Hanabi Design Documentation

Mels Kolozyan
Xudong Li
Xufeng Lin
Evan Semenoff
Tiandu Xie

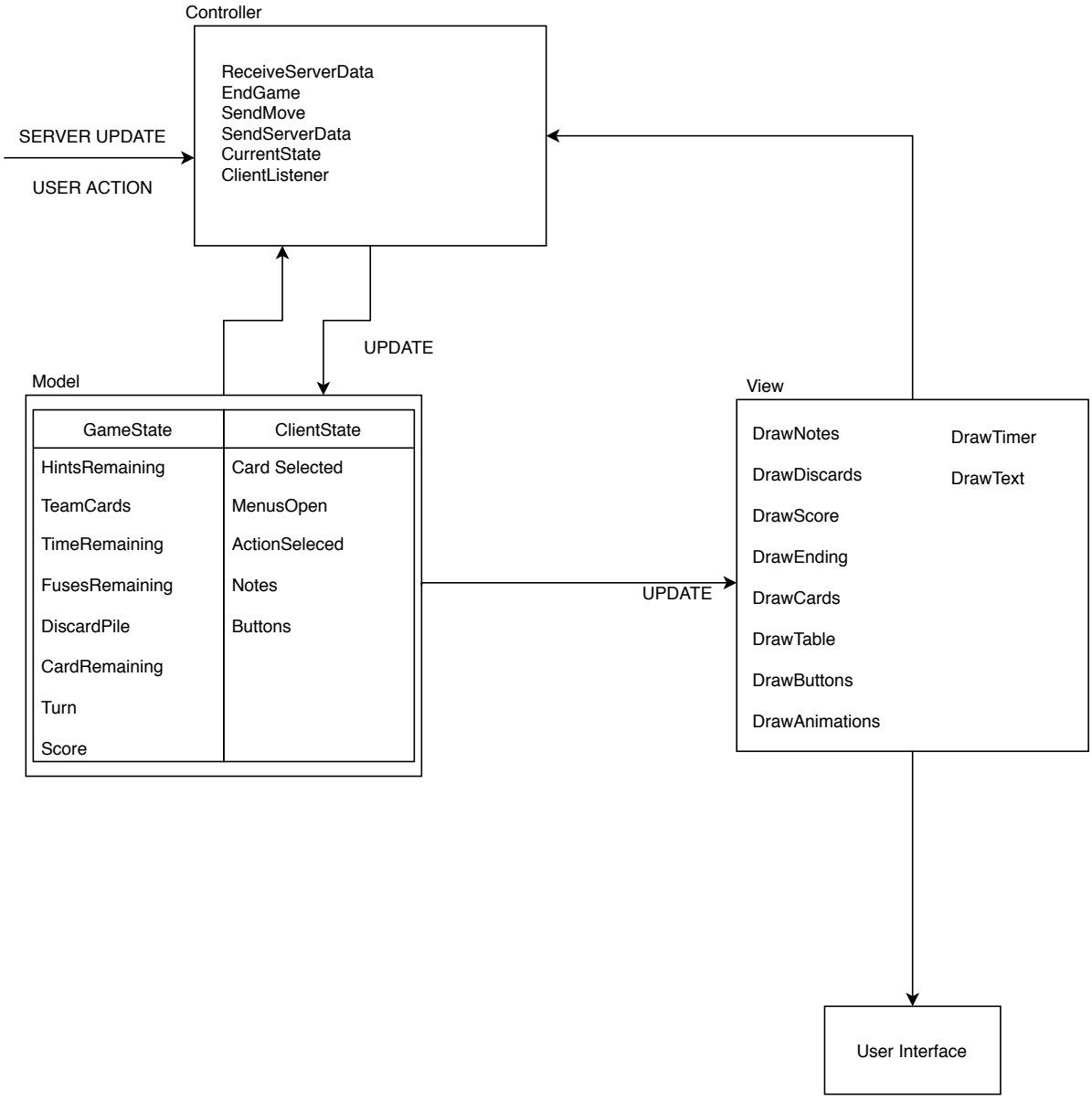
March 3, 2019

Contents

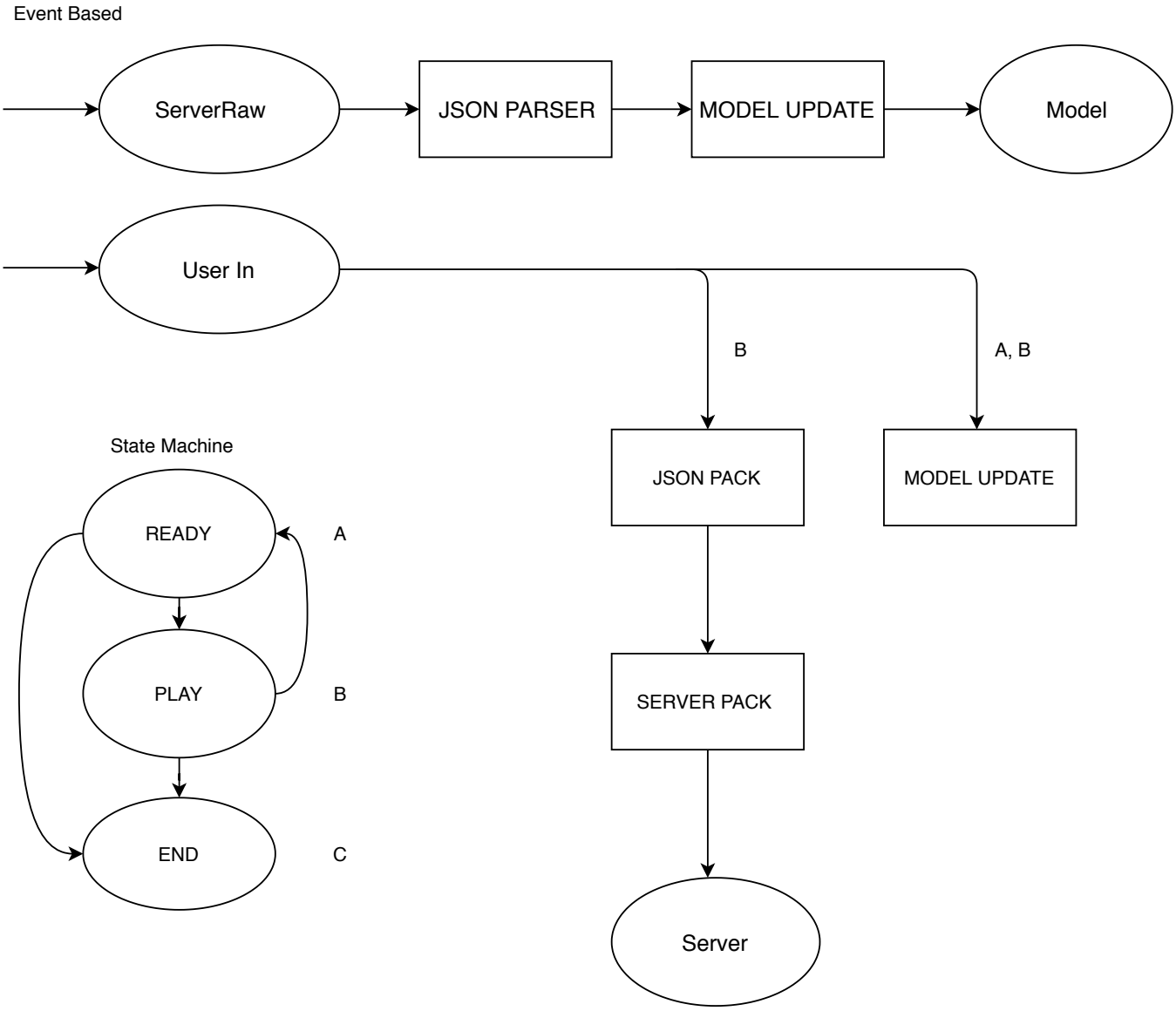
Part I
Architecture

1 Diagrams

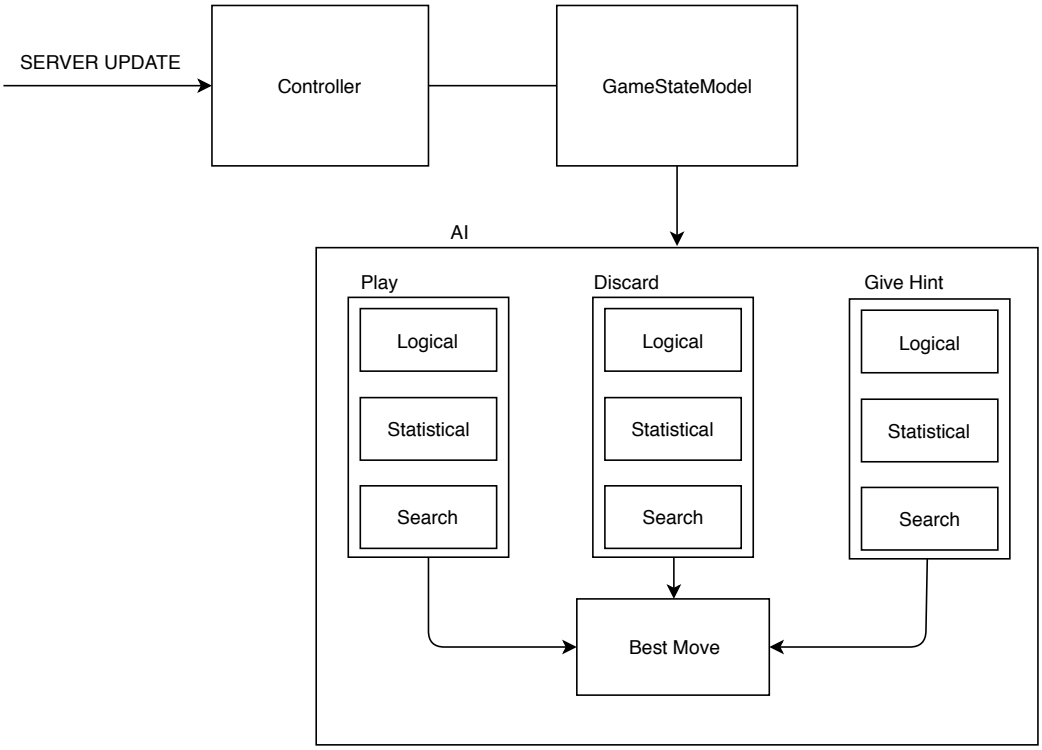
1.1 General Architecture



1.2 Controller Architecture



1.3 AI Architecture



2 Explanation

Client :

The architecture picked was a simple hybrid architecture. The outermost architecture is model-view-controller (MVC) and the controller itself is pipes-and-filters (P&F).

MVC was picked because it best represents the way the game is actually played (in a largely abstract sense), and allows the separation of the various aspects into concise, logical modules. Additionally, by separating the three main concerns of the program, it will allow our team to work in parallel during the implementation phase, and allow us to all focus on the various parts which we have experience or expertise. It was also decided to not have an explict, singular model class, but to seperate the model into two different classes, 'GameState', and 'ClientState'. The game state is responsible for keeping track of the current state of the game being played, where client state is responsible for keeping track of the local client data. By seperating these, we expect that it will be easier to reason about state change that requires server communication, and state change that doesn't.

P&F was picked for the controller because the controller will essentially have a frequent stream of data to process and pass along to the model, from both the network, and to the network. The output of this stream of data is always reflected by a change in the model, as well as (occasionally) the packing and sending of network packets. Because the generation and reading of these network packets have multiple layers to their creation and parsing, the P&F architecture was picked to allow us to separate these layers into distinct, yet cohesive parts.

AI :

The architecture picked for the AI was built to externally match the view class. This way, the AI can essentially sit in the exact same place as the view, and reuse both the controller for communicating with the server, and the model for storing information. In fact, from the perspective of the controller and the model the AI is really no different than a player. This means in order for the AI to 'pick' a move, it sends all the same types of messages a player would send when clicking on cards.

Internally the structure of this AI class will be a pipes and filters architecture, which will be composed of a logic layer, a statistical layer, and finally (if time allows) a search layer. The idea of this is that the AI will have hard coded logic for obviously strong moves, and will fall back to the lower layers only if it can't find a strong move in the above layer. The AI will instantiate three of these layers which will be computed in parallel for the three moves (play, discard, give hint) and finally the evaluation of those three moves will be piped into a final decision layer which picks the move with the highest expected value.

3 Comparison with Other Architectures

Client :

Blackboard : Blackboard was not picked because it doesn't appropriately address the concerns of implementing a Hanabi client. The problem keeping track of the current game state and the control flow of such a program are well defined, concrete problems. Although it would be possible to implement the client with a blackboard style system, the increased complexity and potential concurrency issues would make it inappropriate.

This was the only other architecture considered (at a top level) for the client. All other architectures (database, finite state, interpreter) seemed so poorly suited for the process that they were trivially dismissed.

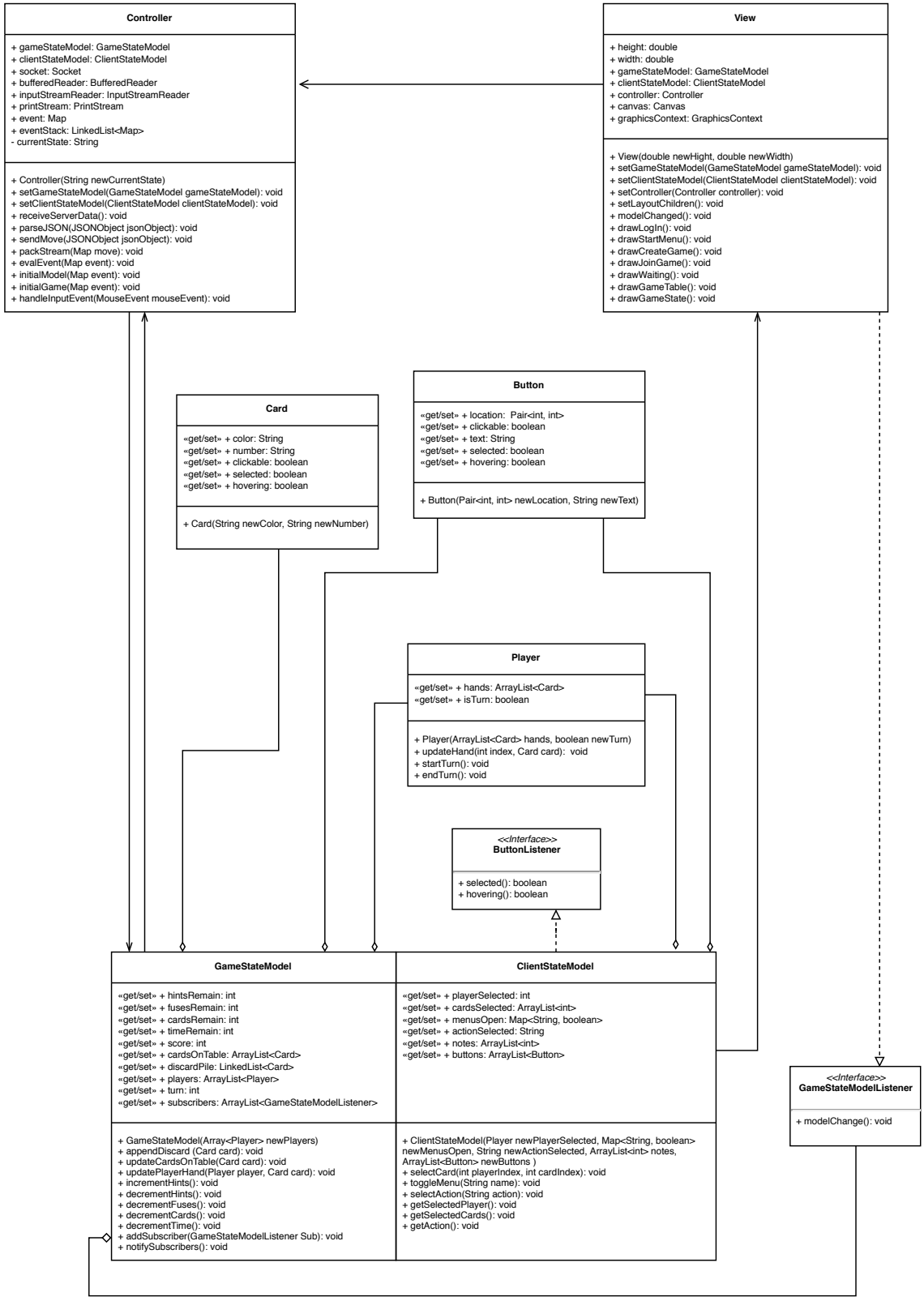
AI :

Blackboard : Here blackboard was again considered, but with more deliberation. The idea of putting all of the game state data in a public blackboard that various agents had access to to perform computations was considered, but ultimately it was decided that being able to reuse some of the pipes and filters architecture from the client would allow more code reuse while remaining effective.

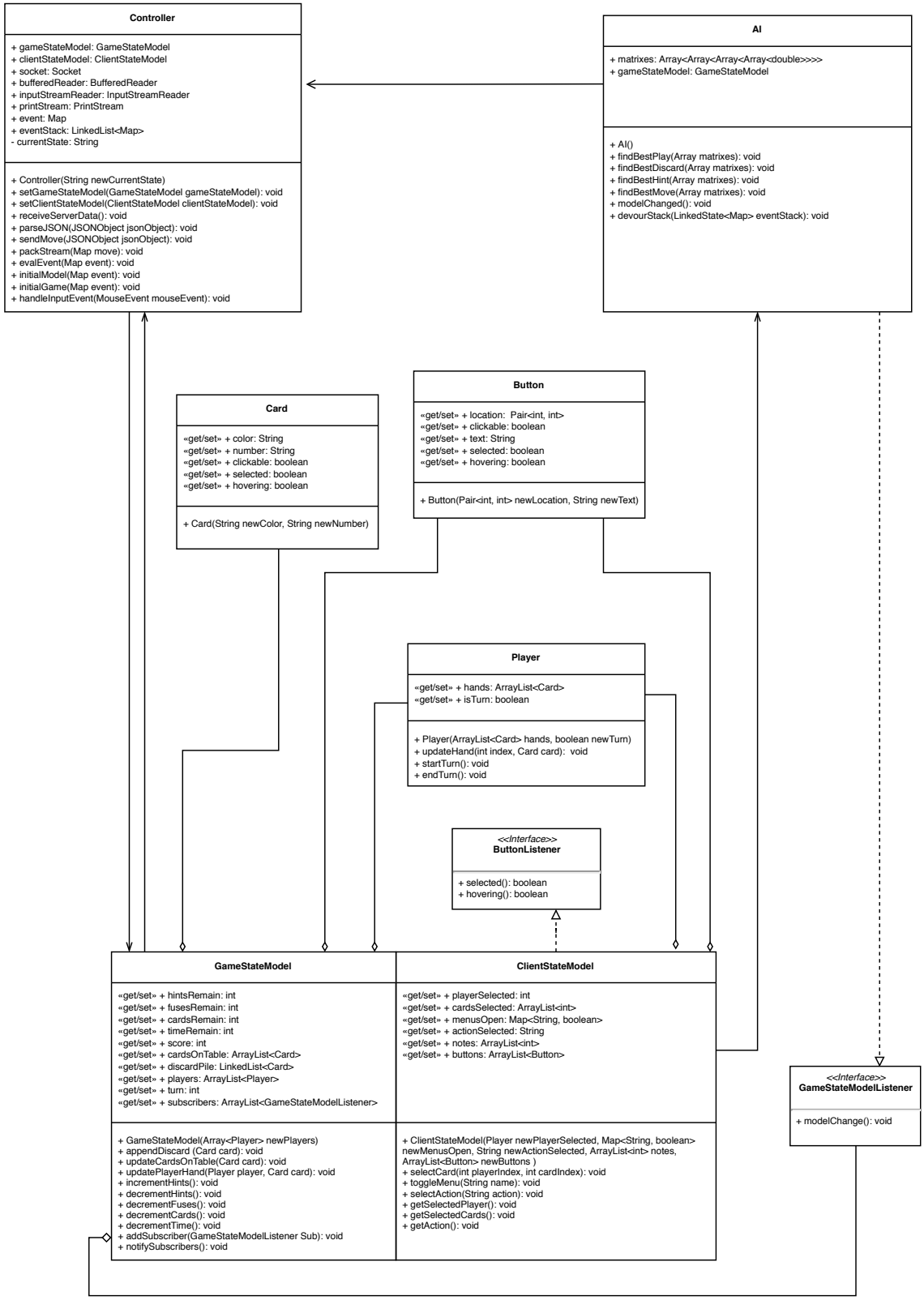
Part II
Design

4 Diagrams

4.1 MVC


Controller
+ gameStateModel: GameStateModel
+ clientStateModel: ClientStateModel
+ socket: Socket
+ bufferedReader: BufferedReader
+ inputStreamReader: InputStreamReader
+ printStream: PrintStream
+ event: Map
+ eventStack: LinkedList<Map>
- currentState: String
+ Controller(String newCurrentState)
+ setGameStateModel(GameStateModel gameStateModel): void
+ setClientStateModel(ClientStateModel clientStateModel): void
+ receiveServerData(): void
+ parseJSON(JSONObject jsonObject): void
+ sendMove(JSONObject jsonObject): void
+ packStream(Map move): void
+ evalEvent(Map event): void
+ initialModel(Map event): void
+ initialGame(Map event): void
+ handleInputEvent(MouseEvent mouseEvent): void
View
+ height: double
+ width: double
+ gameStateModel: GameStateModel
+ clientStateModel: ClientStateModel
+ controller: Controller
+ canvas: Canvas
+ graphicsContext: GraphicsContext
+ View(double newHight, double newWidth)
+ setGameStateModel(GameStateModel gameStateModel): void
+ setClientStateModel(ClientStateModel clientStateModel): void
+ setController(Controller controller): void
+ setLayoutChildren(): void
+ modelChanged(): void
+ drawLogIn(): void
+ drawStartMenu(): void
+ drawCreateGame(): void
+ drawJoinGame(): void
+ drawWaiting(): void
+ drawGameTable(): void
+ drawGameState(): void
Card
«get/set» + color: String
«get/set» + number: String
«get/set» + clickable: boolean
«get/set» + selected: boolean
«get/set» + hovering: boolean
+ Card(String newColor, String newNumber)
Button
«get/set» + location: Pair<int, int>
«get/set» + clickable: boolean
«get/set» + text: String
«get/set» + selected: boolean
«get/set» + hovering: boolean
+ Button(Pair<int, int> newLocation, String newText)
Player
«get/set» + hands: ArrayList<Card>
«get/set» + isTurn: boolean
+ Player(ArrayList<Card> hands, boolean newTurn)
+ updateHand(int index, Card card): void
+ startTurn(): void
+ endTurn(): void
<<Interface>>
ButtonListener
+ selected(): boolean
+ hovering(): boolean
GameStateModel
«get/set» + hintsRemain: int
«get/set» + fusesRemain: int
«get/set» + cardsRemain: int
«get/set» + timeRemain: int
«get/set» + score: int
«get/set» + cardsOnTable: ArrayList<Card>
«get/set» + discardPile: LinkedList<Card>
«get/set» + players: ArrayList<Player>
«get/set» + turn: int
«get/set» + subscribers: ArrayList<GameStateModelListener>
+ GameStateModel(Array<Player> newPlayers)
+ appendDiscard (Card card): void
+ updateCardsOnTable(Card card): void
+ updatePlayerHand(Player player, Card card): void
+ incrementHints(): void
+ decrementHints(): void
+ decrementFuses(): void
+ decrementCards(): void
+ decrementTime(): void
+ addSubscriber(GameStateModelListener Sub): void
+ notifySubscribers(): void
ClientStateModel
«get/set» + playerSelected: int
«get/set» + cardsSelected: ArrayList<int>
«get/set» + menusOpen: Map<String, boolean>
«get/set» + actionSelected: String
«get/set» + notes: ArrayList<int>
«get/set» + buttons: ArrayList<Button>
+ ClientStateModel(Player newPlayerSelected, Map<String, boolean> newMenusOpen, String newActionSelected, ArrayList<int> notes, ArrayList<Button> newButtons)
+ selectCard(int playerIndex, int cardIndex): void
+ toggleMenu(String name): void
+ selectAction(String action): void
+ getSelectedPlayer(): void
+ getSelectedCards(): void
+ getAction(): void
<<Interface>>
GameStateModelListener
+ modelChange(): void

4.2 AI


Controller
+ gameStateModel: GameStateModel
+ clientStateModel: ClientStateModel
+ socket: Socket
+ bufferedReader: BufferedReader
+ inputStreamReader: InputStreamReader
+ printStream: PrintStream
+ event: Map
+ eventStack: LinkedList<Map>
- currentState: String
+ Controller(String newCurrentState)
+ setGameStateModel(GameStateModel gameStateModel): void
+ setClientStateModel(ClientStateModel clientStateModel): void
+ receiveServerData(): void
+ parseJSON(JSONObject jsonObject): void
+ sendMove(JSONObject jsonObject): void
+ packStream(Map move): void
+ evalEvent(Map event): void
+ initialModel(Map event): void
+ initialGame(Map event): void
+ handleInputEvent(MouseEvent mouseEvent): void
AI
+ matrixes: Array<Array<Array<Array<double>>>>
+ gameStateModel: GameStateModel
+ AI()
+ findBestPlay(Array matrixes): void
+ findBestDiscard(Array matrixes): void
+ findBestHint(Array matrixes): void
+ findBestMove(Array matrixes): void
+ modelChanged(): void
+ devourStack(LinkedState<Map> eventStack): void
Card
«get/set» + color: String
«get/set» + number: String
«get/set» + clickable: boolean
«get/set» + selected: boolean
«get/set» + hovering: boolean
+ Card(String newColor, String newNumber)
Button
«get/set» + location: Pair<int, int>
«get/set» + clickable: boolean
«get/set» + text: String
«get/set» + selected: boolean
«get/set» + hovering: boolean
+ Button(Pair<int, int> newLocation, String newText)
Player
«get/set» + hands: ArrayList<Card>
«get/set» + isTurn: boolean
+ Player(ArrayList<Card> hands, boolean newTurn)
+ updateHand(int index, Card card): void
+ startTurn(): void
+ endTurn(): void
<<Interface>>
ButtonListener
+ selected(): boolean
+ hovering(): boolean
GameStateModel
«get/set» + hintsRemain: int
«get/set» + fusesRemain: int
«get/set» + cardsRemain: int
«get/set» + timeRemain: int
«get/set» + score: int
«get/set» + cardsOnTable: ArrayList<Card>
«get/set» + discardPile: LinkedList<Card>
«get/set» + players: ArrayList<Player>
«get/set» + turn: int
«get/set» + subscribers: ArrayList<GameStateModelListener>
+ GameStateModel(Array<Player> newPlayers)
+ appendDiscard (Card card): void
+ updateCardsOnTable(Card card): void
+ updatePlayerHand(Player player, Card card): void
+ incrementHints(): void
+ decrementHints(): void
+ decrementFuses(): void
+ decrementCards(): void
+ decrementTime(): void
+ addSubscriber(GameStateModelListener Sub): void
+ notifySubscribers(): void
ClientStateModel
«get/set» + playerSelected: int
«get/set» + cardsSelected: ArrayList<int>
«get/set» + menusOpen: Map<String, boolean>
«get/set» + actionSelected: String
«get/set» + notes: ArrayList<int>
«get/set» + buttons: ArrayList<Button>
+ ClientStateModel(Player newPlayerSelected, Map<String, boolean> newMenusOpen, String newActionSelected, ArrayList<int> notes, ArrayList<Button> newButtons)
+ selectCard(int playerIndex, int cardIndex): void
+ toggleMenu(String name): void
+ selectAction(String action): void
+ getSelectedPlayer(): void
+ getSelectedCards(): void
+ getAction(): void
<<Interface>>
GameStateModelListener
+ modelChange(): void

5 Classes and Methods

5.1 Model

GameState:

This class keeps track of the state of the game being played.

Attributes:

hintsRemaining(Integer): An integer field that keeps track of how many information tokens are available.

fusesRemaining(Integer): An integer field which keeps tracks of how many fuses remain.

cardsRemaining(Integer): An integer which keeps track of how many cards are remaining.

timeRemaining(Integer): An integer which keeps track of how much time is remaining on the current turn.

score(Integer): An integer which keeps track of what the current collective score is.

cardsOnTable(Array<Card>): A boolean array which marks off which cards have been played on the table at this moment. The position of the index corresponds to color and to number (e.g. CardsOnTable[0] corresponds to red, [1] to blue, etc.). An array has been chosen to represent the cards on the table as the order that each color pile is layed out on the table will stay consistent through all games and there can be no more than 5 piles on the table because of the rules of Hanabi.

discardPile(LinkedList<Card>): A linked list which keeps track of all of the cards that have been discarded up to the current point in the game. As it would be wasteful to have an array with the size of the maximum possible discarded cards in any Hanabi game. Putting a card on top of the discard pile also becomes intuitive and consistent with inserting it at the beginning of the linked list. This is also necessary as the latest discarded card is always on display.

players(Array<Player>): An array which keeps track of each player in the game. When a game is being initialized the number of players is specified and does not change throughout the game (player(s) leaving causes the termination of the game) which is why we use an array for this field.

turn(Integer): An integer which keeps track of whose turn it is. This integer is then used to index into the players array to select the one who's turn it is.

Methods:

GameStateModel(Array<Player>players): The constructor method for GameState that initiates a new GameState object given an array of Player objects

Preconditions: A game was created and a JSON packet with the information on hands received

Postconditions: Players array field of the GameState is set to the provided array of players.

Inputs: The Array of initialized players

Outputs: The reference to a new GameStateModel

appendDiscard(Card card): Appends a discarded card to the linked list of discards

Preconditions: A discard event took place.

Postconditions: The card which was discarded is appended to the list of discards.

Inputs: The Card which was discarded

Outputs: None

updateCardsOnTable(Card card): Updates the cards which lay face up on the table after a valid card was played.

Preconditions: A valid card was played by any player

Postconditions: The CardsOnTable array is updated to reflect the new state of the table.

Inputs: The Card which was played

Outputs: None

updatePlayerHand(Player player, Card card, Int pos): Updates the hand of a player in the array of players.

Preconditions: A valid card was played by the player

Postconditions: The new card is now part of the player's hand

Inputs:
player: The player that made the valid move
card: The new card that was drawn
pos: The card position in the player's hand that should be replaced by the new card

Outputs: None

incrementHints(): Increments the hints remaining by one.

Preconditions: A valid discard was played, or a stack of cards was completed.

Postconditions: The HintsRemaining field is increased by one.

Inputs: None

Outputs: None

decrementHints(): Decreases the hints remaining by one.

Preconditions: A player gave another player a hint, and at least one hint token was available.

Postconditions: The HintsRemaining field is decreased by one.

Inputs: None

Outputs: None

decrementFuses(): Decreases the fuses remaining by one.

Preconditions: A player played an invalid card.

Postconditions: The FusesRemaining field is decreased by one.

Inputs: None

Outputs: None

decrementCards(): Decreases the number of cards remaining in the draw pile by one.

Preconditions: A player has been dealt a new card.

Postconditions: CardsRemaining field is decreased by one.

Inputs: None

Outputs: None

decrementTime(): Decreases the time remaining for the current turn by one second.

Preconditions: One real time second has passed.

Postconditions: The amount of time remaining is reduced by one.

Inputs: None

Outputs: None

Client State Model:

This class represents the current state of the client. It keeps track of all of the client side operations such as: cards selected, actions selected, menus open etc. etc. In other words, this class makes up the part of the model which is separate from the actual game being played. The reason this class was created was to decouple the client logic, from the game logic within the model.

Attributes:

playerSelected(Integer): An index representing the where the player who is selected sits at the table.

cardsSelected(Array<Integer>): A list of indices representing the set of cards which are currently selected by the player.

menusOpen(Dictionary(String:Boolean): A dictionary of strings, which map to booleans. Each string represents a menu that is currently open by the player.

actionSelected(String): A string which represents the current action selected by the player.

notes(Array<Hints>): An array which keeps track of the various hints the player has marked on their cards.

buttons(Array<Button>): An array of all of the buttons which exist.

Methods:

ClientStateModel(Array<Player>): Constructor method for the class.

Preconditions: None

Postconditions: A new class is made that represents the client state.

Inputs:

Array<Player>: An array of the players, with their hands.

Outputs: None

selectCard(Integer PlayerIndex, Integer CardIndex): A method which takes in a player index, and a card index, and sets the card at that array index to Selected.

Preconditions: None

Postconditions: The card at CardIndex held by the player at PlayerIndex is set to selected.

Inputs:

Integer PlayerIndex: An integer which indexes into the player array

Integer CardIndex: An integer which indexes into a hand array

Outputs: None

toggleMenu(String MenuName): Takes in a string and indexes into the MenusOpen dictionary, toggling the boolean value inside.

Preconditions: None

Postconditions: The boolean at the MenusOpen index of the dictionary is toggled.

Inputs:

String MenuName: A string that is the key for the dictionary of open menus.

Outputs: None

selectAction(String ActionName): Takes in a string and sets the current action selected value to that string.

Preconditions: None

Postconditions: The action selected attribute is set to a string reflecting the current action

Inputs:

String ActionName:

Outputs: None

getSelectedPlayer(): Returns the index of the currently selected player.

Preconditions: A player is currently selected.

Postconditions: None

Inputs: None

Outputs: **Integer PlayerSelected:** The index of the currently selected player.

getCardsSelected(): Returns a list of integer indices of the cards currently selected.

Preconditions: A player is currently selected, and a card(s) is selected from their hand

Postconditions: None

Inputs: None

Outputs: **List<Integer>:** A list of indices of currently selected cards.

getAction(): Returns the currently selected action by the player as a string.

Preconditions: At least one card is selected.

Postconditions: None

Inputs: None

Outputs:

String ActionSelected: A string encoding the current action selected.

Player:

This class keeps track of the information about a player's hand and their turn.

Attributes:

hand(Array<Card>): This Array stores all cards that a player has (only used to keep track of other players' cards as the player's own cards are not visible to themselves). The number of cards that each players has is defined at the creation time of the game and does not change during the game. This is why an array was chosen to represent a player's hand.

isTurn(Bool): A boolean which indicates if it is this player's turn to play. Note that at a given time only one player's isTurn variable is allowed to be set to True, meaning it can't be multiple people's turn at the same time.

Methods:

Player(Hand, Turn): The constructor method for Player that initiates a new player object given the cards that they were dealt and whether they're the first one to play.

Preconditions: A game was created and a JSON packet with the information on hands received

Postconditions: A player object is created

Inputs:
Hand: Array of Cards that make up this player's hand
Turn: A Bool that indicates if it's this players turn or not

Outputs: None

updateHand(Player,Card, Pos): Updates the hand of the player with a new card

Preconditions: A valid card was played by the player

Postconditions: The new card is now part of the player's hand

Inputs:
Card: The the new card that was drawn
Pos: The card position in the player's hand that should be replaced by the new card

Outputs: None

startTurn(): Sets the isTurn field for the current player to true, indicating that their turn has started

Preconditions: It is the player's turn to play

Postconditions: isTurn field is set to true

Inputs: None

Outputs: None

endTurn(): Sets the isTurn field for the current player to false, indicating that their turn has ended

Preconditions: The player has made a valid move

Postconditions: isTurn field is set to false

Inputs: None

Outputs: None

Card:

This class represents a card in game. It was picked to be a basic representation of the cards in the game to be held by the player class in an array. The decision to pick individual characters as the representation for colors and rank was made to make the class creation easy after JSON parsing. The decision to pick a set of booleans to represent how the card will be drawn or interacted with was just for readability/simplicity.

Attributes:

color(String): A string which represents the color of the card. The color mapping to character is as follows:

$$r \rightarrow Red$$

$$b \rightarrow Blue$$

$$g \rightarrow Green$$

$$w \rightarrow White$$

$$y \rightarrow Yellow$$

number(String): A string which represents the rank of the card.

clickable(Boolean): A boolean which determines if a card can be clicked on or not (whether or not you can interact with it). This attribute exists for the purpose of the view (so it can draw the cards greyed out), and so that the listener of the card knows not to accept any clicks.

selected(Boolean): A boolean that reflects whether or not a card is selected. This attribute exists for the purpose of the view to alter the way a card which is selected (for hint giving) is drawn.

hovering(Boolean): A boolean the reflects whether or not a mouse is hovering over a card. This attribute allows the view to convey information about which card will be selected if a click happens.

Methods:

Card(String Color, String Number): The constructor method for the card class, takes in two strings, one that represents the rank of the card, and one that represents the color of the card.

Preconditions: None
Postconditions: A card object is created
Inputs:
String Color: A single character which represents the color of the card.
String Number: A single character which represents the number of the card.
Outputs: A reference to a card object

setClickable(Boolean Bool): A method that sets the clickable attribute of a card.

Preconditions: None
Postconditions: The card clickable field is changed
Inputs:
Boolean Bool: A boolean, True if the card is clickable, false otherwise.
Outputs: None

setSelected(Boolean Bool): A method that sets the selected attribute of a card.

Preconditions: None
Postconditions: The card selected field is changed
Inputs:
Boolean Bool: A boolean, True if the card is selected, false otherwise.
Outputs: None

setHovering(): A method that sets the hovering attribute of a card.

Preconditions: None
Postconditions: The card hovering field is changed
Inputs:
Boolean Bool: A boolean, True if the card is being hovered over, false otherwise.
Outputs: None

getClickable(): Returns a whether or not a card is clickable.

Preconditions: None
Postconditions: None
Inputs: None
Outputs: A boolean

getSelected(): Returns whether or not a card is selected.

Preconditions: None
Postconditions: None
Inputs: None
Outputs: A boolean

getHovering(): A method that sets the hovering attribute of a card.

Preconditions: None
Postconditions: None

Inputs: None

Outputs: A boolean

getColor(): A method that returns the color of a card.

Preconditions: None

Postconditions: None

Inputs: None

Outputs: A string which represents the color of a card.

getNumber(): A method that returns the color of a card.

Preconditions: None

Postconditions: None

Inputs: None

Outputs: A string which represents the rank of a card.

5.2 View

View:

This class is responsible for drawing the information obtained from the Model.

Attributes:

height(Double): This double keeps the information about the height of the screen

width(Double): This double keeps the information about the width of the screen

gameStateModel(GameStatemodel): This field keeps track of the state of the game and its fields

clientStateModel(ClientStateModel): This field keeps track of the state of the client and its fields

controller(Controller): The controller from MVC that is responsible for communicating with the server and getting/setting information from/to the Model and the user

canvas(Canvas): Canvas is an image that can be drawn on using a set of graphics commands provided by a GraphicsContext

graphicsContext(GraphicsContext): This field is used to issue draw calls to the Canvas.

Methods:

View(Double height, Double Width): This is the constructor method for the View class which creates a new instance of it given height and width specifications

Preconditions:

Postconditions: None

Inputs: None

Height: a double that specifies the height for the canvas

Width: a double that specifies the width for the canvas

Outputs: a reference to a new View object

setGameStateModel(GameStateModel gameStateModel): Sets the gameStateModel field of the current View object to the given GameStateModel

Preconditions: None

Postconditions: The field gameStateModel is set to the provided model.

Inputs: The GameState to set View's field to

Outputs: None

setClientStateModel(ClientStateModel clientStateModel): Sets the clientStateModel field of the current View object to the given ClientStateModel

Preconditions: None

Postconditions: None The field clientStateModel is set to the provided model.

Inputs: The ClientState to set View's field to

Outputs: None

setController(Controller controller): Sets the controller field of the current View object to the given controller

Preconditions: None

Postconditions: The field controller is set to the provided Controller object.

Inputs: The Controller to set View's field to

Outputs: None

setLayoutChildren(): Gets height and width of the screen and sets those to the canvas

Preconditions: None

Postconditions: None

Inputs: None

Outputs: None

modelChanged(String event): This method is used for notifying the View about changes in the Model and redrawing only the changed parts, which are identified by the Event argument.

Preconditions: a change in the model happened

Postconditions: The components that have changed in the model are redrawn

Inputs: Event is a String that indicates which field in the model has been changed.

Outputs: None

drawLogIn(): Draws the LogIn Screen

Preconditions: The game is launched

Postconditions: The login page is displayed

Inputs: None

Outputs: None

drawStartMenu(): Draws the initial selection screen with Join and Create game options

Preconditions: The login screen redirected the to this screen

Postconditions: The Start Menu interface is displayed

Inputs: None

Outputs: None

drawCreateGame(): Draws the Interface for creating a game

Preconditions: The create game option is selected

Postconditions: The create game interface is displayed

Inputs: None

Outputs: None

drawJoinGame(): Draws the Interface for joining a game

Preconditions: The join game option is selected

Postconditions: The join game interface is displayed

Inputs: None

Outputs: None

drawWaiting(): Draws the waiting screen while all players have not yet joined the game

Preconditions: Either joined or created a game

Postconditions: The waiting interface is drawn

Inputs: None

Outputs: None

drawgameTable(): Draws the card piles that are face up on the table along with all players' cards

Preconditions: The game has started

Postconditions: The players' cards and the cards on the table are drawn

Inputs: None

Outputs: None

drawGameState(): Draws information about the game such as Hint and Fuses remaining, the top card on the discard pile, time remaining to make a move, etc.

Preconditions: The game has started

Postconditions: All the information about the game contained in the client state is drawn

Inputs: None

Outputs: None

Button:

This class represents a button in game. The decision made to have a button class was to make later coding process easier to track every object interacting wiht the user's mouse, such as clickable, selectable and hoverable.

Attributes:

location(Pair<Integer, Integer>): A pair which contains two integers to indicate a button's location.

clickable(Boolean): A boolean which indicates if this button can be clicked or not.

text(String): A string which shows the button's name or message.

selected(Boolean): A boolean which indicates if the button object is currently selected.

hovering(Boolean): A boolean which indicated if the user moves mouse pointer upon the button.

Methods:

Button(Pair<Integer, Integer>newLocation, String newText): A constructor which creates a button object, and it takes two arguments, one is location, and one is text message on button.

Preconditions: None

Postconditions: A button is create.

Inputs:

Pair<Integer, Integer>newLocation: A pair data structure which contains two integer elements indicating two dimensional coordinates on screen layout.

String newText: A string which shows a message on the button.

Outputs: A reference to this button.

setLocation(Pair<Integer, Integer>newLocation): A method that sets the location attribute of a button

Preconditions: None

Postconditions: A new value of coordinate is new assigned to the location attribute

Inputs:

Pair<Integer, Integer>newLocation: A pair data structure which contains two integer elements indicating two dimensional coordinates on screen layout.

Outputs: None

getLocation(): Retruns a button's location

Preconditions: None

Postconditions: None

Inputs: None

Outputs: A pair of integers indicating two dimensional coordinate

setClickable(Boolean newBool): A method that sets the clickable attribute of a button.

Preconditions: None

Postconditions: A new boolean valeu is now assigned to clickable attribute of a button.

Inputs:

Boolean Bool: A boolean, True if the button is clickable by a user's mouse, false otherwise

Outputs: None

getClickable(): Returns a boolean indicating if the button is clickable or not.

Preconditions: None

Postconditions: None

Inputs: None

Outputs: A boolean

setText(String newText): A method that sets a new text to a button.

Preconditions: None

Postconditions: A new text is assigned to text attribute of a button

Inputs:

String newText: A string showing the message of the button to users

Outputs: None

getText(): Returns the text message of a button.

Preconditions: None

Postconditions: None

Inputs: None

Outputs: A string that shows the message of a button

setSelected(Boolean newBool): A method that sets the seleted attribute of a button.

Preconditions: None

Postconditions: A new boolean value is assigned to the selected attribute.

Inputs:

Boolean newBool: A boolean, True if the button is being selected, false otherwise

Outputs: None

getSelected(): Returns a boolean indicating if the button is being selected or not.

Preconditions: None

Postconditions: None
Inputs: None
Outputs: A boolean

setHovering(Boolean newBool): A method that sets the hovering attribute of a button.

Preconditions: None
Postconditions: The hovering value is changed
Inputs:
Boolean newBool: A boolean, True if the button is being hovering, false otherwise
Outputs: None

getHovering(): Returns a boolean indicating if the button is being hovering or not.

Preconditions: None
Postconditions: None
Inputs: None
Outputs: A boolean

5.3 Controller

Controller:

This class is to establish connection to the server in order to deal with external data, and to handle all the game play evets and states.

Attributes:

gameStateModel(GameStateModel): A game state model, one of the models to keep game attributes.

clientStateModel(ClientStateModel): A client state model, one of the models to keep game attributes.

socket(Socket): A socket object which has a port number connecting to the server, and it needs an integer to construct itself.

bufferedReader(BufferedReader): An data reader reads message from the server.

inputPrintStream(InputPrintStream): An input data receiver which receives server data through the socket connection, and this object needs InputReader and Socket objects as constructors.

printStream(PrintStream): An object which outputs data to the server through the socket connection, and this object needs a Socket object as a constructor.

event(Map<String:String>): A map data structure which contains server messages parsed by ParseJSON method.

eventStack(LinkedList<Map>): A linked list data structure in which each element contains an event received from server.

currentState(String): A string which keeps track of the current game state.

Methods:

Controller(String newCurrentState): Constructs controller object one string value representing current state.

Preconditions: None

Postconditions: A controller object is created.

Inputs: A current state that controller starts tracking to

Outputs: A reference to the created controller object

setGameStateModel(GameStateModel gameStateModel): Assigns a game state model object to gameStateModel field.

Preconditions: None

Postconditions: The controller object now has gameStateModel.

Inputs: A game state model object

Outputs: None

setClientStateModel(ClientStateModel clientStateModel): Assigns a client state model object to clientStateModel field.

Preconditions: None

Postconditions: The controller object now has clientStateModel.

Inputs: A client state model object

Outputs: None

receiveServerData(): Receives messages from the server.

Preconditions: Socket connection to the server succeeds, and a server message is expected.

Postconditions: A JSON object containing server message is now ready to be parsed.

Inputs: None

Outputs: A JSON object which contains server messages

initGame(Map<String:String>event): Brings the user into the game program.

Preconditions: An login message has been received from the server.

Postconditions: The user has authority to run program functions.

Inputs: An event which starts the program

Outputs: None

parseJSON(JSONObject jsonObject): Converts JSON object (server messages) into a map data structure.

Preconditions: ReceiveServerData method successfully output a JSON object.

Postconditions: An event in map is acquired and ready to be assigned to the event field.

Inputs: A JSON object which contains an event data from the server

Outputs: An event which can be read for other operations in the program

sendMove(JSONObject jsonObject): Sends move messages to the server.

Preconditions: A player's move is made and parsed by PackStream method.

Postconditions: The server receives a move message from the player.

Inputs: JSON object containing the player's move messages

Outputs: A JSONpacket sent to the server

packStream(Map<String:String>move): Convert a move message from the player's action into a JSON object which can be read by the server.

Preconditions: A player made a move.

Postconditions: A JSON object is converted.

Inputs: A map which contains the player's move message

Outputs: A JSON object ready to be sent to the server

evalEvent(Map<String:String>event): Analizes messages, executes actions in the game.

Preconditions: An event takes place to make changes.

Postconditions: An event has been executed to make system changes.

Inputs: An event which makes changes in system

Outputs: None

initModel(Map<String:String>event): Initializes or establishes a game table with some game settings specified by the game master.

Preconditions: The game master gets all initial setting options of the game set.

Postconditions: Game data in Model object is all set to initial values, and the players are able to see game table with all cards has been arranged. The game state is at the first player's term.

Inputs: An event which contains all messages needed to initialize and start a game

Outputs: None

handleInputEvent(MouseEvent mouseEvent): Listens to mouse events and does functions based on different requests.

Preconditions: A mouse event is received.

Postconditions: Program does functions to change the user's views and data held in the models.

Inputs: A mouse event

Outputs: None

5.4 AI

AI:

The AI class which is instantiated when the game is run from the command line with an argument. The AI uses the same controller and model as the client, and acts indistinguishably from the view by sending the same update messages to the controller, and reading from the model in the exact same way. In general, what it does is use a layered architecture to search for the best move in parallel. It simultaneously calculates the best play, discard, and hint to give.

The layers of the desicion process are as follows:

- Logic The AI first searches for a logical property in the probability matrices to determine a logical move. For example, if it knows with 100% certainty that it has a red 1, and a red 1 is playable, it will play that.
- Statistics If the AI cannot find a move which relies only on logic, then it evaluates its statistical model to attempt to look for a move which is probabilistically very strong. On this layer the AI might play a red 1 if it's 90% sure that playing it is correct.
- Search Finally, if the AI cannot find a move which it considers strong with a statistical analysis, then it attempts to bolster its statistical model with a random sampling method like Monte Carlo tree search or Markov Chain Monte Carlo. The actual implementation of this layer could be tried in many different ways, and we haven't committed ourselves to one specific strategy. As a result, the Strategy design pattern will probably be used for this layer so we can examine the various results of different search methods.
  This layer will be considered a 'nice-to-have', as it is both far more complicated than the previous layers, but also a lot more theoretical.

Attributes:

conditionalProbMatrices(Array(Integer)): An array of integers (one array for each card held) which keeps track of the current conditional probability of a cards expected suit, and rank as a 2D array.

observedCPMatrices(Array(Integer)): An array of integers (one for each card, for each player) that keeps track of the current expected conditional probability tables of the other players at the table. The purpose of this is to aid the AI in giving hints by approximating what other players know about their own hand.

Methods:

AI(): Constructor class for an AI. This should follow the singleton design pattern.

Preconditions: None

Postconditions: An AI class is instantiated

Inputs: None

Outputs: None

findBestDiscard(Integer Time): A method that looks for the best discard, it first looks for a logical property to exploit, then a statistical one, and finally it performs a search.

Preconditions: It is the AI's turn

Postconditions: None

Inputs: Time, the amount of time in seconds that the AI must return a result before. This is for limiting the search method.

Outputs: The best discard.

findBestPlay(Integer Time): A method that looks for the best play, it first looks for a logical property to exploit, then a statistical one, and finally it performs a search.

Preconditions: It is the AI's turn

Postconditions: None

Inputs: Time, the amount of time in seconds that the AI must return a result before. This is for limiting the search method.

Outputs: The best card to play.

findBestHint(Integer Time): A method that looks for the best hint to give, it first looks for a logical property to exploit, then a statistical one, and finally it performs a search.

Preconditions: It is the AI's turn

Postconditions: None

Inputs: Time, the amount of time in seconds that the AI must return a result before. This is for limiting the search method.

Outputs: The best hint to give.

updateCPT(Event): A method that updates the conditional probability tables of the AI after any change in the game state occurs which will allow stronger statistical inference (a new card is exposed, or a hint is given)

Preconditions: Some new information is revealed to the AI

Postconditions: The AI has its conditional probability tables updated

Inputs: An event which encodes the change of state of the game.

Outputs: None

devourStack(LinkedList): A method that when given the stack of all recent moves played, moves through each move one by one to create a statistical model for the current point in the game. This method is called whenever the AI player is sent into the game to replace a player during an AFK period.

Preconditions: The player has set themselves to AFK.

Postconditions: The AI has a current statistical model of the game.

Inputs: A linked list of all of the moves made up to that point.

Outputs: None