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.

1.4 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.

2.1.1 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, and Rainbow, and a numeric rank from 1 to 5. 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.

2.1.2 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.

2.1.3 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 2.1.2 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 a 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.

2.1.4 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 single 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.

2.1.5 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 is still notified of game changes by 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 AI 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.