# SOFTWARE DESING SPECIFICATION

# 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

March 4, 2019

# Contents

**1 Introduction** **3**
1.1 Purpose . . . 3
1.2 Project Scope Review . . . 3
1.3 The Game . . . 3
1.4 The Team . . . 4
1.5 Document Conventions . . . 4
1.6 Glossary . . . 4
1.7 Overview of Remaining Chapters . . . 5

**2 System Architecture** **6**
2.1 Considered architectures . . . 6
2.1.1 Model-View-Controller . . . 6
2.1.2 Event-Driven . . . 6
2.1.3 Pipe-And-Filter . . . 6
2.1.4 Blackboard . . . 7
2.2 Hybrid Architecture . . . 7
2.2.1 Top-Level . . . 7
2.2.2 Mid-Level . . . 8
2.2.3 Low-Level . . . 9

**3 Detailed System Design** **11**
3.1 Overview . . . 11
3.2 Classes . . . 12
3.2.1 Model . . . 12
3.2.2 View . . . 17
3.2.3 State Machines . . . 18
3.2.4 Menus . . . 23
3.2.5 Initialization Classes . . . 31
3.2.6 Server Communication Classes . . . 36
3.2.7 User Turn Classes . . . 38

# 1 Introduction

## 1.1 Purpose

The Purpose of this document is to provide a detailed description of the game's overall design. More specifically, it will describe the structure/ backbone of the program in terms of architecture and class relationships. It will illustrate the architecture's purpose, how it ensures stable and consistent development, and overall design patterns useful for class diagrams/ relationships. Classes and class descriptions are also expanded upon, in addition their dependencies, uses and high-level descriptions of classes. Note that this design will be accomplished via an object-oriented approach. This document is meant to be presented to the code team and development team, as reference for the testing/coding phase.

## 1.2 Project Scope Review

This document also makes reference to the requirements document. The following section first reviews the scope of the project. The product/software will be a game developed for the University of Saskatchewan. It is designed to replicate the experience of the physical card game, "Hanabi," but on a digital platform. By letting users interact with a digital version of this game, the system will allow human players to play alongside computer players.

The program will present the user with an intuitive interface, that they will use to play the game. It will automatically perform essential functions of the game, such as drawing or shuffling cards. It will also work by communicating with a remote server.

## 1.3 The Game

It is then worth reviewing what the game called "Hanabi" is. Hanabi is a cooperative card game in which players are aware of other players' cards but not of their own. They must attempt to play a series of cards in sequential order to create the largest fireworks show possible. The players must accomplish this by:

- giving other players information on the cards in their hand,
- discarding cards they deem unnecessary to the fireworks display, and
- playing cards to grow the display's size.

This must be done while also managing limited pools of information tokens and fuses.

## 1.4 The Team

With this Design introduction, we also re-introduce the skills of those who will be working on the project going forward. The development team is made up of five member currently enrolled in Computer Science program at the University of Saskatchewan. The group possesses relevant course experience in artificial intelligence, computer networking, game design and development, Java programming, group and pair programming, English writing, and engineering.

## 1.5 Document Conventions

Furthermore, this team has followed a convention for writing this document. It is written using the ISO/IEC/IEEE International Standards for Requirements Engineering.

## 1.6 Glossary

As part of the IEEE convention, this document employs a set of terms whose meanings need standardization.The following table contains commonly used words and phrases in the document that is specific to this project:

**User**: The human interacting with the program (is a Player).

**Player**: The name of a specific actor (human of computer controlled, can be the user).

**System**: The program on the local machine.

**Server**: What the client is bouncing information off of.

**Card Stack**: This refers to the available cards that can be distributed to a player.

**Discard Pile**: All of the used cards.

**Hint Tokens**: A hint token gives a player the ability to use the hint action. there are total of 8 tokens and each hint takes one out.

**Fuse Tokens**: This token decreases each time a invalid card is played.There are 3 token in total. Once all tokens are spent the game ends.

**Fire works stack**: Stacks that are created at the center of the game. These stacks of cards are sorted by color and numbers. Players needs to place them from their card hand.

**NSID** : Network Services ID of the University Of Saskatchewan. An unique ID given to each student.

## 1.7 Overview of Remaining Chapters

This concludes the design introduction and requirements review. Following that, the document will first explain considered architectures. It will then discuss the rationale behind our chosen architecture, which is a Hybrid. It will discuss its necessity and purpose of each layer in detail. Then, class descriptions will be given in thorough detail, outlining attributes, functions and dependencies. Finally, we conclude by presenting a large class diagram, which shows the overall relationships between the classes and the architecture.

# 2 System Architecture

## 2.1 Considered architectures

With the introduction finished, the document will lay out the events by which the architecture's design was chosen. To do so, considered architectures are first discussed and reasoned with.

### 2.1.1 Model-View-Controller

A Model-View-Controller architecture was initially considered, as it is suitable for this sort of software. Since it is an adaptation of a card game, and is planned to have a Graphical User Interface, a Model-View-Controller architecture fits well.

However, given the requirements demanded by the software, the Controller module of this architecture would need a complex internal structure that should be described as its own sub-architecture. Thus, while Model-View-Controller is sufficient as a high-level architectural description of the software, it is not sufficient as a low-level description.

### 2.1.2 Event-Driven

In addition to MVC, an Event-Driven architecture was considered. Writing software based on this architecture would be advantageous, as it would produce multiple decoupled modules, making the final software highly maintainable.

However, it has the disadvantage of poor organization compared to a Model-View-Controller architecture. Thus, it was decided that the software would be structured partially around states, and an Event-Driven architecture would be needed for the management of a finite state machine. In conclusion, this architecture could be used as the structure of a mid-level component to best meet its requirements.

### 2.1.3 Pipe-And-Filter

Another software architecture considered was a Pipe-And-Filter architecture. This was not found to be useful for the highest levels of our software's design. This is especially true, given the software requirements included a Graphical User Interface, which this architecture does not favor. Conversely, it was decided that a Pipe-And-Filter would better meet the software requirements as a low-level component architecture.

### 2.1.4 Blackboard

Finally, the Blackboard software architecture was considered. However, it was deemed unsuitable for this project, since it is most suitable for problems in which no deterministic solution strategies are known.

## 2.2 Hybrid Architecture

With considerations in mind, the following section will introduce the chosen architecture. Since none of the architectures satisfied all of the software requirements on their own, a hybrid architecture using Model-View-Controller, Event-Driven, and Pipe-And-Filter was decided upon. This architecture consists of 3 levels: top-level, mid-level, and low-level. In the following, the document describes each of the levels, and provides reasoning behind their inclusion.

### 2.2.1 Top-Level

As previously discussed, the Model-View-Controller fits best into a system that takes constant human input. For this reason, the top level of the architecture is a Model View Controller. As shown in the figure 2.1, the Controller was modified to include two communication modules instead of one. Having this allows the system to update both the view and the model depending on the input.

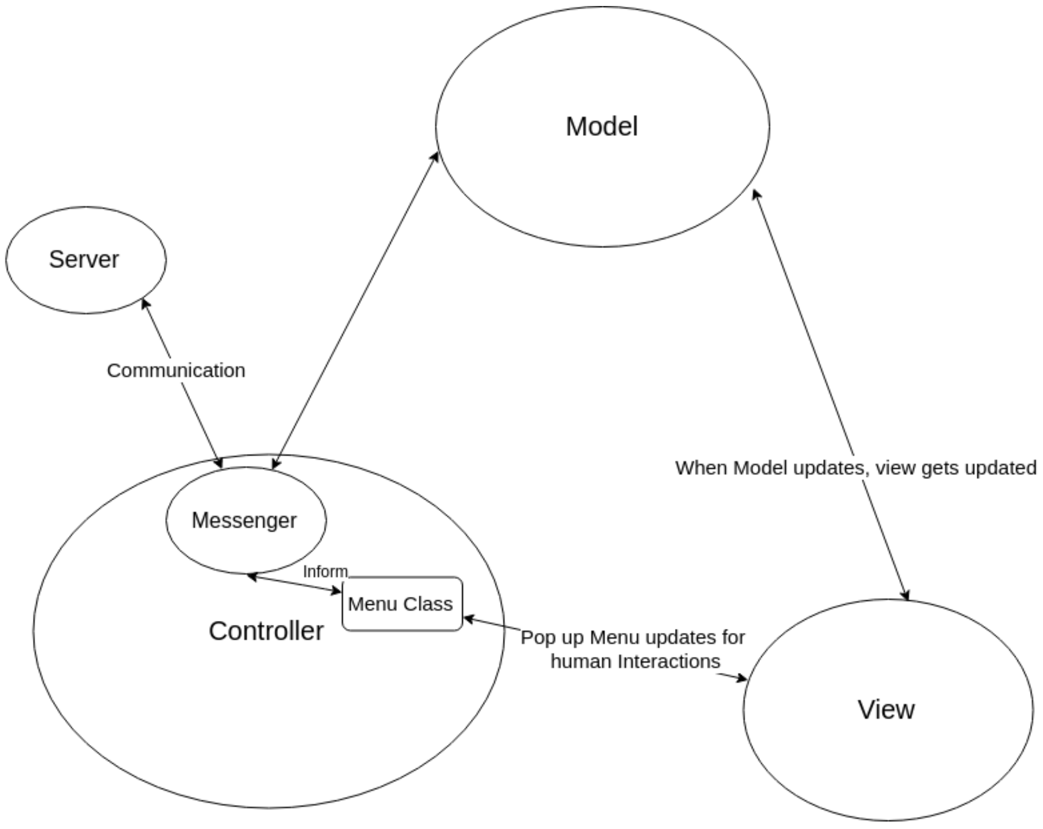

Model
Server
Communication
When Model updates, view gets updated
Messenger
Inform
Menu Class
Controller
Pop up Menu updates for human Interactions
View

Figure 2.1: High-level architectural design of our software incorporating Model-View-Controller

### 2.2.2 Mid-Level

With the high-level software architecture visualized and reasoning laid out, the mid-level architecture is described next. With team collaboration, a decision was made to include an intermediate architectural layer between the high-level Model-View-Controller and the low-level Pipe-And-Filter.

To best fulfill the software requirements and represent the Hanabi game, a finite state machine is included. A finite state machine, would allow the Hanabi game to be broken down into manageable pieces, thus allowing better control the flow of events. It was determined that this mid-level component of the software is best implemented with an Event-Driven architecture.

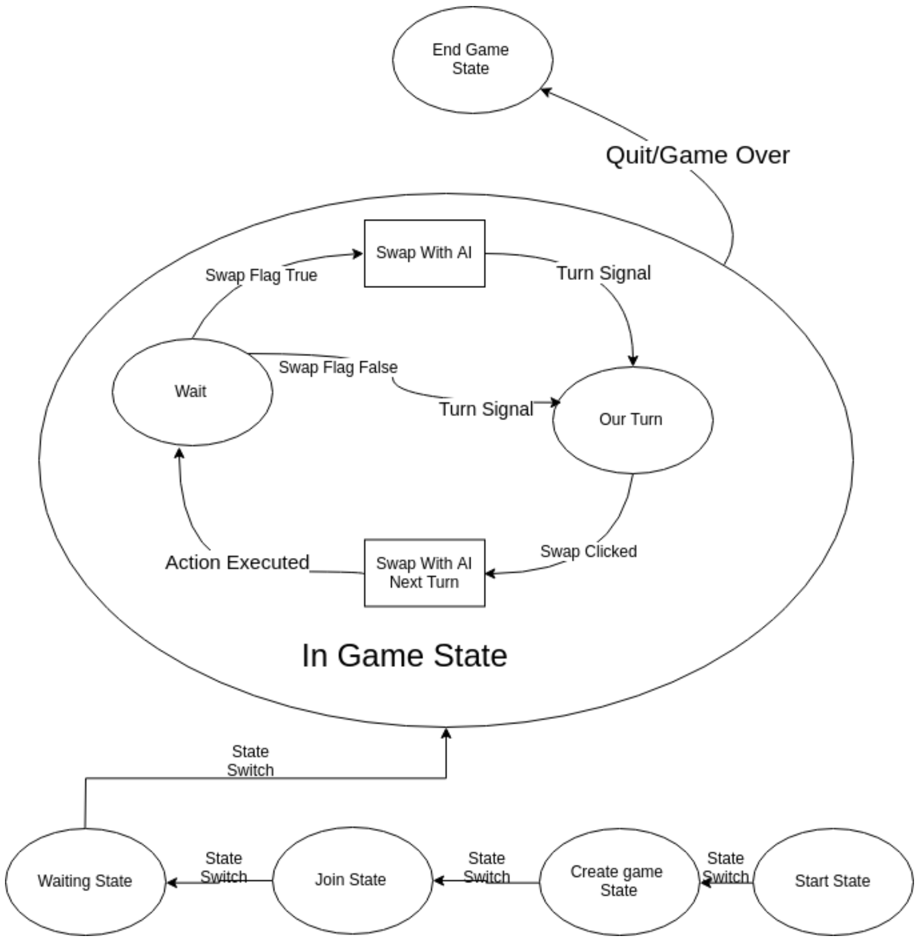


Figure 2.2: Mid-level architectural design of our software incorporating Model-View-Controller

### 2.2.3 Low-Level

The final section of the hybrid architecture is on the structure of low-level components of our software. A Pipe-And-Filter architecture was decided for the internal structure of the "Our Turn" state of our finite state machine, as described in figure 2.2 (above). This architecture best describes the events of this state, because it describes a series of very predictable steps with simple and manageable branching paths. This effectively fulfills the software requirements.

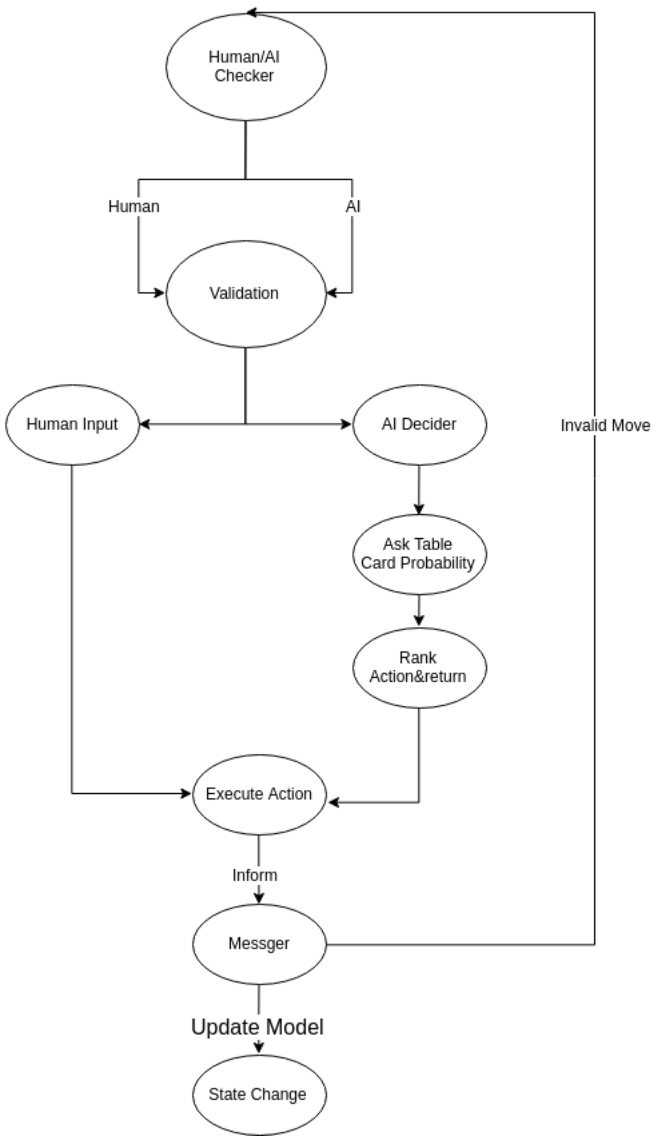


Figure 2.3: Low-level architectural design of our software incorporating Model-View-Controller

# 3 Detailed System Design

## 3.1 Overview

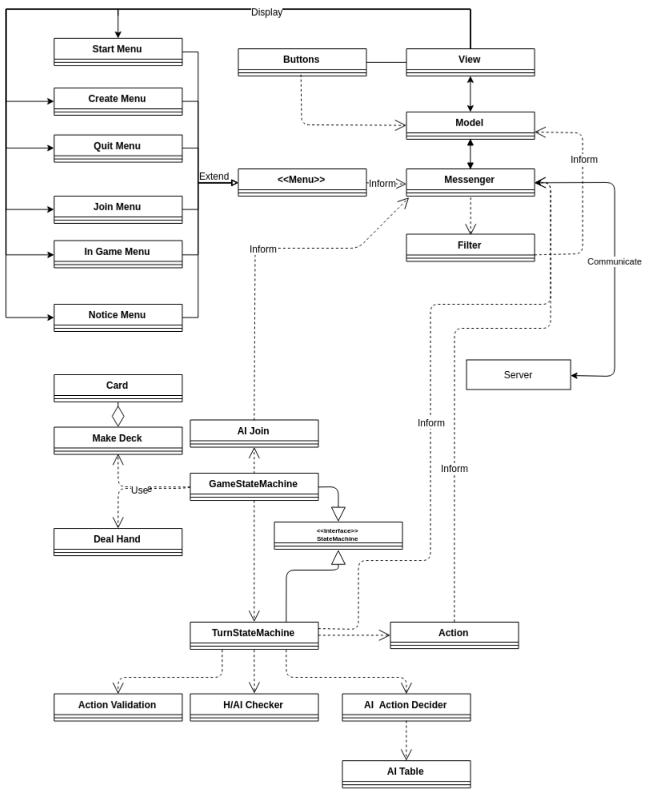
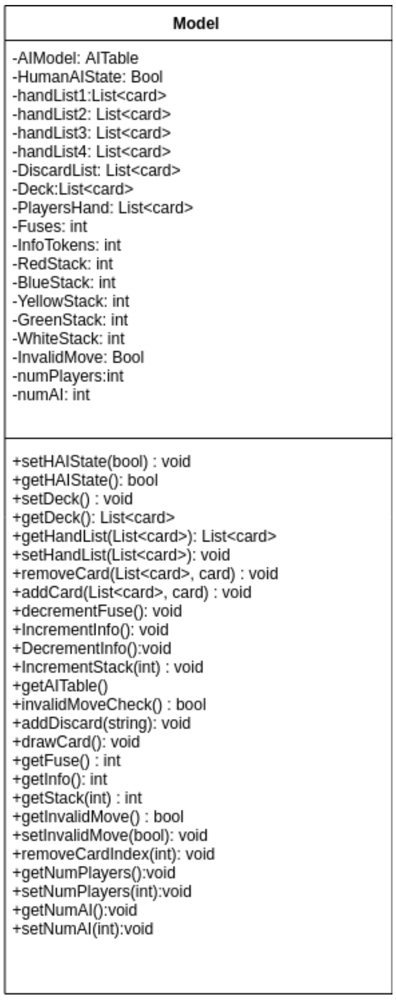


Figure 3.1: Abstracted overview of complete UML diagram

Now that the system architecture has been highlighted and explained, the following section outlines and details all classes and their relationships. Each class is carefully chosen to match the architecture and purpose, as per each layer. Thus, all relationships are deliberate, consistent, and demonstrate that they carefully follow the architecture as a guideline.

## 3.2 Classes

### 3.2.1 Model



```
Model
-AIModel: AITable
-HumanAIState: Bool
-handList1:List<card>
-handList2: List<card>
-handList3: List<card>
-handList4: List<card>
-DiscardList: List<card>
-Deck:List<card>
-PlayersHand: List<card>
-Fuses: int
-InfoTokens: int
-RedStack: int
-BlueStack: int
-YellowStack: int
-GreenStack: int
-WhiteStack: int
-InvalidMove: Bool
-numPlayers:int
-numAI: int

+setHAIState(bool) : void
+getHAIState(): bool
+setDeck() : void
+getDeck(): List<card>
+getHandList(List<card>): List<card>
+setHandList(List<card>): void
+removeCard(List<card>, card) : void
+addCard(List<card>, card) : void
+decrementFuse(): void
+IncrementInfo(): void
+DecrementInfo():void
+IncrementStack(int) : void
+getAITable()
+invalidMoveCheck() : bool
+addDiscard(string): void
+drawCard(): void
+getFuse() : int
+getInfo(): int
+getStack(int) : int
+getInvalidMove() : bool
+setInvalidMove(bool): void
+removeCardIndex(int): void
+getNumPlayers():void
+setNumPlayers(int):void
+getNumAI():void
+setNumAI(int):void
```

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 my many other classes to get the necessary information. All of the attributes in this class are privet 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 attributes 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 used
- PlayersHand: List$< card >$: List of empty cad object representing the missing information
- Fuses: int: Integer to represents 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 $< cards >$, 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¿ : 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

### 3.2.2 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

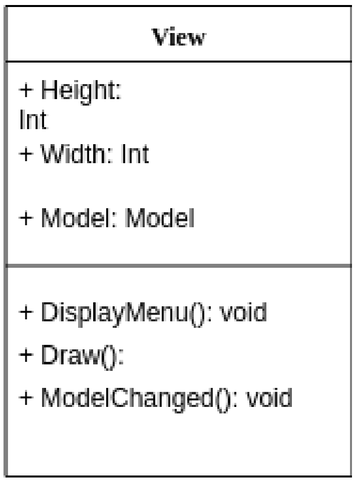


Figure 3.3: View Class UML Segment

**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**:

- DisplayMenu():void
  Displays the current appropriate menu based on model information
- Draw(): void
  Draw into a frame the appropriate graphical elements
- ModelChanged(): void
  Listens for the Model changing to know to update the view

### 3.2.3 State Machines

#### 3.2.3.1 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

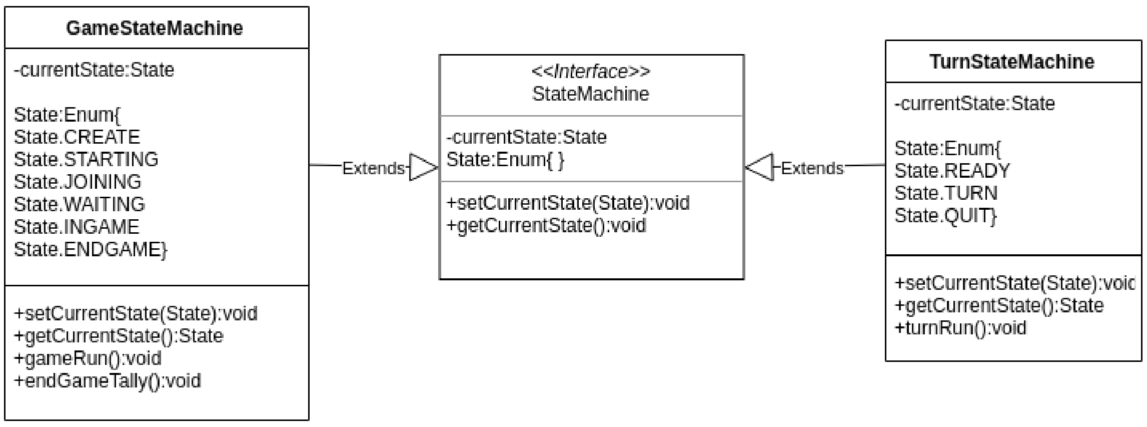


Figure 3.4: "State Machine Interfaceand Relationships"

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

#### 3.2.3.2 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

#### 3.2.3.3 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 quiting 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

### 3.2.4 Menus

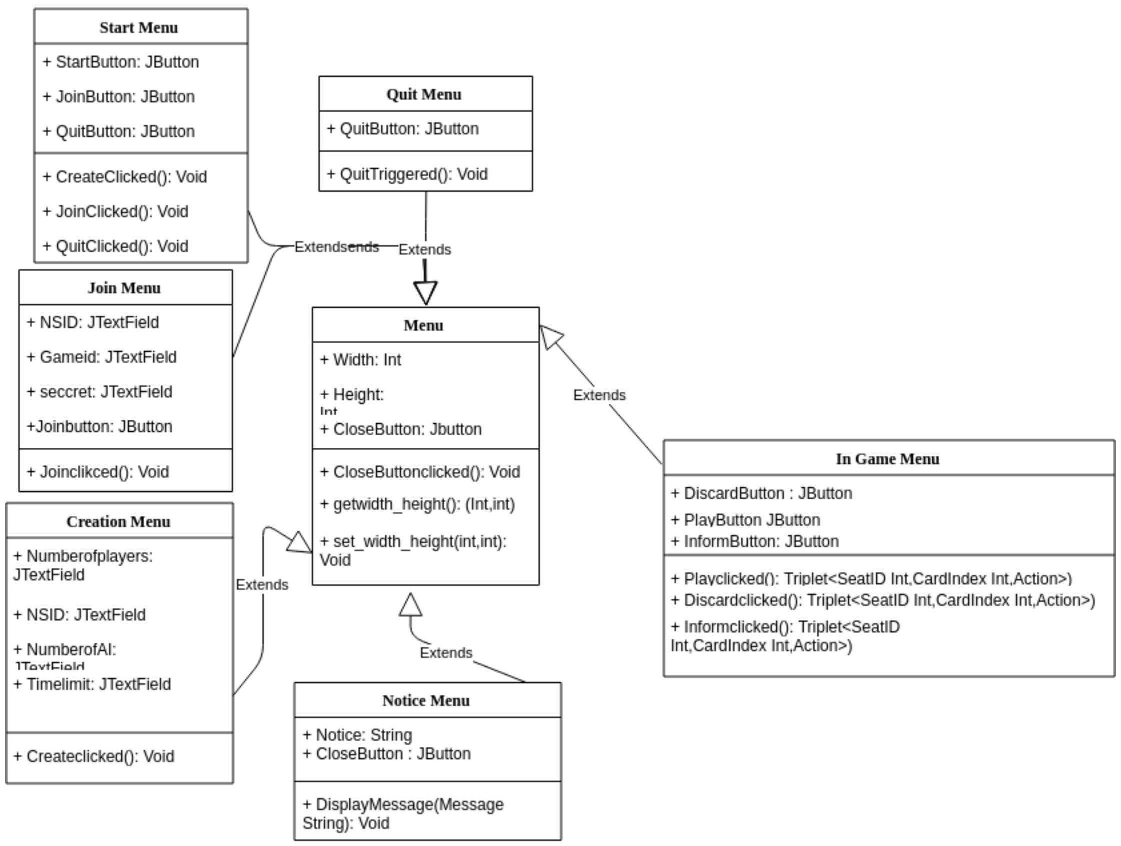


Figure 3.5: Menu Superclass and derived Subclasses

#### 3.2.4.1 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

#### 3.2.4.2 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

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

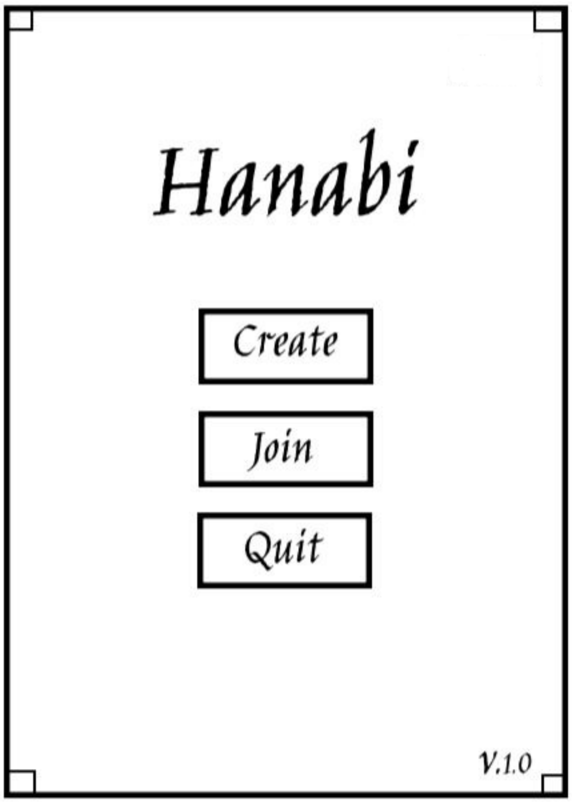


Figure 3.6: GUI of the start menu

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.

- JoincClicked 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

#### 3.2.4.3 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:

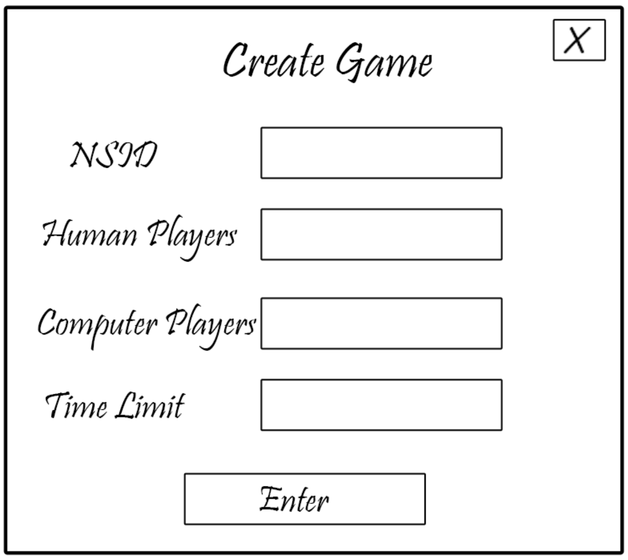


Figure 3.7: GUI of the create menu

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

#### 3.2.4.4 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

#### 3.2.4.5 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**:

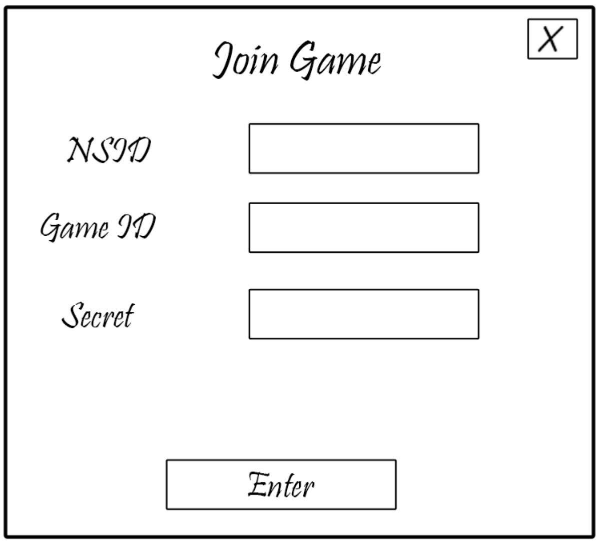


Figure 3.8: GUI of the Join menu

- 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

#### 3.2.4.6 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).

#### 3.2.4.7 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.

### 3.2.5 Initialization Classes

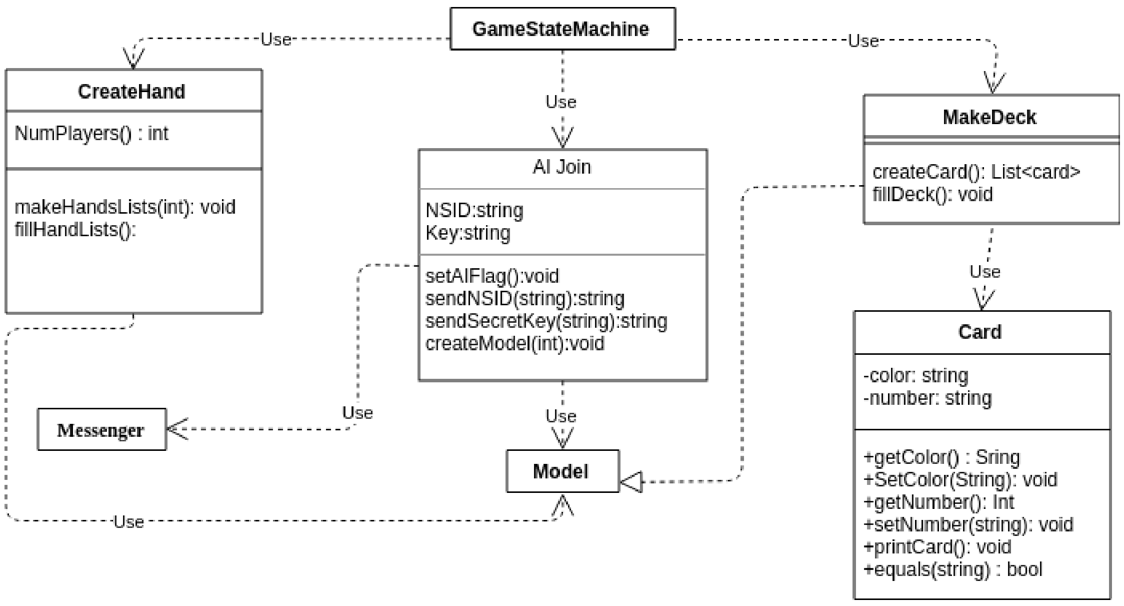


Figure 3.9: Initialization Relationship Diagram

#### 3.2.5.1 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 of 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

#### 3.2.5.2 Make 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

**Attributes**:None

**Functions**:

- createCard() :: A method that creates a list of all the cards available to the game
  Preconditions: None
  Input: none
  Output: A List of ¡card¿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

#### 3.2.5.3 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
  Postcondiditons: 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

#### 3.2.5.4 AI Join

The AI_Join 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

### 3.2.6 Server Communication Classes

#### 3.2.6.1 Messenger

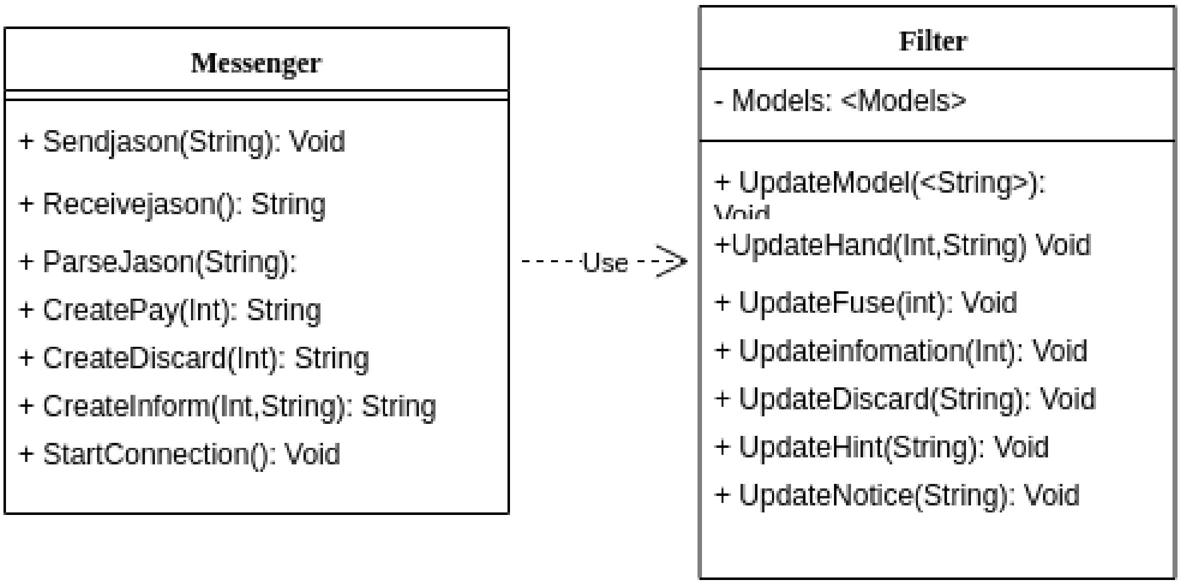

Messenger
+ Sendjason(String): Void
+ Receivejason(): String
+ ParseJason(String):
+ CreatePay(Int): String
+ CreateDiscard(Int): String
+ CreateInform(Int,String): String
+ StartConnection(): Void
Use
Filter
- Models: <Models>
+ UpdateModel(<String>): Void
+UpdateHand(Int,String) Void
+ UpdateFuse(int): Void
+ Updateinfomation(Int): Void
+ UpdateDiscard(String): Void
+ UpdateHint(String): Void
+ UpdateNotice(String): Void

Figure 3.10: 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

#### 3.2.6.2 Filter

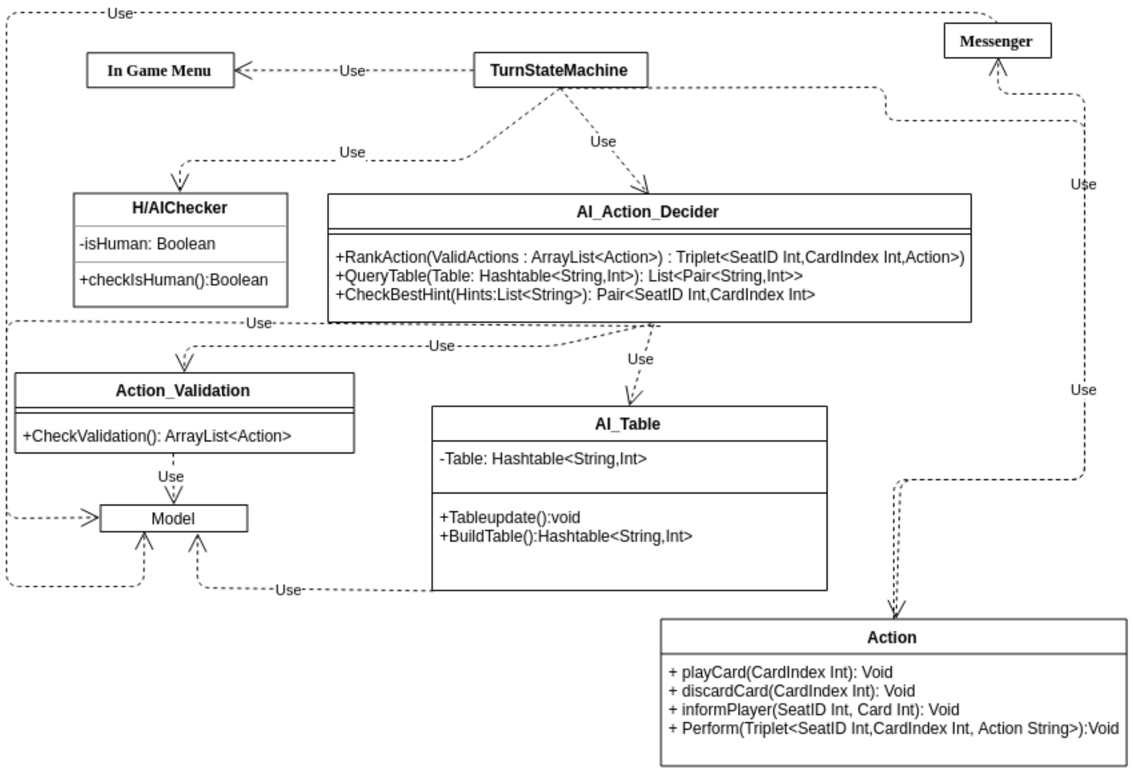
Class Filter is responsible for taking in a parsed JSON string and updating the needed model.
Attributes: A list of models

Functions

- UpdateModel(¡String¿): 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.

### 3.2.7 User Turn Classes



Use
Messenger
In Game Menu
Use
TurnStateMachine
Use
Use
Use
H/AIChecker
-isHuman: Boolean
+checkIsHuman():Boolean
AI_Action_Decider
+RankAction(ValidActions : ArrayList<Action>) : Triplet<SeatID Int,CardIndex Int,Action>)
+QueryTable(Table: Hashtable<String,Int>): List<Pair<String,Int>>
+CheckBestHint(Hints:List<String>): Pair<SeatID Int,CardIndex Int>
Use
Use
Use
Use
Action_Validation
+CheckValidation(): ArrayList<Action>
AI_Table
-Table: Hashtable<String,Int>
+Tableupdate():void
+BuildTable():Hashtable<String,Int>
Use
Model
Use
Action
+ playCard(CardIndex Int): Void
+ discardCard(CardIndex Int): Void
+ informPlayer(SeatID Int, Card Int): Void
+ Perform(Triplet<SeatID Int,CardIndex Int, Action String>):Void

Figure 3.11: Turn Relationship Diagram

#### 3.2.7.1 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 HumanAIState 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.

#### 3.2.7.2 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.

- informPlaye(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 String¿): Void
  Given a Triplet holding two integers and a string, communicates to the Messenger that exact action to be performed.

#### 3.2.7.3 Action Validation

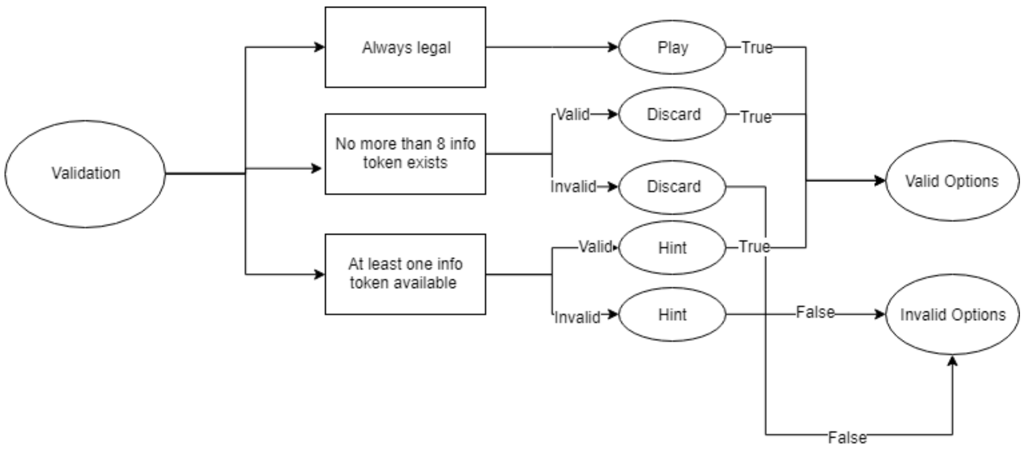


Figure 3.12: Action Validation

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.

#### 3.2.7.4 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).

#### 3.2.7.5 AI Table

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

**Dependencies**

- Class AI_Table 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.