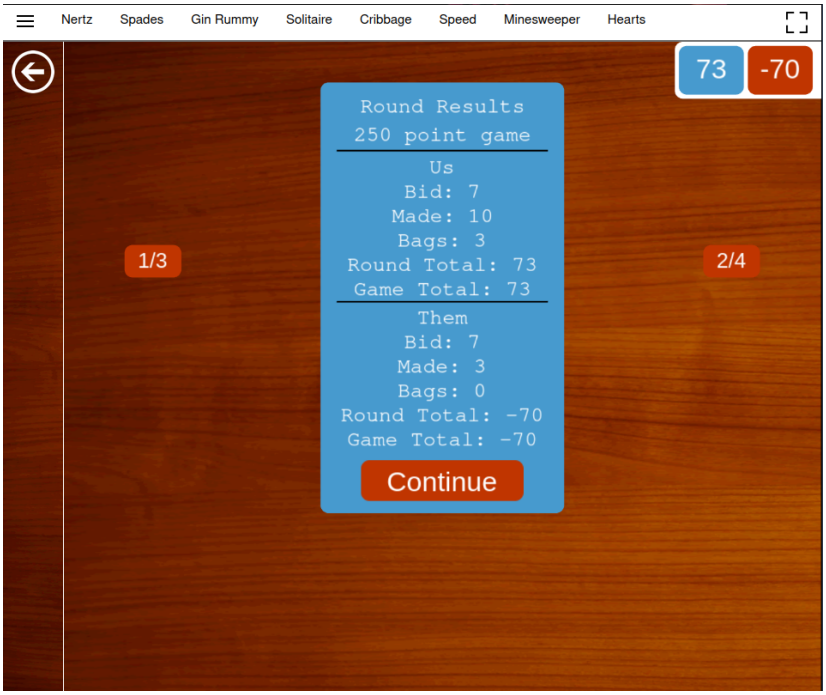


Figure 5.7: The scoreboard in cardsjd.com/spades

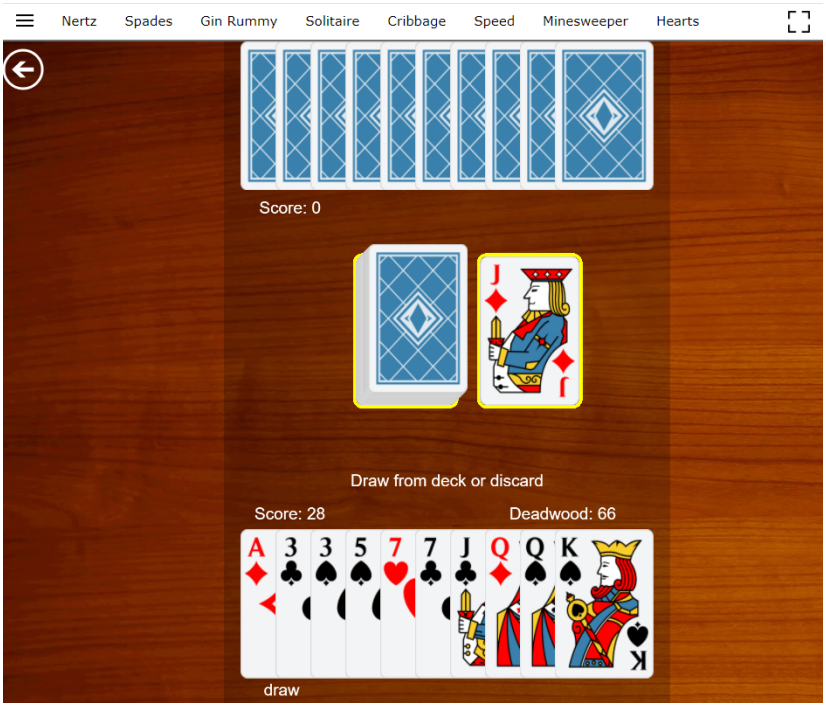


Figure 5.8: cardsjd.com offers other games such as Gin Rummy

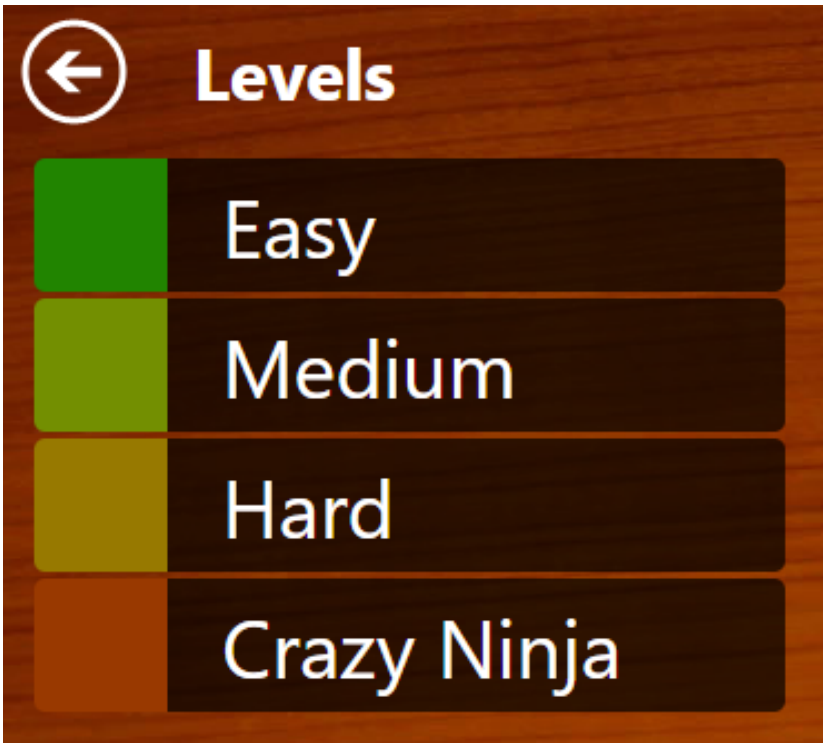


Figure 5.9: cardsjd.com allows users to change the difficulty for some games

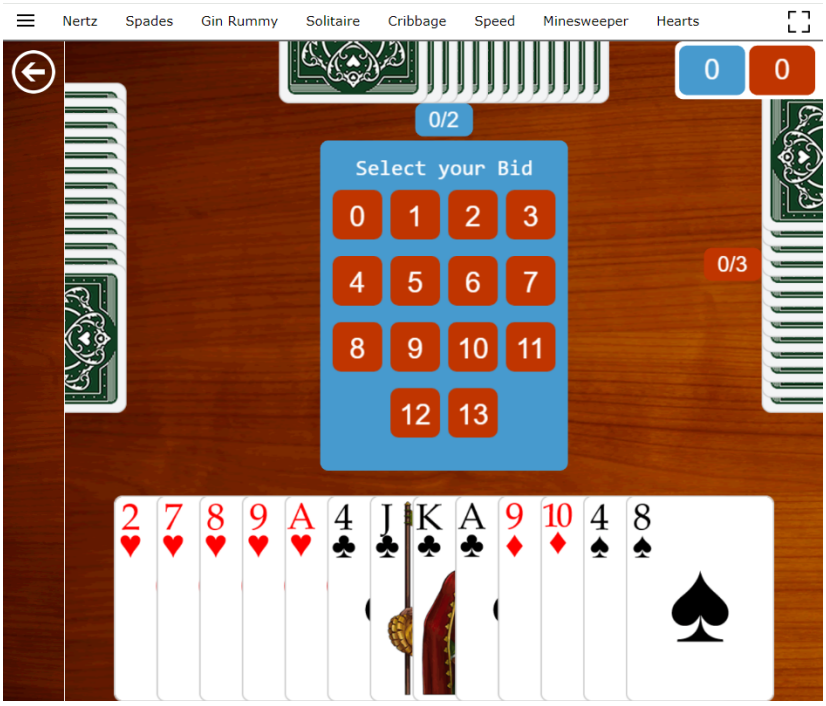


Figure 5.10: cardsjd.com/spades has similar visuals to online-spades.com

5.1.4 pokerpatio.com

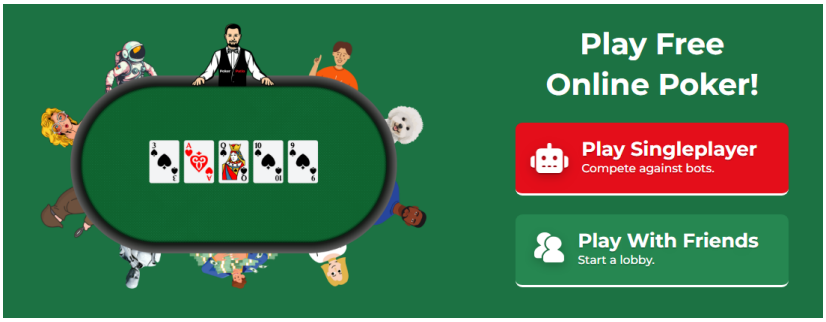


Figure 5.11: The homepage of pokerpatio.com

Figure 5.12: The host approves each player

Figure 5.13: Clear and simple UI

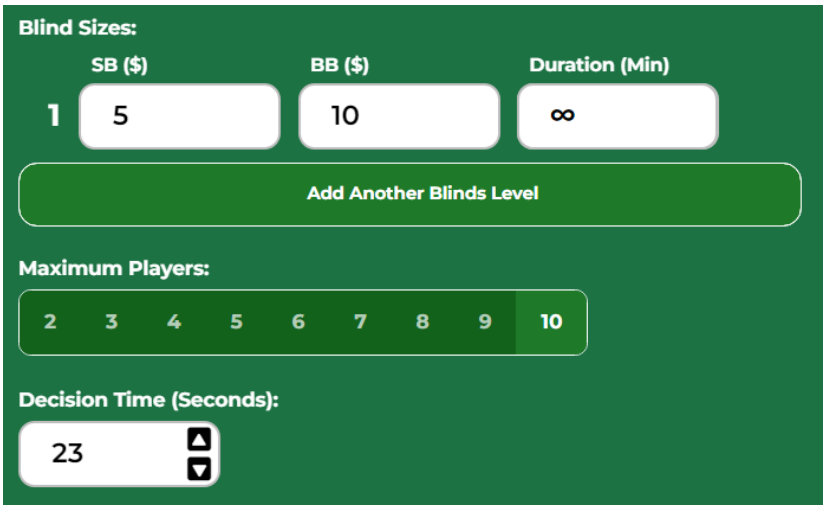


Figure 5.14: Game settings the host can change

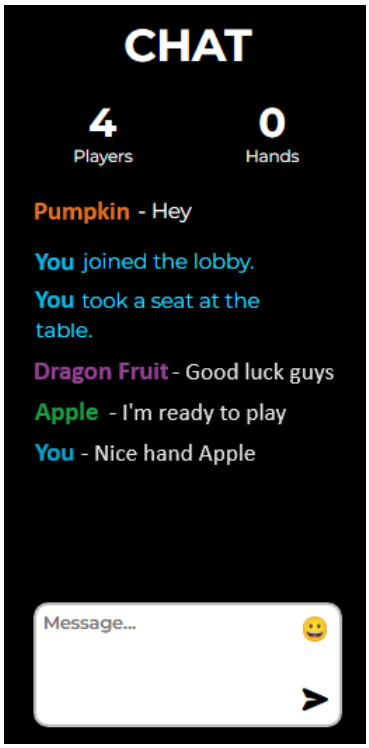


Figure 5.15: Built-in chat function

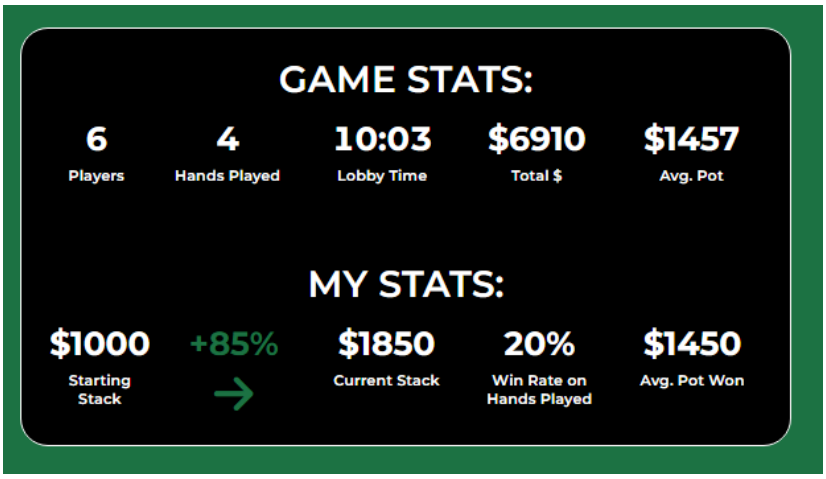


Figure 5.16: Stats page for the session

Beginner's Guide to Playing Poker:

Poker is one of the most beloved card games in the world, known for its depth, variety, and social aspect. As such, it can be played in various formats, three of which we will explore today – **Texas Hold'em, Seven Card Stud, and Omaha**.

These games share some common elements, such as the standard 52-card deck and the fundamental hand rankings, but they each bring their own distinct set of rules and dynamics. For someone interested in learning poker, understanding these three popular versions is a great place to start.

This comprehensive guide provides an introduction into the basics of poker, explores the rules of popular variants, and beginner strategies.

Poker Hands

Poker is played with a standard deck of 52 cards, with cards ranked from 2 (lowest) to Ace (highest). The deck has four suits: Hearts, Diamonds, Clubs, and Spades. No suit is higher than another.

Before we start with the game types and different rules, let's understand the possible combinations of cards or 'hands' you can get in poker, ranked from highest to lowest:

- **Royal Flush**: The best possible hand in poker. It's an ace high straight flush.
- **Straight Flush**: A five-card straight, all in the same suit.
- **Four of a Kind**: Four cards of equal value.
- **Full House**: A three of a kind of one value and a pair of another value.
- **Flush**: Any 5 cards, all of the same suit.
- **Straight**: Any 5 cards of sequential value.
- **Three of a Kind**: Three cards of the same value.
- **Two Pair**: A pair of one value and a pair of another value.
- **Pair**: Two cards of the same rank.
- **High Card**: When you don't have any of the above, your highest card plays. If two or more players hold the highest card, a kicker comes into play.

Figure 5.17: The website has guides for beginners

Figure 5.18: The game can also be played vertically

5.1.5 Design

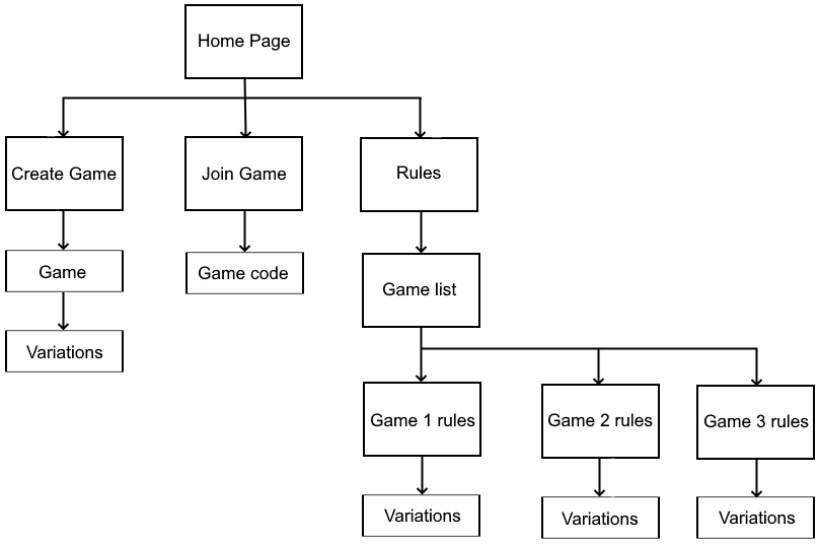


Figure 5.19: A system diagram for web page hierarchy

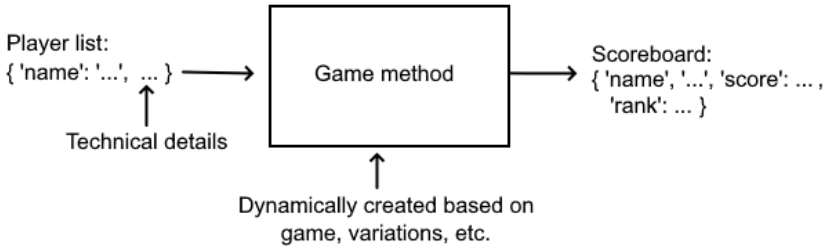


Figure 5.20: An abstract view of the game method

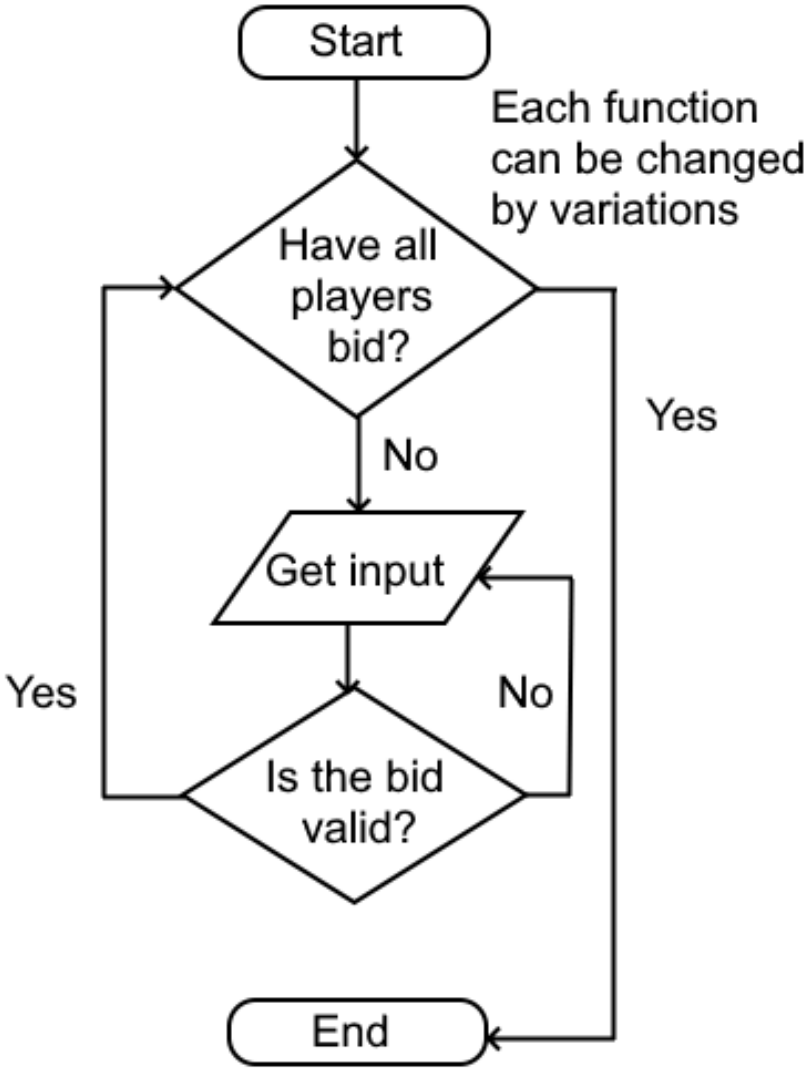


Figure 5.21: The default bid method

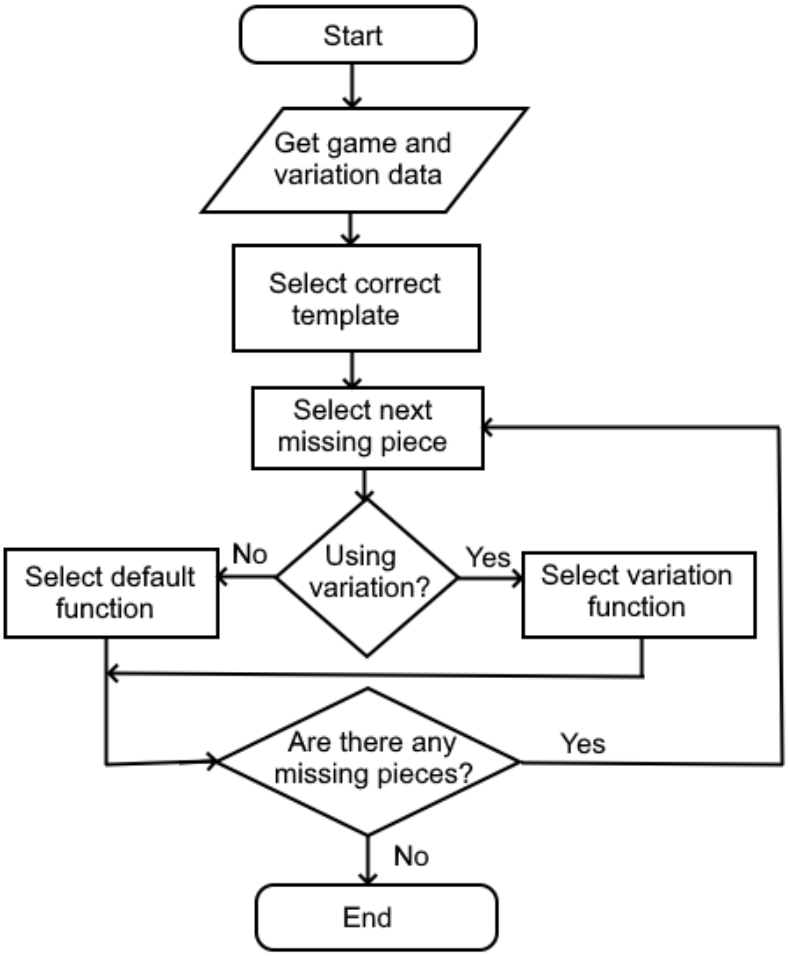


Figure 5.22: A flowchart for the game method creator function

```
Card:
    rank: int
    suit: enum
    name: string
    image: image path
```

Figure 5.23: Pseudo code and data types for the card class

```
scoreboard = [
    { 'name': string, 'score': int, 'rank': int }
]
```

Figure 5.24: Pseudo code and data types for the scoreboard

```
Player:
    name: string
    score: int
    rank: int
    ... # Other attrs vary
    # by game & variation
```

Figure 5.25: Pseudo code and data types for the player class

```
game_start:
    create_scoreboard -> ()
rubber_start:
    pick_partners -> ()
round_start:
    pick_trumps -> ()
    deal_cards -> ()
playing_trick:
    get_valid_cards -> ()
    determine_winner -> ()
round_end:
    determine_winner -> ()
rubber_end:
    determine_winner -> ()
game_end:
    output_scoreboard -> ()
```

Figure 5.26: Pseudo code for a whist game method template

```
{
    "game": "whist",
    "variations": {
        "solo": false, "bid": false, ...
    },
    "updateNo": 13,
    "players": [
        {
            "id": 0,
            "name": "Mark",
            "partnerID": 3,
            "score": 3,
            "rank": 1,
            "tricksWon": 4,
            "hand": ["AS", "2S"]
        },
        {
            "id": 1,
            "name": "Luke",
            "partnerID": 2,
            "score": 2,
            "rank": 2,
            "tricksWon": 2,
            "hand": ["3H", "7D"]
        },
        {
            "id": 2,
            "name": "John",
            "partnerID": 1,
            "score": 2,
            "rank": 2,
            "tricksWon": 5,
            "hand": ["8C", "5H"]
        },
        {
            "id": 3,
            "name": "Judas",
            "partnerID": 0,
            "score": 3,
            "rank": 1,
            "tricksWon": 0,
            "hand": ["6D", "10C"]
        }
    ]
}
```


Figure 5.27: Pseudo code for a whist game state

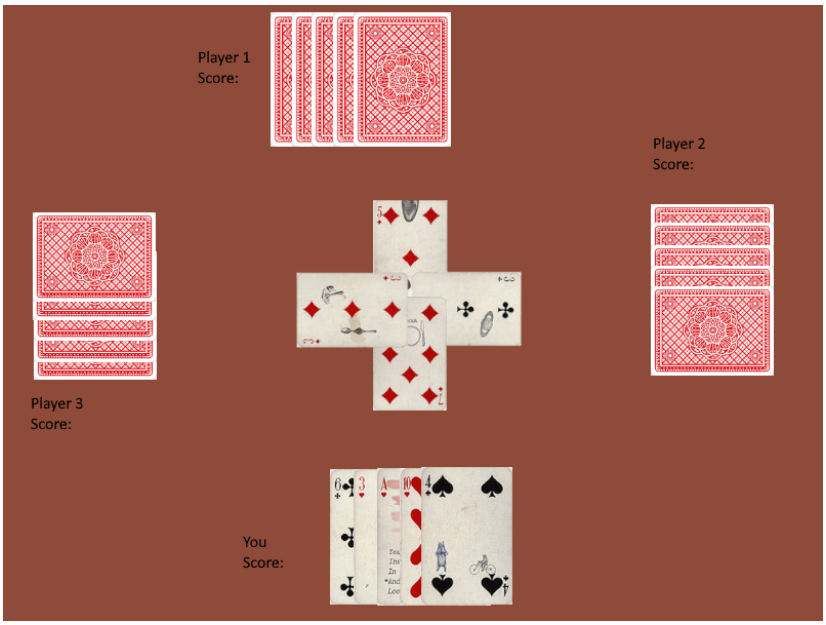


Figure 5.28: Example user interface for whist or spades

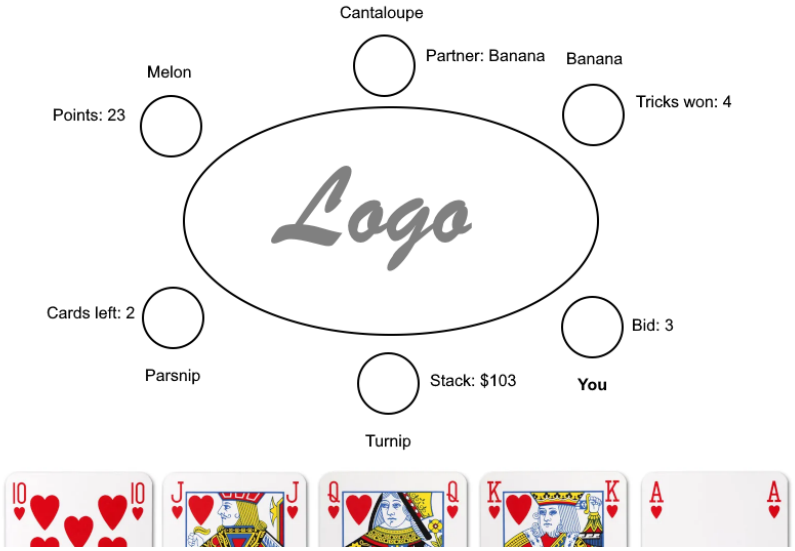


Figure 5.29: A revised, more general user interface

Figure 5.30: Example user interface for making a bet in Poker

Figure 5.31: Example user interface for player a trick in Whist or Spades

```
{
    "game": "whist",
    "variations": {...},
    "updateNo": 7,
    "updateType": "choice",
    "choiceType": "pick_card",
    "options": {
        "2H", "7H", "8H", "JH"
    }
}

```

Figure 5.32: Pseudo code for a choice in Whist

5.1.6 Implementation

Version 1

```python
from flask import Flask, render_template, request

def create_app():
    # Gives current path to flask
    app = Flask(__name__)

    @app.get("/")
    def index():
        return render_template("index.html")

    @app.get("/rules/")
    def rules():
        return render_template("rules.html")

    @app.get("/rules/whist")
    def rules_whist():
        return render_template("rules_whist.html")

    @app.get("/games/new")
    def games_new_get():
        return render_template("games_new.html")

    @app.post("/games/new")
    def games_new_post():
        game = request.form["game"]
        return render_template("games_new_post.html", game=game)

    @app.get("/games/join/")
    def games_join_get():
        return render_template("games_join.html")

    return app
```

Figure 5.33: A basic server written in Python, using Flask

```html
<html>
  <head>
    <title>onlinecards.uk</title>
  </head>
  <body>
    <a href="/"><h1>onlinecards.uk</h1></a>

    <ul>
      <li><a href="games/new">New Game</a></li>
      <li><a href="games/join">Join Game</a></li>
      <li><a href="rules">Rules</a></li>
    </ul>
  </body>
</html>
```

Figure 5.34: The home page of the website

```html
<html>
  <head>
    <title>Join Game</title>
  </head>
  <body>
    <a href="/"><h1>onlinecards.uk</h1></a>

    <form>
      <label>Game ID</label>
      <input type="number">
      <input type="submit" value="Join">
    </form>
  </body>
</html>
```

Figure 5.35: A form for joining an existing game

```html
<html>
  <head>
    <title>New Game</title>
  </head>
  <body>
    <a href="/"><h1>onlinecards.uk</h1></a>

    <form id="settings">
      <div><label id="game_label" for="game">Game:</label></div>
      <select id="game" name="game" required>
        <option id="default_game" selected disabled hidden>Choose a game</option>
        <option value="whist">Whist</option>
        <option value="promise">Promise</option>
        <option value="poker">Poker</option>
      </select>
    </form>

    <form id="whist" style="display: none" action="/games/new" method="POST">
      <select name="game" style="display: none">
        <option value="whist" selected></option>
      </select>

      <div><label>Players:</label></div>
      <div><p>4</p></div>

      <div><label for="scoring">Scoring:</label></div>
      <select name="scoring">
        <option value="british_short_whist">British Short Whist</option>
        <option value="american_short_whist">American Short Whist</option>
        <option value="long_whist">Long Whist</option>
      </select>

      <div><label for="length">Length:</label></div>
      <select name="length">
        <option value="game">1 game</option>
        <option value="rubber">1 rubber</option>
      </select>

      <div><input type="submit" value="Create Game"></div>
    </form>

    <form id="promise" style="display: none" action="/games/new" method="POST">
      <select name="game" style="display: none">
        <option value="promise" selected></option>
      </select>
      <div><label>Coming soon!</label></div>
    </form>

    <form id="poker" style="display: none" action="/games/new" method="POST">
```

Figure 5.36: A form for creating a new game

```
    <form id="promise" style="display: none" action="/games/new" method="POST">
      <select name="game" style="display: none">
        <option value="promise" selected></option>
      </select>
      <div><label>Coming soon!</label></div>
    </form>

    <form id="poker" style="display: none" action="/games/new" method="POST">
      <select name="game" style="display: none">
        <option value="poker" selected></option>
      </select>
      <div><label>Coming soon!</label></div>
    </form>

    <script src="{{ url_for('static', filename='js/games_new.js') }}"></script>
  </body>
</html>
```

Figure 5.37: A form for creating a new game

```
function updateForm() {
  let game = document.getElementById("game");
  let forms = document.querySelectorAll("form");
  for (form of forms) {
    if (form.id == "settings") {
      // Do nothing
    } else if (form.id == game.value) {
      form.style.display = "inline";
    } else {
      form.style.display = "none";
    }
  }
}

function bindFunctions() {
  document.querySelector("#game").onchange = updateForm;
}

bindFunctions();
```

Figure 5.38: Javascript code that automatically updates the above form

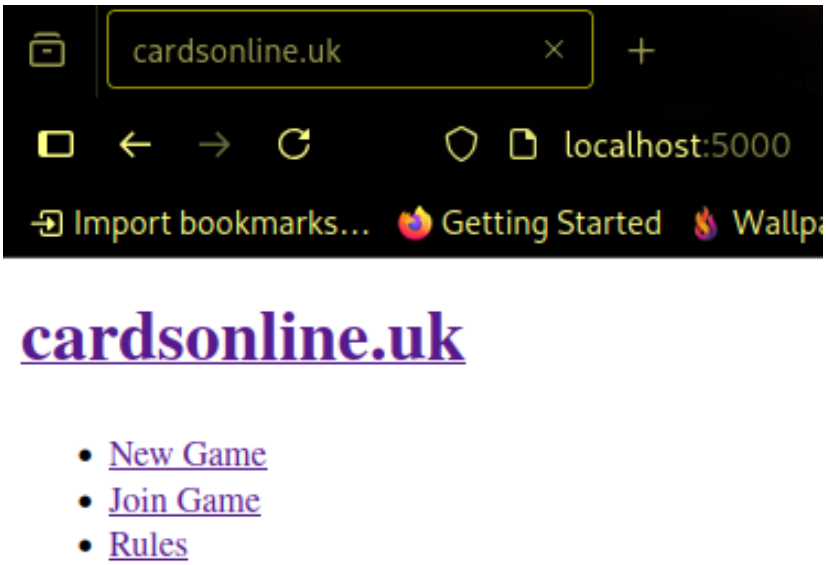


Figure 5.39: The first version of the home page

Figure 5.40: A HTML form for customizing a game and picking variations

Version 2

```python

import websockets, asyncio, json

import games
from chatroom import handle_chatroom

async def handler(websocket):
    async for eventJSON in websocket:
        print(eventJSON)
        event = json.loads(eventJSON)
        match event["type"]:
            case "create":
                await create(websocket, event["gameID"])
            case "join":
                await join(websocket, event["gameID"])
            case _:
                config = json.loads(event["config"])
                match config["game"]:
                    case "chatroom":
                        await handle_chatroom(websocket, eventJSON)

async def create(websocket, gameID):
    games.GAMES[gameID] = set()
    print(f"Created Game {gameID}")

async def join(websocket, gameID_str):
    gameID = int(gameID_str)
    try:
        connected = games.GAMES[gameID]
        games.GAMES[gameID] = connected | { websocket }
        print(f"New player joined game {gameID}")
    except KeyError:
        print(f"Error could not find {gameID}")
        event = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(event))

async def main():
    async with websockets.serve(handler, "", 8001):
        # Wait for a promise that will never be fulfilled - run forever
        await asyncio.Future()

if __name__ == "__main__":
    asyncio.run(main())
```

Figure 5.41: A websocket server written in Python

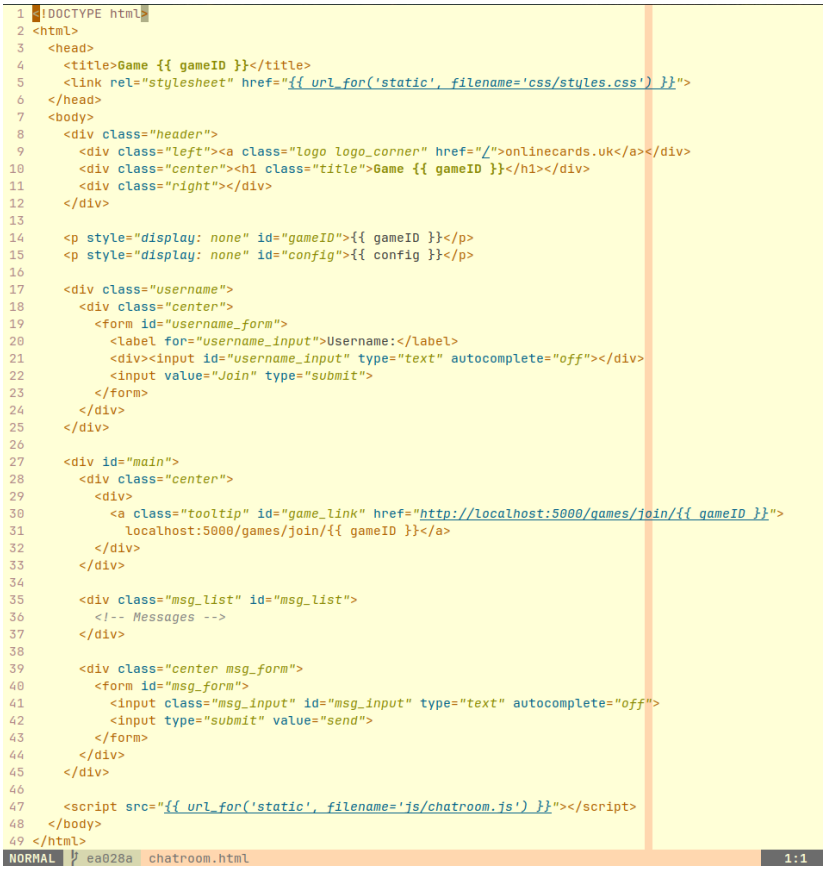


Figure 5.42: A minimalist webpage for a chatroom

```
function setUsername(form, input, main) {
  main.style.display = "none";
  form.onsubmit = (e) => {
    e.preventDefault();
    username = input.value;
    form.style.display = "none";
    main.style.display = "unset";
  }
}

function joinGame(websocket, gameID) {
  websocket.onopen = () => {
    let event = { type: "join", gameID: gameID};
    websocket.send(JSON.stringify(event));
  }
}

function sendMessages(form, websocket, gameID, config) {
  form.onsubmit = (e) => {
    e.preventDefault()
    let msg_input = document.getElementById("msg_input");
    const message = msg_input.value;
    msg_input.value = "";
    const event = {
      type: "message",
      gameID: gameID,
      config: config,
      username: username,
      message: message,
    };
    websocket.send(JSON.stringify(event));
  }
}
```

Figure 5.43: Javascript code for a chatroom using websockets

```
34
35 function recieveMessages(msg_list, websocket) {
36   websocket.onmessage = ({ data }) => {
37     const event = JSON.parse(data);
38     switch (event.type) {
39       case "message":
40         message = document.createElement("p");
41         message.innerHTML = event.username + ": " + event.message;
42         msg_list.appendChild(message);
43         msg_list.scrollTo(0, msg_list.scrollHeight);
44         break;
45       case "error":
46         console.log(event.message);
47         break;
48       default:
49         console.log(`Unsupported event type '${event.type}'`)
50     }
51   }
52 }
53
54 function copyLink(event) {
55   event.preventDefault();
56   let game_link = document.getElementById("game_link");
57   let clipboard = navigator.clipboard;
58   if (clipboard != undefined) {
59     clipboard.writeText(game_link.href);
60     game_link.style.setProperty("--message", '"Copied"');
61   } else {
62     game_link.style.setProperty("--message", '"Error"');
63   }
64 }
65
```

Figure 5.44: Javascript code for a chatroom using websockets

```
65
66 var username = "placeholder";
67
68 function bindFunctions() {
69   const username_form = document.getElementById("username_form");
70   const username_input = document.getElementById("username_input");
71   const main = document.getElementById("main");
72   const msg_form = document.getElementById("msg_form");
73   const msg_list = document.getElementById("msg_list");
74   // localhost needs to be replaced with hostname in production so this requires a better solution
75   const websocket = new WebSocket("ws://localhost:8001/");
76   const gameID = document.getElementById("gameID").innerHTML;
77   const config = document.getElementById("config").innerHTML;
78   setUsername(username_form, username_input, main);
79   joinGame(websocket, gameID);
80   sendMessages(msg_form, websocket, gameID, config);
81   recieveMessages(msg_list, websocket);
82
83   document.getElementById("game_link").onclick = copyLink;
84 }
85
86 bindFunctions();
```

Figure 5.45: Javascript code for a chatroom using websockets

```python
#!/usr/bin/env python

import json

import games

async def handle_chatroom(websocket, eventJSON):
    event = json.loads(eventJSON)
    match event["type"]:
        case "message":
            await send_message(websocket, eventJSON)
        case _:
            await error(websocket, eventJSON)

async def send_message(websocket, eventJSON) -> None:
    gameID = int(json.loads(eventJSON)["gameID"])
    try:
        connected = games.GAMES[gameID]
        for websocket in connected:
            await websocket.send(eventJSON)
    except KeyError:
        print(f"Error could not find {gameID}")
        event = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(event))
    return

async def error(websocket, eventJSON) -> None:
    print("Error handling event")
    return
```

Figure 5.46: Code that handles chatroom-specific events

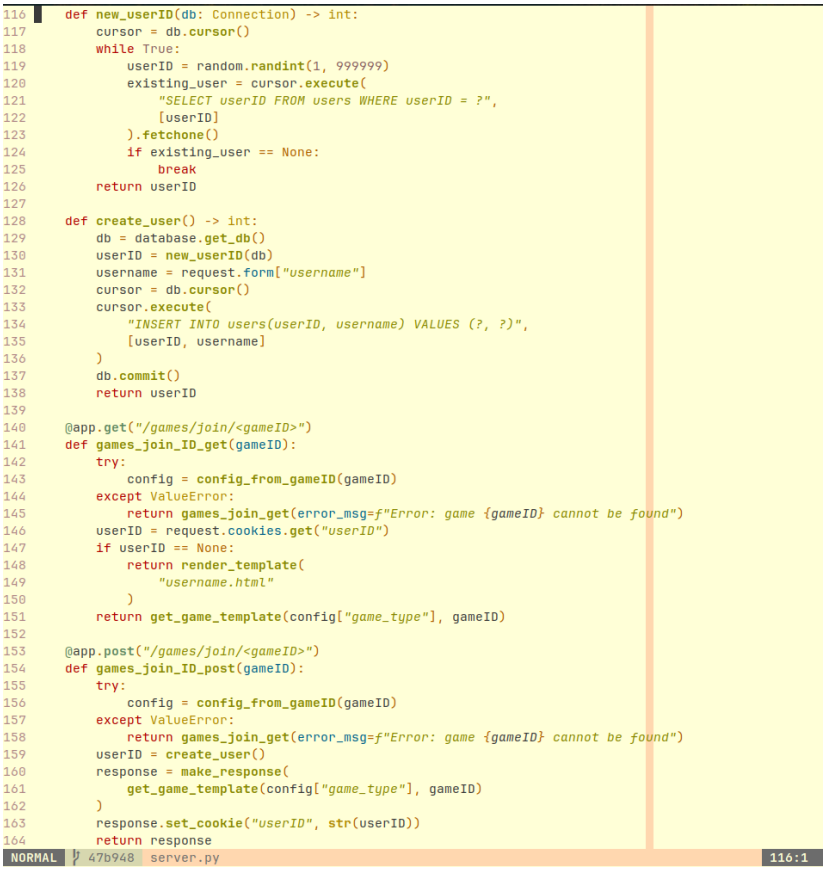
Version 3

```
28
29  def new_gameID(db: Connection) -> int:
30      cursor = db.cursor()
31      while True:
32          gameID = random.randint(1, 999999)
33          existing_game = cursor.execute(
34              "SELECT gameID FROM games WHERE gameID = ?",
35              [gameID]
36          ).fetchone()
37          if existing_game == None:
38              break
39      return gameID
40
41  def create_config(form):
42      game_type = form["game_type"]
43      match game_type:
44          case "whist":
45              return {
46                  "game_type": game_type,
47                  "scoring": form["scoring"],
48                  "length": form["length"],
49              }
50          case _:
51              return {"game_type": game_type}
52
53  def create_game() -> int:
54      db = database.get_db()
55      gameID = new_gameID(db)
56      config = create_config(request.form)
57      configJSON = json.dumps(config)
58      cursor = db.cursor()
59      cursor.execute(
60          "INSERT INTO games(gameID, config) VALUES (?, ?)",
61          [gameID, configJSON]
62      )
63      db.commit()
64      return gameID
65
66  async def ws_create_game(gameID):
67      async with websockets.connect("ws://127.0.0.1:8001") as ws:
68          data = {
69              "type": "create",
70              "gameID": gameID,
71          }
72          await ws.send(json.dumps(data))
```

Figure 5.47: The web server uses the websocket server to create games

```python

@app.post("/games/new")
def games_new_post():
    game_type = request.form["game_type"]
    if game_type not in ["chatroom", "whist"]:
        return "Unsupported game {}".format(game_type), 400
    gameID = create_game()
    asyncio.run(ws_create_game(gameID))
    return redirect(f"/games/join/{gameID}")

@app.get("/games/join/")
def games_join_get(error_msg=""):
    return render_template("games_join.html", error=error_msg)

def get_game_template(game_type, gameID, configJSON=""):
    match game_type:
        case "chatroom":
            return render_template(
                "chatroom.html", gameID=gameID
            )
        case "whist":
            return render_template(
                "whist.html", gameID=gameID
            )
        case _:
            return render_template(
                # This should be removed / replaced later with an error message
                "games_join_post.html", gameID=gameID, config=configJSON
            )

def config_from_gameID(gameID: int) -> dict[str, Any]:
    db = database.get_db()
    cursor = db.cursor()
    result = cursor.execute(
        "SELECT * FROM games WHERE gameID = ?",
        [gameID]
    ).fetchone()
    if result == None:
        raise ValueError
    configJSON = result["config"]
    config = json.loads(configJSON)
    return config

```

Figure 5.48: The web server uses the websocket server to create games



```python
def new_userID(db: Connection) -> int:
    cursor = db.cursor()
    while True:
        userID = random.randint(1, 999999)
        existing_user = cursor.execute(
            "SELECT userID FROM users WHERE userID = ?",
            [userID]
        ).fetchone()
        if existing_user == None:
            break
    return userID

def create_user() -> int:
    db = database.get_db()
    userID = new_userID(db)
    username = request.form["username"]
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users(userID, username) VALUES (?, ?)",
        [userID, username]
    )
    db.commit()
    return userID

@app.get("/games/join/<gameID>")
def games_join_ID_get(gameID):
    try:
        config = config_from_gameID(gameID)
    except ValueError:
        return games_join_get(error_msg=f"Error: game {gameID} cannot be found")
    userID = request.cookies.get("userID")
    if userID == None:
        return render_template(
            "username.html"
        )
    return get_game_template(config["game_type"], gameID)

@app.post("/games/join/<gameID>")
def games_join_ID_post(gameID):
    try:
        config = config_from_gameID(gameID)
    except ValueError:
        return games_join_get(error_msg=f"Error: game {gameID} cannot be found")
    userID = create_user()
    response = make_response(
        get_game_template(config["game_type"], gameID)
    )
    response.set_cookie("userID", str(userID))
    return response
```


Figure 5.49: The game uses cookies to identify users

```
12 async def handler(websocket):
13     async for eventJSON in websocket:
14         print(eventJSON)
15         event = json.loads(eventJSON)
16         match event["type"]:
17             case "create":
18                 await create(websocket, event)
19             case "join":
20                 await join(websocket, event)
21 
22         # Ensure the same DB connection is used rather than creating a new one
23         with server.app.app_context():
24             gameID = int(event["gameID"])
25             cursor = database.get_db().cursor()
26             result = cursor.execute(
27                 "SELECT config FROM games WHERE gameID = ?",
28                 [gameID]
29             ).fetchone()
30             config = json.loads(result["config"])
31         game_handler = get_game_handler(config["game"])
32         await game_handler(websocket, event)
33 
34 def get_game_handler(game_type):
35     async def error(*_):
36         print(f"Error could not find handler for '{game_type}'")
37 
38     map = {
39         "chatroom": handle_chatroom,
40         "whist": handle_whist,
41     }
42     try:
43         return map[game_type]
44     except KeyError:
45         return error
46 
```

Figure 5.50: Refactored `ws_server.py` that can handle both chatroom and Whist games

```python
import json

import games

WHIST = {}

async def handle_whist(websocket, event):
    match event["type"]:
        case "create":
            pass
        case "join":
            pass
        case _:
            await error(websocket, event)

async def waiting(websocket, event):
    gameID = int(event["gameID"])
    try:
        connected = games.get_websockets(gameID)
        response = {
            "type": "waiting",
            "players": len(connected),
            "players_required": 4
        }
        responseJSON = json.dumps(response)
        for websocket in connected:
            await websocket.send(responseJSON)
    except KeyError:
        print(f"Error could not find {gameID}")
        response = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(response))

async def error(websocket, event):
    print("Error handling event")
```

Figure 5.51: An incomplete handler for Whist games

Version 4

```python
import json

import games

WHIST = {}

async def handle_whist(websocket, event):
    match event["type"]:
        case "create":
            pass
        case "join":
            await join(websocket, event)
        case _:
            await error(websocket, event)

async def join(websocket, event):
    gameID = int(event["gameID"])
    # 2 players should not be able to join simultaneously so this should work
    if len(games.get_userIDs(gameID)) != 4:
        await waiting(websocket, event)
        return
    await test_game_state(websocket, event)

async def waiting(websocket, event):
    gameID = int(event["gameID"])
    try:
        connected = games.get_websockets(gameID)
        response = {
            "type": "waiting",
            "players": len(connected),
            "players_required": 4
        }
        responseJSON = json.dumps(response)
        for websocket in connected:
            await websocket.send(responseJSON)
    except KeyError:
        print(f"Error could not find {gameID}")
        response = {
            "type": "error",
            "message": f"Game {gameID} does not exist",
        }
        await websocket.send(json.dumps(response))

```

Figure 5.52: A basic handler for Whist games

```
43
44 async def test_game_state(websocket, event):
45     gameID = int(event["gameID"])
46     try:
47         connected = games.get_websockets(gameID)
48         response = {
49             "type": "game_state",
50             "players": [
51                 {
52                     "username": "matti",
53                 },
54                 {
55                     "username": "test1",
56                 },
57                 {
58                     "username": "test2",
59                 },
60                 {
61                     "username": "private",
62                 }
63             ]
64         }
65         responseJSON = json.dumps(response)
66         for websocket in connected:
67             await websocket.send(responseJSON)
68     except KeyError:
69         print(f"Error could not find {gameID}")
70         response = {
71             "type": "error",
72             "message": f"Game {gameID} does not exist",
73         }
74         await websocket.send(json.dumps(response))
75
76 async def error(websocket, event):
77     print("Error handling event")
```

Figure 5.53: A basic handler for Whist games

```
response = {
    "type": "game_state",
    "players": [
        {
            "username": "matti",
            "bid": 4,
            "tricks_won": 2,
            "hand": [
                "", "", "", "",
            ],
        },
        {
            "username": "test1",
            "bid": 2,
            "tricks_won": 0,
            "hand": [
                "6H", "JD", "7H", "3S",
            ],
        },
        {
            "username": "test2",
            "bid": 5,
            "tricks_won": 3,
            "hand": [
                "KC", "2C", "8S", "TS",
            ],
        },
        {
            "username": "private",
            "bid": 2,
            "tricks_won": 1,
            "hand": [
                "8D", "9S", "QD", "5C",
            ],
        }
    ],
    "community_cards": [
        "AC", "3C", "KD",
    ],
    "trump_suit": "H",
}
```

Figure 5.54: A complete test game state, including community cards used in poker

```
23 
24 function renderTable(event) {
25   const tableTemplate = document.getElementById("table_template");
26   const playerTemplate = document.getElementById("player_template");
27   const linebreak = document.getElementById("flex_linebreak_template");
28 
29   let table = tableTemplate.content.cloneNode(true).querySelector(".table");
30   table.id = "table";
31 
32   let players = [];
33   for (let i = 0; i < 4; i++) {
34     let fragment = playerTemplate.content.cloneNode(true);
35     let wrapper = fragment.querySelector(".player");
36     wrapper.id = "player" + i;
37     wrapper.style.flexFlow = "row wrap";
38     players.push(wrapper);
39   }
40 
41   let html = [
42     players[0],
43     linebreak.content.cloneNode(true),
44     players[1], table, players[2],
45     linebreak.content.cloneNode(true),
46     players[3]
47   ];
48 
49   content.replaceChildren(...html);
50 }
```

Figure 5.55: Javascript code to render a game state

```
51 
52 function renderTableInfoBox(event) {
53   let table = document.getElementById("table");
54 
55   let messages = []
56   // I am not sure if this is the place where the code should branch. The
57   // alternative is to have separate methods for rendering a waiting scene and a
58   // normal gameplay scene, as I do currently, and have these either call
59   // different methods or pass arguments to the methods they call (such as this
60   // one) that changes what is outputted, rather than the method deciding
61   // itself.
62   if (event.type === "waiting") {
63     messages = [
64       "Waiting for players to join...",
65       `${event.no_players} out of ${event.players_required}`,
66     ];
67   } else {
68     let map = {
69       "C": "Clubs",
70       "D": "Diamonds",
71       "H": "Hearts",
72       "S": "Spades",
73     };
74     let trump_suit = map[event.trump_suit]
75     messages = [`Trump Suit: ${trump_suit}`];
76   }
77   addMessages(table, ...messages);
78 }
```

Figure 5.56: Javascript code to render a game state

```
79
80 function renderPlayerInfoBoxes(event) {
81   for (let i in event.players) {
82     let infoBox = document.createElement("div");
83     infoBox.style.margin = "0.5em";
84     infoBox.className += "info_box";
85
86     let player = event.players[i]
87     addMessages(
88       infoBox,
89       `<strong>${player.username}</strong>`,
90       `Bid: ${player.bid}`,
91       `Tricks Won: ${player.tricks_won}`
92     );
93     const wrapper = document.getElementById("player" + i);
94     const hand = wrapper.querySelector(".hand");
95     wrapper.insertBefore(infoBox, hand);
96   }
97 }
98
99 function getCardHTML(cardName) {
100   const wrapper = document.createElement("div");
101   let url;
102   if (cardName === "") {
103     url = `/static/cards-fancy/2B.svg`;
104   } else {
105     url = `/static/cards-fancy/${cardName}.svg`;
106   }
107   fetch(url).then((response) => {
108     // Add error handling here later
109     return response.text();
110   }).then((text) => {
111     // The card svg has an approx. aspect ratio of 100:72
112     wrapper.style.width = `${5 * 0.72}em`;
113     wrapper.style.height = "5em";
114     wrapper.innerHTML = text;
115   })
116   return wrapper;
117 }
118
```

Figure 5.57: Javascript code to render a game state

```
118
119 function renderHands(event) {
120   const content = document.getElementById("content");
121   const linebreak = document.createElement("div");
122   linebreak.className += "flex_linebreak";
123   linebreak.style.marginBottom = "1em";
124   content.appendChild(linebreak);
125   for (let i in event.players) {
126     player = event.players[i];
127     const playerDiv = document.getElementById("player" + i)
128     const handWrapper = playerDiv.querySelector(".hand");
129     const linebreak = document.createElement("div");
130     linebreak.className += "flex_linebreak";
131     playerDiv.insertBefore(linebreak, handWrapper);
132     for (let card of player.hand) {
133       // Queries the server for the corresponding svg file
134       handWrapper.appendChild(getCardHTML(card));
135     }
136   }
137 }
138
139 function renderCommunityCards(event) {
140   const communityCards = document.querySelector(".community_cards");
141   for (let card of event.community_cards) {
142     communityCards.appendChild(getCardHTML(card));
143   }
144 }
145
146 function renderGameState(event) {
147   renderTable(event);
148   renderTableInfoBox(event);
149   renderPlayerInfoBoxes(event);
150   renderHands(event);
151   renderCommunityCards(event);
152 }
153
154 function renderWaiting(event) {
155   renderTable(event);
156   renderTableInfoBox(event);
157   renderPlayerInfoBoxes(event);
158 }
159
```

Figure 5.58: Javascript code to render a game state

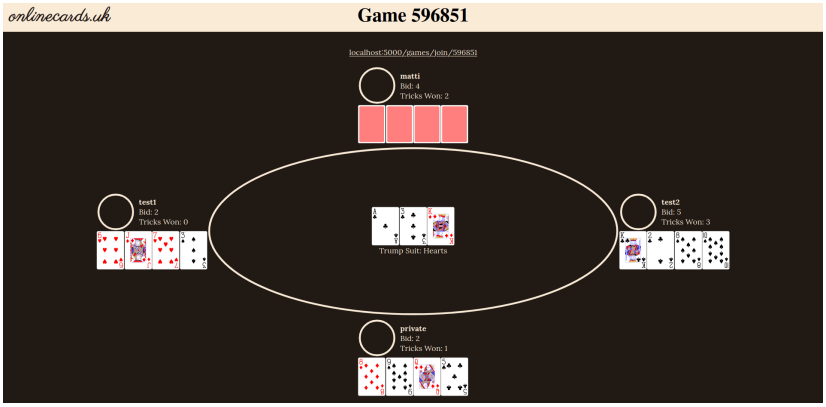


Figure 5.59: The test state rendered by the browser

Version 5

```
 97
 98 def create_deck_default(shuffle=True) -> list[str]:
 99     deck: list[str] = []
100     for suit in ["C", "D", "H", "S"]:
101         for i in range(1, 14):
102             if i in range(2, 10):
103                 card = str(i) + suit
104                 deck.append(card)
105                 continue
106             match(i):
107                 case 1:
108                     card = "A" + suit
109                 case 10:
110                     card = "T" + suit
111                 case 10:
112                     card = "T" + suit
113                 case 11:
114                     card = "J" + suit
115                 case 12:
116                     card = "Q" + suit
117                 case 13:
118                     card = "K" + suit
119                 case _:
120                     raise ValueError
121             deck.append(card)
122     if shuffle:
123         random.shuffle(deck)
124     return deck
125
126 def set_partners_default(players: list[dict[str, Any]]) -> None:
127     random.shuffle(players)
128     partners = [2, 3, 0, 1]
129     for i, player in enumerate(players):
130         player["partner"] = partners[i]
131
132 def deal_hand_default(game_state: dict[str, Any]) -> None:
133     deck = game_state["deck"]
134     players = game_state["players"]
135     while len(deck) != 0:
136         for player in players:
137             try:
138                 player["hand"].append(deck.pop())
139             except KeyError:
140                 player["hand"] = [ deck.pop() ]
141
```

Figure 5.60: An initial game state is created and broadcast

```
141 
142 def initialize_default(game_state: dict[str, Any]) -> None:
143     game_state["deck"] = create_deck_default()
144     set_partners_default(game_state["players"])
145     deal_hand_default(game_state)
146 
147 async def func_default(gameID: int, game_state: dict[str, Any]) -> None:
148     initialize_default(game_state)
149     await broadcast_game_state(gameID, game_state)
150     print(game_state)
151 
152 def get_whist_func() -> Callable:
153     return func_default
```

Figure 5.61: An initial game state is created and broadcast

Version 6

```
 1 import json
 2 
 3 from typing import Any, Callable
 4 
 5 from server import app
 6 import handlers.utils as utils
 7 
 8 def default(
 9     censored_state: dict[str, Any],
10     game_state: dict[str, Any],
11     key: str,
12     userID: int,
13 ) -> None:
14     """
15     Copy the key / value pair to censored_state from game_state.
16     """
17     censored_state[key] = game_state[key]
18 
19 def default_list(
20     censored_state: list[dict[str, Any]],
21     game_state: list[dict[str, Any]],
22     key: str,
23     userID: int,
24 ) -> None:
25     """
26     Given 2 lists, copy the key / value pair for every item.
27     """
28     for i, item in enumerate(game_state):
29         censored_item = censored_state[i]
30         censored_item[key] = item[key]
31 
32 def ignore(
33     *_
34 ) -> None:
35     return
36 
```

Figure 5.62: Python code to censor a game state for a user with support for variations

```python

def censor_hands(
    censored_state: dict[str, Any],
    game_state: dict[str, Any],
    key: str,
    userID: int,
) -> None:
    for i, player in enumerate(game_state["players"]):
        censored_player = censored_state["players"][i]
        if player["userID"] == userID:
            censored_player["hand"] = player["hand"]
        else:
            censored_player["hand"] = [ "" for _ in player["hand"] ]

def censor_hands_first_trick(
    censored_state: dict[str, Any],
    game_state: dict[str, Any],
    key: str,
    userID: int,
) -> None:
    """
    Deal the final card to the dealer face up.
    """
    for i, player in enumerate(game_state["players"]):
        censored_player = censored_state["players"][i]
        if player["userID"] == userID:
            censored_player["hand"] = player["hand"]
        elif i == game_state["dealer"]:
            censored_player["hand"] = [ "" for _ in player["hand"] ]
            censored_player["hand"].pop()
            censored_player["hand"].append(player["hand"][-1])
        else:
            censored_player["hand"] = [ "" for _ in player["hand"] ]

def censor_userIDs(
    censored_state: dict[str, Any],
    game_state: dict[str, Any],
    key: str,
    userID: int,
) -> None:
    for i, player in enumerate(game_state["players"]):
        censored_player = censored_state["players"][i]
        username: str = utils.get_username(player["userID"], app)
        censored_player["username"] = username

```

Figure 5.63: Python code to censor a game state for a user with support for variations

```python

def create_players_func(
        censor: dict[str, Callable],
        add_funcs: list[Callable]
    ) -> Callable:

    def censor_players(
        censored_state: dict[str, Any],
        game_state: dict[str, Any],
        _: str,
        userID: int,
    ) -> None:
        censored_state["players"] = [ {} for _ in game_state["players"] ]
        # Assuming that all players have identical keys
        for key in game_state["players"][0]:
            if key not in censor:
                default_list(
                    censored_state["players"],
                    game_state["players"],
                    key,
                    userID
                )
                continue
            func = censor[key]
            func(censored_state, game_state, key, userID)
        for func in add_funcs:
            func(censored_state, game_state, userID)
        # There may be a more efficient approach for this
        player: dict[str, Any]
        for player in game_state["players"]:
            if player["userID"] == userID:
                censored_state["current_user"] = game_state["players"].index(player)
    return censor_players

def set_event_type(event_type: str) -> Callable:
    def event_type_setter(
        censored_state: dict[str, Any],
        *_,
    ) -> None:
        censored_state["type"] = event_type
    return event_type_setter

```

Figure 5.64: Python code to censor a game state for a user with support for variations

```python

def create_main_func(
        censor: dict[str, Callable],
        add_funcs: list[Callable]
    ) -> Callable:

    def censor_main(game_state: dict[str, Any], userID: int) -> str:
        censored_state: dict[str, Any] = {}
        key: str
        for key in game_state:
            if key not in censor:
                # Copy the key/value pair into the `censored_state`
                default(censored_state, game_state, key, userID)
                continue
            # Otherwise mutate the `censored_state` by calling a replacement / variation function
            func = censor[key]
            func(censored_state, game_state, key, userID)
        for func in add_funcs:
            func(censored_state, game_state, userID)
        return json.dumps(censored_state)

    return censor_main

```

Figure 5.65: Python code to censor a game state for a user with support for variations

```python

def get_whist_censor_func(variation: str) -> Callable:
    match variation:
        case "first_trick":
            # Show the dealers last card which decides the trump suit
            censor: dict[str, Callable] = {
                "hand": censor_hands_first_trick,
                "userID": censor_userIDs,
            }
            add_funcs: list[Callable] = []
            censor_players = create_players_func(censor, add_funcs)
            censor: dict[str, Callable] = {
                "players": censor_players,
                "func": ignore,
                "deck": ignore,
            }
            add_funcs: list[Callable] = [set_event_type("game_state")]
            return create_main_func(censor, add_funcs)
        case _:
            censor: dict[str, Callable] = {
                "hand": censor_hands,
                "userID": censor_userIDs,
            }
            add_funcs: list[Callable] = []
            censor_players = create_players_func(censor, add_funcs)
            censor: dict[str, Callable] = {
                "players": censor_players,
                "func": ignore,
                "deck": ignore,
            }
            add_funcs: list[Callable] = [set_event_type("game_state")]
            return create_main_func(censor, add_funcs)
```

Figure 5.66: Python code to censor a game state for a user with support for variations

Version 7

```
82
83 def get_valid_cards_default(game_state: dict[str, Any], player_index: int) -> list[str]:
84     player = game_state["players"][player_index]
85     trick = game_state["trick"]
86     lead_card = trick["played"][trick["lead"]]
87     if lead_card == None:
88         return player["hand"]
89     lead_suit = lead_card[1]
90     valid = []
91     for card in player["hand"]:
92         if card[1] == lead_suit:
93             valid.append(card)
94     if valid != []:
95         return valid
96     return player["hand"]
97
98 def play_card_default(
99         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
100    ) -> None:
101    userID = int(event["userID"])
102    index = None
103    for i, player in enumerate(game_state["players"]):
104        if player["userID"] == userID:
105            index = i
106            break
107    if index == None:
108        return
109    if index != game_state["trick"]["next_player"]:
110        return
111    card = event["choice"]["chosen"]
112    player = game_state["players"][index]
113    player["hand"].remove(card)
114    game_state["trick"]["played"][index] = card
115
```

Figure 5.67: An 'event-based' system for playing card games

```
115 
116 def get_trick_winner_default(
117         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
118     ) -> int:
119     def get_value(card):
120         try:
121             value = int(card[0])
122         except ValueError:
123             value = value_map[card[0]]
124         return value
125 
126     trick = game_state["trick"]
127     lead = trick["played"][trick["lead"]]
128     trump_suit = game_state["trump_suit"]
129     value_map = {
130         "T": 10,
131         "J": 11,
132         "Q": 12,
133         "K": 13,
134         "A": 14,
135     }
136     winner = trick["lead"]
137     winning_card = lead
138     for i, card in enumerate(trick["played"]):
139         suit = card[1]
140         value = get_value(card)
141         winning_suit = winning_card[1]
142         winning_value = get_value(winning_card)
143         if suit == winning_suit and value > winning_value:
144             winner = i
145             continue
146         if suit != trump_suit:
147             continue
148         if winning_suit != trump_suit:
149             winner = i
150             continue
151         if value > winning_value:
152             winner = i
153             continue
154     return winner
155 
156 def get_lead_default(gameID: int, game_state: dict[str, Any]) -> int:
157     return 1
```

Figure 5.68: An ‘event-based’ system for playing card games

```
158 
159 def play_trick_default(gameID: int, game_state: dict[str, Any]):
160     lead = get_lead_default(gameID, game_state)
161     game_state["trick"] = {
162         "lead": lead,
163         "played": [ None for _ in game_state["players"] ],
164         "next_player": lead,
165     }
166 
167 def check_trick_default(
168         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
169     ) -> None:
170     trick = game_state["trick"]
171     if None in trick["played"]:
172         next = trick["next_player"] + 1
173         if next == len(game_state["players"]):
174             next = 0
175         trick["next_player"] = next
176     else:
177         winner = get_trick_winner_default(gameID, game_state, event)
178         game_state["players"][winner]["tricks_won"] += 1
179         trick["prev_winner"] = winner
180 
181 async def ask_card_default(gameID: int, game_state: dict[str, Any]):
182     player_index = game_state["trick"]["next_player"]
183     event = {
184         "type": "choice",
185         "choice": {
186             "type": "play_card",
187             "options": get_valid_cards_default(game_state, player_index),
188         },
189     }
190     player = game_state["players"][player_index]
191     websocket = utils.get_websocket(player["userID"])
192     await websocket.send(json.dumps(event))
193 
194 async def choice_default(
195         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
196     ) -> None:
197     match event["choice"]["type"]:
198         case "play_card":
199             play_card_default(gameID, game_state, event)
200             return
201         case _:
202             return
203 
```

Figure 5.69: An ‘event-based’ system for playing card games

```python

async def waiting_default(
        gameID: int, game_state: dict[str, Any], event: dict[str, Any]
    ) -> None:
    players: list[dict[str, Any]] = game_state["players"]
    response = {
        "type": "waiting",
        "no_players": len(players),
        "players_required": 4,
        "players": players,
    }
    await broadcast_game_state(gameID, response)

```

Figure 5.70: An 'event-based' system for playing card games

```python

def get_whist_state_handler() -> Callable:
    async def state_handler_default(
        gameID: int, game_state: dict[str, Any], event: dict[str, Any]
    ) -> None:
        # Call setup and teardown functions after each event
        match event["type"]:
            case "waiting":
                return
            case "start":
                play_trick_default(gameID, game_state)
                await ask_card_default(gameID, game_state)
                return
            case "choice":
                match event["choice"]["type"]:
                    case "play_card":
                        check_trick_default(gameID, game_state, event)
                        await broadcast_game_state(gameID, game_state)
                        await ask_card_default(gameID, game_state)
                    case _:
                        return
            case _:
                return

    return state_handler_default

def get_whist_event_handler() -> Callable:
    def error(event):
        return lambda *_: print(f"Error could find handler for event {event}")

    def func_default(event: str) -> Callable:
        try:
            return events[event]
        except KeyError:
            return error(event)

    events: dict[str, Callable] = {
        "start": initialize_default,
        "waiting": waiting_default,
        "choice": choice_default,
    }

    return func_default

```

Figure 5.71: An 'event-based' system for playing card games

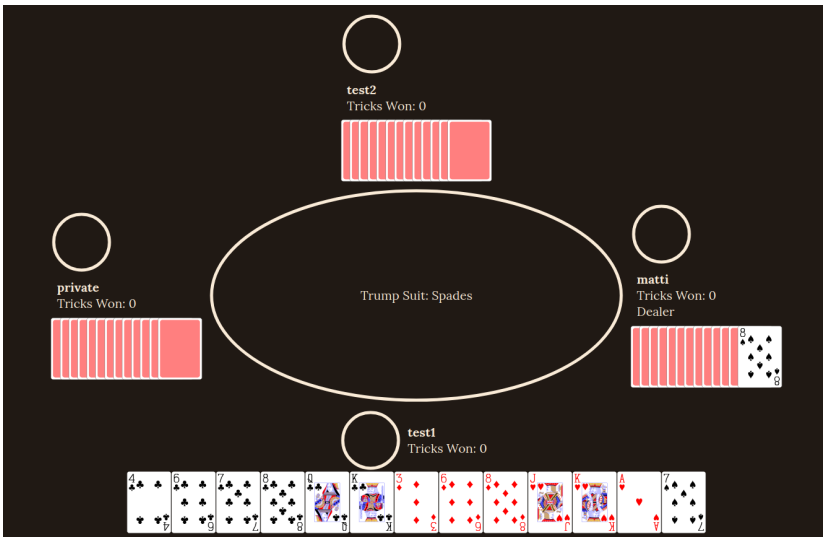


Figure 5.72: Users playing a trick in Whist

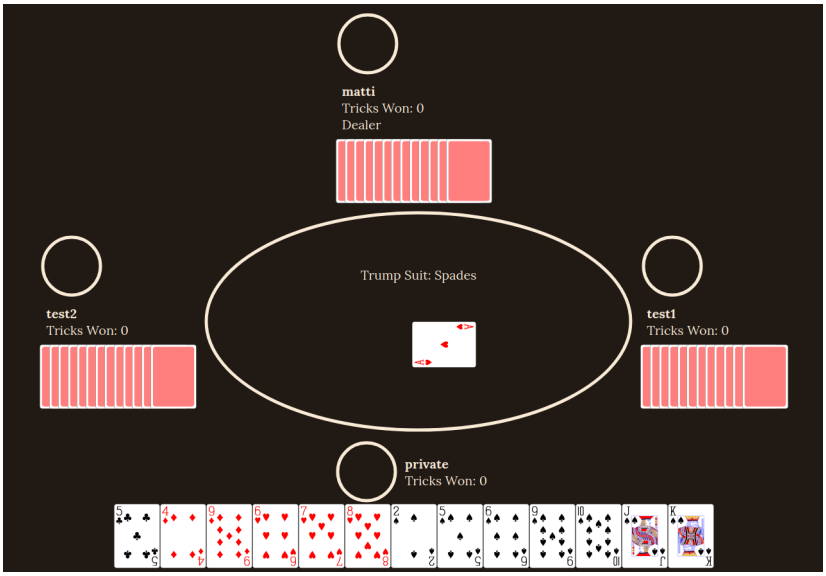


Figure 5.73: Users playing a trick in Whist

Version 8

```
304
305 def get_whist_state_handler() -> Callable:
306     async def state_handler_default(
307         gameID: int, game_state: dict[str, Any], event: dict[str, Any]
308     ) -> None:
309         # Call setup and teardown functions after each event
310         match event["type"]:
311             case "waiting":
312                 return
313
314             case "start":
315                 play_trick_default(gameID, game_state)
316                 await ask_card_default(gameID, game_state)
317
318             case "choice":
319                 match event["choice"]["type"]:
320                     case "play_card":
321                         end = check_trick_end_default(gameID, game_state, event)
322                         await broadcast_game_state(gameID, game_state)
323                         if not end:
324                             await ask_card_default(gameID, game_state)
325                         else:
326                             event["type"] = "end_trick"
327                             print('event["type"]', event["type"])
328                             await state_handler_default(gameID, game_state, event)
329
330                     case _:
331                         return
```

Figure 5.74: A completed `state_handler` for Whist

```
332 
333             case "end_trick":
334                 end = check_hand_end_default(gameID, game_state)
335                 if not end:
336                     play_trick_default(gameID, game_state)
337                     await broadcast_game_state(gameID, game_state)
338                     await ask_card_default(gameID, game_state)
339                 else:
340                     event["type"] = "end_hand"
341                     print('event["type"]', event["type"])
342                     await state_handler_default(gameID, game_state, event)
343 
344             case "end_hand":
345                 update_points_default(gameID, game_state)
346                 await broadcast_scoreboard(gameID, game_state)
347                 end = check_game_end_default(gameID, game_state)
348                 # TODO remove logging once fully tested
349                 if not end:
350                     print("game not done")
351                     initialize_hand_default(gameID, game_state)
352                     play_trick_default(gameID, game_state)
353                     await broadcast_game_state(gameID, game_state)
354                     print("trick initialised")
355                     await ask_card_default(gameID, game_state)
356                     print("asked card")
357                 else:
358                     event["type"] = "end_game"
359                     print('event["type"]', event["type"])
360                     await state_handler_default(gameID, game_state, event)
361 
362             case "end_game":
363                 print('event["type"]', event["type"])
364                 await broadcast_scoreboard(
365                     gameID, game_state, "game_end"
366                 )
367 
368             case _:
369                 return
370 
371     return state_handler_default
372 
```

Figure 5.75: A completed `state_handler` for Whist

Version 9

```
317 
318 def get_whist_state_handler(config: dict[str, Any]) -> Callable:
319     async def default(
320             *_
321         ) -> None:
322         return
323 
324     async def start(
325             gameID, game_state, event
326         ) -> None:
327         play_trick_default(gameID, game_state)
328         await ask_card_default(gameID, game_state)
329 
330     def get_choice_handler(config: dict[str, Any]) -> Callable:
331         # This can recurse as many times as necessary for more granular control
332         async def default(
333                 *_
334             ) -> None:
335             return
336 
337         async def play_card(
338                 gameID, game_state, event
339             ) -> None:
340             end = check_trick_end_default(gameID, game_state, event)
341             await broadcast_game_state(gameID, game_state)
342             if not end:
343                 await ask_card_default(gameID, game_state)
344             else:
345                 event["type"] = "end_trick"
346                 print('event["type"]', event["type"])
347                 await state_handler_default(gameID, game_state, event)
348 
349         map: dict[str, Callable] = {
350             "play_card": play_card,
351             "_": default,
352         }
353 
354         async def choice_handler(
355                 gameID: int, game_state: dict[str, Any], event: dict[str, Any]
356             ):
357             func = map.get(event["choice"]["type"], map["_"])
358             await func(gameID, game_state, event)
359 
360         return choice_handler
361 
```

Figure 5.76: The final version of the Whist `state_handler`

```python

    async def end_trick(
            gameID, game_state, event
        ) -> None:
        end = check_hand_end_default(gameID, game_state)
        if not end:
            play_trick_default(gameID, game_state)
            await broadcast_game_state(gameID, game_state)
            await ask_card_default(gameID, game_state)
        else:
            event["type"] = "end_hand"
            await state_handler_default(gameID, game_state, event)


    def get_end_hand(config: dict[str, Any]) -> Callable:
        # This structure can be copied to implement more variations
        match config["scoring"]:
            case "american_short_whist":
                game_end_func = check_game_end_american_short_whist
            case _:
                game_end_func = check_game_end_default

        async def end_hand(
                gameID, game_state, event
            ) -> None:
            update_points_default(gameID, game_state)
            await broadcast_scoreboard(gameID, game_state)
            end = game_end_func(gameID, game_state)
            if not end:
                initialize_hand_default(gameID, game_state)
                play_trick_default(gameID, game_state)
                await broadcast_game_state(gameID, game_state)
                await ask_card_default(gameID, game_state)
            else:
                event["type"] = "end_game"
                print('event["type"]', event["type"])
                await state_handler_default(gameID, game_state, event)

        return end_hand

    async def end_game(
            gameID, game_state, event
        ) -> None:
        print('event["type"]', event["type"])
        await broadcast_scoreboard(
            gameID, game_state, "game_end"
        )

```

Figure 5.77: The final version of the Whist `state_handler`

```python

    # Potentially each of these functions should have a standard interface
    # `get_X_handler(config)`, but currently functions that do not have
    # variations have their defaults hard-coded
    map: dict[str, Callable] = {
        "start": start,
        "choice": get_choice_handler(config),
        "end_trick": end_trick,
        "end_hand": get_end_hand(config),
        "end_game": end_game,
        "_": default,
    }

    async def state_handler_default(
        gameID: int, game_state: dict[str, Any], event: dict[str, Any]
    ) -> None:
        # Call setup and teardown functions after each event
        func = map.get(event["type"], map["_"])
        await func(gameID, game_state, event)

    return state_handler_default

```

Figure 5.78: The final version of the Whist `state_handler`

Prototypes

```python

def game_state_for_user(game_state: dict[str, Any], userID: int) -> str:
    copy: dict[str, Any] = {"type": "game_state"}
    for key, value in game_state.items():
        if key != "players":
            copy[key] = value
            continue
        if key == "players":
            copy["players"] = []
            for player in game_state["players"]:
                copy_player = {}
                for k, v in player.items():
                    if k not in ["hand", "userID"]:
                        copy_player[k] = v
                        continue
                    if k == "userID":
                        copy_player["username"] = games.get_username(v, server.app)
                        continue
                    if player["userID"] == userID:
                        copy_player["hand"] = player["hand"]
                        copy["current_user"] = game_state["players"].index(player)
                    else:
                        copy_player["hand"] = [ "" for _ in player["hand"] ]
                    copy["players"].append(copy_player)
    print(f"{userID} {copy}")
    return json.dumps(copy)

```

Figure 5.79: The first version of what would become the Whist `censor_func`

```
72 
73 def censor_game_state(game_state: dict[str, Any], userID: int) -> str:
74     def default(
75         censored_state: dict[str, Any],
76         game_state: dict[str, Any],
77         key: str,
78         userID: int,
79     ) -> None:
80         censored_state[key] = game_state[key]
81 
82     def censor_hand(
83         censored_player: dict[str, Any],
84         player: dict[str, Any],
85         hand: str,
86         userID: int,
87     ) -> None:
88         if player["userID"] == userID:
89             censored_player[hand] = player[hand]
90         else:
91             censored_player[hand] = [ "" for _ in player[hand] ]
92 
93     def censor_userID(
94         censored_player: dict[str, Any],
95         player: dict[str, Any],
96         userID: str,
97         current_userID: int,
98     ) -> None:
99         censored_player["username"] = games.get_username(player[userID], server.app)
100 
101     def censor_player(
102         censored_players: list[dict[str, Any]],
103         player: dict[str, Any],
104         _,
105         userID: int,
106     ) -> None:
107         censored_player: dict[str, Any] = {}
108         censor: dict[str, Callable] = {
109             "hand": censor_hand,
110             "userID": censor_userID,
111         }
112         key: str
113         for key in player:
114             if key not in censor:
115                 default(censored_player, player, key, userID)
116                 continue
117             func = censor[key]
118             func(censored_player, player, key, userID)
119         censored_players.append(censored_player)
120 
```

Figure 5.80: A Whist `censor_func` that uses a `dict` that links each key to a `censor` function

```
120
121     def censor_players(
122         censored_state: dict[str, Any],
123         game_state: dict[str, Any],
124         players: str,
125         userID: int,
126     ) -> None:
127         censored_state["players"] = []
128         player: dict[str, Any]
129         for player in game_state[players]:
130             censor_player(censored_state["players"], player, "", userID)
131             if player["userID"] == userID:
132                 censored_state["current_user"] = game_state["players"].index(player)
133
134     censored_state: dict[str, Any] = {"type": "game_state"}
135     censor: dict[str, Callable] = {
136         "players": censor_players,
137     }
138     key: str
139     for key in game_state:
140         if key not in censor:
141             default(censored_state, game_state, key, userID)
142             continue
143         func = censor[key]
144         func(censored_state, game_state, key, userID)
145     print()
146     print(f"{userID} {censored_state}")
147     return json.dumps(censored_state)
148
```

Figure 5.81: A Whist `censor_func` that uses a `dict` that links each key to a `censor` function

```
13 
14 def censor_game_state(game_state: dict[str, Any], userID: int) -> str:
15     def default(
16         censored_state: dict[str, Any],
17         game_state: dict[str, Any],
18         key: str,
19         userID: int,
20     ) -> None:
21         """
22         Copy the key / value pair to censored_state from game_state.
23         """
24         censored_state[key] = game_state[key]
25 
26     def default_list(
27         censored_state: list[dict[str, Any]],
28         game_state: list[dict[str, Any]],
29         key: str,
30         userID: int,
31     ) -> None:
32         """
33         Given 2 lists, copy the key / value pair for every item.
34         """
35         for i, item in enumerate(game_state):
36             censored_item = censored_state[i]
37             censored_item[key] = item[key]
38 
39     def ignore(
40         censored_state: dict[str, Any],
41         game_state: dict[str, Any],
42         key: str,
43         userID: int,
44     ) -> None:
45         return
46 
47     def censor_hands(
48         censored_state: dict[str, Any],
49         game_state: dict[str, Any],
50         key: str,
51         userID: int,
52     ) -> None:
53         for i, player in enumerate(game_state["players"]):
54             censored_player = censored_state["players"][i]
55             if player["userID"] == userID:
56                 censored_player["hand"] = player["hand"]
57             else:
58                 censored_player["hand"] = [ "" for _ in player["hand"] ]
59 
```

Figure 5.82: A Whist `censor_func` that ensures a consistent interface for each different variation

```python

    def censor_userIDs(
        censored_state: dict[str, Any],
        game_state: dict[str, Any],
        key: str,
        userID: int,
    ) -> None:
        for i, player in enumerate(game_state["players"]):
            censored_player = censored_state["players"][i]
            username: str = utils.get_username(player["userID"], server.app)
            censored_player["username"] = username

    def censor_players(
        censored_state: dict[str, Any],
        game_state: dict[str, Any],
        _: str,
        userID: int,
    ) -> None:
        censored_state["players"] = [ {} for _ in game_state["players"] ]
        # Assuming that all players have identical keys
        for key in game_state["players"][0]:
            censor: dict[str, Callable] = {
                "hand": censor_hands,
                "userID": censor_userIDs,
            }
            if key not in censor:
                default_list(
                    censored_state["players"],
                    game_state["players"],
                    key,
                    userID
                )
                continue
            func = censor[key]
            func(censored_state, game_state, key, userID)
        # There may be a more efficient approach for this
        player: dict[str, Any]
        for player in game_state["players"]:
            if player["userID"] == userID:
                censored_state["current_user"] = game_state["players"].index(player)

```

Figure 5.83: A Whist `censor_func` that ensures a consistent interface for each different variation

```
 99
100     censored_state: dict[str, Any] = {"type": "game_state"}
101     censor: dict[str, Callable] = {
102         "players": censor_players,
103         "func": ignore,
104         "deck": ignore,
105     }
106     key: str
107     for key in game_state:
108         if key not in censor:
109             default(censored_state, game_state, key, userID)
110             continue
111         func = censor[key]
112         func(censored_state, game_state, key, userID)
113     return json.dumps(censored_state)
114
```

Figure 5.84: A Whist `censor_func` that ensures a consistent interface for each different variation

Errors

```
def create_config(form):
    game_type = form["game_type"]
    match game_type:
        case "whist":
            return {     ■ "{" was not closed
                "game_type": game_type,
                "scoring": form["scoring"]
                "length": form["length"],      ■■■■ Code is unreachable
            }
        case _:
            return {"game_type": game_type}
```

```
def create_config(form):
    game_type = form["game_type"]
    match game_type:
        case "whist":
            return {
                "game_type": game_type,
                "scoring": form["scoring"],
                "length": form["length"],
            }
        case _:
            return {"game_type": game_type}
```

Figure 5.85: A syntax error caused by a missing comma

```
def create_game() -> int:
    db = database.get_db()
    gameID = new_gameID(db)
    config = create_config(request.form)
    configJSON = json.dumps(config)      ■ "configJSON" is not accessed
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO games(gameID, config) VALUES (?, ?)",
        [gameID, configuration]      ■ "configuration" is not defined
    )
    db.commit()
    return gameID
```

```
def create_game() -> int:
    db = database.get_db()
    gameID = new_gameID(db)
    config = create_config(request.form)
    configJSON = json.dumps(config)
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO games(gameID, config) VALUES (?, ?)",
        [gameID, configJSON]
    )
    db.commit()
    return gameID
```

Figure 5.86: A name error caused by changing the name or variable in all but one instance

```
<form id="chatroom" style="display: none" action="/games/new" method="POST">
  <select name="game_nam" style="display: none">
    <option value="chatroom" selected></option>
  </select>

  <div><input type="submit" value="Create Game"></div>
</form>
```

```
<form id="chatroom" style="display: none" action="/games/new" method="POST">
  <select name="game_type" style="display: none">
    <option value="chatroom" selected></option>
  </select>

  <div><input type="submit" value="Create Game"></div>
</form>
```

Figure 5.87: An error caused by changing 1 file without changing a second that depends on it

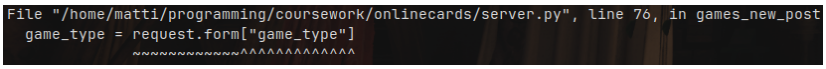


Figure 5.88: An error caused by changing 1 file without changing a second that depends on it

5.2 Interviews

5.2.1 Jamie

1. What do you most like about card games?

   Card games require a level of skill, problem solving and on the spot thinking which brings entertainment. I also like the various forms of card games available (e.g. Solitaire, Promise and Sevens).

2. What do you most dislike about card games?

   Many card games online have silly rules that you cannot edit. Also, some card games cannot be found online such as Promise.

3. What would make card games more convenient for you?

   Online playability where I can connect with friends and/or random players. A variety of card games would also be helpful.

4. What advantages do you think a computerized card game has?

   You can play at any time, with anyone. It also allows for people to access their favourite card games from wherever they are.

5. What would you improve from apps you have used?

   Some apps I have used in the past can be quite laggy and finding your friends on a "server" can prove near impossible.

6. Do you have any other ideas?

   If you could, make a function that allows you to pick which "table" you want to play your card game on so that you can play with your friends and/or random players.

5.2.2 Duncan

1. What do you most like about card games?

   I like card games because they are simple and easy to play but require a high level of skill to master - posing a challenge. I also like the thrill of winning big and overcoming taunts from my friends who may goad me just before a big win.

2. What do you most dislike about card games?

   I do not like that you have to be physically with someone to play a card game. The world is changing and in 2024 you should be able to play a card game with a customisable rule set and incentives to replay and win.

3. What would make card games more convenient for you?

   If card games were online and customisable I would be enamoured with it as I would be able to engage with my friends from the confines of my abode. Similarly, I am 17 and my friends will be going to university next year - I would like to play with them without the cost of a return train ticket or 3-hour car journey.

4. What advantages do you think a computerized card game has?

   I think that a computerised card game can increase the speed of play - when I play in person with my friends it takes a while for cards to be shuffled and organised. This time saving would be mitigated with the introduction of algorithms to take place for human shuffling. This also reduces the chance of rigging the deck to a very low possibility unless the programmer deliberately distributes a specific deck every x amount of passes where $x \in \mathbb{Z}^+$.

5. What would you improve from apps you have used?

   I would introduce further customisability and a multiplayer function - on my current game where I am restricted to play one standard version of 'Spades', I am forced to play against a computer. I would also introduce a dark/light theme which could be toggled in settings and a user guide.

6. Do you have any other ideas?

   No

5.3 Development log

During development I kept a log of my progress and discussed the issues that I found and how I designed solutions for them. Combined with `git` commit messages they provide the most complete log of the development, my thought process at the time, my priorities, etc. A copy of the log up until the final version is below:

Realised that GAME global variable, as recommended by websocket documentation would not work since global variables are local to their own modules, and so cannot be referenced across files. With that design I would also have 2 copies of the same data, one in the database on disk, and one in Python in memory. Instead I have opted to use only a database and have the Flask server initialise the game when it is created.

Thu 24 Oct 16:38:07 BST 2024:

This is also proving difficult since accessing the database from the `chatroom.py` file requires using flask requires a reference to the `current_app`.

I could do this by importing the app from `server.py` but this could cause a circular import if I then want to use code from `chatroom.py` in `server.py`.

Thu 24 Oct 16:56:17 BST 2024

Wondering if I should add error checking and type hints (recurring theme)

Thu 24 Oct 17:25:21 BST 2024

Running into `TypeError: cannot pickle '_asyncio.Future' object`. I am trying to serialise a set of websocket connections (using pickle), store them in an sqlite database and them retrieve them later. I will need to store them somewhere, and currently the backend creates a game by creating an row in a database with an ID, config and empty set of connected users. If I am going to use this method, I should have other parts of the program query the database to get the information they need about each game. For the chatroom, it currently only needs to know who has connected so that it can broadcast messages to every connected user. If I am going to use a database, as designed above, the chatroom would need to be able to get the connected users from the database. If this is not possible, I may have to rethink the design here.

Thu 24 Oct 17:44:33 BST 2024

I have mostly confirmed that websockets cannot be pickled. Source

> Objects with open web connections are the classic example of unpickleable objects. The problem isn't the 'complexity', its that you can't store something with an open web connection in memory and then retrieve the object in the same state.

This means that I will have to rethink the design of the of the database. Websockets can be held in memory, so I could keep the list of connections in a Python dict, using the gameID as a key. However, I would like to keep the database in some form rather than having all of its contents stored in memory. The redesign may include the database storing a gameID and configJSON, and a global variable in the `chatroom` module being a dict linking gameID to a set of websocket connections.

Thu 24 Oct 23:45:09 BST 2024

When creating a new game, the Flask server now uses a websocket to connect to the websocket server and sends a create game message. This is done using asyncio.run, which ironically was part of my first failed solution. The websocket recieves this message and creates an entry in the `GAMES` dict which links a `gameID` to the set of all connected

websockets. When a player joins, they are added to the set of connections and when a message is recieved, it is then rebroadcast to all connected websockets.

Mon 4 Nov 17:29:49 GMT 2024

I had noticed that when enough messages had been sent, they began to overlap with the input box and when scrolling down the input box did not scroll with the view. I also did not like the browser suggesting previously entered messages and usernames, but this was easily fixed with `autocomplete="off"`. I previously had the message form absolutely positioned relative to the `main` div and tried setting its position to be fixed. However the messages still overflowed and went beneath it. Then I tried setting a `max-height` for the message list before settling on setting the `height` to be `100vh - used_space` and positioning the `msg_form` beneath the `msg_list`.

Mon 4 Nov 20:38:13 GMT 2024

I have decided to push the latest version to production, deploying it on my Dad's old work laptop. I have ssh'd in, added my public key to the authorised keys and I am currently working on writing a bash script to launch the server. I have had to install some dependencies I had forgotten about (python's websocket package) and have been reminded of docker containers. I should create one (or look into python's venv solution) to help manage dependencies for the project.

Tue 5 Nov 18:45:27 GMT 2024

I have decided to refactor the websocket server. Previously I had a `chatroom.py` file that handled all websocket (ws) connections (since it was the only game that worked). However, with adding support for multiple games, I decided that I should have only a single ws server, but I also want the code for each type of game (whist, poker, etc.) in separate files. This means that I need some way of importing functions and running them, which is proving difficult. Python's websocket library is by default asynchronous which is part of the difficulty. `TypeError: 'coroutine' object is not callable` I also foresee that I will have the same challenges with importing global variables across files that I did when creating the server.

Wed 13 Nov 10:46:28 GMT 2024

I have successfully created a main `ws_server.py` file and a separate `chatroom.py` file for the websocket server. This was permitted by using a third `games.py` file to hold the `GAMES` dict that links a `gameID` to the set of connected websockets (ws) to prevent a circular import. Otherwise, `ws_server.py` would have to import functions to handle ws connections from `chatroom.py`, which would have to import the `GAMES` dict from the main server file. I also fixed the TypeError by awaiting

the function before attempting to use its return value.

Wed 13 Nov 12:23:08 GMT 2024

I am currently working on creating the network code required to get Whist working, which requires deciding on a serialisation format. I had this specified in a file `docs/serialisation.md` that I should likely commit to git, so that I have version history of it. However, I am discovering now that the initial design, where each message has a `gameID` and a `config` parameter is a poor design decision. This will increase the size of every message that has to be sent and if the server believes what it is sent then it could lead to exploitation. Instead I have decided that the `GAMES` global variable should link a `gameID` not only to the connected websockets, but to the game's `config`. This means that each packet does not need to send the `config` data for the game and the server can use a trusted source to get the `config`.

Wed 13 Nov 19:06:53 GMT 2024

I am working on new documentation after updating the serialisation format. I am also realising that I should take more steps to prevent or reduce cheating – which requires more server-side validation. For example, I should check that when a message is sent, even if the `gameID` is valid, the websocket is connected to that game. If someone has been barred from entering a game (they will not be added to the set of connections) they should not be able to influence it by sending messages the server believes is valid. I also think the database is no longer required since a game will work if and only if the websocket server has an entry for that `gameID`. Duplicating this information in a database and using it for some tasks and not others only causes more bugs.

Thu 14 Nov 10:04:13 GMT 2024

I am attempting to remove the database and use only the `GAMES` global variable to track which games have been created. I currently have 2 files, `server.py` and `ws_server.py` that are started separately to handle the Flask and Websocket servers respectively. However this means that they have different instances of the `GAMES` global variable, since they are started separately. If the architecture is changed so that the same file starts both servers, then they would share the same `GAMES` global. However this would require multithreading or some other asynchronous technique to allow both servers to run simultaneously. Alternatively, the Flask server could connect to and then query the websocket server for each request, to see if a game exists.

Thu 14 Nov 17:34:03 GMT 2024

I am reconsidering how the http and websocket servers interact. It is proving difficult to have both programs started simultaneously and sharing the same memory such as the `GAMES` global variable. I am now

wondering if that is even desirable as would make running the 2 servers on separate machines much more difficult. However, there does need to be some communication between the 2 servers which I am trying to redesign now. This is because I want to send different HTML based on whether the user attempted to join a game with a valid gameID or not.

Fri 15 Nov 19:16:26 GMT 2024

I have realised again that I need to perform more input validation. I should ensure that if the website is used correctly there are no errors, but I also need to protect it against malicious data and requests, which I am realising now.

Fri 15 Nov 2024

During a discussion with my stakeholders about my plans to remove the database, they argued that this would provide a worse experience for them. Whilst I was correct that an entry that existed in the database, but not in `ws_server.py`'s memory did not constitute a working game – it did in a user's mind. If they had created that game, they would expect it to exist, even if the server crashed and had to be restarted. Even further, a user might hope that the server could recover their game after a crash. Since I have planned to serialize the game state into JSON already, this makes storing it so that it can be recovered easier. This will also influence the design of the serialisation format. I have planned to have a `player` class that included a player's hand, points and other data, and link each user to an instance of this class. In order to identify users, I had planned to use websockets, however in the event of a crash or having to reconnect, the browser would create a new websocket connection, which would no longer link to their player. This could be fixed by instead linking each instance of the `player` class to a `userID`, which in turn links to a websocket. This means that the websocket could be replaced without breaking the game state. Whilst this is not an immediate priority, I think it is a good feature that I would like to add.

Sun 17 Nov 15:13:36 GMT 2024

I have managed to revert the project to a previous, functional state using git. The non-functional changes are now part of a commit, if I wish to look back on them, though I might want to add a warning that they are non-functional. Now I am looking to refactor some parts of the `ws_server` to better support multiple game types and more complex card games too.

Sun 17 Nov 19:34:39 GMT 2024

I have decided that the next step in development should be getting the chatroom to survive reloading, which it cannot currently. What I believe happens is that when the HTML is loaded, `chatroom.js` is

fetched from the server and ran. This causes a new Websocket to be created and connect to the `ws_server`. However, this means that the server now has 2 websockets for the same user, an old that does not exist and a new one for the current connection. Particularly since I am hoping to move towards identifying users by a `userID` I am going to make that change as part of the next step. I also need to find a way to detect when a user leaves or disconnects and call the relevant handler function to allow each game type to account for this. How this will relate to implementing a `userID` system I am not sure.

Sun 17 Nov 22:51:32 GMT 2024

Haven given it a bit of thought, I don't believe that getting server to be able to load games from a file on start up will be particularly difficult. The most basic version of creating entries in the `GAMES` global for each `gameID` in the database should be very easy and a feature that I will add soon. If I can successfully design a serialisation format that can turn the state of a card game into JSON then loading this should not be difficult either. Since this is already a requirement and something that I have created some mock-ups for I believe it is possible and recovering from it should not be particularly difficult. If this is to protect against crashes, the server should routinely write the state of each game into a file which may be difficult to implement. Initially I have 2 ideas: 1. At fixed intervals (5 minutes for example) the server stops processing and writes the state of every game into a single file (or maybe multiple). This would likely be simpler to implement but could cause significant lag spikes if there is a large number of games. Particularly if games are saved frequently there could also be frequent lag. 2. The server continuously iterates through each game saving it and then moving to the next one. Theoretically the compute spent writing games to the disk would be constant, preventing lag spikes. This would likely be most effective if the state of each game is stored in a separate file e.g. `game_states/<gameID>.json`. So that they can be written to separately. However ensuring that every game is saved every X minutes may be difficult and this method may require more time to implement correctly

Sun 17 Nov 23:06:32 GMT 2024

I am currently trying to implement a `userID` so that games can link to connected `userID`s and then to websockets so that the websocket can be replaced without breaking the game. This could prevent the same browser from joining multiple games. Deciding how I want to implement this is the current issue as while authentication for HTTP servers (using cookies) is quite standard, there is no such solution for websockets. The docs for the python package discusses this and gives recommendations, but I have not decided which approach I will take. I am also wondering if I should implement a traditional login system

with creating and verifying accounts first. I don't want to force a user to login, but it may be good practice for creating the userID system.

Tue 19 Nov 10:19:01 GMT 2024

After having committed the new cookie update, I have now remembered that I need to create a cookie notice to comply with GDPR. I have also realised that using a database to store `userID`s that have been given out is good for preventing accidental duplication of `userID`s, if the browser automatically deletes these cookies (since they are session cookies) the database will end up with lots of `userID`s that are no longer in use. Either I need to switch to a method of generating `userID`s that can create an (almost) limitless amount or find some way of recovering lost ones. The former is likely a better plan.

Tue 19 Nov 10:25:56 GMT 2024

I have managed to implement using `userID`s to track connected users rather than websockets. This means that when the page reloads the browser creates a new websocket and joins the game again, replacing their old connection. However the current design means the webpage resets their username. Like the config this is something that I will want to move server-side. Now that I have a database that for users, I can use add username as a column. I also want to prevent userID spoofing, but that will likely require encryption and it may be best if I use an external library like Flask-Login for this. When I add this, I will likely want to also ensure that the user is actually connected to a game before processing the request, to prevent gameID spoofing.

Tue 19 Nov 17:55:51 GMT 2024

I have remembered that I need to add error handling for unexpected websocket closing and attempting to send connections to websockets that no longer exist / cannot be reached. Earlier today I was also considering the interface that I should use for the functions. In my design I discussed how getting each function variation to use the same interface could be difficult. This is because different variations may require different information. However if all functions are given a copy of the complete game state, then they have access to all the information available, and all the information they could need. This feels slightly like a 'brute force' approach and may have some drawbacks, but I think it is likely the best solution.

Tue 19 Nov 18:17:46 GMT 2024

I have decided to move username into the database and that it should be fetched from the database rather than sent with each message. I also do not want users to be able to change their username by reloading the page. I am currently looking at how best to design this. With a typical login and user system, a user would set their username when

the sign up (along with their password, email etc.), however I do not want to force a user to sign up to use my app. This means that a guest user needs to be able to create or join a game and set their username at a convenient point. I think it makes the most sense for a user to set their username when they join a game. If they are entering a `gameID` by hand, I can add an extra input to the form to set a username at this point. However if they have clicked on a link, they should be shown a separate page where they can enter a username. When this form is submitted, the server can then create an entry in the `users` table and set the `userID` cookie. I could use the presence of this cookie to check whether a user needs to create a username and branch on it accordingly.

Wed 20 Nov 09:14:31 GMT 2024

I'm not sure if API is the correct phrase, but as I am writing the server-side code I am trying to design it in a way that makes easy for another programmer to contribute by adding another card game. This is why the main `ws_server.py` handles the creating, joining, (and in the (near) future) leaving of games, whilst still sending events to the individual handlers. This is also the intention behind splitting handlers into individual files and using a map to find the appropriate handler function. I may even decide to move the map into a separate file. This may make it easier to create a new handler as one would not have to dig through the `ws_server.py` source code to find the correct dict. This is also why I always pass the same 2 arguments to every function `websocket, event` so that another programmer can have a standard interface for their own code. The intention behind `games.get_userIDs(gameID)` and `games.get_websockets(gameID)` is to provide some abstraction for another programmer as well as encapsulate some parts of the game class. (It may be interesting to note that I am trying to use encapsulation despite not using object oriented programming. Whilst it may not be possible to prevent other parts of the program from accessing `GAMES` directly, I might still gain some of the advantages of the paradigm without using OOP).

Sat 23 Nov 14:59:56 GMT 2024

I have looked through some websites (online-spades.com, cardgames.io/spades, pokerpatio.com) to see how their user interfaces were created. Most seemed to use standard HTML elements, including lots of divs – rather than canvas (which I could also use). I think I am also going to use standard HTML elements, though I do want to add flexboxes in the future for mobile / vertical support. Given how dynamic I want the system to be (supporting any number of players) I will have to give some thought to how I am going to style the elements. Some styling should definitely be part of a css file, but the placement of a div may be unique. In this case it might be beneficial to add these styles directly to the element. Creating this in vanilla javascript may also introduce

some issues, since creating HTML can be quite tedious. This means I might want to try to use templates wherever possible (potentially sending these within the main HTML file with `display="none"` so they can be fetched / copied / modified by javascript later on (if this is possible)). Particularly since my current design is rather vague, and scope rather large, I think it is best if I create a simple, fully functioning example before I expand the scope or set anything in stone. In this spirit it is quite likely that the system will undergo significant change between the first working version and the second. This will be me redesigning the system so that it is more general, easy to use as I have done so with most parts of the program (for example encapsulating the `GAMES` global variable). Hopefully I can finish the docs on this feature as well (and document other parts of the program that contributors may want to add to).

ps. I should add this file and the docs to version control, in some manner

Sat 23 Nov 22:16:58 GMT 2024

I have managed to get 4 players (circles) placed evenly around a table (ellipse) using HTML and CSS. I am using a flex box with:

```
.table {
    display: flex;
    flex-flow: row wrap;
    justify-content: center;
    align-items: center;
}
```

which allows for multiple rows by using divs to create linebreaks

```
.flex_linebreak {
    width: 100%;
    height: 0;
}
```

However, I am not particularly happy with this solution since it does not easily allow for even distribution of different numbers of players. For example with 3 players they would be arranged like

```
            player 1
player 2     table
            player 3
```

rather than

```
            player 1
             table
    player 2       player 3
```

with even spacing around each player. This is not a significant issue but it may make having more than 4 players difficult. This could potentially be mitigated by having the table be a stadium shape (a rectangle with semi circles at each end) and having multiple players along the long edge. This could look better than an ellipse table particularly if players are not evenly distributed, so I may change to this later. Currently the "#table" div holds an .svg file which is the table ellipse (and may later include a logo). This makes it difficult to overlay a text box above the table. If I can set this .svg file to be the `background-image` this may make it easy to add a text box above the table. This would be used to display information that is public and affects all players, such as the trump suit in whist or the pot and community cards in poker. However this may only be possible if the image is moved into a separate file which may prevent it from referencing things defined by the browser, such as colour names and width percentages.

Fri 29 Nov 14:41:33 GMT 2024

I have encountered a bug for the second time where every user's info box has the first player's name added. This results in their own name and the first player's name being displayed. This occurred both times after changing browsers and reloading the page on that other browser. I am not sure what is causing the issue and it is currently low priority.

Fri 29 Nov 20:06:09 GMT 2024

After reading the MDN web docs, and trying to follow their recommendations (using semantic HTML as recommended by Mozilla) I learned that I should replace my `div id="template"` with an actual <`template`> tag. This removes the need to use `display: none` and better mirrors what I am actually trying to achieve. This may require some minor refactoring and replacing `template.cloneNode()` with `template.content.cloneNode()`, but it may come with some benefits. For one, I should probably change the id for the template and real node so that I can accurately `document.getElementById()`, which I am currently having issues with. I am likely going to use a naming scheme such as `...Template` so that I can automatically generate the real id by removing `Template` from the end of the id. For duplicates, such as multiple players, I can then append other identifiers, such as a number, giving `playerTemplate -> player1, player2, ....`

Sat 30 Nov 15:51:56 GMT 2024

I have finished most of the rendering logic for Whist (including adding support for community cards that will be needed for poker) but the last feature I need to add is focusing on the current user. I want the user to have their icon on the bottom of the screen and have the table rotated as needed to accommodate this.

```
user 1       user 2
```

```
        table
    [ current user ]
```

Currently each player receives an identical copy of the game state, but this will need to be changed in the future to reduce cheating. If a player is given the entire game state, they will be able to see other player's cards, even if they are not rendered, by parsing the raw input themselves. This means that each user will have to be sent a different game state containing the only information that they can see. Since the game state is changed based on which user it is sent to, it would be easy to change it to tell the renderer which is the current user. This means that the renderer could then focus on this user, changing whatever it needs to do to accommodate this.

There are 2 ways that I initially thought I could communicate who the current user is. Since each player is a dictionary, they could have an attribute `current_player: bool` that states if this player is the user's, but this seems that it would make the game state unnecessarily large. I could have this set only on the current player, but this would require that I handle errors when I check if this key exists on a player. Either of these variations would require that I search through each player to find the current one. Alternatively, I could give the game state another attribute `current_player_index: int` which is the index for the `players` list that gives the current player. This is likely the best solution (or good enough that any improvement will be minor).

Wed 4 Dec 16:13:57 GMT 2024

I have just come across a bug in my previous commit where the game freezes when creating a new game. It is attempting to rotate the player list till the current player is at the bottom, but `current_user` is not defined in this message. This has also raised more significant design questions about how I am going to broadcast game states. This problem becomes significantly more complex when each player needs to receive a different game state. My previous solution was rather hard-coded and I mentioned in the commit message that it needs to be updated. It seems that I will have to update that now and I am currently considering how I should design that.

Wed 11 Dec 16:13:07 GMT 2024

I am currently working on how I can broadcast game states and refactoring the code, however censoring a game state seems like a much more complex problem than I initially thought. The current plan is to begin with an empty `reduced_state`, iterate through every `key` / `value` pair in the game state and either copy them verbatim or call a censor function, passing the `value`, and set the `value` to what is returned. This works well for simple use cases where the function is mathematically pure – for example censoring an opponents hand – where the function

returns a list of the same length as the input, with empty strings instead of card names (`"AS" -> ""`, `"3D" -> ""`, `"5C" -> ""`, etc.). However this example also reveals the problems with this approach. In order to determine if this player's hand should be censored, we need to know if this player's `userID` matches the `userID` of the user we are sending this `reduced_state` to. This information isn't (and shouldn't) be stored in the game state and instead is passed in as an argument to the `censor_game_state` / `game_state_for_player` function. Another problem is that when censoring `userID`, it is used to get that user's `username` which is then added as a key (rather than `userID`) to the `reduced_game` state. Neither of these use cases are supported by the above model.

In order to provide maximum flexibility every function should be able to read and modify the entire game state, but this provides no guarantees about what each part of the system may be doing. For example the function `censor_hand` should not change a user's `"username"` but this guarantee cannot be provided if the function can read and write to the entire game state. This is likely to be a persistent problem across the entire codebase since I am planning to use this technique for the main game method because of the flexibility it provides.

Method 1:

```
def censor_players(players: dict[str, Any]) -> dict[str, Any]:
    reduced_players = players.copy()
    for key, value in players.items():
        # Decide whether to copy or mutate each key, value pair
        if ... :
            reduced_players[key] = value
        # This cannot read other parts of the `game_state` or what `userID` is
        ...
    return reduced_players

players: dict[str, Any] = censor_players(game_state[players])
reduced_state["players"] = players
```

Method 2:

```
def censor_players(
    reduced_state: dict[str, Any],
    game_state: dict[str, Any],
    userID: int
) -> None:
    for key, value in game_state["players"]:
        # Decide whether to copy or mutate each key, value pair
        if ... :
            reduced_state["players"][key] = value
        # This can read and write to other parts of the `game_state` and `userID`
```

```
        ...
    return

censor_players(reduced_state, game_state,  userID)
```

Fri 13 Dec 14:36:12 GMT 2024

I have now managed to refactor the code so that it follows Method 2 as detailed above. This was quite difficult, as I have already described, but once I had decided on the overall design things did seem to fall into place. I have also tried to ensure that the code follows a similar structure where possible, detailed below.

```
def default(
    censored_state: dict[str, Any],
    game_state: dict[str, Any],
    key: str,
    userID: int,
) -> None:
    # By default copy the key / value pair in its entirety
    censored_state[key] = game_state[key]

def censor_hand(
    censored_player: dict[str, Any],
    player: dict[str, Any],
    hand: str,
    userID: int,
) -> None:
    # Here we should copy the hand without changing it, if it's 'our' hand
    # and replace every card with an empty string, if it is another player's hand.
    if player["userID"] == userID:
        censored_player[hand] = player[hand]
    else:
        censored_player[hand] = [ "" for _ in player[hand] ]

censor: dict[str, Callable] = {
    # We only want to censor the hand and copy everything else, such as bids
    "hand": censor_hand,
}
key: str
for key in game_state:
    if key not in censor:
        default(censored_state, game_state, key, userID)
        continue
    func = censor[key]
    func(censored_state, game_state, key, userID)
```

This repeating pattern could allow for further functional abstraction, since each function is just a variation on the above structure, but I think that would likely lead to far too much unnecessary complication. In general I am hesitant and considering whether this is the correct approach to take and if this is overcomplicating the matter. I have not exactly followed the above Method 2 and in many cases have passed a subset of the `game_state` rather than the complete version. An example of this includes `censor_hand` as above that has `censored_player, player` as parameters rather than the general `censored_state, game_state` that default has. This is required to maintain each function taking the same number of arguments (otherwise censor hand would have to be separately told which hand it should be censoring). However this does break the design principle of giving maximum modularity and power to each function.

This could be fixed by rearchitecting the system again to use a different design. Instead of iterating over each player and fully censoring them before moving onto the next player, players would be censored using passes. For each key there would be a censor function that takes the list players and the current list of censored players. Then the function would censor and add the key / value pair to each copy. For example, when censoring bids, each players bid would be directly copied to their censored version, but when censoring hands, the function would use the current `userID` to determine whether to censor the hand or not. The advantage that this gives is that every function may be able to consider the entire game state at once, since it knows which key it should be looking for. It will also not get confused about which player it should be considering, as it is considering every player at once. However I am not sure that this would lead to an improvement and so I will refrain from changing what works until I discover a problem.

Tue 17 Dec 17:39:16 GMT 2024

I now have a bit more confidence in my design to use small modular functions / procedures that can be easily interchanged. After fixing a small bug, I was able to easily reuse the code for broadcasting a `"game_state"` to broadcast a `"waiting"` state. This is because I already had code for censoring the attributes I needed to (including `userID`) and other attributes were copied by default, without throwing errors. This gives me confidence that this may be able to scale. Automatically generating or choosing variations of these functions for different rule sets is still something that I am not sure about and have not tried.

```
# This gives the whole game state
players = WHIST[gameID]
# rather than this which gives only the players (which is what I wanted)
players = WHIST[gameID]["players"]
```

Above is the bug I encountered, which I discovered due to a runtime error and traced the cause using print statements.

```
The logging printed this:
{ ... 'players': {"players": [ {"userID": ... }, ... ]}, ... }
rather than the expected:
{ ... 'players': [ {"userID": ... }, ... ], ... }
```

Tue 17 Dec 19:04:32 GMT 2024

I am now considering how the games should run, from a high level, and what approaches / paradigms I could use. Many would promote an object oriented approach to the system (potentially including the OCR exam board) but I am not sure that this would provide the flexibility that I am looking for. I have also used OOP for another project and would like to use something new. For this reason I have designed the system differently. Every game would have a `game_state` that stores all necessary data about the game. There would also be a `game_method` that mutates the `game_state` to progress the game. At times this would require user input, for this an event would be sent to the requisite user which displays their options. Once they have chosen, their choice would be sent back to the server, which would use this to update the `game_state` until it requires user input again.

The flexibility in this system could come from the ease of replacing functions with other variations. To implement different bidding rules, one could simply replace the old bidding function with a variation.

For Whist in particular the structure of a game would be:

1. Initialize game
    1. Randomly pick partners
    2. Shuffle the deck
    3. Deal cards to each player
    4. Broadcast the initial state to each player
2. Play a trick
    1. Determine the starting player
    2. For each player
        1. Determine which cards each player can play
        2. Send the choice to the user
        3. Validate the user's response
        4. Update the `game_state`
        5. Broadcast the new `game_state`
    3. Once every player has played a card, determine the trick winner
    4. Increment the trick winner's number of tricks won
    5. Broadcast the new game state

3. Play tricks until cards are exhausted
4. Calculate the number of tricks won in each partnership
5. Calculate the number of points the winning partnership gains
6. Update the scoreboard
7. Continue until a partnership wins

The current file structure of the project is as such:

```
onlinecards/
|-- server.py
|-- ws_server.py
|-- handlers/
    |-- chatroom.py
    |-- games.py
    |-- whist.py
```

But I need to find where I am going to put the game logic, decide on naming conventions and finish writing the documentation for the back end. My plan is to have a `game_method` that updates and runs the game. However this will be created dynamically by a `game_method_creator` / `game_method_factory` function. This will take the user's chosen variations as an argument and generate a corresponding `game_method`.

```
onlinecards/
|-- server.py
|-- ws_server.py
|-- handlers/
|-- |-- chatroom.py
|-- |-- games.py
|-- |-- whist.py
|-- game_methods/
    |-- whist.py
```

Wed 18 Dec 09:53:59 GMT 2024

I have been running into many an `ImportError` today now I am coming to the opinion that restructuring the project may have been a mistake. At the very least it has not fixed my issue and has taken time and effort. I tried to follow this solution after I initially came across an `ImportError` but it has not fixed the problem. The problem was that I was trying to import `onlinecards/game_methods/whist.py` from `onlinecards/handlers/whist.py`. This may have a simpler solution but the structure he suggested seemed to be a better one that I currently had. It also seemed convenient and intuitive to have imports based on the project root.

Fri 20 Dec

Returning to this issue a few days later, I tried to reproduce the error in a separate test case, but was unable to. Instead I managed to produce a working import system based on the project root, which was quite similar to what I wanted in the first place. I then attempted to discern what differentiated my test case from the actual environment and in desperation deleted both `__pycache__/` and `__init__.py` which seemed to fix the bug. After hours of researching python's package and module system, I still don't really know what fixed the issue, but it seems to work now – even when using a venv. I am now going to create a few commits for this and then continue work.

Recreating a blank `__init__.py` does not seem to cause the issue to return which leads me to believe it may have been a caching error (which yes I **have** experienced before).

Sat 21 Dec 21:26:07 GMT 2024

Fixed a bug where `pyright` was raising import errors incorrectly. Using `LspInfo` inside `nvim` revealed that it was running in the `coursework/` directory rather than `coursework/onlinecards/`. This answer fixed the issue.

Sun 22 Dec 12:11:57 GMT 2024

I have created a new branch to track the devlog, todo etc. so they can be backed up without uploading them to github. However this seems to have removed them from local main. I want the files to be tracked in the `devlog` branch but untracked in the `main` branch and I have not yet been able to achieve this so I may have to ask a question on stack overflow about this.

Sun 22 Dec 12:22:52 GMT 2024

I have made a function that can initialise a new Whist game but this has made clear that the current code for rendering a Whist game state is not good enough. It cannot display each player with their starting hand of 13 cards and instead causes errors with wrapping and scrolling. Looking at other websites, they have cards that overlap, so that only the indexes can be seen, rather than having each card fully side-by-side. They also have a mix of cards arranged horizontally and vertically depending on whether a player is above / below or to the side of the table. Both of these are ideas that I shall add to my renderer and it will hopefully fix this issue. This is the next thing that I will have to fix before I can continue developing gameplay.

Sun 22 Dec 17:35:02 GMT 2024

When reading the rules for Whist on Wikipedia I learned that the dealer is supposed to deal the final card face up to themselves, which sets the trump suit. When it is their turn to play, they draw the card and play as normal. This means that for the first trick, the dealer's last

card should be uncensored for all players. This means that I have to use 2 separate functions to decide which cards should be censored, which is exactly what I have designed my system for. However, I quickly discovered that the variation function would need to know who the dealer was, which the default function did not, and since I was not passing the entire game state to the function this information was not even available. This would require that I either design 2 separate interfaces for the 2 different variations (which would make interchangeability very difficult) or I redesign the functions so that they use the same interface.

The easiest way to ensure that the functions will share the same interface is to pass the entire game state every time. This means that no matter what the function needs to read or where it needs to write to, it can. However with the current version of the default function, it deals with a single player at a time and cannot take the entire game state. If it did, it would have to determine which player it should be dealing with, which would require another parameter, which would change the interface. Above I discussed a potential alternate method that addresses this issue: instead of censoring each player one at a time, every player is censored at once. Rather than censoring every attribute for player 1 before moving on to player 2, the first attribute for every player would be censored before moving on to the next attribute. This could break if not all players share the same attributes, which I will have to consider.

I will also have to write a constructor that can create a new censor function based on the variations specified. This is a central problem that I will need to solve for the project to function as designed that I have not tackled yet. Unrelated, unforeseen issues, such as whether the function that runs a Whist game is blocking or non blocking have also emerged that I do not have solutions for and I may have to ask for advice online.

My main concern is that this could be a rather large rewrite to something that I thought I had finished and again casts doubt on the central idea that I have designed this project around. If I want this to succeed I think I will have to ensure that every function takes a complete game state (which I have not done), though I am not sure that it will.

Mon 23 Dec 21:59:00 GMT 2024

I have made the above changes which were much easier than anticipated. What has been the exact opposite is constructing a `game_method_creator` function – or `censor_method_creator` in this case. I have found the limitations of currying (which I had based this idea on) are much stricter than I remembered. I am certain my vague plan of having replaceable functions is possible but I am not sure that it would work in the manner that I had envisioned it. Nailing down a specific design is what I am struggling with at the moment. I particularly want to use only

a functional approach, rather than an object-oriented one, which may add further difficulties.

Further thinking my design for `censor_game_state` I am wondering if I should add a mechanism for adding data that does not exist within the `game_state` to the `reduced_state`. This is required in the instance of specifying the `"type"` of the event sent to the user, which is currently hard-coded as `"game_state"`. Such a mechanism would allow for more flexibility and could ensure that the function's behaviour is entirely controlled by the `censor` dict. This would be useful as it could allow for the function to take this dictionary as a parameter, which could then be passed as an argument or by currying. This is the design that I shall go for, but I am still not yet sure about how I am going to implement it.

Tue 24 Dec 20:21:48 GMT 2024

Call it a Finnish Christmas Eve present! Turns out that the task of turning the `censor_game_state` function into something dynamically created at run time, using currying, was actually rather easy. It seems that I accidentally stumbled onto exactly the solution that I wanted. In particular the decision to use dictionaries to map from keys to functions was useful since the program now does change these dictionaries at run time and the same key may point to 2 different functions for 2 different variations. I hope that this design will also work for the main game method and implementing variations for that!

Tue 24 Dec 21:41:09 GMT 2024

After spending most of Thursday failing to find a JavaScript bug (that wasn't a JavaScript bug) and after adding some logging that printed out the events the websocket received, it became clear that after broadcasting the `choice` state, which **was** rendered, I immediately broadcast a `game_state` that over wrote that and re rendered the screen. This was fixed by commenting the line to broadcast the game state.

Sat 28 Dec

When trying to get user input, I send a `choice` event to the user, which is rendered, and once the user has made their choice they send a response. When the `game_method` tries to await the response a `RuntimeError` is caused since both the `ws_server` and `game_method` would be waiting for the response. Yesterday I realised that this design may be slightly strange, and implementing it today I have come across the error. Normally the `ws_server` handles all requests and passes the response down the chain to the correct handler (`handle_whist` in this case). I like this design for basic events such as create or join a game and it also works well for a chatroom. However, the way I want the game to run is that a `choice` is sent to the user, they respond and then code

execution continues from the next line. This could be implemented using `websocket.recv()` fairly easily, but could cause the program to freeze waiting for a response and it also causes this `RuntimeError`. Using the other design of the `ws_server` handing down responsibility to the `whist` handler and then to the `game_method` could alleviate these problems. Abstractly, the `game_method` would run as long as it could, updating the game as much as it could, before requesting user input for a decision. Once the `choice` event has been sent, the server could continue handling other requests, until the response arrives and then the `game_method` could continue execution.

However I am not sure how to implement this in code. 1. I could have the `game_method`, after having sent a `choice` event, return a function to call, with the response, to continue execution. 2. I could create a wrapper function `get_response` that abstracts this detail away and is what is returned and called again. 3. I could have the response appended to a list of user actions and call the game method with the starting game state. Each action could then be applied to reach the current game state.

In case 1 or 2 my concern is that the call stack would be destroyed meaning that when the `get_choice` function returns it will not give its return value to the correct function, breaking the program. Case 3 requires the game being completely deterministic which makes adding randomness more difficult. It also seems incredibly processor intensive and wasteful.

I will have to settle on a design before I can continue.

Sun 29 Dec 21:51:05 GMT 2024

There are a number of conditions that I have to follow for getting user input: 1. The `game_method` cannot just `await websocket.recv()` since this would be blocking and stop **all** games until a response is received. 2. Further the `game_method` must return when it is expecting user input since that would also be blocking. 3. The program must be able to continue execution as normal once it has received user input. 4. Returning a continuation function from the top-level `game_method` every time user input is required seems to severely hinder encapsulation. 5. Once the function has returned the call stack has been destroyed. This could be alleviated by not using functions, but this would cause it's own issues.

Overall these issues are large and I cannot think of a good way of resolving them. This has made me question the overall design of the `game_method`. Currently, the `game_method` updates the game_state as much as possible before requiring use input, then sends a request for user input, and then continues. This is quite different to the `chatroom` which waits for user input, then processes it, and then broadcasts the `game_state`. The latter is much more similar to how the `ws_server` already works for creating and joining games and is the pattern is also

used by the `python-websockets` tutorial. To me this suggests that this may have to be the design pattern that the `game_method` takes rather than what I had previously designed. This will require a large redesign of the `game_method`.

Whilst this is frustrating, it does seem that it may be easier to implement variations as I simply have to change which function deals with each event, which can be done using a dictionary.

Mon 30 Dec 15:29:57 GMT 2024

Design for an event-based Whist game method

One of the issues that I have with an event-based design is that card games have a strict order of events. You cannot play a card if it is not your turn, you cannot change your bet after you have seen your opponent's cards, etc. Unlike a chatroom (where you can send a message at any time) this does not seem to suit an event-based architecture as well. To offset this I will likely have to check before processing an event if it is valid, which needs to be added across the program.

Program flow:

1. `"start"` : This is sent internally once the required number of players (4) join.
2. `"play_trick"` : This is also sent internally to perform the setup for the next trick. The program now waits for `"play_card"` events.
3. `"play_card"` : Each user, in turn, plays a card.
4. `"get_trick_winner"` : The program calculates the winner of the trick, and updates their score.
5. `"update_scores"` : Update the scores for each partnership – by default the winning partnership score 1 point for every trick in excess of 6 that they win.
6. `"get_game_winner"` : Determine if a partnership has won the game, if so update the scoreboard.
7. `"get_rubber_winner"` : Determine if a partnership has won the rubber, if playing longer than 1 game, and update the scoreboard.

event	action	after	before
`"start"`	create the deck, set partnerships, set dealer	N/A	`"play_trick"`
`"play_trick"`	perform setup required for the next trick, determining the lead, etc.	`"start"`	`"play_card"`

event	action	after	before
`"play_card"`	during a trick, adds a (valid) card to the list of played cards	`"play_trick"`	`"get_trick_winner"`
`"get_trick_winner"`	calculate the winner of the trick	`"play_card"`	`"update_scores"`
`"update_scores"`	update the scoreboard to show the number of points each partnership scores	`"get_trick_winner"`	`"get_game_winner"`
`"get_game_winner"`	determine if a partnership has won the game, if so update the scoreboard	`"update_scores"`	`"get_rubber_winner"`
`"get_rubber_winner"`	determine if a partnership has won the rubber, if playing longer than 1 game, and update the scoreboard	`"get_game_winner"`	N/A

Mon 30 Dec 16:28:49 GMT 2024

I have managed to refactor the game to use an 'event-based' architecture but I am not happy with the result. Currently the code handling events after a card being played is overly complex. This is because a trick will only end once a card is played, and the same is true of a hand, a game, etc. I hope to refactor this so that these are moved to separate events. I think the easiest way to do this would be with recursive calls to the `state_handler`.

Separately, I have been running into issues with the 'data oriented' approach that I have taken so far. Storing the game state in a dict has worked but has limited static analysis tools compared to what is available for a class. For example, type checking with a `dict[str, Any]` is almost non-existent since attributes can be of type `Any`, whilst with a class, each attribute could have a type specified. I have also come across `KeyError`s when I have forgotten the name of an attribute that were only raised a runtime, but could have been identified by static

analysis if I had used a class.

Previously there were 3 reasons I had used a 'data oriented' approach: 1. I thought it would be more suited for a functional approach that I believed would provide more flexibility 2. `json.dumps` expects a `dict`-like object to serialise. 3. I had used an 'object oriented' approach for a previous project and wanted to try something different

However researching this, I discovered that the second issue could be overcome by replacing `json.dumps(game_state)` with `json.dumps(vars(game_state))` (or the equivalent `json.dumps(game_state.__dict__)`). The first reason is a misunderstanding – in my use case having a function take a `dict` or a `class` is not very different. Python's `dataclasses` seem to provide me a way to define the `game_state` as a class and signal that it should not have methods. This would mean that it could behave very similarly to a `struct` in C or the `dict` that the code currently uses. Importantly, refactoring to use a class would provide improved static analysis and type checking.

Tue 7 Jan 10:47:40 GMT 2025

There has been a bug with rendering the scoreboard for some time where the scoreboard would flash by too quick to read before the next hand was started. I have previously added a delay to ensure that the program waits a second before rendering the next event, but this was not working. Once an event is received an 'early' `resume` time is set which is when the program can render the next event, and once the event has been rendered a 'late' `resume` time is set. The early resume time ensures that the next event must wait at least a second after the previous event has been received before it can be rendered and the late time is more accurate, being set for a second after the rendering has actually finished.

The way the rendering is waited for is by using `setTimeout` with the and setting the wait time to `resume - Date.now()`. However, if 2 events are sent in short succession, `resume` will be set to its early value when the timeout for the second event is set. This means that the second event will be rendered 1 second after the first has been received. If the first event also takes a long time to render (`0.7` seconds for example), this means that the second event will be rendered very shortly after the first has been shown (`0.3` seconds in this example). To fix this, the program will have to always use the late `resume` time, which cannot be done using `setTimeout`.

I can overcome this issue by first setting the early `resume` to `undefined` rather than a number. Then when a message is received the program will check what the value of `resume` is. If `resume` is undefined, the program will `sleep` for a short period (`0.1` seconds for example) before checking again. If it is a number, the program will `sleep` until the resume time and then render the `event`. When rendering, `resume` will first be set to `undefined` and once the `event` has been rendered it will

be set to 1 second in the future.

Again this introduces another bug if 3 or more events are sent in quick succession. Events should be rendered in order – first in first out – but this method would not ensure this and could allow for the third event to be rendered before the second. To fix this a queue is required that lists each event in order and ensures that they run in the order that they are received. This suggests that perhaps I should redesign the system to create a `Promise` when an event is rendered that resolves 1 second after the `event` has been rendered. The next event would then `await` this `Promise` before rendering.

Mon 13 Jan 20:20:13 GMT 2025

I have underestimated the complexity of 'just using a `Promise`' to order events. I think that the `resume` variable should be replaced with a new variable `currentEvent`, which is either a `Promise` or `undefined`. If it is `undefined` this represents that there is not current event so the next event can be rendered straight away. If `currentEvent` is a `Promise` then the next event must `await` it before it can be processed. When processing an event `Array.shift` can be used to get the first item from an `Array`, which should then set `currentEvent` to a `Promise`. 1 second after the event has been rendered, the `Promise` can then resolve.

Mon 13 Jan 20:49:50 GMT 2025

Nothing beside remains ... The lone and level sands stretch far away.