

---

**Final Report**

**for**

**Game Client for Hanabi Card  
Game**

**Version 1.0**

**Prepared by Alexander Lavis, Caim Chen,  
Justin Pointer, Noah Kovacs, Sajith Rajapaksa**

**Group E3, CMPT 370, Department of Computer  
Science - University of Saskatchewan**

**April 9, 2019**

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>User Manual</b>	<b>4</b>
2.1	Create Game . . . . .	4
2.2	Join Game . . . . .	6
2.3	Game board Overview . . . . .	7
2.4	Play Action . . . . .	8
2.5	Discard Action . . . . .	9
2.6	Hint Action . . . . .	11
2.7	Quit Action . . . . .	13
2.8	Swap Action . . . . .	13
2.9	Tutorial . . . . .	14
<b>3</b>	<b>As Built Documentations</b>	<b>15</b>
3.1	Model . . . . .	16
3.1.1	View . . . . .	22
3.1.2	State Machines . . . . .	23
3.1.3	Menus . . . . .	28
3.1.4	Initialization Classes . . . . .	35
3.1.5	Server Communication Classes . . . . .	40
3.1.6	User Turn Classes . . . . .	42
<b>4</b>	<b>Amendments to the Design and Requirements</b>	<b>48</b>
4.1	Messenger and Filter . . . . .	48
4.2	Menus . . . . .	48
4.3	Model . . . . .	48
4.4	AI . . . . .	48
<b>5</b>	<b>Known Bugs and Requires Implementation</b>	<b>50</b>
5.1	Action . . . . .	50
5.2	AI Join . . . . .	50
5.3	Thread Issue . . . . .	50
<b>6</b>	<b>Summary</b>	<b>51</b>

# 1 Introduction

During the process of software engineering, the project concludes with one additional document outlining what has been completed, and what needs completing. In the case of Group E3 (at the University of Saskatchewan), this document will explain the outcome and usage of the "Hanabi" project, in addition to a user manual. The User manual will be discussed first, explaining what is required how to use the product. The rules of the game will also be outlined in this section, via a tutorial. Any further queries about the functionality of the game should be investigated within the requirements document. Then, the latter half of this document will then elaborate on the code documentation. While this is similar to the design document, this section of the document will discuss what was implemented, rather than what was planned. Expanding on this idea, any amendments made to the design will also be discussed, as well as anything that was not successfully built. Now that this introduction has been concluded, the document will begin to explain the user manual.

## 2 User Manual

This section is intended to outline the basic functionality and usage of the as built graphical interface of the Game. It will outline how to install, create, join, and play the game at this time with easy to understand image examples of the game included. To begin with the installation process the user must clone the Main directory from E3 Project Repository next open the IntelliJ IDE. Then create a new project from the cloned Main directory. Next go find the GameState.java file this is the main program used to start the game. Running the GameState.java main function will result in the Start Menu appearing as shown below:

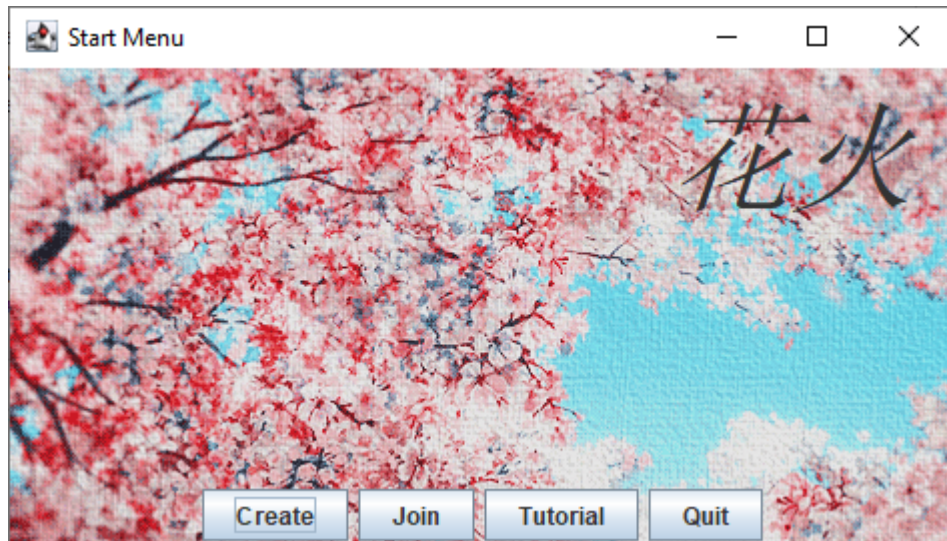


Figure 2.1: Start Menu

As shown above you can see there are four options available Create, Join, Tutorial, and Quit each actions results will be discussed in all the following subsections.

### 2.1 Create Game

Create game action is taken by the host user to create a game. This is initiated by clicking the create button located in the start menu. Below is a detailed description of the paths a user may take by this action.

**Trigger :** The User clicks the Create button.

**Create Menu**

Select Number of Human Players:

☒ 2 ☐ 3 ☐ 4 ☐ 5

Enter the number of AI Players

☒ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4

Enter your NSID:

Enter your secret player key

Enter Timeout Limit:

Select Wild Card Mode:

None

Enter

Figure 2.2: Create Menu

**Basic Path :**

1. The User launches the game from their computer.
2. The User will click the create button.
3. The System will prompt the User to enter the Network Services ID (NSID) of the University Of Saskatchewan, Number of human players, Number of computer players and the turn time limit.
4. The User will then enter the NSID, number of human players up to 4, number of bots between 0 to 4 (such that the total number of players is less than five) and the turn time limit.
5. The System will send the information to the Server.
6. The Server will create a game id and a secret key for the game and notify the System.
7. The System will display the game id and secret key to the host User.
8. The System will then change the interface to the waiting area.
9. Once all required Players have joined, the System will then display the game's interface.

### Alternative Paths :

- At any point before the step 7 host User can quit, this will result in a game not being created.
- At step 4 if the NSID does not match:
  1. Server will notify the System of a mismatch.
  2. System will prompt the user again for the NSID.

**Post condition :** A game ID and a secret key will exist in the Server.

Alternatively a user may instead choose to join an already existing game that they have the secret key for.

## 2.2 Join Game

Join game action is taken by the all human Players except for the host User to join an existing game in the Server.

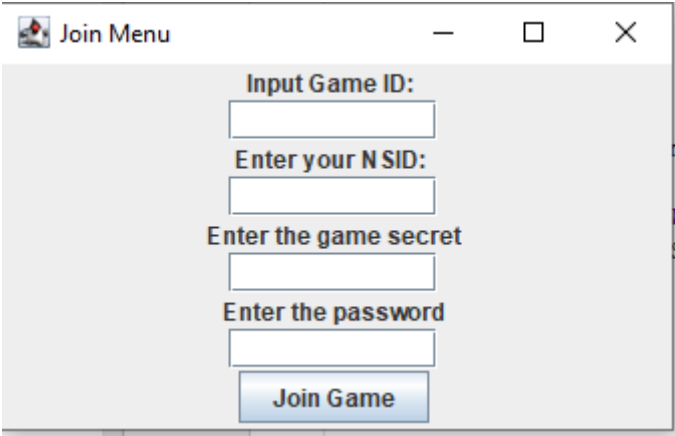


Figure 2.3: Join Menu

**Trigger :** The Player clicks the Join button.

### Basic Path :

1. The Player launches the game from their computer.
2. The System will prompt the player to enter the Network Services ID (NSID) of the University Of Saskatchewan.
3. The System then prompts the Player if the Player would like to join or create a game.

4. The Player will click the join button.
5. The System will prompt the player to enter game id and the secret key.
6. The Player will be moved to a waiting screen.
7. Once all required Players are joined The System will then display the game interface.

#### Alternative Paths :

- At step 5 if the game id or secret key does not match:
- At step 4 if the NSID does not match:
  1. Server will notify the system of mismatch.
  2. System will prompt the user again for the information again.

**Post condition :** The Player will be connected with the Server for the given game id.

## 2.3 Game board Overview

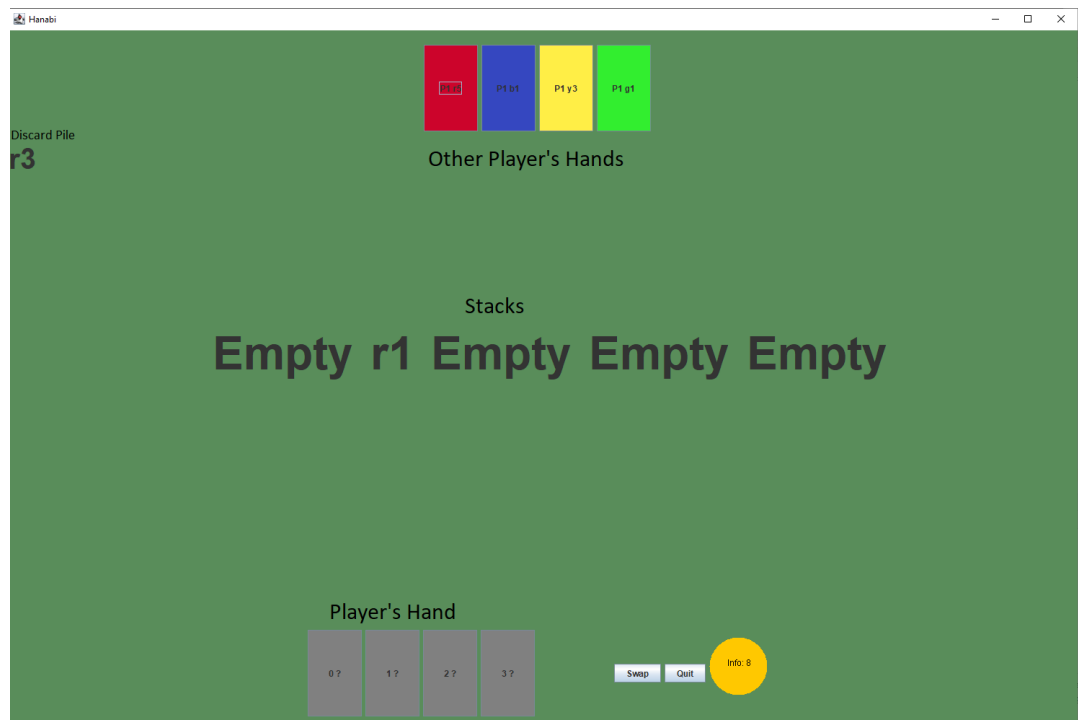


Figure 2.4: Game Board Menu

## 2.4 Play Action

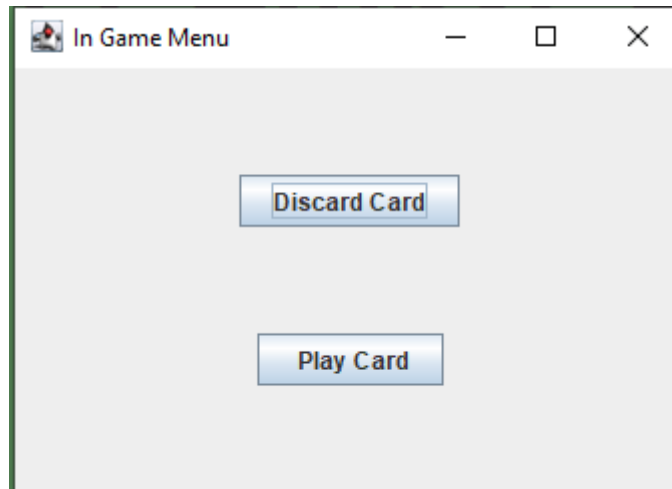


Figure 2.5: Play Menu

**Brief Description:** The play card action can be taken by any Player on their turn for placing a card from their hand to the a firework card stack.

**Trigger :** The Player attempts to play a card from their hand.

**Precondition:** Must be the Player's turn.

**Basic Path :**

1. The Player selects the card they wish to play within time limit.
2. The System displays message confirming the Player's selection with pop-up buttons yes/no assume yes.
3. The System notifies the server.
4. The Server notifies all players.
5. The System checks if the card played is valid for being placed on any of the stacks and returns with a positive result.
6. The card is moved from the Player's hand to the appropriate stack.
7. The Server checks if the deck has cards to draw and return with a positive result.
8. The Server draws a new card for Player from the deck and adds it to their hand.
9. The System displays new game state.



10. The Server then moves onto the next player's turn.

**Alternative Paths :**

- In step 5, if the System responds with an invalid card to be placed on stack result.
- The Server receives the signal that the Player has unsuccessfully played a card. The Server then burns one of the Fuse Tokens reducing their number by one and moves the card played from the Player's hand to the discard pile. Return to step 7.
- In step 7, if the deck no longer contains any cards to draw the Server returns a negative result.
- 6. The Server sets final turn flag to on. Return to step 9
- In step 5, if the card is a valid match and completes a suit of a color a information token will be granted to the game board given it does not exceed 8 information tokens.

**Post condition :** The Player's hand has a new card drawn from the deck or final turn flag set.

**Exception Paths :** The Player may cancel a Play action on Step 2 of the basic path. If the Player fails to act in Step 1 a pop-up message from the System will appear asking them to select an action. If they fail to act within the time limit more than 2 times, the game will terminate.

**Other :** The cards in the Player's hand will act as a clickable button/object that triggers the Play action or Discard action. This scenario assumes the Play action was selected at that time.

## 2.5 Discard Action

**Brief Description:** The discard a card action can be taken by any Player on their turn for placing a card from their hand to place it on the a discard card stack to obtain a hint token. At least one hint token should be spent.

**Trigger :** The Player attempts to discard a card from their hand.

**Precondition:** Must be the Player's turn and less than eight information tokens available.

**Basic Path :**

1. The Player selects the card they wish to discard within time limit.

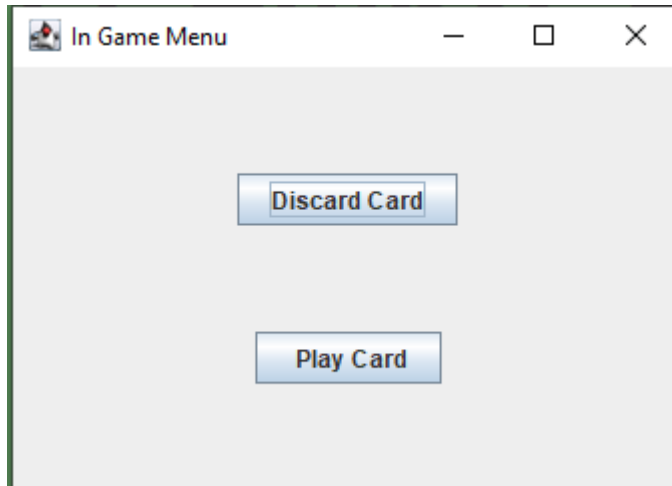


Figure 2.6: Discard Menu

2. The System displays message asking the Player if they wish to discard the selected card. With two buttons yes and no. Assume the Player selects yes.
3. The System checks if the information tokens have less than 8 available and receives a positive signal.
4. The System flips an information token to available position.
5. The Server checks if the deck has cards to draws and return with a positive result.
6. The Server draws new card for Player from the deck and adds it to their hand.
7. The System displays new game state.
8. The Server then moves onto next players turn.

### Alternative Paths :

- In step 3, if the Server responds with a negative valid to be placed on stack result.
- 4. The Server receives the signal that the Player has unsuccessfully played a card. The Server then burns one of the Fuse Tokens reducing their number by one and moves the card played from the Player's hand to the discard pile. Return to step 5.
- In step 5, if the deck no longer contains any cards to draw the Server returns a negative result.
- 6. The Server sets final turn flag to on. Return to step 7.

**Post condition :** The Player's hand has a new card drawn from the deck or final turn flag set.

**Exception Paths :** The Player may cancel play action on Step 2 of the basic path. If the Player fails to act in Step 1 a pop-up message from system will appear asking for them to select an action. If they fail to act within the time limit more than 2 times, the game will terminate.

**Other :** The cards in Player's hand will act as a clickable button/object that triggers the Play action or Discard action this scenario assumes discard was selected at that time.

## 2.6 Hint Action

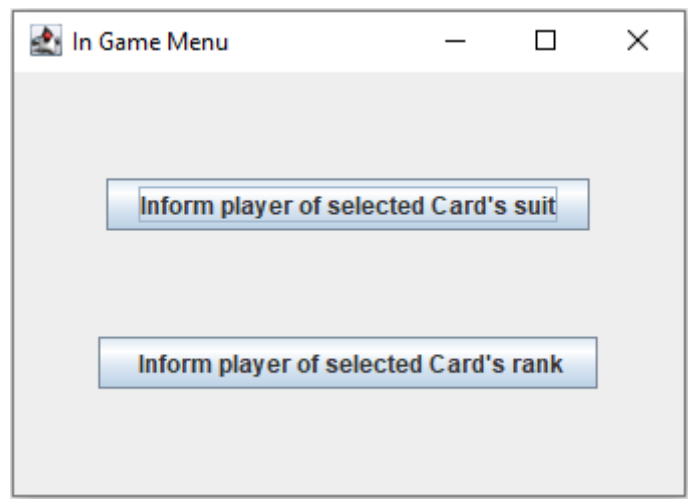


Figure 2.7: Hint Menu

**Brief Description:** The hint action can be taken by any Player on their turn to give another Player a hint about their hand. This can be either the number of a given colour or given number. One hint token should be available to use this action.

**Trigger :** The Player attempts to give hint about the cards in another Player's hand.

**Precondition:** Must be Players turn and at least one of eight information tokens available.

**Basic Path :**

1. The Player clicks a card in another player's hand within the turn time limit.
2. The System displays a pop-up message asking if the Player wants to give a hint to the Player holding the selected card. With two buttons Yes and No. Assume Yes is selected.
3. The System checks if the information token pool has at least 1 token available and receives a positive signal.
4. The System displays a message to the Player asking them if they wish to give a hint about the specific suit color or values of the cards in the other Player's hand.
5. The System then displays the appropriate pop-up for Player selection. For colours the pop-up will have buttons for all colour cards in other Player's hand, For Values the pop-up will have buttons for all the different value of cards in other Player's hand.
6. The System flips an information token to the unavailable position.
7. The System displays a new game state.
8. The System then moves onto the next Player's turn.

**Alternative Paths :**

- In step 3, if the System responds with a negative signal for the availability of information tokens.
- 5. The System signals to the Player that that is an invalid action then exits the hint action and lets Player select different action.

**Post condition :** Available information tokens decrease by 1.

**Exception Paths :** The Player may cancel hint action on Step 2 of the basic path. If Player fails to act in Step 1 a pop-up message will appear asking for them to select an action. If they fail to act for 2 more minutes the game will end.

## 2.7 Quit Action

**Brief Description:** The quit action can be taken by any human Player at any given time. Once the game has started, quitting the game will cause the whole game to terminate.

**Trigger :** The Player clicks on the quit button.

**Precondition:** None

**Basic Path :**

1. Player clicks the Quit button located at the game board.
2. The System displays a message asking if they wish quit the game with a YES or NO button pop up.
3. The Player clicks YES.
4. System then notifies the Server to end the game.
5. All Players are notified that the game has ended by the Server.
6. The System displays the end game screen.

**Alternative Paths :**

- In step 2 if the Player clicks NO the game resumes normally.
- If the game has not started and the Player quits, the System will notify the server and wait for another Player.

**Post condition :** The game will be at end game state

## 2.8 Swap Action

**Brief Description:** The swap action can be taken by any Player during anytime of the game. This will cause an computer Player to play on the human Player's behalf. The Player can also turn this off at any point of the game.

**Trigger :** The Player clicks on the swap button.

**Basic Path :**

1. The Player clicks on the Swap button.
2. The System displays a message to confirm that the Player wants to swap IN if a computer Player is playing or OUT if human Player is currently playing.

3. The Player clicks YES.
4. The System starts to use the computer Player to make decisions on the Player's turns.

**Alternative Paths :**

- On step 2, If the Player clicks NO, the game will resume normally.
- On step 4, If the Player is swapping back in, the System will wait for the human Player's input for the Player's turns.

## **2.9 Tutorial**

This option can only be accessed at the start menu it is intended to present a link to a online video detailing the rules and play of Hanabi.

### 3 As Built Documentations

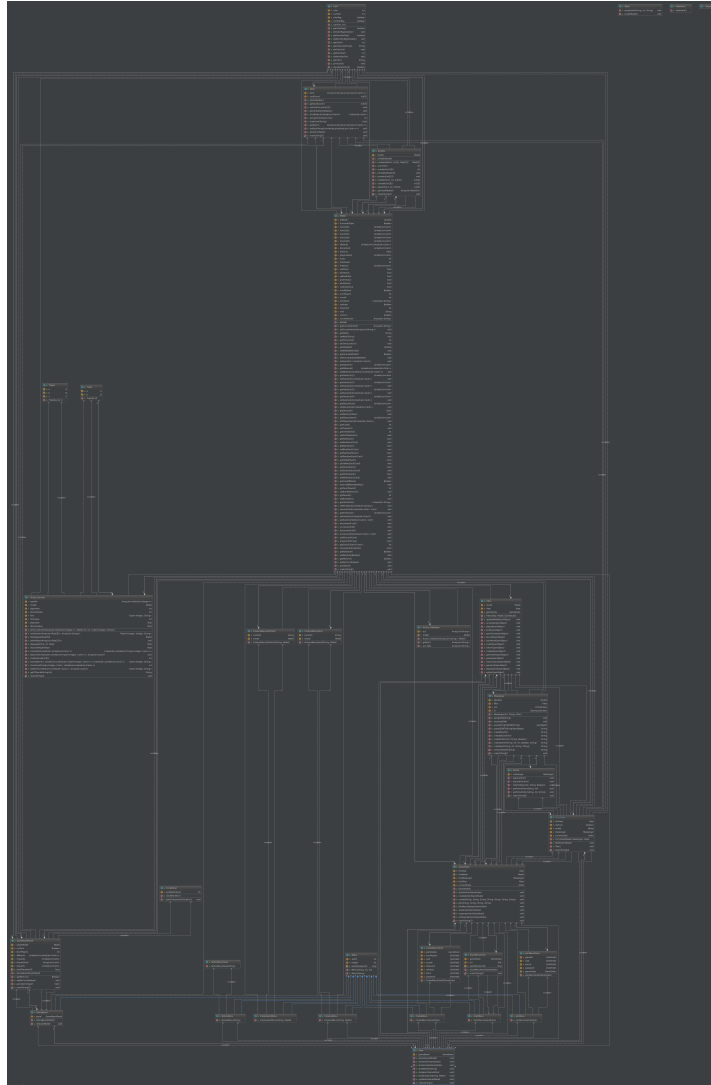


Figure 3.1: Overview

Figure 3.1 displays the complexity of the entire system. Because of this reason in this chapter we will brake each package in to its own section to explain classes and methods. For the most reason documentation please visit <https://git.cs.usask.ca/370->

19/e3/tree/Main/Main/src/JavaDoc where the JavaDoc can be found.

### 3.1 Model

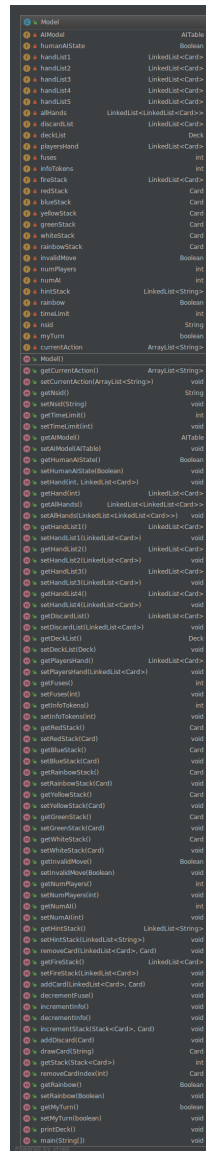


Figure 3.2: Model Class UML Segment



In this class using the MVC convention we store all the data needed to run the Hanabi game. This class is used by many other classes to get the necessary information. All of the attributes in this class are private but can be manipulated by the methods.

Dependencies: Messenger: the Messenger is responsible for updating the model with the information sent via the server

Attributes:

- AIModel: AITable : This attribute stores a AI table object. This keeps the necessary statistical information needed for AI's calculations to determine the best possible action.
- HumanAIState: Bool: This is a Boolean value True means that the model belongs to an AI player and False means a human player.
- handList1: List< *card* >: List of cards representing cards of the player one, if this is the hand of the user it is an empty list.
- handList2: List< *card* >: List of cards representing cards of the player two, if this is the hand of the user it is an empty list.
- handList3: List< *card* > List of cards representing cards of the player three, if this is the hand of the user it is an empty list.
- handList4: List< *card* > List of cards representing cards of the player four, if this is the hand of the user it is an empty list.
- DiscardList: List< *card* >: List of cards that represents all the cards that have been discarded.
- Deck:List<card> : List of cards that represents all the cards that has yet to be used
- PlayersHand: List< *card* >: List of empty card object representing the missing information
- Fuses: int: Integer to represent the number of fuse remaining.
- InfoTokens: int: integer representing the number of information token remaining
- RedStack: Card : Contains the top card of the Red suit
- BlueStack: Card: Contains the top card of the blue suit
- YellowStack: Card Contains the top card of the blue suit
- GreenStack: Card Contains the top card of the Green suit
- WhiteStack: Card Contains the top card of the White suit
- InvalidMove: Bool: Boolean type True if an Invalid move occurred, False if no invalid Move occurred in the last turn.

- numPlayers:int : number of players present in the class.
- numAI: int Number of AI player represents in the class. 0 if the User is not the host
- numAI: int Number of AI player represents in the class. 0 if the User is not the host
- HintStack: List< *String* >: list of strings containing previously given out hints to other players.

Functions:

- setHAIState(Bool)::Changes the variable HumanAIState with the given boolean  
Inputs Boolean : New human/AI state  
Output None
- getHAIState() :: Returns the Current Human or AI state.  
Inputs None  
Output Boolean : Current human or AI.
- setDeck()::Updates the Deck variable with the new given list of cards.  
Inputs List of cards  
Output None.
- getDeck() :: Gets the current deck of the game.  
InputsNone  
Output < *cards* > : returns the list of cards that is currently in the game .
- getHandList()::Gets the list of cards that belongs to a given hand  
Inputs : None  
Output < *cards* > : returns the list of cards that is currently in the game .
- setHandList():: Sets the cards to any given hand  
Inputs  
A player or AI's hand, as a List of< *cards* >  
Outputs
- removeCard() :: A mutator method that removes a particular card from a given list of cards  
Input:

A List of  $\langle cards \rangle$ , representing the a player hand or Deck.  
 A specific card object that is to be removed from the given list  
 Outputs: None  
 Postconditions: A card is removed from a list

- `addCard()`  
 :: A mutator method that adds a card to a given list of cards  
 Input  
 List|card<sub>i</sub> : A list of cards to be added to a list, either a hand or discard pile Card:  
 A specific card to be added to the given list  
 Output: None  
 Postconditions: A Card is added to one of the give lists
  
- `decrementFuse()` :: A mutator method that subtracts the fuse token count by 1  
 Input: None  
 Output: None  
 Postconditions: fuse count is subtracted by 1
  
- `IncrementInfo()` :: increases the number of info tokens available by 1  
 Preconditon: The info token count does not equal 8  
 Input: None  
 Output: None  
 Postconditions: the InfoToken attribute is increased by 1
  
- `decrementInfo()` :: reduce the number of info tokens available by 1  
 Precondtions: the info token does not equal 1  
 Input: None  
 Output: None  
 Postconditions: the InfoToken attribute is reduce by 1
  
- `IncrementStack()` :: Increases the number of cards placed on a given stack  
 Inputs: A stack where cards are placed, represented by an integer  
 Outputs: None  
 Postconditions: The given stack is incremented by one, indicating a card was added to the stack
  
- `getAITable()` :: A getter for obtaining the AITable  
 Inputs: None  
 Outputs: The AI table

Postconditions: None

- `invalidMoveCheck()` :: A method that checks whether a given move is valid once it post data-flow check  
Inputs: None  
Outputs: True if invalid; false otherwise  
Postconditions: None
- `addDiscard()` :: Adds a card to the discard pile, given a string input, representing the card  
Inputs: A string, representing the cards color+number pair  
Output: None  
Postconditions: A card is added to the discard pile
- `drawCard()` :: Removes a card from the deck, and adds it to a players hand  
Inputs: None  
Outputs: None  
Postconditions: A card from the top of the deck is removed, and placed in a players hand
- `getfuse()` :: A getter that obtains the current fuse token quantity  
Inputs: None  
Outputs: An integer representing the current fuse tokens  
Postconditions: None
- `getInfo()` :: A getter that obtains the available information tokens  
Inputs: None  
Outputs: An interger representing the current information tokens available  
Postconditions: None
- `getStack()` :: A getter that obtains the value of a given stack  
Inputs: A stack, as an int  
Outputs: The value of the stack  
Postconditions: None
- `getHintStack()` :: A getter that returns a list of given hints  
Input: None  
Outputs: A list of all the hints  
Postconditions: None

- `getInvalidMove()` :: A getter that returns an invalid move performed by the player  
Input: None  
Output: an Boolean indicting an invalid move  
Postcondition: None
- `setInvalidMove()` :: sets an invalid move  
Input: None  
Output: None  
Postconditions: Sets the invalid move to true if give true, and false given false
- `removeCardIndex()` :: Removes a card from a players hand, given by an index  
Preconditions: The given argument is between 1 and 4  
Input: A number indicating the index of the card to be removed  
Output: None  
PostConditions: a card is removed from the players hand
- `getNumPlayers ()` :: Gets the number of human players in the game  
Input: None  
Output: the number of players in the game, as an integer  
Postconditions: None
- `setNumPlayers()` :: Sets the number of human players participating in the game  
Preconditions: the number of players being added to the game is between 1 and 5  
Input: An integer, indicating how many human players are participating  
Output: None  
Postconditions: the number of human players is set
- `getAIplayers()` :: sets the number of AI players that are participating in the game  
Preconditions: the number of players being added to the game is between 1 and 5  
Input: An integer, indication how many human AI are participating  
Output: None  
Postconditions: The number of AI players is set
- `setNumPlayers ()` :: Gets the number of human players in the game  
Input: None  
Output: the number of players in the game, as an integer  
Postconditions: None

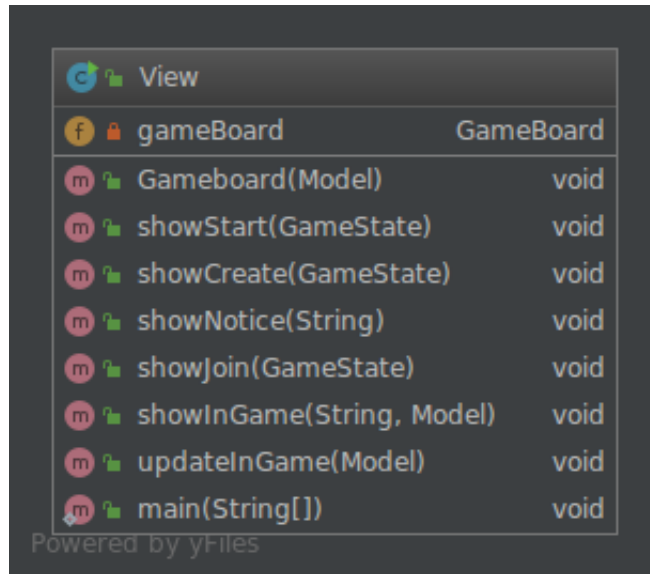


Figure 3.3: View Class UML Segment

### 3.1.1 View

Role of visually representing the elements of the game as it changes such as menus, GUI, and other graphical elements

#### Dependencies:

- Uses Model to determine what to display on the user's screen
- Java Frame and Graphics Libraries: Necessary to create a window and draw interactive graphical elements and menus

#### Attributes:

- Height: Integer indicating the height of the window of the game
- Width: Integer indicating the width of the window of the game
- Model: a Model that will be used to accurately display the game

#### Functions:

- `gameboard():void`  
Displays the current game menu based on model information
- `showStart():void`  
Displays the current appropriate start menu based on model information
- `showCreate():void`  
Displays the appropriate create menu based on model information
- `updateInGame(): void`  
Draw into a frame the appropriate graphical elements
- `ModelChanged(): void`  
Listens for the Model changing to know to update the view

### 3.1.2 State Machines

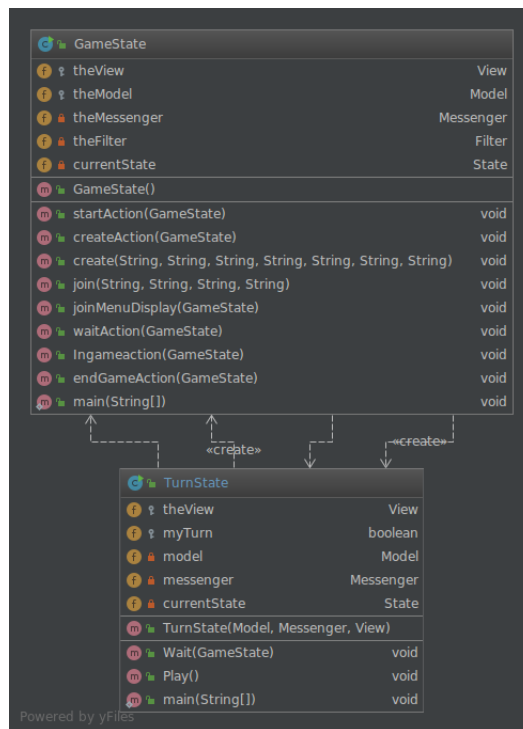


Figure 3.4: "State Machine Interface and Relationships"

### 3.1.2.1 Interface State Machine

The StateMachine class is an interface designed to show the basic required variables and methods to be implemented by all StatteMachines in the program. In this Design Document both GameStateMachine and TurnStateMachine extend StateMachine interface.

#### Attributes:

- currentState : this attribute represents the overall state the program is in at the moment it is used to determine what actions should be occurring at the time it defaults to the STARTING State initially.
- State:Enum: The attribute State: Enum is an enumerated list of all the states that will occur in the program using easily readable and self explanatory names for programmer readability while acting as differently numbers cases for a switch based program structure to be implemented.

#### Functions:

setCurrentState(State):void

- Function intended to set the current state attribute to a specific enum state, needs to be implemented

- Inputs

State: A specific State from the functions enum list of states

- Outputs

Void

getCurrentState(): State

- Function intended to return the current state attribute to a specific enum state, one of the StateMachine Interface's required functions to be implemented.

- Inputs

None

- Outputs

Void

### 3.1.2.2 Game State Machine

The GameStateMachine class is intended to monitor the current state of the game overall including starting a new game, creating a new game, joining a game, waiting for all players to join, playing the game, and finally the end of the game. This class's overall purpose is to act as the main function of the program calling other classes as needed. The



class implements the StateMachine Interface and depends on several classes including CreateHand, AI Join, MakeDeck, Model, Messenger and Finally the TurnStateMachine.

#### **Attributes:**

- `currentState` : this attribute represents the overall state the program is in at the moment it is used to determine what actions should be occurring at the time it defaults to the STARTING State initially.

- `State:Enum`  
`State.CREATE`  
`State.STARTING`  
`State.JOINING`  
`State.WAITING`  
`State.INGAME`  
`State.ENDGAME`

: The attribute `State: Enum` is an enumerated list of all the states that will occur in the program using easily readable and self explanatory names for programmer readability while acting as differently numbers cases for a switch based program structure.

#### **Functions:**

`setCurrentState(State):void`

- `FSets` the current state attribute to a specific enum state, one of the StateMachine Interface's required functions

- Inputs

State: A specific State from the functions enum list of states

- Outputs

Void

`getCurrentState(): State`

- returns the current state attribute to a specific enum state, one of the StateMachine Interface's required functions

- Inputs

None

- Outputs

Void

gameRun():void

- General main function that initializes at the start of the game and contains all the logical switch cases and state transitions.

- Inputs

None

- Outputs

Void

endGameTally():void

- Checks the Model for the final tally of all the Stacks at the end of the game and computes the final score

- Inputs

None

- Outputs

Void

### 3.1.2.3 Turn State Machine

The GameStateMachine class is intended to monitor the current state of the players turn cycle including being ready for their turn, taking their turn, and quitting the game.

This class's purpose is to act as the controller for INGAME State behaviors like taking your turn and calling all the action pipelines contained within. The class implements the StateMachine Interface and depends on several classes including H/AI Checker, AI\_Action\_Decider, Action, In Game Menu, Model, Messenger, Action.Validation and finally the AI Table.

Attributes:

- currentState : this attribute represents the overall state the program is in at the moment it is used to determine what actions should be occurring at the time it defaults to the READY State initially.

- State:Enum  
State.READY  
State.TURN  
State.QUIT  
: The attribute State: Enum is an enumerated list of all the states that will occur in the program using easily readable and self explanatory names for programmer readability while acting as differently numbers cases for a switch based program structure.

Functions: setCurrentState(State):void

- Sets the current state attribute to a specific enum state, one of the StateMachine Interface's required functions
  - Inputs  
State: A specific State from the functions enum list of states
  - Outputs  
Void

getCurrentState(): State

- returns the current state attribute to a specific enum state, one of the StateMachine Interface's required functions
  - Inputs  
None
  - Outputs  
Void

turnRun():void

- General function that initializes at the start of the INGAME state of the GameStateMachine and contains all the logical switch cases and state transitions use to monitor the players turn cycle while playing.
  - Inputs  
None
  - Outputs  
Void

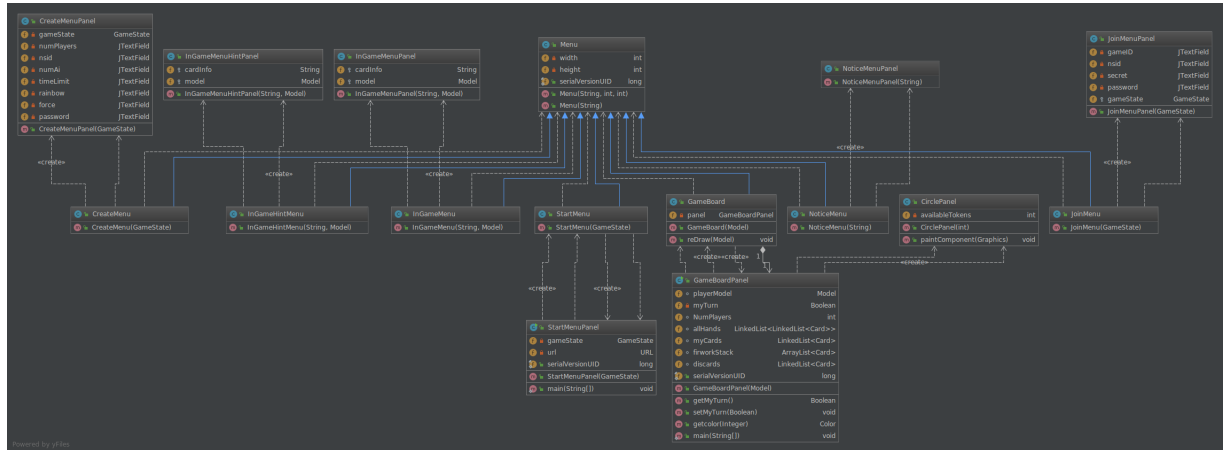


Figure 3.5: Menu Superclass and derived Subclasses

### 3.1.3 Menus

#### 3.1.3.1 Abstract Class Menu

Class Menu is an abstracted class. Purpose of this class is to be the super class of all the menus used for human interactions.

##### Dependencies

Menu is used by view to create the pop ups for the human interactions and menus have action listens for each interaction.

##### Attributes

- Width int : Width of the frame that is used to create the given menu.
- Height int : Height of the frame that is used to create the given menu
- CloseButton: JButton : A JButton object which is used to add a close button to the frame.

##### Functions

CloseButtonClicked(): Void

- This Method is a action listener to the CloseButton. When pressed it notifies the View to close the frame.
- Inputs
  - None
- Outputs
  - Void

getwidthheight(): (Int,Int)

- Method for getting for width and height
  - Inputs
    - None
  - Outputs
    - The height and width of the menu frame in a pair

### 3.1.3.2 Start Menu

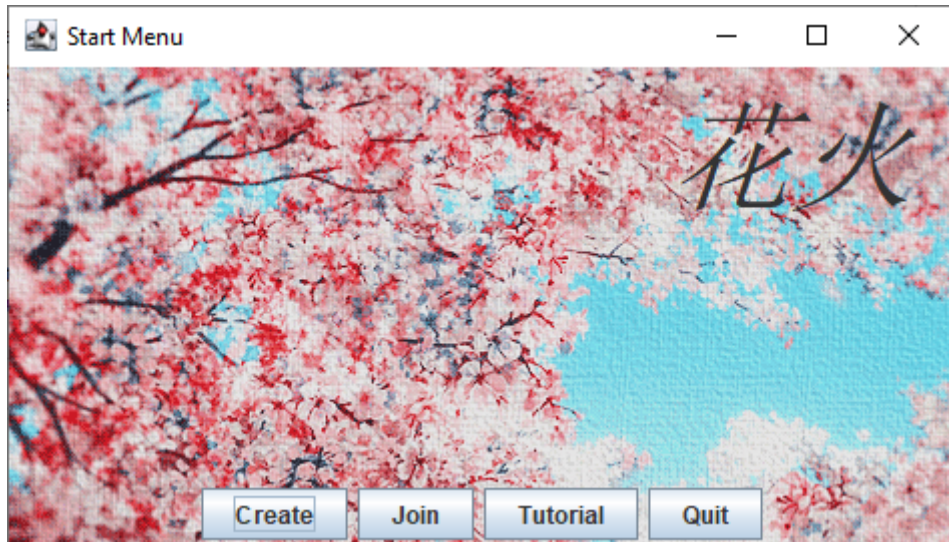


Figure 3.6: GUI of the start menu

StartMenu class keeps in track of which button is clicked in start menu. There are Three options player can select according the figure above.

1. Start: Start hosting a Hanabi Game
2. Join: Join a public Hanabi Game
3. Quit: Quit the Hanabi Game
4. Tutorial: Tutorial the Hanabi Game

Once a button is clicked, StartMenu class triggers the corresponding functionality and inform the messenger class. In addition, AI players do not use StartMenu to interact with the game since AI players live in host's program.

#### **Dependencies:**

StartMenu class inform the Messenger class to communicate with server. StartMenu

class extends the menu class since menu class contains common menu elements that StartMenu needs.

#### Attributes

- StartButton: JButton: A button corresponding to start game option.
- JoinButton: JButton : A button corresponding to join game option.
- QuitButton: JButton : A button corresponding to quit game option.

#### Function

CreateClicked (): Void

- Pre-condition: Player is currently in Create game state.
- JoinClicked function listens to StartButton. Once the StartButton is clicked, CreateClicked function changes current game state to create game state.

- Inputs

None

- Outputs

The height and width of the menu frame in a pair

JoinClicked (): Void

- Pre-condition: Player is currently in Create game state.
- JoinClicked function listens to JoinButton. Once the JoinButton is clicked, JoinClicked function changes current game state to join game state.

- Inputs

None

- Outputs

The height and width of the menu frame in a pair

QuitClicked (): Void

- Pre-condition: Player can be in any game state except in game state.
- QuitClicked function listens to QuitButton. Once the QuitButton is clicked, the QuitClicked function confirms with player again before terminates the program.

- Inputs

None

- Outputs

The height and width of the menu frame in a pair

### 3.1.3.3 Create Menu

Figure 3.7: GUI of the create menu

CreateMenu class listens to the CreateButton. Once players confirm that they desired to create, CreateMenu creates a game and inform the messenger class. In addition, AI player cannot use this Menu class.

Dependencies:

CreateMenu class extends the menu class since menu class contains common menu elements that CreateMenu needs. CreateMenu class interacts with the state of the game

Attributes:

- NumberOfPlayer: a JTextField where a user can enter the number of players they wish to have in their new game

- NSID: a JTextField where a user can enter the NSID required to start a new game
- NumberOfAI: a JTextField where a user can enter the number of AI controlled player they want present in their new game
- TimeLimit: a JTextField where the user can enter the time of a player's turn in this new game

Functions:

CreateClicked(): void

Listens to CreateButton. Once the CreateButton is clicked, the function informs the messenger of the game the user wants to create and changes the current state to a waiting state.

#### 3.1.3.4 Quit Menu

QuitMenu class listens to the QuitButton. Once players confirm that they desired to quit, QuitMenu terminates the program and inform the messenger class. In addition, AI player cannot use QuitMenu.

#### Dependencies

- QuitMenu class inform the Messenger class to communicate with server.
- QuitMenu class extends the menu class since menu class contains common menu elements that QuitMenu needs.

#### Attribute

- QuitButton: JButton

A button for player to terminate the program at any time.

Function

QuitTriggered(): Void

- Pre-condition: Player can quit in any game state except in game state.
- Once quit button is pressed, this function confirms with players if they desire to quit the game. If player desired to quit the game, this function terminates the program. Otherwise players remain in game.

- Inputs

None



- Outputs

The height and width of the menu frame in a pair

### 3.1.3.5 Join Menu



Figure 3.8: GUI of the Join menu

Join Menu class Takes in the information of the player and game the player will join. After all the required information entered by the player and the player also clicks join button. Join Menu class alters player's current state to join game state and inform the messenger class. In addition, AI players do not use Join Menu to join a public Hanabi game since AI players live in host's program.

#### Dependencies:

JoinMenu class inform the Messenger class to communicate with server. JoinMenu class extends the menu class since menu class contains common menu elements that JoinMenu needs.

#### Attributes:

- NSID: JTextField : Store the NSID for the player.
- Gameid: JTextField : Store the game ID for an existing game.
- secret: JTextField : Store the secret code for join a specific game.
- JoinButton: JButton : A button for confirmation to join the specified game.

#### Functions:

Joinclikced(): Void :Joinclikced function listens to the JoinButton. Once the Join button is clicked, the function alters the player's current state to join game state.

Output:None

### 3.1.3.6 In Game Menu

InGameMenu class keep in track of which action button has been pressed. Once an action button is pressed, InGameMenu class informs the action class to execute the corresponding action. In addition, AI players do not use InGameMenu to select their desired actions.

#### Dependencies

InGameMenu class inform the Messenger class to communicate with server. InGameMenu class extends the menu class since menu class contains common menu elements that InGameMenu needs.

#### Attribute

- PlayButton JButton : A button corresponding to play action.
- DiscardButton : JButton : A button corresponding to discard action.
- InformButton: JButton :A button corresponding to inform action.

#### Function:

- Playclicked():( Triplet< *SeatIDInt*, *CardIndexInt*, *Action* >) :: Playclicked function listens to PlayButton. Once the PlayButton is clicked, the function informs the action class which card has been played. Returns a returns a Triple which contains SeatID(The position for a player who effects by the action) , CardIndex(Index of the desired card) and a play action( the action to be performed).
- Discardclicked(): Triplet< *SeatIDInt*, *CardIndexInt*, *Action* >):: Discardclicked function listens to DiscardButton. Once the DiscardButton is clicked, the function informs the action class which card has been discarded. Returns a returns a Triple which contains SeatID (The position for a player who effects by the action.), CardIndex (Index of the desired card) and a Discard action( the action to be performed).
- Informclicked(): Triplet< *SeatIDInt*, *CardIndexInt*, *Action* >) :: Informclicked function listens to InformButton. Once the InformButton is clicked, the function informs the action class which card has been played. Returns a returns a Triple which contains SeatID (The position for a player who effects by the action.), CardIndex (Index of the desired card) and a Inform action(the action to be performed).

- Return: Triple which contains SeatID(The position for a player who effects by the action.) ,CardIndex(Index of the desired card) and a hint action(The action need to be performed).

### 3.1.3.7 Notice Menu

Class NoticeMenu informs player when the class receive exception messages from filter class. The message player receive should describe what situation player is in (such as: Time out; Invalid move.).

#### Dependencies

NoticeMenu class receive a message from filter class and store the given message. NoticeMenu class extends the menu class since menu class contains common menu elements that NoticeMenu menu needs.

#### Attributes

- Notice: String :: Store the received message as a string type variable.
- CloseButton: JButton :: A button to close the display message pop-up window.

#### Functions:

- DisplayMessage(String): void DisplayMessage takes in a string of message and display the message in a pop-up window to inform the player's situation. DisplayMessage class also listens to the CloseButton. If CloseButton is clicked, the DisplayMessage window disappears.

## 3.1.4 Initialization Classes

### 3.1.4.1 Card

The card class represents a card object, which has a colour and a number. By default, the values of a card are unknown, and updated as they revealed to the players who are not the holder of the card. The deck, hands, and discard pile are all lists of cards, and are all kept within the model.

**Dependencies:** None

#### Attributes:

- Color: String :: A string representation of the cards colour, denoted by a single character for the valid colors

•Number: String :: A string representation of the cards' number, denoted by a single character of the valid numbers

**Functions:**

•getColor() :: A method that returns the cards color

Input: None

Output: a single character representing the color

Postconditions: None

•setColor() :: A method that sets the cards color

Precondition: the colors value is one of the following values (R,B,Y,W,G,?)

Input: A single character denoting the number of the card

Output: None

Postconditions: the color of the card is set/changed

•getNumber() :: A method that returns the cards current number

Input: None

Output: The number of the card as a string

Postconditions: None

•setNumber() :: A method that sets the value of the cards Number

Precondition: the numerical value being added is a value between 1 and 5

Input: A string of the valid numbers

Output: None

Postconditions: The number of the card is set/updated

•printCard() :: A method that prints the current information of the card, in a color+number format

Preconditions: None

Input: None

Output: A string representation of the card, in a color+number format

Postconditions: None

•Equals() :: A method that checks if the cards current string representation is equal to the value given

Preconditions: None Input: A string to compare the information Against

Output: true if the strings are equal; false otherwise

Postconditions: None

### 3.1.4.2 Deck

A Class that creates all cards available to the game, and adds them to the a deck

**Dependencies:**

Card :: This class depends on card. Without the use of the card class, it is unable to generate all necessary cards, and thus cannot insert meaningful data into the deck

Model :: This class uses the deck attribute in the model to store all cards it generates. It does not depend on it, but uses it as a place to store the deck once all the cards have been generated

**Functions:**

- createCard() :: A method that creates a list of all the cards available to the game  
Preconditions: None  
Input: none  
Output: A List of jcard's containing all available cards  
Postconditions: None
- filleDeck() :: A method updates the models deck with a list of all cards  
Preconditions: None  
Input: None  
Output: None  
Postconditions: models deck attribute is filled with all the games cards

### 3.1.4.3 Deal Hand

This class is responsible for the generation for the creation of a players hand.

**Dependencies:**

- Model :: It uses the model to gain access to the available hands for each of the players. When the hand is created, it updates the attributes stored in the model

**Attributes:**

NumberPlayers(): int :: Indicates how many players in the current game are human, thus indicating the rest are AI

**Functions:**

- makeHandsLists() :: A method that generates a hand for a given player, denoted by an integer index  
Preconditions: The integer given as an argument must be between 1 and 5  
Inputs: An integer index, which indicates which players are in need of a list  
Output: None

Postconditons: The players of the given index are given List of *< cards >* that have none of their values set

- `fillHandLists()` :: A method that updates the cards in the players hand to have a color and number  
Precondition: None  
Input: None  
Output: None  
Postconditions: All cards in the players' hands are updated to have the correct color and number

#### **3.1.4.4 AI Join**

The `AIJoin` class handles signaling to have a new AI controlled player join the game

##### **Dependencies:**

- `Messenger`: needs to hold a reference to the `Messenger` to pass information on to it
- `Model`: needs to employ some of the methods of a `Model`

##### **Inputs:**

- `String`: the NSID or secret key needed to join a game
- `Integer`: the number indication the number of AI controlled players will be included

##### **Attributes:**

- `NSID`: A string which stores the NSID to be used to join a AI controlled player to the game
- `Key`: A string which store the Key to be used to join a AI controlled player to the game

##### **Functions:**

- `setAIFlag():void` Use the method of a Model to flag the state of a player to AI
- `sendNSID(string): string` Given a string, send the argument to the Messenger to be handle
- `sendKey(string): string` Given a string, send the argument to the Messenger to be handle
- `createModel(int): void` Given a positive integer, create a number of models equal to this number

### 3.1.5 Server Communication Classes

#### 3.1.5.1 Messenger

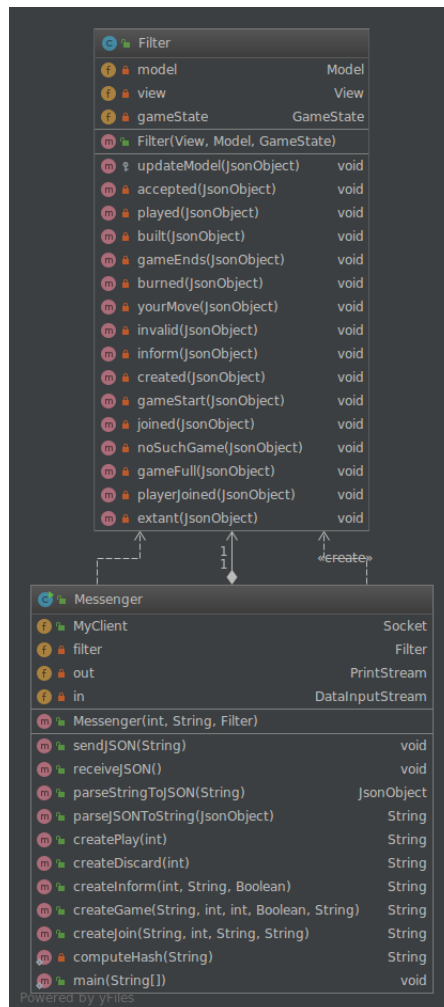


Figure 3.9: Server Communication Classes

Class Messenger is responsible for the communication between the server and the system. Main tasks being sending and receiving information from the server.

#### Dependencies:

Messenger has many dependencies. It is used by the Menu class, Action class. Messenger uses the Filter class to updates the Model.

**Attributes:**None

**Functions**



- `Sendjason(String): Void ::` Takes in a String that is in the format of JSON and sends to the server. These JSON strings will be created using methods of this class.  
Input: String in the form of JSON  
Output: Void
- `Receivejason(): String ::` Recieves JSON messages from the connected server. Once the message has been received it calls the `ParseJason` method to parse. After parsing `Filter.UpdateModel` is called to update the model of the received information.  
Input: None  
Output: String in the standard of JSON
- `ParseJason(String): < String > ::` This method take in a string in the standard of a JSON. Then it separates the string and returns the list of strings.  
Input: JSON string  
Output: list containing the parsed elements of the JSON String.
- `CreatePay(Int): String ::` This method is used to create the JSON object needed to send the play action notification to the server.  
Input: Integer noting the index of the card in your hand.  
Output: JSON string in the form of play action.
- `CreateDiscard(Int): String ::` This method is used to create the JSON object needed to send the discard action notification to the server.  
Input: Integer noting the index of the card in your hand.  
Output: JSON string in the form of discard action.
- `CreateInform(Int,String,String): Void ::` This method is used to create the JSON object needed to send the Inform action notification to the server.  
Input: Integer noting the player number, String with whether colour or number noting the information you want to send another String noting the card.  
Output: JSON string in the form of inform action.
- `StartConnection(): Void ::` Starts a connection with the server.  
Input: None

Output::Void

### 3.1.5.2 Filter

Class Filter is responsible for taking in a parsed JSON string and updating the needed model.

Attributes: A list of models

Functions

- UpdateModel(iString): Void :: Updates the model using the built in methods.
- UpdateHand(Int,String) Void :: Updates a hand list.
- UpdateFuse(int): Void :: Updates Fuse counter.
- Updateinfomation(Int): Void :: Updates the information tokens.
- UpdateDiscard(String): Void : : Updates Discard pile by adding cards.
- UpdateHint(String): Void :: Updates the hint list.
- UpdateNotice(String): Void :: Calls the notice display.

### 3.1.6 User Turn Classes

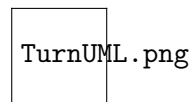


Figure 3.10: Turn Relationship Diagram

#### 3.1.6.1 HumanAI Checker

Class HumanAIChecker determine the player for the current turn is human or not.  
**Dependencies**

- Turn state machine use Action.Validation class to lock the player from performing invalid actions.
- HumanAIState: Check if the player is a human or not by accessing the HumanAIS-tate in model.

**Attribute**

isHuman : Boolean If true, then current player is human. Otherwise the current player is an AI player.

**Function**

- checkIsHuman():Boolean  
This function accesses HumanAIState in model to determine if player is human or not.  
Returns true if player is human, false otherwise.  
Output:  
Return: a Boolean flag indicate if the player is human or not.

**3.1.6.2 Action**

The Action class handles taking the details of the action to be performed by a player and passing it on to the Messenger to be handled there.

**Dependencies:**

Communicates with the Messenger.

**Inputs:**

- CardIndex Int: Index of a card
- SeatID Int: Identification of a seat
- Triplet(SeatID Int, CardIndex Int, Action String): Tuple holding the data necessary to describe an action fully

**Attributes:** None

**Functions:**

- playCard(CardIndex Int): Void  
Given an integer that represents a specific card in the current player's hand, communicates to the Messenger the action of playing that card.

- `discardCard(CardIndex Int): Void`  
Given an integer that represents a specific card in the current player's hand, communicates to the Messenger the action of discarding that card.
- `informPlay(SeatID Int, CardIndex Int): Void`  
Given two integers that represent a player and a card in the current player's hand respectively, communicates to the Messenger the action of informing that player about that card.
- `perform(Triplet; SeatID Int, CardIndex Int, Action Stringi): Void`  
Given a Triplet holding two integers and a string, communicates to the Messenger that exact action to be performed.

### 3.1.6.3 Action Validation

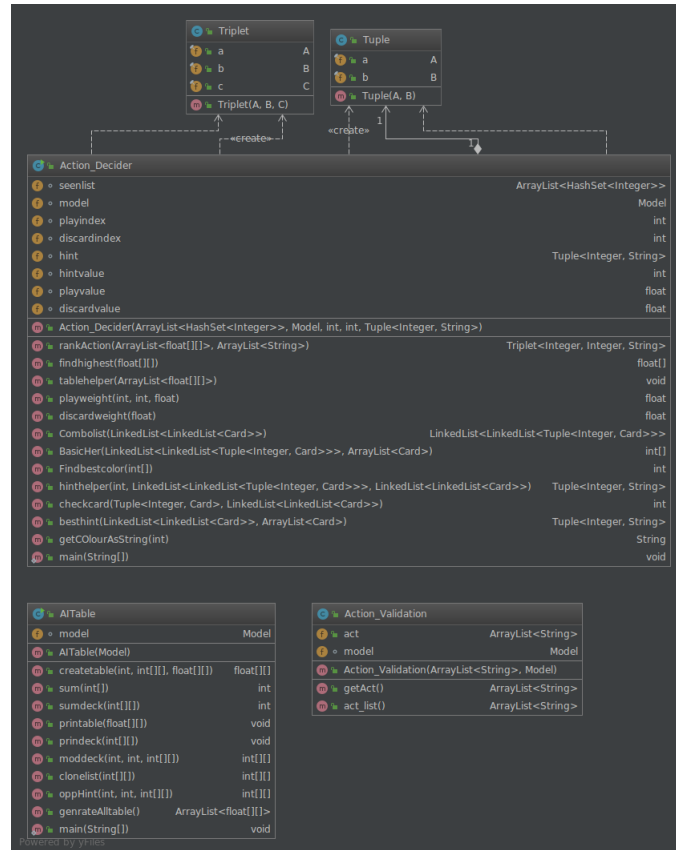


Figure 3.11: Action Validation

Class Action Validation determine all the valid performable actions for the current game state by checking the action requirement with access to the model.

### Dependencies

- Turn state machine will use Action Validation class to lock player from performing invalid actions.
- Model: InfoToken: Number of information tokens left for the current state. Discard and Hint actions both depends on the number of tokens left on the board.

**Attribute** : None

### Function

- CheckValidation(): ArrayList< *Action* >  
Consider all the actions and exclude the actions that cannot be perform for the current turn.  
Returns an Arraylist of valid actions for the current turn.

Output:

Return: an Arraylist of valid actions for the current turn.

#### 3.1.6.4 AI Action Decider

Class AI first takes in a list of perform-able and valid actions that parsed from Action Validation Class. Then AI class query the table within the model to obtain the cards' approximation information in AI player's hand. Base on the returned cards' information, AI class ranks all the actions and return the highest rank action with the card index.

### Dependencies

- Model:  
Hint stack: A stack of given hint for each player.
- Table: A stasis table that contain both the information of possible cards in player's hand and the probability of certainty.

## Inputs

- ValidActions : ArrayList<Action> ::

An Array list of valid actions that parsed from Action.Validation Class. It contains all the valid and perform-able actions for the current turn.

**Attribute** : None

## Function

- RankAction(ValidActions : ArrayList<Action>) : Triplet< SeatIDInt, CardIndexInt, Action >  
First, this class takes an input of ValidActions list and query the table to obtain the needed information to determine the current player's hand set. Then rank all the valid action to select a best action for the current turn. Finally, it returns a Triple which contains SeatID(The position for a player who effects by the action) ,CardIndex(Index of the desired card) and an action(a best action to be performed).

- QueryTable(Table: Hashtable<String,Int>): List< Pair < String, Int >>  
This function is a helper function for RankAction function. Querying the table within the model to obtain a list of pairs which contain both the information of possible cards in player's hand and the probability of certainty.

- CheckBestHint(Hints:List< String >): Triplet< SeatIDInt, CardIndexInt, Action >  
This function takes in a list of hints that each player received. It then calculates a best hint AI can give out. Returns a Triple which contains SeatID(The position for a player who effects by the action) ,CardIndex(Index of the desired card) and a hint action(The action need to be performed).

Output:

Return: Triple which contains SeatID(The position for a player who effects by the action) ,CardIndex(Index of the desired card) and a hint action(The action need to be performed).

### 3.1.6.5 AI Table

Class AITable has responsible for creating, updating and storing a stasis table. The AITable object is also stored inside the model.

## Dependencies

- Class AITable has access to the model to obtain all the necessary Board information create, update and store table.
- Required Board information list:
  - 1.TokenNumber: The number of available formation token.
  2. FireworkStack: The Fire work stack that is on the board.
  3. Playerhand: Cards in each players' hand.
  4. DiscardPile: The pile contains all the discarded cards.
  5. Deck: Deck contains all the cards left for the current game.

### Attribute

- Table: Hashtable< *String*, *Int* >
- Table: A stasis table that contain both the information of possible cards in player's hand and the probability of certainty.

### Function

- Tableupdate():void  
This function updates the stasis table since each time model gets updated, the table need also to be updated to match the latest changes.
- BuildTable():Hashtable< *String*, *Int* >  
This function accesses to the model and obtain information in Required Board information list. Use the obtained information to build a probability stasis table.

## 4 Amendments to the Design and Requirements

### 4.1 Messenger and Filter

Multiple amendments needed to be made to both messenger and filter classes. These were made due to changes that was added later o the project and to fix errors within the original design. You can find the most up to date documentation here:

### 4.2 Menus

According to the design document section x.x.x we stated that the Menu classes would have a a direct connection to the View where the panels would have been created. This was changed to include a panel class for each menu frame. This helped us to keep a strong object oriented structure compared to the previously proposed design. This can be seen in the As built documentation section y.y.y.

### 4.3 Model

Many changes were made tot he Model class compared to the Model discussed in the design document.

### 4.4 AI

According to requirement section 4.3, we found three ways we could take to develop our AI. As we progressing, we realized Monte Carlo tree search and Reinforcement Learning were not good choices because those two way need to simulate the game state transition to mimic the game in order to predict future moves, and this is heavy cost. To avoid doing this, we took probability feature from naive bayes combined with heuristic evaluation. With this new way, we would be able to implement a advanced AI to make decisions. The implementation of this AI tends to avoid risks and aid human players. The AI keep in track of a probability table for each card in its hand, which support the AI when it wants to make play and discard decision. With the probability table, AI knows more about what cards in its hand while the game gradually progress. Meanwhile, AI also keep in track of how many cards each player has for every color to determine which color can be the highest potential color to focus on. With heuristic function applied to each organized card pile, each color will be weighted differently base



on the number on the cards and how many cards. Once AI finds the best color to focus on, it will go through each color for that color to figure out what hint to give out to the specific player. Since the player might receive hints before which entails player might know something about the cards in hand. To avoiding giving duplicate or useless hint on card that is already hinted, AI could check on the flag associated the card. For play and discard, AI can use the probability table we built for each turn to check what cards are in AI's hand and how certain AI feel about that card. Once AI makes sure the card in its hand is a good card to play, AI could select from few action option it has base on the weight on every action AI could perform. Only the highest weighted action will be returned as the best action for the current game state.

As build design, according to what we have in design for the class in AI package. Everything went across our plan. However, we did end up with adding in more helper function in the AI action decider class to keep the single responsibility for each main function. There are many functions server as helper function such as: sum function, calculate highest probability for card function, find highest heuristic function. Beside from the extra functions, we also added in Tuple class and Triplet class as helper class to store and return multiple elements. Since sometimes not only we need the card object and also the player's ID who holds the card, a tuple or triplet can be really handy to keep in track of the extra infos.

Overall, the AI changed from requirement, and follows the design. The final AI acts really similar to what we described in the design doc. For user, AI should felt like same as we stated in the requirement documentation.

## 5 Known Bugs and Requires Implementation

In this section we present the know bugs that exists within our software and potential solutions to them and the classes that needs to be implemented to successfully execute the game as intended.

### 5.1 Action

Once the player clicks a action such as play card the action class should be called to send the correct message to the server. This class was not implemented due to time constrains. You can find all the required information necessary under the design document to implement this class.

### 5.2 AI Join

In the late stage of the project, the issue of AI join was found and appeared different than what group was expecting. The way of implementation that stated in design stage was not suitable for solving this problem. The solution to this issue is re-implement separate special filter class for AI to join the game, However,due to limit of time resource, AI join feature was not able to be implemented.

### 5.3 Thread Issue

When a player create a game and enters the in game state Java Swing panels stops displaying. Once of the suspected cause of this bug is that we attempted to run both the server listener and the Java Swing frames on the same thread. Therefore a solution would be to add a extra thread to the View class so Messenger and View do not block each other.

## 6 Summary

Now, the document is concluded with final summary. First, the user manual was discussed, outlining each action and a tutorial. A brief description of the UI was also provided in this section. Next, the "As built Documentation" was discussed. This explained what had been completed, and how it differed from what was specified at design. Overall, these changes varied from significant to small, only revealing small issues within the design process. This shown in the amendments. One amendment, for example, was to the menus, which showed how a panel was to be included. Finally, the document concludes by discussing things that were not implemented, and known bugs regarding threading issues. Overall, though not everything works, this document has revealed what needs to be changed, and what has been made so far