Hanabi Client: Design Document

CMPT 370 - Group F1

Minh Hoang - mph952
Sharjeel Humayun - shh738
Donovan Lavoie - dbl599
Georgi Nikolov - gnn308
Faiz Qureshi - faq588
Jason Wong - jaw845

Contents

1. Introduction

1.1 Purpose

This document describes the design of a client program that lets a user play Hanabi. It describes the design at two different levels: a high-level architectural design that describes the major components and their organization, and a low-level detailed design that describes the classes and objects that compose the system and their interfaces.

1.2 Scope

The design described in this document is that of the Hanabi Client, which lets 1 person play Hanabi with other people over a network connection. This design covers how the Client is organized so that the state of the game is maintained and displayed to the user as inputs from human and AI users ask to create and join a game, add AI Players, or make a move. It also specifies how communication between Players is handled using a Server as an intermediary and which parts of the Client deal with sending and receiving Server messages. Finally, the design of AI Players and how they make moves is discussed.

1.3 Glossary

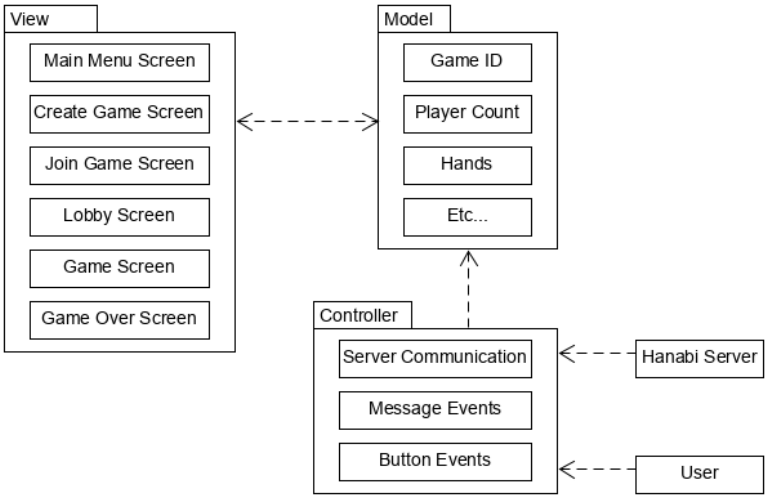
Term	Description
AI Player	A computer Player that makes moves without user input.
Game ID	A numeric identifier for games on the Server, used by Players to join a game.
Hanabi Client/Client	The program described in this document, which lets a user play Hanabi with other people over a network connection
Hanabi Server/Server	The server that handles communication between Players as they join and play games of Hanabi
Human Player/Player	A user running the Client program and playing a game of Hanabi
JSON	Javascript Object Notation
Token	A string value tied to games on the Server, used by Players to join a game.
State Diagram	Abstract descriptions of the behaviour of a system, representing it as states and transitions between states.

1.4 Overview of Document

The next section, the Architectural Design, describes the architecture used by the Client and the resulting organization of its major components. Some of the components have architectures themselves, which are also described there. This is followed by the Detailed Design section, which provides class diagrams and a listing of interfaces for the classes that the Client is composed of so that it takes user input, maintains the model, displays the game and menus, and sends and receives Server messages.

2. Architectural Design

2.1 Client Architecture



The overall architecture of the Hanabi Client is structured as a Model-View-Controller (MVC) application, which is depicted in the above diagram. The model stores the state of a game of Hanabi and the overall Client, including the game ID and Token, the number of Players, and the Player's hands, among other aspects of the gameplay and Client state. The view sees and exposes this game model to the Players as they play, with updates to the game state being pushed to the view as they occur. There are multiple possible views, with the view during gameplay only being the one that shows most of the model; various menu views present buttons and fields that let a Player create or join a game, leave a game, and add AI Players. Communication between the user, via button selections, and the system is handled by a controller, which turns user inputs into changes in the model and displayed view.

The controller's handling of inputs and their corresponding model and view changes is done using an event-driven state machine. Selection of displayed buttons by the user generates events that the controller listens for, and these events are used to trigger state transitions that alter the model; the main view may change as well as part of these events. Server communication is dealt with within the controller by treating received messages as events, which the controller responds to with model changes, and treating sending messages as actions to take in response to other events alongside model changes.

The next several sections will describe the structure of the architectural components: the model, view, and controller. The data stored in the model will be described, and then the state machines that describe the views and the controller operations will be covered.

2.2 Client Model

The model of the Hanabi Client keeps track of the Client's core state separate from the other components. This includes the state of an active game of Hanabi, like the hands and token counts, alongside variables that track the general state of the Client.

Upon the creation of the game, a Game Creator will define both the game's maximum number of Players, a value from 2 to 5, and a timeout period from 1 to 120 seconds for turns. Each game has its own unique ID and secret Token given upon creation that other players use to join. The current number of Players in a game can be anywhere from 1 to the maximum number of Players allowed (up to 5) and is updated as Players join and leave the game, including the addition of AI Players, and the game starts when they are equal.

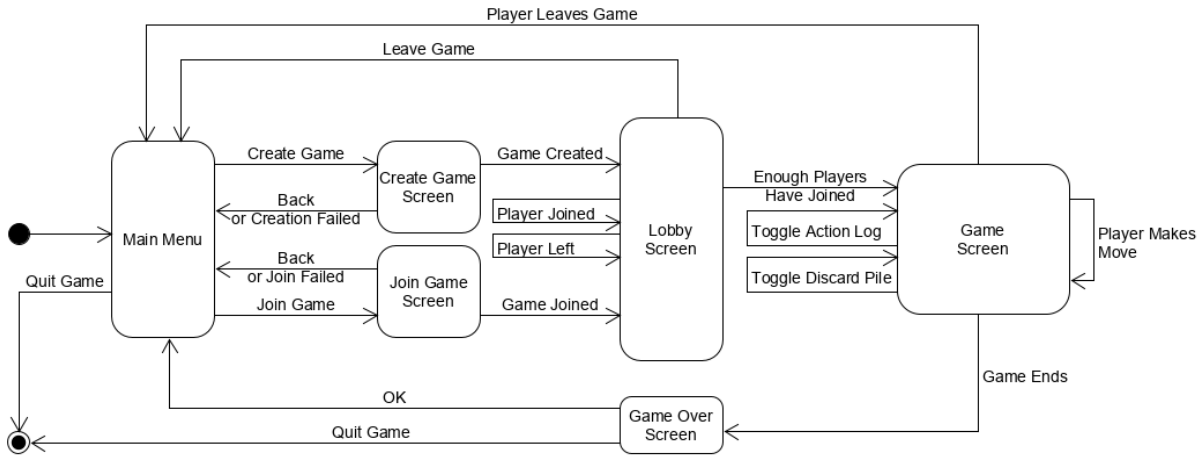
Once a game starts, the model keeps track of all the cards in the game. Each card has two properties: a colour from a list of Red, Blue, White, Yellow, Green, Rainbow, and Null, and a numeric rank from 0 to 5. The Null colour and 0 rank are used for when a Player doesn't know the colour or rank of one of their own cards. Cards are stored in the Player's hands, firework piles and discard pile, which are all containers of cards. The number of hands is equal to the number of players in the game, and each hand has room for 4 cards (with 4-5 Players) or 5 cards (with 2-3 Players) at any time; when cards are played, another card is drawn to replace it except on the last turn. A single firework pile exists for each card colour (i.e. there are 6), and each one holds up to 5 cards of their colour in numeric order ({1}, {1,2}, ...). The discard pile then holds any of the 0+ cards that have been discarded throughout the game.

The model also tracks Hanabi tokens, which are represented by a count and are split into two categories: fuse and information tokens. The fuse token count starts at 3 and decreases by one when Players make mistakes in playing a card (ex. playing a Red 3 when the Red firework pile has up to a Red 1). The number of information tokens, meanwhile, goes up and down by one as Players discard cards (to gain one) or give information about each other's hands (to lose one).

Finally, the model also contains an action log, a listing of turns in chronological order that the view can display as a list of statements about moves, which can then be used as information by the Players.

2.3 Client Views

The Hanabi game model will be displayed through 6 main views, which are shown in the architecture diagram in section 2.1. These main views and the actions that cause them to change are a state machine. The View State Diagram below shows the different views as states and the system actions and events that change between them as transitions.



The starting view is the Main Menu screen, where the Player is not in a game and the Game ID and Token are not set. It gives a Player the option to create or join a game as well as close the client. Closing the client goes to the end of the state machine, while selecting the Create Game and Join Game options causes a transition to the Create Game or Join Game screens.

The Create Game or Join Game views then appear to get the required options for creating or joining a game. Game creation requires the number of Players, the timeout period, whether to force create a game for Game Creators who have left another game, and the Player's NSID. Joining a game instead requires a Game ID and Token with the Player's NSID. Once these are given by the Player, selecting to create or join a game with those parameters will bring them to the Lobby view if the game is successfully created or joined; if they fail, then the Player is notified with a message as the view changes back to the Main Menu. The Player can also choose to go back to the Main Menu by selecting the Back option.

When a Player is at the Lobby view, they are in a game with a Game ID, Token, and the number of Player slots as well as a count of how many Players are currently in the game. The Lobby view shows these properties as well as options to add AI Players and to leave the game. The Player slots are updated whenever another Player joins or leaves the game. The Player can also choose to leave the game themselves, which will take them back to the Main Menu. Once enough Players have joined the game, the game starts and the view transitions to the Game view.

When a Player is in an active game, the Game view shows all the information needed for a Player to play Hanabi: a set of cards for all the Players, the fireworks piles, the remaining information and fuse tokens, a discard pile, and a log of the game's actions so far. A card in a Player's hand will separately show their color and their number based on the information that a Player knows about them; a card back is shown when neither property is known about a card in the Player's own hand. Whenever any Player makes a move, the Game view updates to show the new game state; the action log is always updated with a message about the move. Selecting the discard pile or the Log option will toggle the Game view's

displays of the discarded cards and the action log on and off at any time during a game. The Player can also choose to leave the game before it ends, which will take them back to the Main Menu view.

Once a move ends the game, whether because the fireworks piles are finished, the fuse tokens have run out, or every Player has had one move without the chance to draw a card, the Player is taken to the Game Over view, which presents the final score. The Player can select from an OK option that takes them back to the Main Menu again, or they can choose to close the client, going to the end of the view state machine. Either transition from the Game Over view also sets the game model to a state of not being in a game since the Player is disconnected from a game when it ends.

2.4 Client Controller

Besides the views, the entire Hanabi Client is itself a state machine, and transitions between these states are handled by a controller that listens for events from the view and messages from the Hanabi Server. The state transitions involve the acting on the model to change its current state and the displayed view and sending and receiving messages from the Server.

The View State Diagram in section 2.3 already describes the states of the whole Client from starting up the Client to entering a game, though it does not fully cover the operations that the controller does in those transitions. The next several of paragraphs will describe the controller's actions on the model and communications with the Server during the transitions between menus besides changing the view.

When creating or joining a game, the controller needs to send an appropriate message to the Server to create a game (with or without force) or join a game. It will then receive a message from the Server with the identifying information about the game: the Game ID, Token, and timeout period. This information is given to the model so that it knows the Player is in a game and the model is told to store the game identification. If the received server message instead indicates that it failed to create or join a game, then the model is simply not notified.

While waiting for more Players to join a game, the controller listens for Server messages about other Players joining and leaving. As these messages come in the controller tells the model of the change in Player count. If the Player leaves the game, however, then the model is told to clear the game info since the Player is no longer in that game.

When enough Players have joined the game, the controller will get a server message about the hands of every Player; the Player's own hand will be empty, however, since they don't see their own hand. The model is given these hands by the controller as it is told to start the game. Starting a game involves initializing the empty log and discard pile, the starting 8 information tokens and 3 fuse tokens, the empty fireworks piles, and a tracker of whose turn it is.

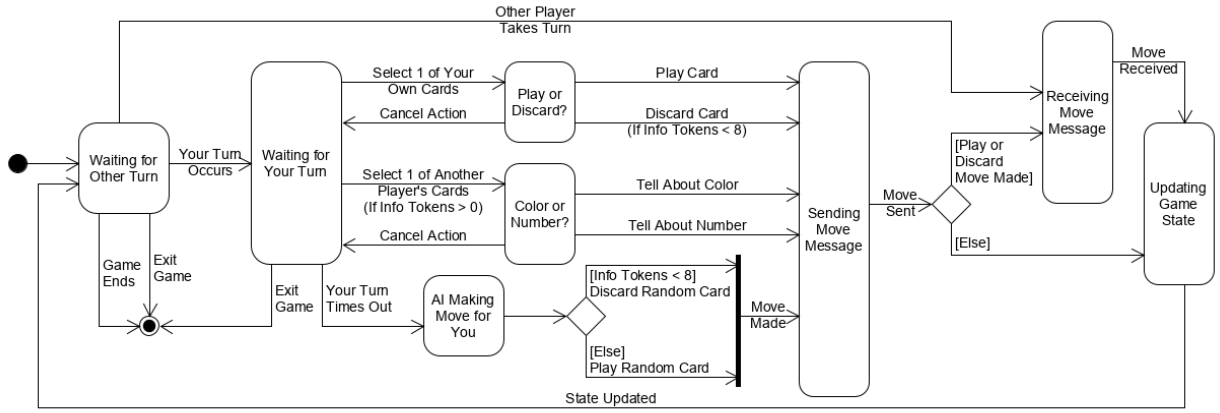
While a game is being played, the controller is occupied with sending and receiving Server messages about when it is the Player's turn, the moves that Players make, timeout prompts, and when the game ends. As moves are made, the details of those moves are given to the model, which applies the move to

the current game state and updates the Player's hands, the token counts, the log, and the discard and fireworks piles as necessary.

Once the Server sends a message saying that the game has ended, the controller tells the model to change the view from the Game view to the Game Over view; the game state is not cleared yet to facilitate score display. Once a transition occurs from the Game Over view, then the controller tells the model to clear the game state, since ending a game involves disconnecting from it.

2.5 State Machine for Human Player Turns

The controller and Game view have a more detailed state machine that is used during a game as Human Players take turns. This was abstracted into one simple transition in the View State Diagram in section 2.3 but is shown in more detail in the state diagram below.



Once a game starts, the controller and Game view enter a state of waiting for the Server to say that it is the Player's turn. If it is another Player's turn, then a Server message with their move will be received by the controller and is used to make the model update the game. Once it is the Player's turn, a Server message will indicate this to the controller and it will wait for the Player to start making a move. The game can also end during this state due to a Player leaving the game or the game ending naturally, which brings the Game state machine to its end.

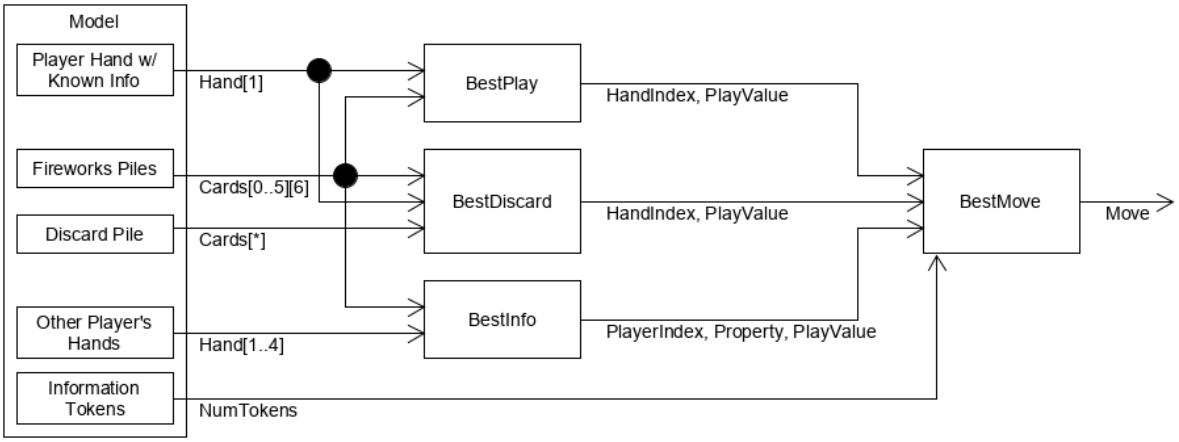
Once the controller and Game view are waiting for the Player's turn, the Player can start to make a move by selecting a card in any of the Player's hands. If they select one of their own cards, the Game view will prompt them to clarify whether they want to play or discard that card; discarding can only be selected if there are less than 8 information tokens in the game. If they instead select one of another Player's cards and there is at least 1 information token, then they want to give information to them. The Game view prompts for whether to tell them about the number or color of that card and other cards in their hand with the same property. During either prompt the Player can also elect to cancel the action, sending the controller and Game view back to waiting for a move from the Player.

Once the Player's move has been made and fully specified, the controller takes the move information and sends a message to the Server so that the other Players will know about it. After the move has been sent, the controller may also have to wait to receive another message from the server; this message will specify the drawn card if the Player plays or discards a card. Either way, the controller then gives the Player move to the model and tells it to apply that move before going back to waiting for the Player's next turn and receiving and applying other Player's moves.

If a Player takes longer than the timeout period to make a move and a Server message is received to re-prompt them for a move, then it is assumed that the Player is away from the game but still wants to play, so an AI Player will make a move for them. This AI-made move is simplistic, however; it decides to either discard a random card in the Player's hand if there are less than 8 information tokens, or plays a random card if discarding is not possible. This move is given to the controller, which sends it to other Players and makes the model apply it like any other move.

2.6 AI Player Architecture

The process of making a move is different for AI Players, whether they are making a move for a non-present human Player as described above, or they are a standalone Player created at the Lobby screen or through command line execution of the Client. They don't have any displayed view with menus to navigate or buttons to select, and simply need to pick a move and give it to the controller. The view they have displays nothing but can still see the game model, which the AI uses to make decisions.



The above figure shows the pipeline architecture used by the AI Player to decide on a move and give it to the controller. It uses the state of the game (the Player's own hand, the other Player's hands, the discard pile, the fireworks piles, and the token counts) as inputs to 3 separate filters to determine what the best play, discard, and information moves would be. The separate kinds of best move are then given to another filter that decides what to do between the 3 best moves and gives the chosen move to the controller.

As an input, the AI Player's own hand is given to the filters as a Hand structure which contains a list of 4 or 5 cards with individual colour and rank; unknown properties about cards are simply missing from the respective cards. The hands of the other Players are given as a list of their Hand, each with a similar structure, although their cards are guaranteed to have a known colour and rank. The fireworks piles are passed as a list of 6 lists of cards, one for each colour, that contain the cards that have been played throughout the game; played cards in one list are guaranteed to be in numerical order since successful plays must be in numerical order. The discard pile is passed as a list of cards that have been discarded so far but has no guaranteed order. Finally, the information token count is passed simply as a number, NumTokens.

The best play move is determined by the BestPlay filter, which takes in the AI Player's own hand and the fireworks piles. It then returns a description of the best play in terms of the hand index of the card to play and a PlayValue, a number that is smaller for stronger moves. It uses the following cases in deciding on the best play and assigning PlayValues, while preferring higher ranks and choosing the lowest index card in the hand whenever there are multiple cards that fit a case:

- If any card in the AI Player's hand is fully known and is the next card in a fireworks pile, then it is the best play. The returned PlayValue is 0.
- If a card, from a known rank, could be the next card in a fireworks pile, then it is the next best play. The returned PlayValue is 1.
- If a card, from a known colour, could be the next card in a fireworks pile, then it is the next best play. The returned PlayValue is 2.
- If nothing is known about any of the AI Player's cards, then a random card is selected as the best play. The returned PlayValue is 3.

The BestDiscard filter also takes in the AI Player's own hand and the fireworks piles, but also uses the discard pile to determine the best discard move. The returned move description is like the BestPlay filter, returning a hand index for the card to remove and a PlayValue that is smaller for stronger moves. It makes its decision as follows, selecting the lowest rank or the lowest index card for ties within a case:

- If any fully known card in the Player's hand is a duplicate of a card in a fireworks pile, then it is the best discard. The returned PlayValue is 0.
- If a fully known card is not the last copy of that card, because not all the other copies are in the discard pile, then that is the next best discard. The returned PlayValue is 1.
- If a card, from a know colour or rank, is not the last copy of that card, because not all the other copies are in the discard pile, then that is the next best discard. The returned PlayValue is 2.
- If nothing is known about any of the AI Player's cards, then a random card is selected as the best discard. The returned PlayValue is 3.

The best piece of information to give is instead determined by giving the hands of the other Players and the fireworks piles to the BestInfo filter. The best info move is described by an index of which Player to give information to and the property to tell them about (colour or rank), alongside the PlayValue of the

move. It considers the following things in making its decision while preferring to give info about higher rank cards and the lowest index cards when ties occur:

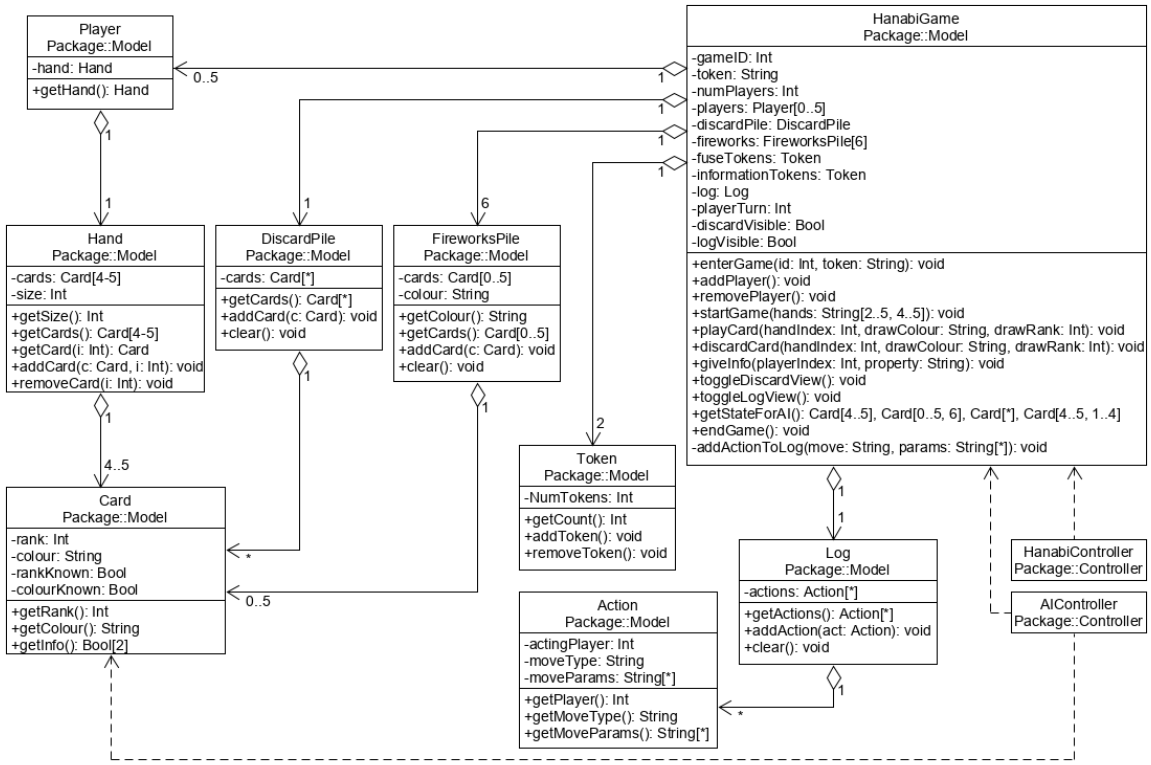
- If a card in any other Player's hand is the next card in a fireworks pile and one of its properties isn't known by them, then info about that card is the best info to give. Colour is preferred over rank if neither is known by the other Player. The returned PlayValue is 1.
- If any other Player's hand has a card with rank 5 and they don't know its rank, then that cards rank is the next best info to give. The returned PlayValue is 2.
- Otherwise, determine the Player who knows the least about their cards, determine the property that would tell them the most, and that property is the best info to give. The returned PlayValue is 3.

Once the best moves of each type have been determined, the parameters and PlayValues of those moves, plus the number of information tokens, are given to the BestMove filter. It uses the PlayValues and an ordering of moves to determine what move to select. The selected move is then given to the controller as a Move structure that says what move to make and the parameters of that move: an index into the Player's hand for plays and discards, or a Player number and a card property for information giving. It will not decide to discard a card if there are 8 information tokens or give information if there are 0 information tokens, but otherwise it will pick the move with the lowest PlayValue. Ties in PlayValue are settled by preferring information giving to plays to discards.

3. Detailed Design

The Hanabi Client's detailed design consists of a collection of classes that work together to present info to the user, accept user input, communicate with the Hanabi Server, and consistently change the model state. This section will describe how the classes work together and how they are grouped into Model, View, and Controller packages. A detailed description of the classes in each package and their fields and methods are also given.

3.1 Model Package



As shown in the class diagram above, the Model package stores the state of a Hanabi game. It does this through encapsulating concepts of a HanabiGame, Log, Action, Token, FireworksPile, DiscardPile, Player, Hand, and Card. HanabiGame is the main class in the model and handles the overall game state, aggregating the various parts of a game and ensuring consistent changes among those parts. A Log, meanwhile, deals with storing all the actions made during a game in the form of Actions, which contain information about a single move made by a Player.

The main primitive entity in the model is the Card, which has a rank, a colour, and flags for whether they are known to a given Player. Each Player has a Hand of cards, and cards can be added to and removed from specific positions in a hand. The DiscardPile and FireworksPile are other card containers that store cards that have been discarded throughout a game and cards that have been successfully played and are

part of built-up fireworks. The model's other primitive entity is the Token, which deals with tracking fuse and information tokens as they are gained and lost during a game.

For the Controller to request model changes, HanabiController depends on an instance of HanabiGame to get access to its public interface, which ensures that consistent changes are made to the model. Meanwhile, AIController uses the Card class and depends on HanabiGame despite the extra coupling to make it easier to pass around game state in the process of deciding on AI Player moves.

3.1.1 Card

Description: A Card represents a single Hanabi card with a rank and color. Both properties may be known or unknown to the Player holding it; this is shown to them by a rank of 0 and a colour of Null respectively, and to other Players with info flags.

Fields:

- Int rank: The rank value of the card, from 1-5. Can be 0 for a Card whose rank is unknown to the Player holding it.
- String colour: The colour of the card, from Red, Blue, White, Yellow, Green, or Rainbow. Can be Null for a Card whose colour is unknown to the Player holding it.
- Bool rankKnown: A flag for whether a Card's rank is known to the Player holding it.
- Bool colourKnown: A flag for whether a Card's colour is known to the Player holding it.

Methods:

- Int getRank(): Returns the numeric rank of the Card.
- String getColour(): Returns the colour of the Card.
- Bool[] getInfo(): Returns a Bool array with the Card info known to the Player holding the Card, in the form of [rankKnown, colourKnown].

3.1.2 Hand

Description: A Hand stores a single Player's hand of cards. The size of a hand is fixed throughout a game and a card stays in its position in a hand until it is removed. A place in a hand can also be empty, which is represented by a null value in a hand.

Fields:

- Card[] cards: The cards in a hand. Places in a hand with no card are represented by a null value.
- Int size: The number of cards in the hand. This depends on the number of Players in a game and is constant throughout its lifetime.

Methods:

- Int getSize(): Returns the number of cards in the Hand.
- Card[] getCards(): Returns the Hand of cards.
- Card getCard(): Returns a card from the hand at a given index.
    - Int i: The index in the hand to get the card from. Ranges from 1-5.
- void addCard(): Adds a new card to the hand at a given index.
    - Card c: The new card to add to the hand.
    - Int i: The index in the hand to place the new card in.
- void removeCard(): Removes a card from the hand from a given index.
    - Int i: The index in the hand to remove the card from.

3.1.3 Player

Description: A Player represents a Player in a game of Hanabi with their own hand.

Fields:

- Hand hand: A Player's hand of cards.

Methods:

- Hand getHand(): Returns the Player's hand of cards.

3.1.4 DiscardPile

Description: A DiscardPile stores the cards that have been discarded throughout a game of Hanabi. Cards are added to it as they are discarded, and the pile is cleared at the end of a game.

Fields:

- Card[] cards: The set of cards in the discard pile.

Methods:

- Card[] getCards(): Returns the set of cards in the discard pile.
- void addCard(): Adds a new card to the discard pile.
    - Card c: The card to add to the discard pile.
- void clear(): Removes all of the cards in the discard pile.

3.1.5 FireworksPile

Description: A FireworksPile stores a firework in Hanabi, which is a mono-colour pile in increasing numeric order from 1-5. Cards are added to it as they are successfully played while the pile is cleared at the end of the game.

Fields:

- Cards[] cards: The cards in a firework pile.
- String colour: The colour of cards in the firework pile.

Methods:

- String getColour(): Returns the colour of the cards in a firework pile.
- Card[] getCards(): Returns the set of cards in a firework pile.
- void addCard(Card c): Add a given card to the firework pile.
    - Card c: The card to add to the firework pile.
- void clear(): Removes all the cards in a firework pile.

3.1.6 Token

Description: A Token stores the count of a set of either fuse or information tokens. The token count can be either increased or decreased by one as the game progresses.

Fields:

- Int numTokens: The number of tokens.

Methods:

- Int getCount(): Returns the number of tokens.
- void addToken(): Increases the number of tokens by one.
- void removeToken(): Decreases the number of tokens by one.

3.1.7 Action

Description: An Action stores a record of a move in a game of Hanabi. This includes what Player made the move, what type of move it was, and the parameters of the move, either the hand index and values of a card for plays and discards, or a Player index and the property that was mentioned for info giving.

Fields:

- Int actingPlayer: An index of what Player performed the action.
- String moveType: The type of move that was made: Play, Discard, or Info.
- String[] moveParams: Describes the parameters of a move. For a Play or Discard move, this will contain the index into their hand and the colour & rank of the card. For an Info move, this will contain the index of the Player that was given info and the property that was passed to them.

Methods:

- Int getPlayer(): Returns the index of the Player that performed the action.
- String getMoveType(): Returns the type of move that was made.

- String[] getMoveParams(): Returns the parameters of a move.

3.1.8 Log

Description: A Log holds all the actions that have been made in a game of Hanabi so far. These actions are added in chronological order as they happen but are all cleared at the end of the game.

Fields:

- Action[] actions: The set of actions made throughout a game, in chronological order.

Methods:

- Action[] getActions(): Returns the set of actions made so far in the game.
- void addAction(Action act): Adds a new given action to the log.
    - Action act: The action to add to the log.
- void clear(): Removes all of the actions from the log.

3.1.9 HanabiGame

Description: HanabiGame represents the state of a single game of Hanabi. This includes the identity of the current game, the number of Players in a game, the Players and their hands, the discard and fireworks piles, the fuse and information tokens, and a log of the game's moves. This also includes trackers for whose turn it is and whether the discard pile and log should be visible to the Player running the Client.

Fields:

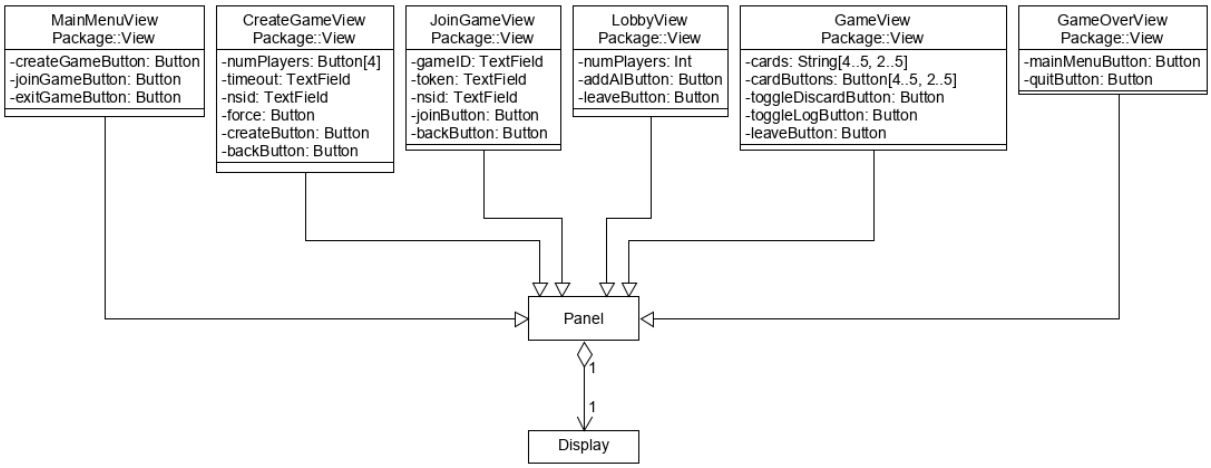
- Int gameID: The Game ID of the current game.
- String token: The Token of the current game.
- Int numPlayers: The number of players in the game.
- Player[] players: The players in the game, each with their own hand of cards.
- DiscardPile discardPile: The discard pile with all the cards that have been discarded during the game.
- FireworksPile[] fireworks: The fireworks piles, one for each colour, with the successfully played cards.
- Token fuseTokens: A set of fuse tokens.
- Token informationTokens: A set of information tokens.
- Log log: A log of all the moves made throughout the game so far.
- Int playerTurn: The index of the Player with the current turn.
- Bool discardVisible: A flag for whether the Player can see the cards in the discard pile.
- Bool logVisible: A flag for whether the Player can see the action in the log.

Methods:

- void enterGame(): Creates or joins a new game with a given identity
    - Int id: The Game ID of the game to create or join
    - String token: The Token of the game to create or join
- void addPlayer(): Adds a player to the game.
- void removePlayer(): Removes a player from the game.
- void startGame(): Starts a game of Hanabi using the cards dealt out to each Player. Each of 2-5 Players gets a hand, and each hand has 5 cards (with 2-3 Players) or 4 cards (with 4-5 Players). The Player running the Client will be given an empty hand, indicating that they don't know what cards they have at the start of the game.
    - String[] hands: The hands of cards that the Players start with, with each card in the form of 2 characters: one for the colour (r,b,g,y,w,m) and one for the rank (1,2,3,4,5).
- void playCard(): Makes the current Player play a card from their hand and replace it with a drawn card. The played card goes into a firework pile if it continues that pile in numerical order; otherwise it goes to the discard pile and a fuse token is lost.
    - Int handIndex: The index of the card to play in the current Player's hand.
    - String drawColor: The colour of the drawn card to place back into the Player's hand.
    - Int drawRank: The rank of the drawn card to place back into the Player's hand.
- void discardCard(): Makes the current Player discard a card from their hand and replace it with a drawn card. The discarded card goes to the discard pile and an information token is gained.
    - Int handIndex: The index of the card to play in the current Player's hand.
    - String drawColor: The colour of the drawn card to place back into the Player's hand.
    - Int drawRank: The rank of the drawn card to place back into the Player's hand.
- void giveInfo(): Tells the given Player information about their hand that matches the given property. That Player learns about that information for the cards in their hand currently, but an information token is lost.
    - Int playerIndex: The index of the Player to give information to.
    - String property: The property of the other Player's hand to tell them about.
- void toggleDiscardView(): Toggles the flag for making the discarded cards visible to the Player running the Client.
- void toggleLogView(): Toggles the flag for making the log of game moves visible to the Player running the Client.
- (Card[], Card[], Card[], Card[]) getStateForAI(): Gets the game state necessary for an AI Player to decide on a move and returns it as a set of Card lists. The provided state includes the AI Player's own hand, the fireworks piles, the discard pile, and the hands of the other Players.
- void endGame(): Ends the game of Hanabi, clearing all of the piles and removing all of the Players from the game.
- void addActionToLog(String move, String[*] params): Adds a move with specified parameters to the log.
    - String move: The type of move to add: Play, Discard, or Info.

- String[] params: The parameters of the move: either the hand index and colour & rank of a card for plays and discards, or a Player index and the property that was mentioned for info giving.

3.2 View Package



The class diagram above shows the View package and its eight classes: MainMenuView, CreateGameView, JoinGameView, LobbyView, GameView, GameOverView, Panel, and Display. A single window frame is made and encapsulated by Display, which then contains one Panel of any of the Client's six possible views and changes throughout the Client's lifetime. Each view has its own class that defines how that view is displayed and organized and how it updates itself when the model changes. Views also contain Buttons which turn user actions into responses from HanabiController and in turn cause changes in the Client model. The CreateGameView and JoinGameView also contain TextFields that are used to get text input from a Player when creating or joining a game.

3.2.1 MainMenuView

Description: The MainMenuView represents the Main Menu screen of the Client. It includes buttons to let the Player create or join a game or close the Client.

Fields:

- Button createGameButton: A button for moving to the Create Game screen.
- Button joinGameButton: A button for moving to the Join Game screen.
- Button exitGameButton: A button for closing the Client.

Methods: None.

3.2.2 CreateGameView

Description: The CreateGameView represents the Create Game screen of the Client. It includes text fields for specifying the timeout value for a game and the Player's NSID, a set of buttons for specifying the number of Players in a game, and buttons for creating a game with the given settings or going back to the Main Menu screen.

Fields:

- Button[] numPlayers: A set of exclusive buttons for selecting the number of players in the game. They allow for selecting Player counts of 2-5.
- TextField timeOut: A text field for specifying the timeout limit of the game.
- TextField nsid: A text field for specifying the Player's NSID.
- Button force: A button for letting a Game Creator create a new game even when an existing game made with their NSID exists.
- Button createButton: A button for creating a game with the specified settings.
- Button backButton: A button for going back to the Main Menu screen.

Methods: None.

3.2.3 JoinGameView

Description: The JoinGameView represents the Create Game screen of the Client. It includes text fields for specifying the Game ID and Token of the game to join and the Player's NSID, and buttons for joining a game with the given settings or going back to the Main Menu screen.

Fields:

- TextField gameID: A text field for specifying the Game ID of the game to join.
- TextField token: A text field for specifying the Token for the game to join.
- TextField nsid: A text field for specifying the Player's NSID.
- Button joinButton: A button for joining a game with the specified settings.
- Button backButton: A button for going back to the Main Menu screen.

Methods: None.

3.2.4 LobbyView

Description: The LobbyView represents the Lobby screen of the Client. It includes a display of the Game ID and Token of the current game so that the Player can give them to other people and let them join the game. It also includes buttons for adding AI Players and leaving the game.

Fields:

- Int numPlayers: The current number of Players in the game.
- Button addAIButton: A button for adding AI Players to the current game.
- Button leaveButton: A button for leaving the current game and going back to the Main Menu screen.

Methods: None.

3.2.5 GameView

Description: The GameView represents the Game screen of the Client, which displays the current game state while playing a game of Hanabi and has a different layout for different numbers of Players in the game. It includes buttons for each card in the game that display the information the Player running the Client knows about them. The card buttons prompt the Player about whether to play or discard a card (for cards from their own hand) or about what property of the card to tell another Player (for cards from other Player's hands). Other buttons allow for toggling of views of the discarded cards and game log and let the Player leave the game early.

Fields:

- String[] cards: The known colours and ranks of the Player's hands, with one character representing the colour and rank. Null colour and 0 rank properties are denoted with spaces.
- Button[] cardButtons: Buttons that display the known colours and ranks of cards in the Player's hands and that let the Player running the Client select a card to play, discard, or give info about during their turn.
- Button toggleDiscardButton: A button for toggling the display of the discarded cards in the discard pile.
- Button toggleLogButton: A button for toggling the display of the moves in the game log.
- Button leaveButton: A button for leaving the game early and going back to the Main Menu screen.

Methods: None.

3.2.6 GameOverView

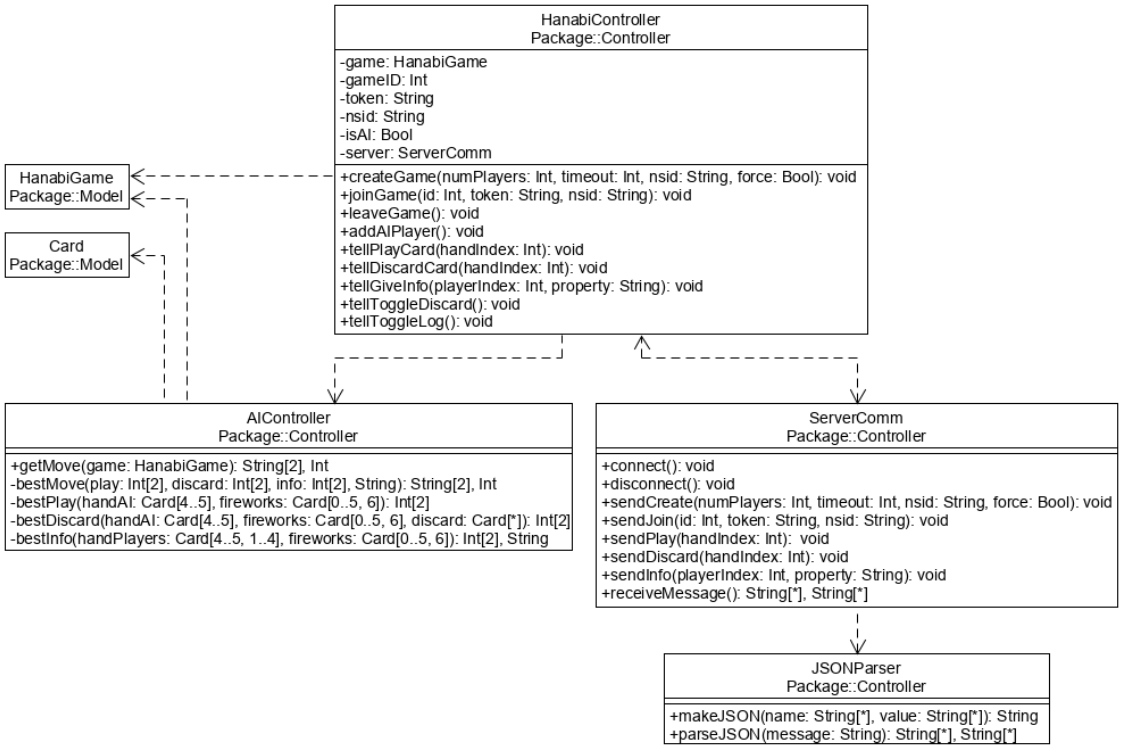
Description: The GameOverView represents the Game Over screen of the Client. It displays the final score of the game with a message proportionate to the score. It also includes buttons for going back to the Main Menu screen or closing the Client.

Fields:

- Button mainMenuButton: A button for going back to the Main Menu screen.
- Button quitButton: A button for closing the Client.

Methods: None.

3.3 Controller Package



As seen in the class diagram above, the Controller package contains four classes: HanabiController, ServerComm, JSONParser, and AIController. HanabiController is the main controller class that receives user input and coordinates server communication and model changes. HanabiController uses ServerComm to handle the task of maintaining a Server connection and sending and receiving Server messages. ServerComm itself then uses JSONParser to create and parse the JSON format used by Server messages. HanabiController is also dependent on AIController to determine the moves of AI Players whenever the Client is running with an AI Player.

The Controller package connects to the model mainly through an instance of the main model class, HanabiGame, that HanabiController maintains and uses to communicate state changes to the model. A reference to this instance is also given to AIController so that it can access the game state in the process of determining AI moves. AIController has some extra coupling too, as it depends on the Card class to encapsulate the game state passed around by its methods into fewer inputs.

3.3.1 HanabiController

Description: The HanabiController handles communication with the game model and the HanabiServer whenever the Player running the Client specifies an action that requires model changes or sent and received server messages.

Fields:

- HanabiGame game: An instance of the Hanabi game model and its stored state.
- Int gameID: The Game ID of the current game.
- String token: The Token of the current game.
- String nsid: The NSID of the Player running the Client.
- Boolean isAI: A flag for whether the Player running the Client is an AI Player.
- ServerComm server: An instance of the server connection that sends and receives Server messages for the Client.

Methods:

- void createGame(): Creates a new game with the given settings on the Hanabi Server and makes the Player join it.
    - Int numPlayers: The number of Player slots for the game.
    - Int timeout: The number of seconds for the timeout period of the game.
    - String nsid: The NSID of the Player creating the game.
    - Bool force: A flag for whether a new game should be forcefully be created if another game exists under the same NSID.
- void joinGame(): Joins the Player to an existing game on the Hanabi Server.
    - Int id: The Game ID of the game to join.
    - String token: The Token of the game to join.
    - String nsid: The NSID of the Player joining the game.
- void leaveGame(): Makes the Player leave the game they are currently in.
- void addAIPlayer(): Creates an AI Player and has them join the Player's current game.
- void tellPlayCard(): Tells the rest of the Players and the game model of what card the current Player is playing.
    - Int handIndex: The index of the card to play in the current Player's hand.
- void tellDiscardCard(): Tells the rest of the Players and the game model of what card the current Player is discarding.
    - Int handIndex: The index of the card to discard in the current Player's hand.
- void tellGiveInfo(): Tells the rest of the Players and the game model about which Player the current Player is giving information to and what information they are giving.
    - Int playerIndex: The index of the Player to give information to.
    - String property: The property of the other Player's hand to tell them about.
- void tellToggleDiscard(): Tells the game model to toggle the flag for displaying the view of discarded cards.
- void tellToggleLog(): Tells the game model to toggle the flag for displaying the view of the game log.

3.3.2 AIController

Description: The AIController deals with determining the moves of AI Players given the current state of the game. See Section 2.6 for the details of the AI move algorithm and the exact meaning of PlayValues.

Methods:

- (String[], Int) getMove(): Gets the AI's next move in the game based on the current game state. Returns the move type as a string and the hand or Player index as an integer, alongside the string information property if necessary.
    - HanabiGame game: A reference to the current game model and its state.
- (String[], Int) bestMove() : Separately determines the best play, discard, and info moves and selects the best move to make. Returns the move type as a string and the hand or Player index as an integer, alongside the string information property if necessary.
    - Int[] play: The index of the prospective card to play and the PlayValue for the play.
    - Int[] discard: The index of the prospective card to discard and the PlayValue for the discard.
    - (String, Int[]) info: The index of the Player to prospectively give information to (as an integer), the prospective information to tell them about (as a string), and the PlayValue for giving that info.
- Int[] bestPlay(): Determines the index of the best card to play in the AI Player's current hand and gives it a PlayValue based on the AI's current hand and the current fireworks piles.
    - Card[] handAI: The cards in the AI Player's current hand.
    - Card[] fireworks: The cards currently in the fireworks piles.
- Int[] bestDiscard(): Determines the index of the best card to discard in the AI Player's current hand and gives it a PlayValue based on the AI's current hand, the current fireworks piles, and the current discard pile.
    - Card[] handAI: The cards in the AI Player's current hand.
    - Card[] fireworks: The cards currently in the fireworks piles.
    - Card[] discard: The cards currently in the discard pile.
- (String, Int[]) bestInfo(): Determines the index of the best player to give info to (as an integer), the best info to give them (as a string), and gives the move a PlayValue based on the other Player's current hands and the current fireworks piles.
    - Card[] handPlayers: The cards in the other player's current hands.
    - Card[] fireworks: The cards currently in the fireworks piles.

3.3.3 ServerComm

Description: ServerComm encapsulates a connection to the Hanabi Server. It handles communication with the Server using JSON messages by sending required messages and constantly waiting for and handling received messages from the Server.

Methods:

- void connect(): Creates a connection to the Hanabi Server
- void disconnect(): Destroys a connection to the Hanabi Server
- void sendCreate(): Sends a create game message to the Server with the specified settings.

    - Int numPlayers: The number of Player slots for the game.
    - Int timeout: The number of seconds for the timeout period of the game.
    - String nsid: The NSID of the Player creating the game.
    - Bool force: A flag for whether a new game should be forcefully be created if another game exists under the same NSID.
- void sendJoin(): Sends a join game message to the Server with the specified settings.
    - Int id: The Game ID of the game to join.
    - String token: The Token of the game to join
    - String nsid: The NSID of the Player joining the game.
- void sendPlay(): Sends a play move message to the Server with the index of the current Player's played card.
    - Int handIndex: The index of the card played from the current Player's hand.
- void sendDiscard(): Sends a discard move message to the Server with the index of the current Player's discarded card.
    - Int handIndex: The index of the card discarded from the current Player's hand.
- void sendInfo(): Sends the info move message to the Server with the index of the Player that was given information and what that information was.
    - Int playerIndex: The index of the Player that was given information.
    - String property: The property of the other Player's hand they were told about.
- (String[], String[]) receiveMessage(): Checks for and receives messages from the server, returning String lists of the tags and values so that the main controller can respond to them

3.3.4 JSONParser

Description: The JSONParser handles the creation and parsing of JSON messages that are sent to and received from the Hanabi Server.

Methods:

- String makeJSON(): Takes a set of name-value pairs and creates a JSON message that encodes those pairs.
    - String[] name: A set of parameter names.
    - String[] value: A set of parameter values corresponding to the names.
- (String[], String[]) parseJSON(): String list data type and parses through the data from the server.
    - String message: A JSON message to parse.

4. Conclusion

We have described the design of a client program that lets a user play Hanabi with others over a network connection. At an architectural level, it is broken down into 3 major components organized as a model, view, and controller. The view and controller components themselves have architectures based on event-driven state machines, while the decision process for AI Players uses an architecture of data flowing through pipes and filters. At a lower level, meanwhile, the detailed design of individual classes in the Client has been described, alongside how they are organized into the model, view, and controller components of the architecture and how they work together to run the Client.

Appendix A: Requirements Changes

The development of the Hanabi Client design was partly based on the requirements that we had determined for it and documented in our requirements document. While developing the design, however, we found that we had to make a change to our requirements due to a mistake in interpreting the needs of the customer.

In our original requirements document, we specified that for a Game Creator to create a new game with force, they would do so from the Lobby screen by pressing a Create Different Game button that would confirm the action with a prompt, make them leave the game, and bring the Game Creator back to the Main Menu screen. This can be seen in the first Alternative Sequence of Functional Requirement 2 (Section 4.2 of the requirements document).

Our new requirement is that force creating a new game is a togglable option on the Create Game screen during the regular process of creating a game. The Alternative Sequence mentioned above would be removed and the Main Sequence for creating a game would change in Steps 2 and 3 to accommodate the new Force toggle as follows:

2. A new “Game Settings” prompt appears, displaying buttons and text boxes for the Player to give the number of Players, the time-out period, the force option, and their NSID
3. The Game Creator selects a button for the Player count, selects the button for whether to force create a new game, and types the time-out period and their NSID into the corresponding text boxes