System Design Document

Table of Contents

1) <u>**Architecture**</u>

1.1) **<u>Our choices for architecture:</u>**

Our choices for the architectures this design will follow include the following: Dataflow, Model-view-controller, and event-based. Our AI will use a dataflow architecture, while the remainder of this project will take a model-view-controller approach. The controller used by clients will be designed using an Event-based architecture.

1.2) **<u>Reasoning for selecting a dataflow architecture for the AI:</u>**

Data flow focuses on information going through our client. Using this information, we can clearly see the requirements for each calculation that the AI will need to complete.

By using a dataflow, we're also able to easily view which computations the AI needs to complete before others, and we're able to see which computations can be completed in parallel.

This architecture will work better than a data centric model because the AI player will depend on the order of inputs and outputs. Likewise, an event-based architecture will be better than an event based system for the AI because no events are actively occurring until the AI decides to play, discard, or tell information. As such, an event architecture would not allow us to, in detail, follow the events of the AI's operation.

Additionally, this system would be better than a model-view-controller architecture because, as specified by the client, the AI must be able to play a game without displaying any information to the screen; to be able to just play from the terminal. Finally, this architecture would be better for the AI than call-and-return because it allows us to see both sequentialism and parallelism.

1.3) **<u>Reasoning behind selecting model, view, controller:</u>**

We chose a model, view, controller architecture for the clear modular and responsibility benefits. A model-view-controller architecture clearly dictates, at a high level, that some classes will be responsible for drawing, some classes will be responsible for maintaining information, and some classes will be responsible for handling events. Having a clear pattern like this will allow us to develop and test several smaller classes at once, rather than develop one larger class.

This architecture works well for this portion of the project for several reasons. The first is a graphical user interface will be developed. Using a model-view-controller architecture provides modularity. By developing our project this way, we're able to replace the graphical view with a textual one while we're developing the controller and model. This will allow us to focus on assuring functionality before aesthetics. Secondly, having separate components, each responsible for one type of action, allows us to clearly think of the pieces that are needed for this project.

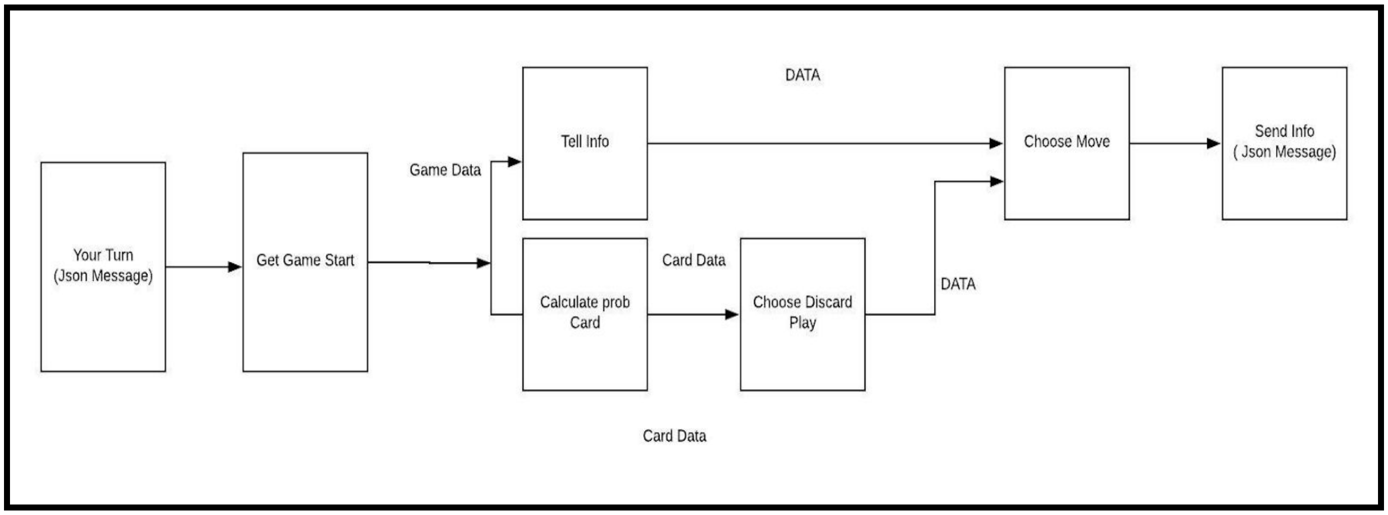
1.4) <u>**Reasoning behind selecting an Event-Based architecture for controller:**</u>

We chose an event-based architecture for our controller for one specific reasons: The first reason is that an event-based architecture allows us to focus on human actions, and consequently reacting to said actions; this architecture allows us to recognize which actions mean what at specific times.

This architecture will work better than a data centric architecture because it will allow us to sequentially follow what should happen each time the user interacts with the client. A data centric model would not show this information in a clear, concise way. This system will work better than a call and return system because we're not worried about function hierarchy, we're mainly worried with client-system interaction. Finally, this system will work better than a dataflow architecture because a client has no reason to interact with our client in any particular order. They can input any number of commands in any order.

2) **AI Architecture - Dataflow**

2.1) **The Data-Flow UML:**



2.2) **Describing the AI Data-Flow Diagram:**

Our client will initially receive a JSON message from the server. This message will state whose turn it is. If it is our client's turn, and our client is an AI, the AI will begin its turn.

Initially, the AI will retrieve information about the gameboard. This information will include information such as what cards are in it's teammates' hands, the cards shown in the center pile, the cards in the discard pile, and how many information tokens are remaining in the game. The AI will then pass this information into two separate processes: 'Tell info' and 'Calculate prob card'.

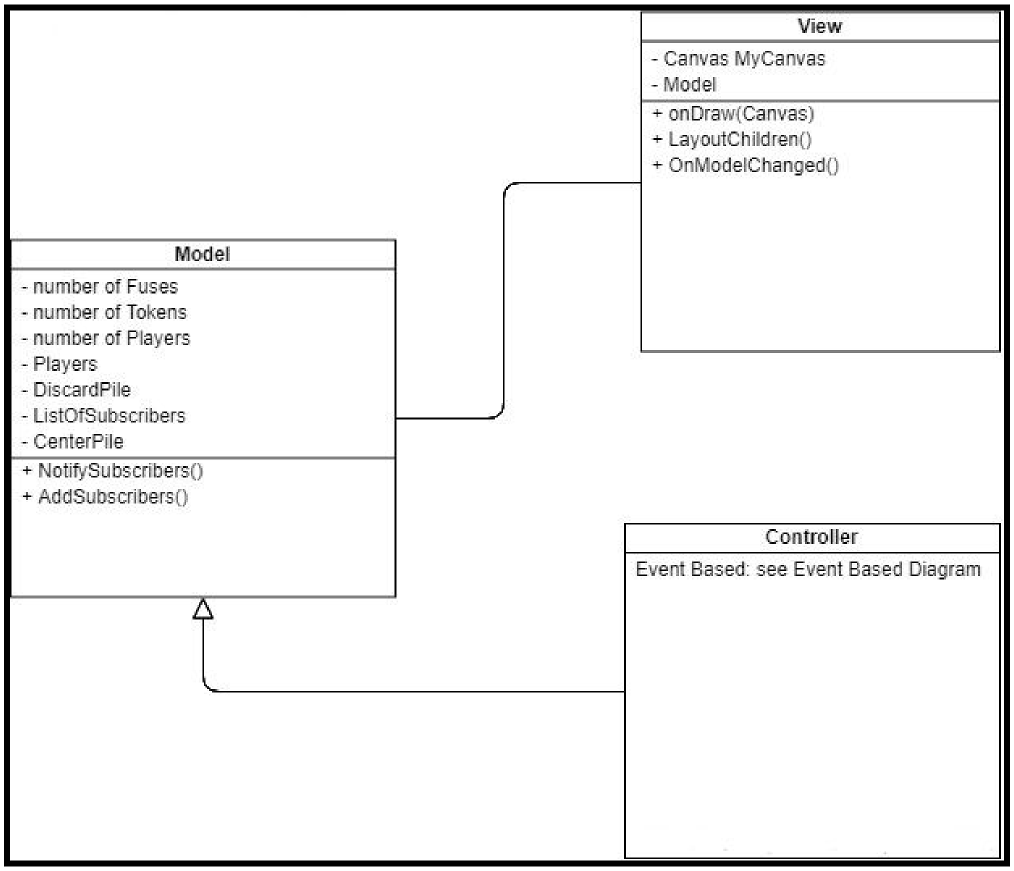
In the pipe responsible for 'Calculate prob card', the AI will look at all of the cards visible in the game and determine which card is most likely in its hand. It will then pass this card over to the next portion of this pipe. In the next portion of the pipe, known as 'chooseDiscardPlay', we will determine if it is better to play or discard the card that has been given, given the current state of the board. This means that, if we have 7 information tokens, this pipe will automatically choose to play the card. This area of the pipe will then create and send a JSON message to merge into the next part of the pipe.

On the separate pipe, ‘Tell info’, the AI will look at the number of information tokens left in the game, as well as the information in it’s teammates’ hands. If we have zero information tokens, the AI will automatically send an output that indicates that telling information is not possible at this time. If we do have information tokens, however, the AI will then determine both the best player to tell information to, and the best information to tell them. We will then send this information to merge with the output from the “calculatePlayDiscard’ pipe.

Finally, given two options (one telling information or one playing/discard a card), our AI will choose which is better at the current time; performing a move or telling another player information. The AI will then send it’s choice back to the server in a JSON format.

3) **Model-View-Controller (Human Architecture):**

3.1) **Model-View-Controller UML**

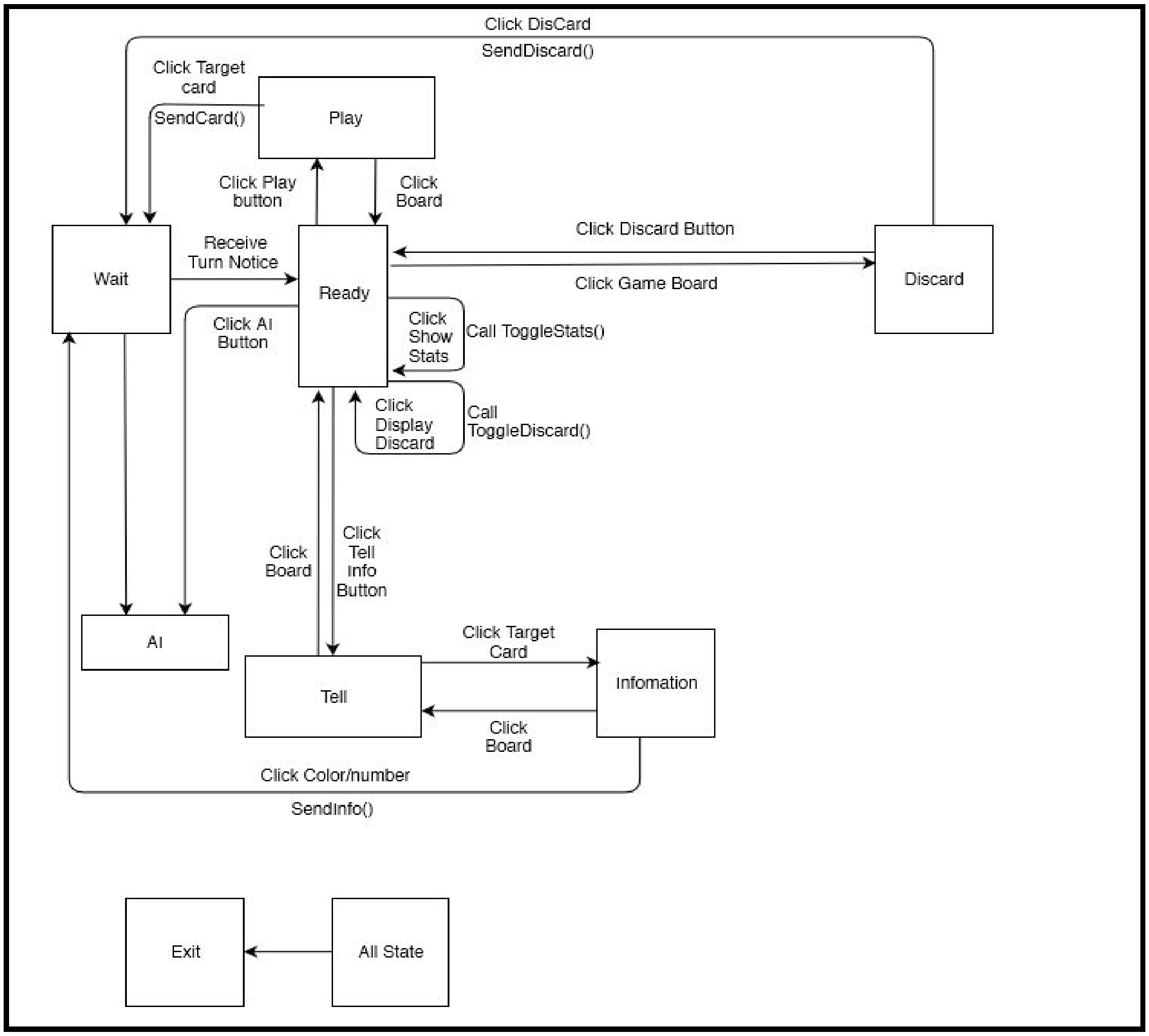


3.2) **Model-View-Controller Description:**

For users of the human variety, we will design their interaction using a model-view-controller approach. This approach will contain three things: A model, a view, and a controller. The model will be responsible for maintaining the state of the game. The view will draw the state of the game in a way that is understandable for the user. And finally, the controller will interpret and manipulate the gamestate based on the actions the user provides to the client. A more detailed description of the architecture for the controller is given below.

4) **Controller Architecture - Event-based:**

4.1) **Event-based Architecture UML:**



4.2) **States and Interactions In The System:**

It's important to note that, when we say "Clicking on the board" refers to clicking on any invalid spot. For instance, from the "Play Card" state, only clicking on one of your cards is a valid click. Clicking on anyone else's cards will result in returning back to the "Your Turn" state.

- Wait
    - Player waits for others' events.
        - Users at this state will not be able to interact with the client. They'll only witness the screen change after another player has made a move.
- Your Turn
    - It is now the users turn. From this state, players are able to click on one of six buttons. These buttons will allow the user to tell another player information, play a card, discard a card, see analytics, see the discard pile, or activate AI mode when clicked on.
    - Clicking on the button that says "Play card" will bring you to the "Play Card" state.
    - Clicking on the button that says "Tell Information" will bring the user to the "Give Information" state. If the game members have zero information tokens, clicking this button will keep you in the state, and a textual indication that this is not currently a possible move will appear on the screen.
    - Clicking on the button that says "Discard Card" will bring the user to the "Discard Card" state. If the game members have seven information tokens, clicking this button will keep you in this state, and a textual indication that this is not currently a possible move will appear on the screen.
    - Clicking on the button that says "Toggle Analytics" Will keep you in this state, but will make visible some statistical measurements of the game. If information is already visible, clicking on this button will make it invisible.

    - Clicking on the button that says “View Discard Pile” will keep you in this state, but will make visible to you a textual description of the discard pile. If this information is visible, clicking on this button will make it invisible.
    - Clicking on the button that says “AI mode” will move you to the AI mode state.
- Give Information
    - User click on their target player’s cards, and choose if they want to provide the number of the card or the color of card to the selected user. This will reveal to the selected user all cards with this field that are in said user’s hand.. If user click on the board from the state, the user goes back to the “your turn” state.
- Play Card
    - Form play, user can click on a card in their hands to send it to the server. If user click on the board from the state,we go back to the “your turn” state.
- Discard Card
    - User are able to click on a card in their hands to discard it and send it to the server. If user click on the board from the state, we go back to the “your turn” state.

- AI Mode
    - Upon entering this state, an AI will perform one move. The AI will then dump the user in the “Waiting” state and turn off.
- Exit Game
    - User are able to exit the game from any state that they want by click on “exit” button.

4.3) **<u>Events:</u>**

- Click to Play Card
  - From “Your Turn” state, the user can click on a button that says “Play card” in order to choose which card that user want to play, and send this card to play pile, then go back to “wait” state.
- Click to Discard
  - From the “Your Turn” state, the user can click on a button that says “Discard card” in order to choose which card that user want to discard , and send this card to discard pile, then go back to “wait” state.
- Click to give information
  - From “your turn” state , clicking on the “tell information”   button, then clicking on team card to go choose “click color” or “click number” to send to server, getting back to “waiting” state when this process finish.
- Click to quit game
  - Available from any state. Clicking on the “Quit Game” button will cause the player to exit the game. Consequently, a player will not be able to reconnect the game they were connected to, and the game will momentarily end for all players in the room.

- Click to view discard pile
  - From the “Your Turn” state, the user can click on a button that says “See Discard Pile” in order to have a textual display of the cards in the discard pile display on the screen.
- Click to view Statistical Analysis
  - From the “Your Turn” state, the user can click on a button that says “See analytics” in order to have a textual display of statistics display on the screen.

These statistics will include several probabilities, including most likely color in the user’s hand and the most likely card in the user’s hand.

- Click game board
    - This action involves clicking on anything that is not a valid place to click. This could include clicking on another player’s card when you’re supposed to be clicking on your own card. This action will always return you back to the “Your Turn” state.
- Click target card
    - After clicking the “give information” button, the user will then need to click on a card in another players hand.
- Click color
    - After clicking on another players card, users are able to choose tell teammate color information by clicking a “color information” button that will appear on the screen
- Click number
    - After clicking “Target Card”, user can choose what number of target card to tell teammate by clicking “number information”.
- Click AI model
    - Click this button user can switch to AI mode. In this state, an AI will perform one move for you. This will leave you in the “Waiting” state.
- Click own card
    - After clicking “discard”, users are able to pick which card they want to throw away by clicking one of their own cards.

4.4) **Event-based architecture functions:**

1) **SendPlayCard(Card)** : This function will send the selected card in the users hand to the server in the form of a JSON string with the following format:

```
{ "action" : "play"
, "card"   : (Index of selected card)         # the card number
(not rank)
}
```

2) **SendDiscardCard(Card)**: This function will send the selected card in the users hand to the server in the form of a JSON string with the following format:

```
{ "action" : "discard"
, "card"   : Card rank      # the card number (not rank)
}
```

3) **SendAllNumber(Int, Player)**: This function will send both a specified user and a specified type of information to a server. This information will be sent in a JSON format with the following format:

```
{ "action" : "inform"
    , "player" : (Player number here)
    , "rank"   : (Number from 1-5)
    }
```

4) **SendAllColor(String, Player)** : This function will send both a specified user and a specified color to the server. This information will be sent in a JSON format with the following format:

```
{ "action" : "inform"
    , "player" : 1
    , "color": "(Color here)"
    }
```

5) **ToggleDiscardPile()** : This function will draw the contents of the discard pile to the screen in a textual way if the information is not already drawn. If the information is already drawn, this function will make it invisible.

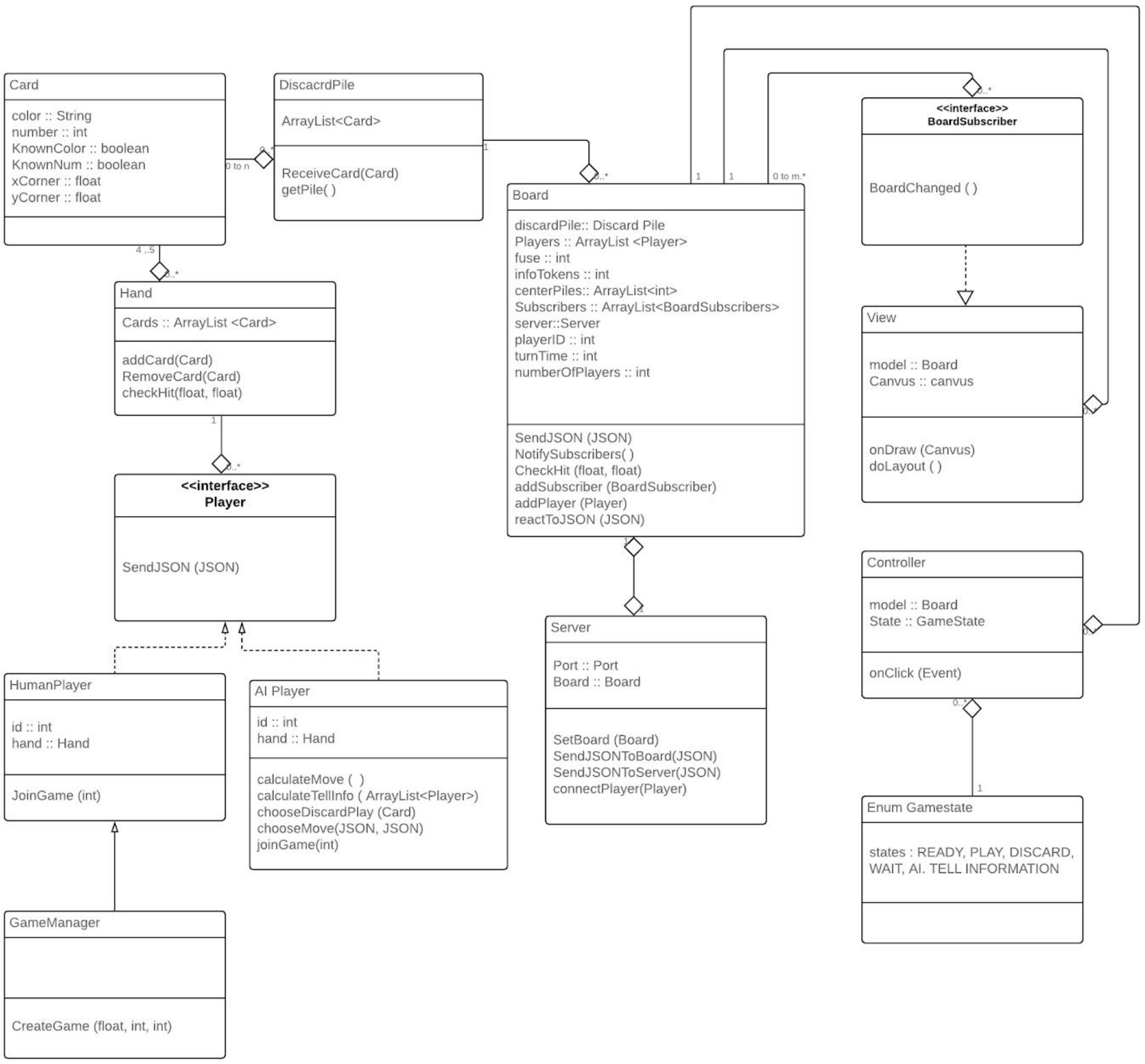
6) **ToggleAnalytics()** : This function will draw the game's statistics to the screen in a textual way if the information is not already drawn. If the information is already drawn, this function will make it invisible.

7) **ActivateAI()**: This function will activate the AI mode for the game, allowing the AI system built into the game to make a move for the player. The player will not be able to interact with the system until the move is complete.

8) **ExitGame()** : This function will disconnect the server from the game and close out the client.

5) <u>Classes and High-Level Design:</u>

5.1) <u>Class UML:</u>



5.2) **<u>Main Responsibilities:</u>**

While main is not a part of our design, a main function must be able to take in a server connection (and create a server object), initialize a model, view, and controller, and hook the view up to the controller. Main will also be responsible for connecting the model to the server, and vice versa. The Main function will provide entrance into the application.

5.3) **<u>Class Descriptions:</u>**

1. Card
   - An Instance of Card will keep the information of one card.
   - Fields:
     - This object will hold the card's colour as a String, in an field called 'color'. Options for this value will be "white", "green", "blue", "red", "yellow", or "multi".
     - The number of a card will get preserved as an integer, in an field called 'number'. The value will be between 0 and 5, inclusive.integers.
     - An field called 'knowColor' will exist as a boolean in order to indicate if this cards color is known to the player who is holding it. This value will be false on creation.
     - An field called 'knowNumber' will exist as a boolean in order to indicate if this cards number is known to the player who is holding it. This value will be false on creation.
     - An field called 'xCorner' will exist as a float. This value will indicate the x position, in pixels, of the top left corner of the card for our display.
     - An field called 'yCorner' will exist as a float. This value will indicate the y position, in pixels, of the top left corner of the card for our display.
   - Behaviours:
     - changeColor():

        - Purpose: To change the ‘knowColor’ field to true.
        - Prerequisites: None
        - Returns: None
        - Post: The value of the ‘knowColor’ field in the instance of Card will be true.
    - changeNumber():
        - Purpose: To change the ‘knowNumber’ field to true.
        - Prerequisites: None
        - Returns: None
        - Post: The value of the ‘knowNumber’ field in the instance of Card will be true.

2. DiscardPile
    - Purpose: An instance of DiscardPile will record all the cards that players in a game have discarded.
    - Fields:
        - An field ‘Cards’, of type ArrayList<Card> will exist. It will record all cards that have been discarded by players.
    - Behaviours:
        - addCard(Card):
            - Purpose: To add a card to the discard pile
            - Prerequisites: This function will require a Card object as input.
            - Return: None
            - Post: The object inputted will be inserted into the field ‘Cards’.
        - getPile():
            - Purpose: To send, in a textual style, all of the cards that have been discarded.
            - Prerequisites: None

        - Return: A string, summarizing all instances of Card stored in the field ‘Cards’.
        - Post: None

3. Hand
    - The Hand object keeps track of the cards that are in the players hand.
    - Fields:
        - An field ‘Cards’, of type ArrayList<Card> will exist in order to record which Cards are in a Hand. The length of ‘Cards’ will never exceed 5, and never be lower than 4.
    - Behaviours:
        - addCard(Card):
            - Purpose: To add a card to the field ‘Cards’.
            - Prerequisites: This function will require an instance of Card.
            - Return: None
            - Post: The instance of Card supplied to the function will now be in the field ‘Cards’.
        - RemoveCard(Card):
            - Purpose: To remove a card from the field ‘Cards’.
            - Prerequisites: This function will require an instance of Card as input.
            - Return: None
            - Post: The instance of Card supplied to the function will no longer be in the field ‘Cards’.
4. Player
    - This class will be an interface. Anything that implements this interface will be considered a Player in our game. Each client will have one Player.
    - Fields:
        - None

- Behaviours:
    - sendJSON(JSON):
        - Purpose: This function will take a JSON message and send it to the game board.
        - Prerequisites: A JSON message must be supplied to this function.This JSON message will come from the controller.
        - Return: This function will simply return the same JSON message that it was supplied. This JSON message will be sent to the game board.
        - Post: None.

5. HumanPlayer
    - Any instance of HumanPlayer will implement the Player interface.
    - Fields:
        - A field ‘id’ of type Integer will exist. The purpose of this field will be to record the id of the player this instance of HumanPlayer will represent.
        - A field ‘hand’ of type Hand will exist. The purpose of this field will be to maintain the cards that a particular instance of HumanPlayer has in their hand.
    - Behaviours:
        - joinGame(Integer):
            - Purpose: To add a player to a game of Hanabi. The game must exist, and the game must not be full.
            - Prerequisites: An integer must be supplied to this function. This Integer must be the id of an existing game of Hanabi.

        - Return: None
        - Post: This function will add a player to a list of Players stored in the instance of Board for the game they're joining. If the game is full, an error message will occur indicating so.

6. GameManager
    - Any instance of GameManager will extend a HumanPlayer. GameManagers are responsible for creating games of Hanabi.
    - Fields:
        - None, except those that exist in a HumanPlayer.
    - Behaviours:
        - createGame(Integer, Integer, Integer):
            - Purpose: To create a game of Hanabi.
            - Prerequisites. This function will take three integers as input. The first Integer will be the identifier of the game of Hanabi that they wish to create. This identifier must not be in use by another game. The second integer will be the amount of players that will be allowed into this game. This number must be between two and five, inclusively. The final integer will be the length of time each player will be allotted per turn. This must be a positive number.
            - Return: None
            - Post: A new game of Hanabi will be created, with the information provided as input. The GameManager will be inserted into a list of players in the instance of Board connected to their game of Hanabi.

7. AIPlayer

- An object that also inherit from player, and it still has all the Fields of player. However, AI player could play by themselves.
- Fields:
    - An field ID which is of type int. The purpose of this field is to represent the ID of the AI player.
    - An field hand which is of type Hand. The purpose of this field is to represent the hand of the AI player.
- Behaviours:
    - CalculateMove():
        - Purpose: calculate several probabilities, including most likely color in the user’s hand and the most likely card in the user’s hand.
        - Prerequisites: None
        - Return: an object of type Card representing the most likely card in the player’s hand.
        - Post: None
    - CalculateTellInfo(players[ ]):
        - Purpose: calculate what kinds of information to tell other players. This function takes an array of type Player as input.
        - Prerequisites: None
        - Return: a Json message to the “ChooseMove()” function. If the team has no information tokens then the function will return a message, indicating such.
        - Post: None
    - ChooseDiscardPlay(card):
        - Purpose: Takes in the most probable card, given the current state of the board and compares the benefit of playing that card versus discarding that card.
        - Prerequisites: None

        - Return: A Json message to function ChooseMove().
        - Post: None
    - ChooseMove(Json, Json):
        - Purpose: Receives Json message from other methods, and decides which move should be taken.
        - Prerequisites: None
        - Return: A Json message to the server with the chosen move.
        - Post: None
    - JoinGame(ID):
        - Purpose: Takes in an integer ID as input and joins a created game.
        - Prerequisites: None
        - Return: A message indicating whether the AI player joined successfully or not.
        - Post: If successful, the AI player is now connected to the server.

8. GameState
    - An enumeration class which keeps track of the game's current state. GameState has seven states: READY, PLAYCLICKED, TELL, WAIT, DISCARD, INFORMATION, AND AI. This class will be used by the Controller to check what state the game is in, so the Controller can perform an appropriate action.
9. Controller
    - The Controller will be responsible for receiving device inputs and determining how to manipulate the model given the inputs. As described above (in section 3.3 of this document), the controller will have several functions

that will send JSON messages from our client to the server.

10. View
    - The View will be responsible for drawing the model. The view class will implement the BoardSubsriber interface.
    - Fields:
        - A field titled 'model', of type Board, will exist in this class. This field will contain the model that the view will draw.
        - A field 'canvas' of type Canvas will exist in this class. This field is what we will draw our model on.
    - Methods:
        - onDraw(Canvas):
            - Purpose: This method will draw our GUI.
            - Prerequisites: This canvas will take in a canvas.
            - Returns: None
            - Post: This function will draw the current model to the screen.
        - doLayout():
            - This method will simply be responsible for handling resizing events. This method will need to resize a canvas and call the draw function. This method will have no prerequisites or return values, but will change the way the client's screen looks.
11. Board
    - Each client will have a single Board instance. The board will be responsible for maintaining all players in the game, the number of fuses remaining, the discard pile, the cards stored in the center piles, and more information that will be described in details below. Creating an object of type

Board will require a Server object in which this client can exchange JSON messages with.

- Fields:
    - A field called ‘fuse’, of type integer, will exist for instances of this class. This field will be responsible for maintaining the number of fuses the team has remaining in the game. This field’s value will be between 3 and 0, inclusively.
    - A field called ‘infoTokens’, of type integer, will exist for instances of the Board class. This field will be responsible for maintaining the number of information tokens the team has remaining. This field’s value will be between 0 and 7, inclusively.
    - A field called ‘playerID’, of type integer, will exist. The purpose of this field will be to record the identity of the user of this particular instance of the client.
    - A field called ‘turnTime’, of type integer, will exist in this class. The purpose of this field will be to record how long our client will have to finish a turn.
    - A field titled ‘numberOfPlayers’, of type integer, will exist in this class. The purpose of this field will be to record how many players are currently connected to the game.
    - A field called ‘subscribers’, of type ArrayList<BoardSubscriber> will exist in this class. The purpose of this field will be to maintain a list of views subscribed to our model.
    - A field titled ‘centerPiles’ will exist in the Board class. This field will record and maintain all played, accepted cards. Each index in this array will represent a different color, and it will store the highest number played for that color.

        - ‘Players’ will be a field that exists in the Board class. This field will be of type ArrayList<Player>. It will record all of the players that are in a game of Hanabi at any given time.
        - A field called ‘discardPile’, of type DiscardPile, will exist in any instance of the Board class. This field will be responsible for maintaining all discarded cards in the game. It will be initialized as empty, and grow as the game progresses.
        - A field titled ‘server’, of type Server, will be a part of the class Board. This field will be responsible for containing an Object of type server. We will send JSON messages to the server hosting our Hanabi game through this field.
    - Methods:
        - sendJSON(JSON):
            - Purpose: Send a JSON file to the Server field associated with this class.
            - Prerequisites: A JSON message containing the move a player chose.
            - Return: The same JSON message that was taken as input.
            - Post: None
        - notifySubscribers():
            - Purpose: To inform any of our views that our model has changed.
            - Prerequisites: None.
            - Return: None
            - Post: Each instance of BoardSubscriber subsribed to our model will perform it’s boardChanged() method.
        - checkHit(float, float):

- Purpose: To check what the mouse click was meant to hit, if anything.
    - Prerequisites: Two floats, the first of which represents the X-coordinate of a mouse click, the second represents the Y-coordinate of a mouse click.
    - Return: A string indicating what was hit.
    - Post: None
- addSubscriber(BoardSubsriber):
    - Purpose: This function will simply add a board subscriber to the field ‘subscribers’.
    - Prerequisites: An object that implements the BoardSubscriber interface.
    - Return: NOne
    - Post: The length of ‘subscribers’ will increase by one.
- addPlayed(Player):
    - Purpose: To add a player to the ‘players’ field.
    - Prerequisites: An object that implements the Player interface. The game must not be full.
    - Return: Nothing
    - Post: The length of ‘players’ will increase by one. a be responsible for maintaining the general state of the game.
- reactToJSON(JSON):
    - Purpose: This function will parse the JSON file provided to it, and decide if our client needs to do anything, or just wait.
    - Prerequisites: A JSON file from the server, indicating whos turn it is.
    - Return: A true value if it is our clients turn, false otherwise.

            - Post: None.An object of type board will have several functions.

12. BoardSubscriber
    - This class will be an interface. Objects that implement this interface will have a method “BoardChanged()”. This function will take no inputs, and will return void. However, it will be responsible for telling the view to redraw it’s model.
13. Server
    - This class will be responsible for maintaining a connection to the server. This connection will exist for us to send JSON messages to and receive JSON messages from the server provided to us.
    - Fields:
        - A field titled ‘connection’, of type Connection, will exist in order to maintain a connection to a server supplying the Hanabi game.
        - A field titled ‘board’, of type Board, will exist. This field will be responsible for having a board for us to send and receive JSON messages to and from.
    - Methods:
        - setBoard(Board):
            - Purpose: This method will set the board associated with the instance of this class calling this method to the Board supplied to the function.
            - Prerequisites: An object of type Board.
            - Return: None
            - Post: The field ‘board’ will contain a new Board.
        - sendJSONtoBoard(JSON):

    - Purpose: This method will receive a JSON message from the server and send it to the clients board.
    - Prerequisites: A JSON object that indicates who’s turn it is.
    - Return: None
    - Post: None
- SendJSONToServer(JSON):
    - Purpose: This method will receive a JSON message from the client and send it to the server.
    - Prequisites: A JSON object that indicates a moved performed by our client.
    - Return: None
    - Post: None
- connectPlayer(Player):
    - Purpose: This method will add a player to our Board’s list of players. A new player will be in our game, and visible to our client.
    - Prerequisites: A Player that is joining our game. The game must not be full.
    - Return: None
    - Post: A new player will be in our board’s list of players.