Hanabi Client: Final Project Report

CMPT 370 - Group F1

Minh Hoang - mph952

Sharjeel Humayun - shh738

Donovan Lavoie - dbl599

Georgi Nikolov - gnn308

Faiz Qureshi - faq588

Jason Wong - jaw845

1. User Documentation

1.1 Installation Instructions

We were not able to finish our project and reach the deployment phase, but we did have a plan for how we would deploy the final Client. The plan was to generate an executable jar file so that there would be no special instructions for installing the Client itself. The Client would work on the Linux computers in the computer labs of 3rd Floor Spinks operated by the Computer Science department but could technically work anywhere with a JRE that supports Java 11. Some further configuration would be required, however, in order to connect to the server: users would have to create a config file named “hanabisecret.txt” in their home directory with their NSID and secret hash in the format of “abc123:hash” on a new line. This would allow the Client to grab a user’s hash from this file and have the Server accept connections.

1.2 Desired Usage Instructions

Once the Client is installed and set up, a Player should open up a command prompt in the directory with the jar file. The Client can be started for a human Player with the command:

java -jar HanabiGame.jar

For an AI Player, this command should also have two extra arguments: -g with the game ID of the game for the AI to join, and -t with the game’s token.

A main menu screen will then appear in the terminal presenting options to create a game, join a game or exit the Client:

Welcome to Hanabi!
1. Create Game
2. Join Game
3. Exit Game
Your choice:

If the Player types a 3, the Client will terminate and return control to the command prompt after displaying:

Thank you for playing!

Otherwise, when the Player types a 1 (to create a game), the Client presents a sequence of prompts to them for the parameters of the game to create:

Please enter number of players (2-5):
Please enter timeout period (1s-120s):
Enter Nsid:
Force Create Game? (Y/N):
Should rainbows be used? (Y/N):
Which kind: Firework('f') or + Wild ('w'):

These parameters are, respectively, the number of players in the game, the maximum time allowed for Players to make a move, their U of S NSID (which the Player must have to connect to the Server), and whether or not to forcefully create a game (if another game has been started with the same NSID). Yes/No prompts can be answered with any case of 'Y' or 'N'. The selections for rainbow cards show up only if the Player says to have them. Selecting "firework" means that rainbow cards are treated as a separate colour while "wild" will allow them to be played on any fireworks pile.

If the user instead types a 2 (to join an existing game), a similar sequence of prompts appears:

Enter Game ID:
Enter Game Token:
Enter Nsid:

These are asking for the Game ID and Token of the game the Player wants to join, which are unique to each game, as well as their NSID.

If an error occurs in creating or joining a game, the Main Menu prompt will appear again after a error message is displayed. Otherwise, successfully creating or joining a game will bring the Player to a Lobby which prompts them as follows (where <n>, <id>, and <token> are placeholders):

Slots left: <n>
To begin the game please wait until the lobby is full
Game ID: <id>
Game Secret: <token>
To go to Main Menu ('b')

As other Players join or leave the game, the Lobby's prompt will be reprinted with the new number of slots left. The ID and secret token are displayed so that the Player can give them to others who want to join the same game. If the Player types in a 'b' before the game starts, they will leave the game and the Main Menu prompt will be displayed again.

Once all of the Players join a game and a game is in session, the hands are dealt to the Players and the starting state of the game is displayed:

F [0b, 0r, 0g, 0w, 0y] Info: 8
P1 [2b, 1r, 1g, 2w, 3b] Fuse: 3
P2 [2m, 1b, 1w, 2b, 3r]
P3 [,,,,,]

The Client Player's hand is the one that starts off with no card information in it. Card information is specified as a numeric rank (from 1-5) and a colour character (Red: 'r', Blue: 'b', Green: 'g', Yellow: 'y', White: 'w', or Rainbow/Multicolour: 'm'). Fireworks use the rank 0 to indicate that they are empty and always have the same colour. The goal of the game is to build up fireworks piles of the same colour (up to 5) until either all of the pile are finished or all 3 fuse tokens are lost.

Players have the choice of playing a card, discarding a card, or giving information to another player on their turn. As other Players make moves, the state is updated and reprinted. Once it becomes the Client Player's turn, a prompt is displayed after the state of the game:

> **Play('p') Discard('d') Info('i') Quit('q')**
> **Your choice:**

The Player then chooses one of these actions and is given a sequence of prompts to get the required parameters. For playing a card onto a firework, the Player needs to specify which card to play and which fireworks pile to play it on, so the prompts are:

> **Which card would you like to play? <n>**
> **Do you want to change your choice? (y/n):**
> **You chose to play card # <n>**
> **Which pile would you like choose? <m>**
> **You chose to to play on pile # <m>**

For discarding a card, which requires that there are less than 8 information tokens, they only need to select which card to discard, so the prompts change to:

> **Which card would you like to discard? <n>**
> **Do you want to change your choice? (y/n):**
> **You chose to discard card # <n>**

For info giving, meanwhile, they must specify the other Player to give info to, which card has the property to tell them about, and what info to give; this requires that there is at least 1 information token. The prompts then become:

> **Choose a player: <m>**
> **Choose a card: <n>**
> **Do you want to change your choice? (y/n):**
> **You chose Player <m> and their card # <n>**
> **Which property should you tell them about (c: Colour, r: Rank)?**
> **Do you want to change your choice? (y/n):**
> **You choose to tell Player <m> about their <colour or rank> cards**

The other Players are then told about the move and the state updates and is redisplayed. Plays and discards result in a new card being drawn and replacing the played or discarded card. Played cards go onto a firework pile if they are the next number in the pile of the same colour (or a rainbow card if they are treated as "wild"). Playing non-consecutive or off-colour cards onto a pile will discard that card and remove a fuse token. If the Client Player is given information about their cards by another Player, that information will be added to to the display of their own hand when the rest of the game state is displayed.

The game can end naturally (all three fuse tokens are lost, all of the fireworks piles are completed, or everyone gets 1 turn without the possibility of drawing a card) or unnaturally (any Player leaves or is disconnected from the game or the Player selects 'q' at the first move prompt). Unnatural ends will

bring the Player straight back to the Main Menu. For natural ends, however, the Game Over prompt will then appear and display the game's final score, which is the sum of the heights of the fireworks piles. The prompt differs based on the game's score (with the examples showing the minimum score for each one):

> **Your score is 0! Oh no! The crowd is getting the eggs ready!**
> **Your score is 6! Poor you! There was hardly any applause!**
> **Your score is 11! Meh! But audience is somehow still awake!**
> **Your score is 16! Nice! They liked you this time!**
> **Your score is 21! Oh WoW! The crowd is cheering!**
> **Your score is 25! Legendary!!! This will go down in history!!**

Once the score is presented, another prompt appears asking the Player whether or not to go back to the Main Menu or exit the Client:

> **("m") for Main Menu or ("q") for Quit:**

If the Player decides to go back, then the Main Menu prompt appears again from which they can create or join another game. If they choose to exit the Client, then the Client ends and control is returned to the command prompt.

1.3 As-built Requirements

Some changes were made in the planned and built Client relative to our original Requirements Document. The most significant change was that we no longer used a GUI to display information to the user or buttons to get their input. We overestimated our capabilities, so we downgraded to building a console application that displayed information as text on Standard Out and took text input from the user on Standard In one parameter at a time.

We also changed some aspects of playing a game. The way that users selected their move waby making them specify a move type before other parameters rather than selecting a card and then selecting a move type based on whether or not it was in their own hand.

Some features had to be scrapped that presented some extra information to the user. We were unable to present NSIDs associated with each Player or anticipate the number of Players actually in the game on the Lobby View since that information was never given to us by the Server.

2. Programmer Documentation

2.1 Build/Compilation Instructions

The Hanabi Client project is maintained in a GitLab repository which can be accessed at https://git.cs.usask.ca/370-19/f1/. The 'master' branch should have the latest version of the complete project. The code is divided internally into three packages: "view", "model", and "controller". Documents related to our group's development process can be found in 'Documents' while 'src' contains the source code and 'doc' contains the JavaDoc for our code. The start of the JavaDoc can be found at:

https://git.cs.usask.ca/370-19/f1/blob/FINAL/doc/index.html

Building the Hanabi Client first requires obtaining its dependencies, which are maintained using the Maven build tool. The POM file at the root of the repository can be used by Maven to import the project dependencies needed to run the source code and its unit tests. Alongside the Maven Surefire plugin for running all of the unit tests at once, the dependencies include:

- GSON 2.8.5 (see https://github.com/google/gson)
- JUnit 5.4.1, particularly the Jupiter API and Jupiter Engine packages (see https://junit.org/junit5/)

Once the Maven project is imported using the POM file, the source code for the Client and tests can be compiled by executing Maven's "compile" goal or with the help of an IDE's build tools. Maven will use its default settings during the compilation process. Our project must be compiled using JDK 11. Once compiled, the project's main method then lies in the MainController class.

Of note is that some classes use the Observer and Observable interfaces in the Java standard library. These interfaces were deprecated in JDK 9, so proper compilation will still produce some deprecation warnings about using these.

2.2 As-built Design

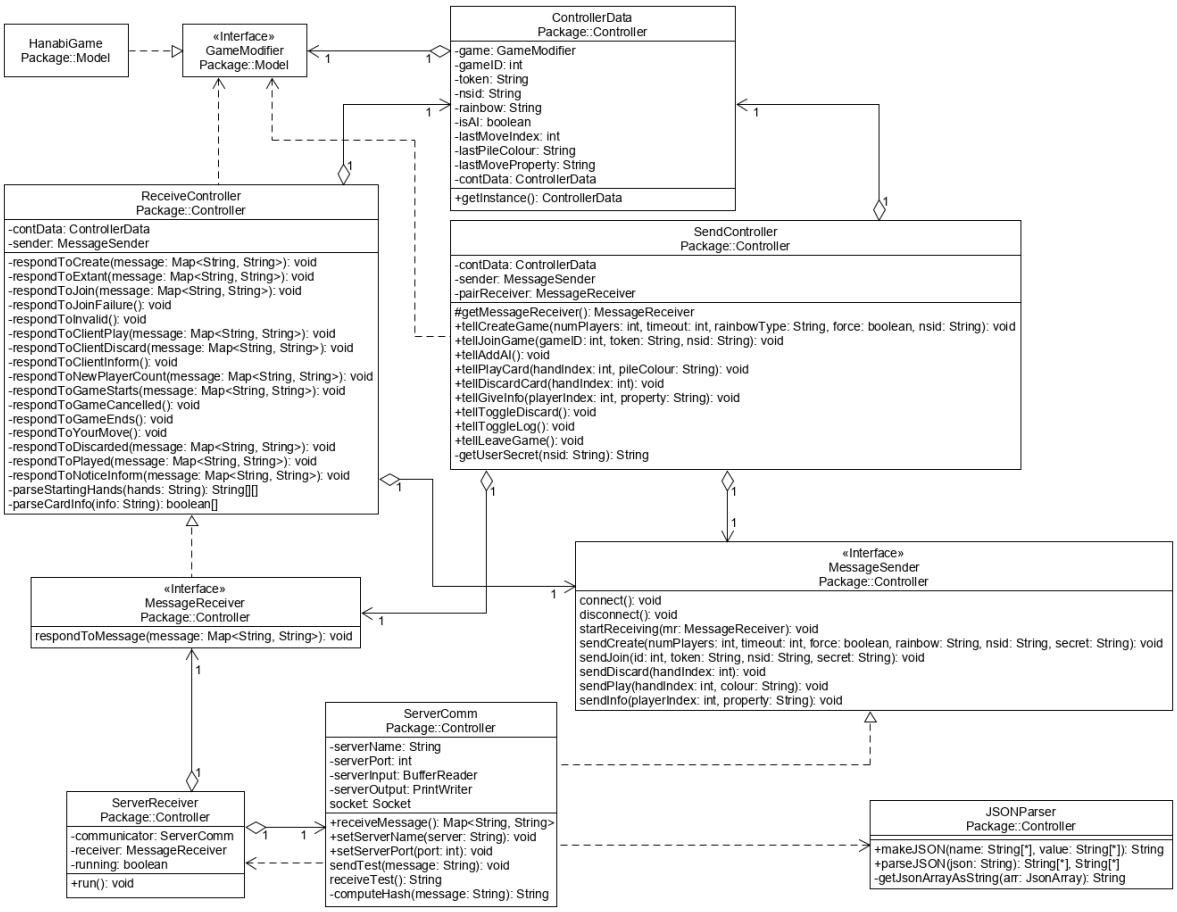
Compared to our Design Document, our overall architecture remained the same; the code is split at a high level into Model, View, and Controller components while a class for determining AI Player moves uses a pipeline architecture internally. We made several large changes, however, to our detailed class design and implemented a new sub-architecture into the Controller.

2.2.1 Controller Changes

In the Controller package of the Client, the single HanabiController class was broken up into 2 classes to separate their responsibilities to send and receive messages with the Server: SendController and ReceiveController. Two new classes were also added. ServerReceiver lets the controller asynchronously receive messages from the Server in a separate thread while other messages can still be sent. ControllerData was added to coordinate common data between the Send and Receive controllers like access to a Model reference and parameters for Client Player moves like hand and players indices that server messages won't echo back. Two interfaces for a MessageSender and

MessageReceiver were also added to allow the various classes more easily use mock classes when developing unit tests.

The new UML class diagram for the Controller package can be seen below. The class diagram also shows that the overall Controller now uses a call-and-return architecture. The core functionality of communicating with the Server is contained in ServerComm. Its functions are then exposed via the MessageSender and MessageReceiver interfaces to the rest of the controller, which can then send and receive messages as required by the View and Model.



SendController contains references to the ControllerData block, a MessageSender and a MessageReceiver that is paired to it. Its methods, meanwhile, all pertain to being told by the View to send various messages to the Server and commands to the Model directly, which may involve setting some fields in the ControllerData block to pass along Player move information to the ReceiveController. It also has a private method which reads the secret hashes for several NSIDs as configured by ‘hanabisecret.txt’ and extracts the hash for the Player trying to play a game.

The MessageSender in SendController is a ServerComm instance that exposes the methods needed to send messages, make its paired receiver start receiving messages, and establish Server connections. Receiving is done by having ServerComm start a ServerReceiver in a new thread which it supplies with references to itself and SendController’s paired receiver. The ServerReceiver then blocks on a call to ServerComm’s receiveMessage() until either the connection is closed or a message

comes in from the Server. These messages are then passed on to the MessageReceiver that it was given by calling that interface's only method, respondToMessage().

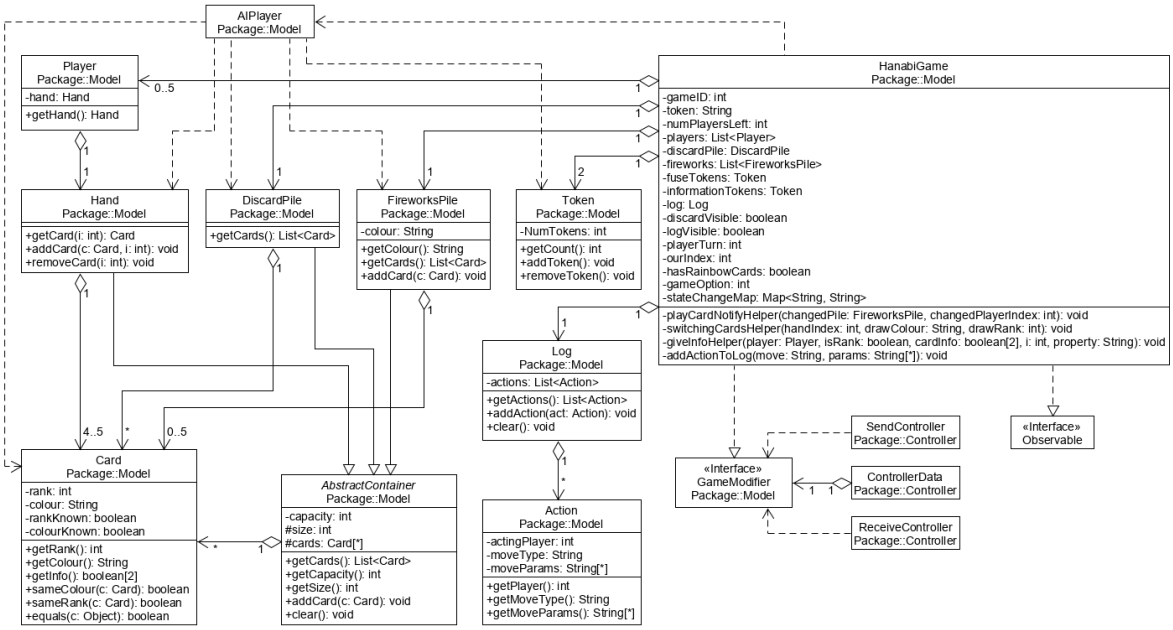
ReceiveController then gets that message and uses the first field (which is tagged with "reply" or "notice" in the original JSON message) to delegate the response to one of many private functions. Most of these functions simply extract message data, grab any required move data from the ControllerData block, and tell the model to update based on the actions the Player took, whether in the game or in a menu. ReceiveController does sometimes need to sever the Server connection, so it contains a MessageSender reference to allow this.

Some other small changes were made throughout the Controller. ServerComm's fields for the Server's name and port number were exposed via setter methods so that the name and port could be redirected to a local mock Server during testing. SendController also uses a protected method to generate a ReceiveController as its paired MessageReceiver during its construction. This method can then be overwritten during testing to instead generate a mock receiver. Many of the sending methods were also updated to accommodate the inclusion of rainbow cards and their settings into the game.

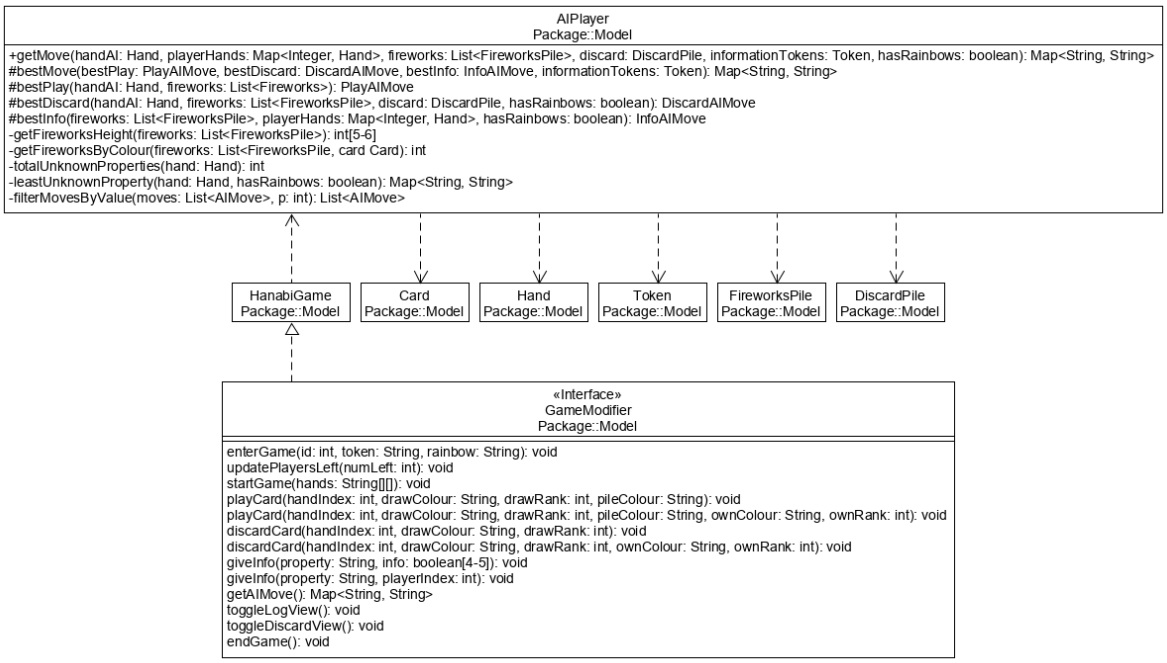
2.2.2 Model Changes

The Model package saw less drastic changes compared to the Controller. A new abstract class, AbstractContainer, was added to place some of the common operations in the Card containers in one class. Most of the public methods in HanabiGame were placed into an interface called GameModifier to facilitate testing of the Controller classes and the methods had come changes to accommodate the new rainbow card modes. The main move methods were overloaded into 2 versions to better handle the differences in Client Player vs other Player moves. The biggest change was turning AIController in the Controller into AIPlayer in the Model since it made more sense for it to have access to Model classes. The Card class also gained some new comparison methods to facilitate AIPlayer's work.

The UML class diagram was split into 2 parts since the AIPlayer class and GameModifier interface got too large on their own to share 1 diagram with them. The first one below shows the part with the original components with the additions of AbstractContainer class and GameModifier. Cards are the fundamental unit in the model and store their own properties and knowledge about them while various containers like discard and fireworks piles and Player hands track them. A discard pile, 5-6 fireworks piles, a log of Actions taken in the game, 2 Token objects for tracking info and fuse tokens, and 2-5 Players are maintained by HanabiGame, which implements the GameModifier interface to expose itself to the Controller. HanabiGame also has to track a game's ID and token to expose to the view, the indices of the current and Client Player, and a flag that determines whether rainbow cards are in the game and how to handle them when they are played since they can either be a separate colour or a set of wildcards.



The other class diagram for the Model expands on the methods for the AIPlayer class and GameModifier interface. GameModifier exposes the functions for basic model manipulations needed by the Controller: entering games and adjusting the remaining number of Players, starting a game from starting hands, making moves like plays, discards, and info giving, and ending the game. It also lets the toggle and discard views be toggled on and off and exposes a way for the Controller to get moves for AI Players. AIPlayer is what produces those moves using a large amount of state for the game passed in through getMove, though passing in a HanabiGame reference would likely have been easier. As described in the original Design Document, getMove uses bestPlay, bestDiscard, and bestInfo to determine the best move of each kind before using bestMove to select from the three based in playValue rankings. Not shown in the class diagram are a set of inner classes in AIPlayer (AIMove, PlayAIMove, DiscardAIMove, and InfoAIMove) used for passing AI moves between the private methods that implement the decision algorithm, storing the move's parameters, and converting their contents to a Map for output to the Controller.



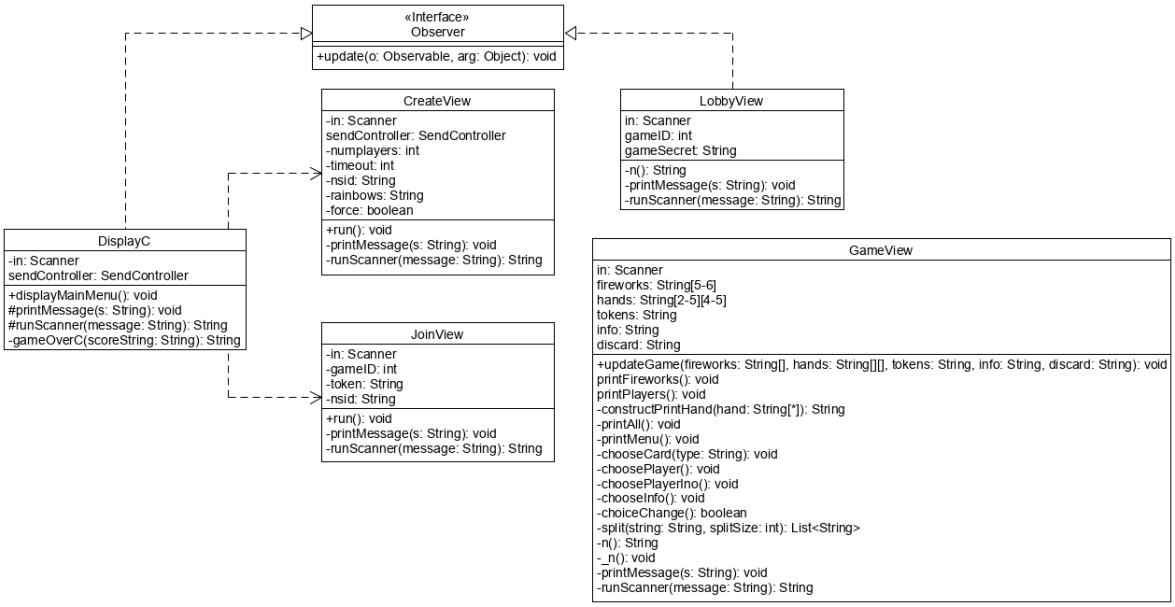
2.2.3 View Changes

The View component had some of the most drastic changes in our design, and this was driven by our attempts to implement our original design.

Initially, the awt and Swing libraries were used to create the various screens described in our Requirements Document. The goal was to have a single window frame encapsulated by the Display class which contained one Panel instance; this instance would switch between the different screens that subclassed Panel throughout the Client's lifetime. We found, however, that building them with Swing was tedious and imposed too many challenges with alignment and image overwriting. One particular issue was the panel having its own layout which had to contain the background image and other, more important, components like buttons and text labels. Images also needed their own layouts, as did groups of components and subgroups of buttons, which made it difficult to build the View classes quickly enough and connect them to the other components.

We then tried to build the View using JavaFX since it seemed more user friendly to build in, but it imposed other issues with cleanly separating the different screens into classes and connecting the button listeners and update methods to the Controller and Model, whether directly or using aspects in AspectJ. The code also turned out to be heavily operating system dependent, hampering the ability for multiple group members to work on the View and build it quickly.

It was decided at the last minute to switch to a console based implementation of the View that directly used Observer and Observable interfaces to connect it to the other components. The design of the View was then changed and what little we were able to accomplish is shown in the UML class diagram below.



The idea was to have different classes handle console displays and interactions for different Client screens using their own display(), print(), run(), get(), and choose() style methods. The DisplayC class would be the starting point of the Client and handle the display and inputs of the Main Menu and the Game Over screens, while the other screens got their own classes: the Create Game, Join Game, Lobby, and Game screens. DisplayC would also take a SendController as input to pass to new screens that it spawns which need to send messages to the Server.

All of the screen classes were also meant to be Observers that were notified of changes to the Client's one Observable: HanabiGame. They would all implement the update() method and get a map of the changed game state passed to them which they would inspect for changes they cared about. They would also hold onto those bits of state that they cared about during their operation. CreateView and JoinView would store the game parameters they obtained from the user in fields while the Lobby view would store the current game's ID, token, and remaining number of Player slots. GameView would have to stash the most state since it required all of the hands, card information, piles, and tokens to display the current state of a running game of Hanabi.

Screen transitions were supposed to be handled by constructing new screen classes and running their main display() method. In cases where a transition required a received Server response first, the View would simply have no methods running at that time and the response would be noted by a change of state in HanabiGame. The update() method would then have the job of recognizing that change of state and constructing the next screen class in the desired sequence.

2.3 Known Bugs, Incomplete Features, and Workarounds

A number of bugs have been identified but are not currently checked for in testing. In HanabiGame in particular, the discard pile always has a capacity of 50 even when rainbow cards are in use. When playing cards, information tokens are not added when a firework is finished and there aren't 8 information tokens (i.e. a 5 is successfully played), actions are not added to the log when a card is burned, and wild rainbow cards are always played on the first pile regardless of the Player's wishes.

The playCard and discardCard methods are also unable to handle reaching the end of the draw pile and getting the properties of a non-existent card, which will either throw an exception or propagate to other methods exhibiting unknown behaviour.

In SendController, meanwhile, the code for getting the user hashes from “hanabisecret.txt” is only confirmed to work on one of the developer’s Windows machines and it is believed to not be cross-platform. The code for spawning a new Client process to run an AI Player is also untested due to the main method in MainController not implementing the detection for an AI Player process being started with command line arguments.

As an unfinished project, several features were unable to be implemented besides the AI Player version of the Client. Most notably, most of the View component was unable to be finished in time; only the Main Menu and Create Game views are connected together and they can only get as far as sending a message to the server before looping back to the Main Menu view again. The togglable displays for the action log and discard pile were also not implemented in any way in the scraps of isolated code in the Game View, and the AI action that was made on a Player’s behalf if their timeout period expired was not implemented in the Controller or Model.

The Controller classes that dealt with TCP sockets used one workaround to deal with the nature of the TCP protocol. When a socket is being read from and blocks a thread, there is no way to stop the reading process on command without closing the connection on either side. When we want to disconnect on command in the Client, ServerComm’s disconnect just goes straight to closing the socket with no regard for potential ongoing reads in ServerReceiver, which is expected to catch a SocketException, recognize this scenario, and do nothing but stop ServerReceiver’s thread since this is a normal situation.

2.4 Highlights

Despite our failure to complete the project, there were some highlights in the development process in terms of the tooling used. The major highlight was our use of JUnit 5 to write our unit tests. It allowed us to write all of the tests we had planned for a specific class in one separate file and made it easy to run all of them at once. This separation of our main and testing code also made it easy to create mock versions of other classes (after we added some interfaces in our class-level design) and make the class being tested use them. There was one case, however, where we couldn’t separate a class from its dependency very easily due to the way the Controller was designed. While a design change would be ideal, we dealt with testing that class in the meantime by splitting generation of that dependency into another method and having the test class make a subclass that overrode that generator method, allowing for testing with minimal changes to the original class.

We also used a subset of the capabilities of the Maven build tool for our project. While we didn’t go to the lengths of using it for continuous integration, we did use it to simplify the project’s dependency management so that we could pass around the POM file and use it to download the same versions of libraries as everyone else. Near the end of the project, we also started to use the Surefire plugin to run the unit tests for all of the classes at once, which simplified the testing process once our code base got rather large.

3. Post Mortem

As has been stated throughout this report, we were unable to finish the project. This section makes an effort to describe what went wrong with our group and the project in general, how we would do things differently if we were to try again, and our potential changes and future work if we had some extra time.

One major problem we had was that there was never a single idea among our members of what the current design was at any given time. This came about because of a lack of communication, whether through detailed meeting agendas and minutes or through outside personal talks, that prevented ideas and changes from propagating among ourselves and our work. As such, the detailed design became fragmented as the Controller made untrue assumptions about the Model's presented interface, the Model made untrue assumptions about what information would be given by the Server, and the View was left to wither in isolation with untrue assumption about how it would communicate with the rest of the Client until the last minute.

We also could not stick to a single implementation of the View throughout the life of the project. We started out using Swing to implement our View classes, but then switched to using JavaFX halfway through the coding phase and switched again to building a console app within a week of the deadline. We were overconfident in our preexisting knowledge of GUI development and our ability to learn new GUI frameworks in time to finish the project. The design we came up with for the View in general was also not fleshed out enough to provide a clear idea of how it was to be constructed in any GUI framework; this itself was caused by poor time management leading to the creation of a detailed design at the last minute of the design phase.

The poor time management of our group and its members was also a leading problem for the project. Some members could not find the time to work on their assigned tasks and eventually turned in half-finished work (which was not labeled as such) when they finally did get to their project work. This lead one member to do and redo almost everything for the requirements and design phases, which backfired on the whole group as they became the only person with a clear view of the whole project and poorly communicated that knowledge to the rest of the group, destroying everyone else's productivity.

If we had another attempt at completing the project, one of the first things we would do is to focus on having more topics at our meetings, fleshing out the agendas and minutes more, and then having everyone promise to use them as a canonical source of knowledge on the project's current state. Better meetings and more diligence would provide an avenue for better communicating any design changes to the rest of the group and keeping everyone on the same page when things change. Quicker communication could be still be handled with the Slack channel we made so that members could quickly ask and answer any questions they may have.

Another major change would be to define some schedules for when work should be done. This allows some time for handling and reassigning undone work and providing constructive criticism on poor work. While the extra time as a group would mitigate the issues, personal problems with getting work done could still be an issue, and with a fixed group we can only do so much to hold them accountable. What we could do, though, would be to try and make an effort to document and

understand each other's personal work schedules to try and more fairly distribute work, using redaction of member names from submitted work as a final resort if they still can't perform their assigned tasks.

Alongside better time management, we would also have to take some time to more thoroughly evaluate our skills and our ability to learn new skills within the timeframe of the project. This chiefly would have prevented our attempts to create a GUI application when none of us had the skills and time necessary to pull it off. This in turn would give us more time to focus on creating better requirements and designs upfront and to confront other problems with the project like incorporating aspects and implementing the subject-observer design pattern.

If we were given an extra week, our future work would focus on redesigning the View component of our Client and having a meeting where everyone is acquainted with the current design of the project. This would lead into an effort to rebuild the View classes and their connections to the Model and Controller until we could get to the point of playing a simple no-rainbows game of Hanabi with 2 Players. Once that was established, we would then split up to implement the missing features like AI Players and viewing the log and discard pile and fix bugs with reading our config file ("hanabisecret.txt"), playing cards, and handling rainbow cards and the end of the draw pile.