Final Report

Mels Kolozyan
Xudong Li
Xufeng Lin
Evan Semenoff
Tiandu Xie

Part I
User Documentation

- **Final Deliverable:** A zip folder with all of the assets and Java .class files required to run the game.

- **Installation Instructions:**
  Extract the contents of the zip file to a folder, and the installation is finished. All that you need to do to start the game is double click on the JAR file in the main directory.

- **Configuration Instructions:**
  The software requires no special configuration files.

- **Basic Usage:**
  The basic usage of the game is meant to be straight-forward, and intuitive. To start the game, it is as simple as double clicking on the JA

- **As-Built Requirements:**
  The biggest change from the requirements phase to the final deliverable product is the introduction of the two extra game modes. To be totally honest, this change was actually anticipated by the team, so although a few small things have been altered in the requirements, they were done in such a way to easily accommodate this change in the future. As far as changes to the requirements that our team made, we improved on some of the original requirements concepts during design, but never radically re-did the requirements, these changes included:

  1. Changing user interaction from clicking buttons to drag/drop style interface
     The reason this was done was we realized that if every single choice a player makes is handled by a pop-up dialog and a series of radio buttons, it quickly reduces the game to 'radio-button-simulator 2019', which seems to take a lot of fun out of the game.
  2. The AI got considerably more complex as the project advanced
     The reason for this change (from the original requirements of the AI to only be able to play better than a player who simply selects

random legal moves) was because it was quickly realized during implementation that the statistical model of finding strong moves was a lot easier to implement than originally thought. As a result a lot of extra time was available to make the AI stronger. This was a result of an under-estimation of our skills.

3. Changes in Libraries Used
   In the end, we decided to not use either EJML (Efficient Java Matrix Library) because we realized it was easier to just write our own simple matrix operations rather than learn a new library. We also ended up adding both GSON and Simple-JSON libraries to deal with JSON.

As far a changes go, this was it. Our requirements documentation very accurately reflected the actual requirements of the project. As a result, any changes that happened afterwards were usually small and had a low overhead, or they were changes that we did because we realized that we could deliver more in certain parts of the project, and that the scalability of our design allowed us to do this with little extra cost.

Part II
Programmer Documentation

1 Compilation Instructions

The program can be built from IntelliJ, provided the following libraries are installed:

- **Simple JSON 1.1.1**
  The standard JSON library we used for creating and dealing with JSON messages
- **GSON 2.8.5**
  This is the JSON library we used strictly for communicating with the server, as it is order preserving (important for hashes)
- **JavaFX 11**
  This was the library used for drawing the game table.

Given that these are installed, compiling the game into a working JAR should be as simple as using the 'build artifacts' method in IntelliJ. Simply go to the build menu, select 'build artifacts' and then click OK.

2 As-Built Design

The architecture as designed, did not change from the design phase to the delivery phase. By the time we were done designing the general architecture of the program, we never felt a need to go back and change anything during implementation. That being said, during the design phase itself we went through multiple refinements of the architecture. Each time we re-examined the architecture during this phase we aimed to slim down the architecture and make it more simple, rather than adding extra complexity. We will discuss why we believe this was an important aspect to our success in our highlights section.

With regards to design (in the strict sense) there several differences between the final project, and the design handed in on March 3rd. We will outline them by highlighting how various classes were split into multiple classes, and how some were merged.

- **Model** The model underwent considerable change, in that it transitioned from being a completely inert object (that only dealt with getters and setters) to being an object that kept track of a lot of important calculations about data (for the view and the AI). Our original design of the model, we realized, was flawed in that the model was responsible strictly for storing data about the game, where any other data that was being calculated using the model data was being stored locally by whatever object called for the calculation. We realized later that if these calculations were a property of some of this game state data, that it would make more sense to get the model to keep track of these properties along with the data that it came from. This way, the calculations were all stored in one place, making the code easier to read/reason about.

  Things that were moved from various parts to the model were:

  - The 'TopCard' variable, which keeps track of a probability distribution for the next card to be drawn.

  - The discard matrix, and the play matrix. Since these elements were logically related to the cardsOnTable matrix, it made the most sense to store them closely.
  - Selected cards, and cards which are being dragged. It seemed to make the most sense to move these out of the view, making it inert with itself. It allowed us to separate the reasoning of how a card is drawn from the logic of why it is being drawn.
  - The addition of playBox, discardBox, and turnBox. These were all moved into the model, for the same reason as the selected cards field was moved over. This again allowed the view to be designed so it only needed to be concerned with drawing the right thing at the right time, not determining if it was the right time.

- **View** The view had the addition of many extra classes. Although they didn't change how the view fit into the architecture, they were constructed to address the various aspects of the main menu system, and the general flow of the software. These were added because

  The view also obtained a few extra fields

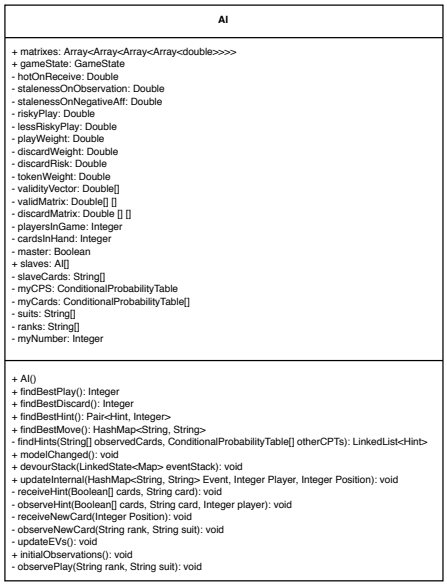
- **AI**

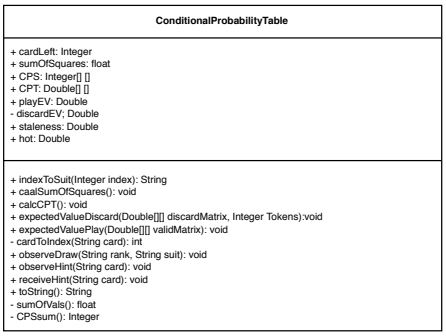
  The AI was expanded considerably during the implementation phase. The first major change was the move from keeping the probability tables as fields of primitive double arrays, to turning them into their own class. This was to aid in modularity, as the probability tables should be able to change how they work independently of the AI and its decision making process (in anticipation of the rainbow card addition).

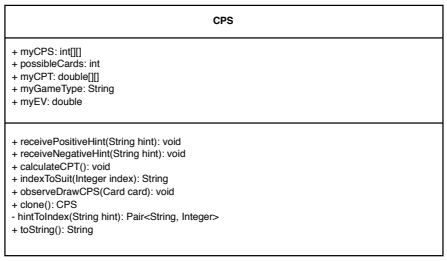
  The next change was the addition of a Hint class for the AI. It might seem strange to have a class specifically for hints, and not for plays and discards, but the discard and play moves are already handled by the card class. Not only that, but hints are considerably more complicated, not only to generate, but to keep track of their expected value.

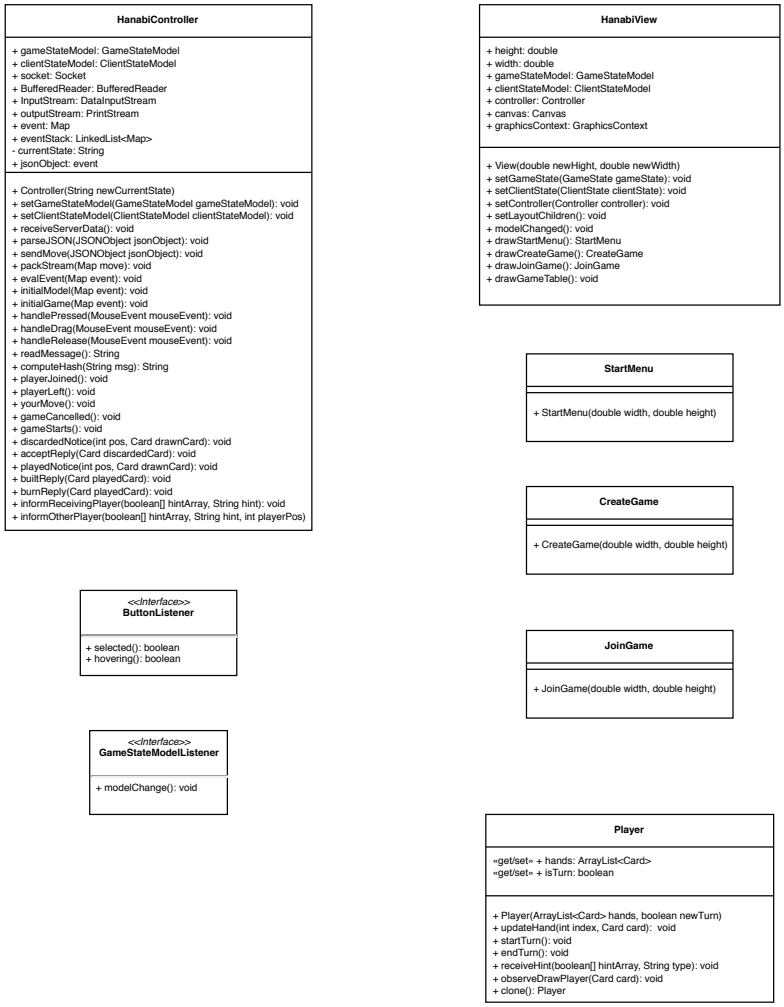
  Next, the devourStack method (a method that would allow the AI to get up-to-date with the current game state when instantiated by a user) was scraped, as we realized that keeping track of all of the AI statistics in the background (even when the user doesn't want the AI playing) was not only easier, but had almost no impact in performance.

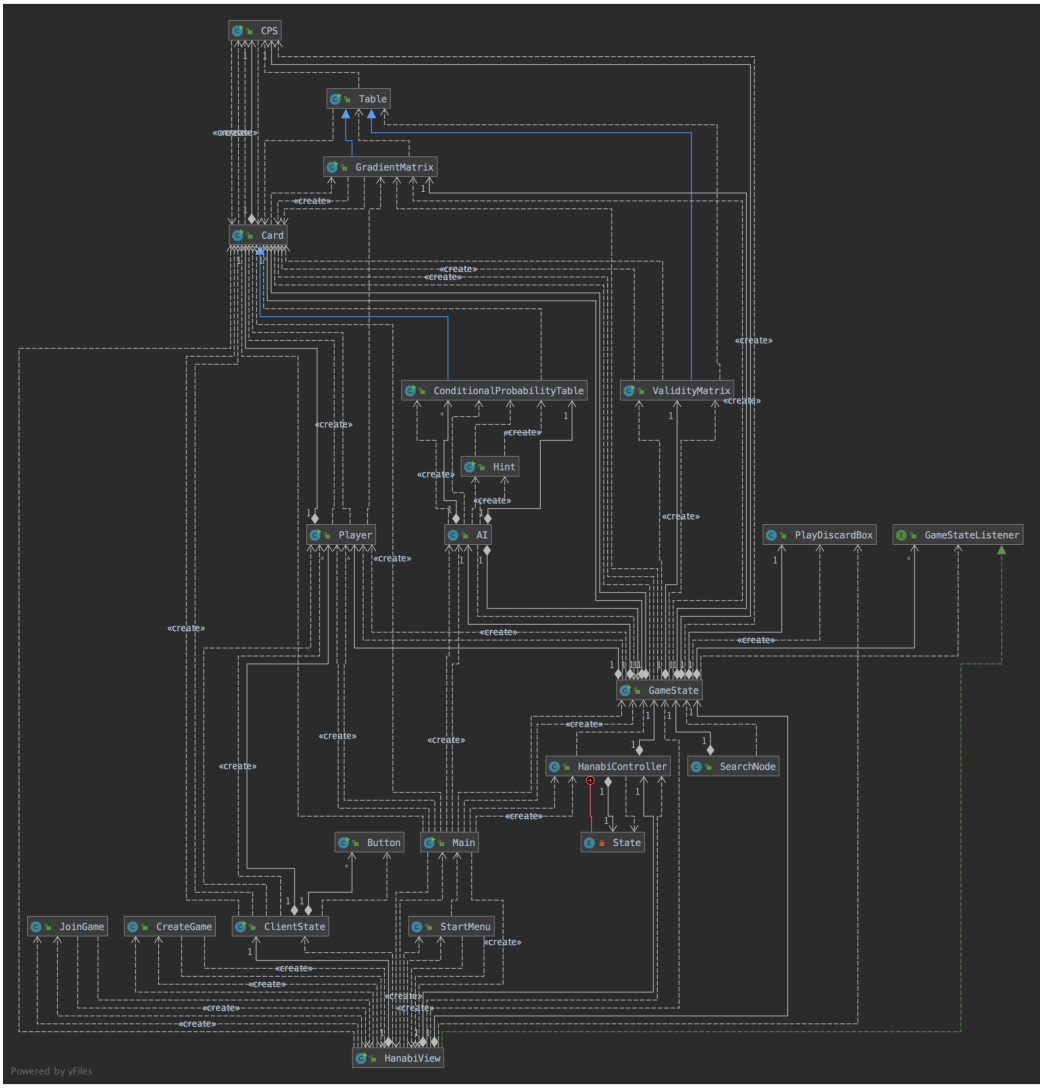
The last, and probably most important change (at least as far as performance goes) in the AI was the addition of many extra fields that aid the AI in its decision making process. One of these was the inclusion of a recursive structure, where the AI has copies of other 'AIs' to keep track of a belief state about the other players belief states, as well as many heuristic values that weighted different expected value calculations. Another field which was added as a 'top card' field which essentially keeps track of what the probability distribution of the top card on the stack is, allowing us to easily update the hand of the AI when a new card is drawn.


AI
+ matrixes: Array<Array<Array<Array<double>>>>
+ gameState: GameState
- hotOnReceive: Double
- stalenessOnObservation: Double
- stalenessOnNegativeAff: Double
- riskyPlay: Double
- lessRiskyPlay: Double
- playWeight: Double
- discardWeight: Double
- discardRisk: Double
- tokenWeight: Double
- validityVector: Double[]
- validMatrix: Double[] []
- discardMatrix: Double [] []
- playersInGame: Integer
- cardsInHand: Integer
- master: Boolean
+ slaves: AI[]
- slaveCards: String[]
- myCPS: ConditionalProbabilityTable
- myCards: ConditionalProbabilityTable[]
- suits: String[]
- ranks: String[]
- myNumber: Integer
+ AI()
+ findBestPlay(): Integer
+ findBestDiscard(): Integer
+ findBestHint(): Pair<Hint, Integer>
+ findBestMove(): HashMap<String, String>
- findHints(String[] observedCards, ConditionalProbabilityTable[] otherCPTs): LinkedList<Hint>
+ modelChanged(): void
+ devourStack(LinkedState<Map> eventStack): void
+ updateInternal(HashMap<String, String> Event, Integer Player, Integer Position): void
- receiveHint(Boolean[] cards, String card): void
- observeHint(Boolean[] cards, String card, Integer player): void
- receiveNewCard(Integer Position): void
- observeNewCard(String rank, String suit): void
- updateEVs(): void
+ initialObservations(): void
- observePlay(String rank, String suit): void


ConditionalProbabilityTable
+ cardLeft: Integer
+ sumOfSquares: float
+ CPS: Integer[] []
+ CPT: Double[] []
+ playEV: Double
- discardEV; Double
+ staleness: Double
+ hot: Double
+ indexToSuit(Integer index): String
+ caalSumOfSquares(): void
+ calcCPT(): void
+ expectedValueDiscard(Double[][] discardMatrix, Integer Tokens):void
+ expectedValuePlay(Double[][] validMatrix): void
- cardToIndex(String card): int
+ observeDraw(String rank, String suit): void
+ observeHint(String card): void
+ receiveHint(String card): void
+ toString(): String
- sumOfVals(): float
- CPSsum(): Integer


CPS
+ myCPS: int[][]
+ possibleCards: int
+ myCPT: double[][]
+ myGameType: String
+ myEV: double
+ receivePositiveHint(String hint): void
+ receiveNegativeHint(String hint): void
+ calculateCPT(): void
+ indexToSuit(Integer index): String
+ observeDrawCPS(Card card): void
+ clone(): CPS
- hintToIndex(String hint): Pair<String, Integer>
+ toString(): String


HanabiController
+ gameStateModel: GameStateModel
+ clientStateModel: ClientStateModel
+ socket: Socket
+ BufferedReader: BufferedReader
+ InputStream: DataInputStream
+ outputStream: PrintStream
+ event: Map
+ eventStack: LinkedList<Map>
- currentState: String
+ jsonObject: event
+ Controller(String newCurrentState)
+ setGameStateModel(GameStateModel gameStateModel): void
+ setClientStateModel(ClientStateModel clientStateModel): void
+ receiveServerData(): void
+ parseJSON(JSONObject jsonObject): void
+ sendMove(JSONObject jsonObject): void
+ packStream(Map move): void
+ evalEvent(Map event): void
+ initialModel(Map event): void
+ initialGame(Map event): void
+ handlePressed(MouseEvent mouseEvent): void
+ handleDrag(MouseEvent mouseEvent): void
+ handleRelease(MouseEvent mouseEvent): void
+ readMessage(): String
+ computeHash(String msg): String
+ playerJoined(): void
+ playerLeft(): void
+ yourMove(): void
+ gameCancelled(): void
+ gameStarts(): void
+ discardedNotice(int pos, Card drawnCard): void
+ acceptReply(Card discardedCard): void
+ playedNotice(int pos, Card drawnCard): void
+ builtReply(Card playedCard): void
+ burnReply(Card playedCard): void
+ informReceivingPlayer(boolean[] hintArray, String hint): void
+ informOtherPlayer(boolean[] hintArray, String hint, int playerPos)
HanabiView
+ height: double
+ width: double
+ gameStateModel: GameStateModel
+ clientStateModel: ClientStateModel
+ controller: Controller
+ canvas: Canvas
+ graphicsContext: GraphicsContext
+ View(double newHight, double newWidth)
+ setGameState(GameState gameState): void
+ setClientState(ClientState clientState): void
+ setController(Controller controller): void
+ setLayoutChildren(): void
+ modelChanged(): void
+ drawStartMenu(): StartMenu
+ drawCreateGame(): CreateGame
+ drawJoinGame(): JoinGame
+ drawGameTable(): void
StartMenu
+ StartMenu(double width, double height)
CreateGame
+ CreateGame(double width, double height)
JoinGame
+ JoinGame(double width, double height)
<<Interface>>
ButtonListener
+ selected(): boolean
+ hovering(): boolean
<<Interface>>
GameStateModelListener
+ modelChange(): void
Player
«get/set» + hands: ArrayList<Card>
«get/set» + isTurn: boolean
+ Player(ArrayList<Card> hands, boolean newTurn)
+ updateHand(int index, Card card): void
+ startTurn(): void
+ endTurn(): void
+ receiveHint(boolean[] hintArray, String type): void
+ observeDrawPlayer(Card card): void
+ clone(): Player


CPS
Table
GradientMatrix
Card
ConditionalProbabilityTable
ValidityMatrix
Hint
Player
AI
PlayDiscardBox
GameStateListener
GameState
HanabiController
SearchNode
Button
Main
State
JoinGame
CreateGame
ClientState
StartMenu
HanabiView
«create»
Powered by yFiles

3 JavaDocs

JavaDocs link on Git repository

4 Known Bugs, Incomplete Features, and Workarounds

4.1 Known Bugs

- **NaN Bug:** This has been a bit of a brutal bug that exists when calculating the probability tables for cards. In its current state, it exists only in one specific instance: when calculating the expected value for a hint. This is because the 'slave' AIs that the AI uses to calculate the EV of hints observe their own draws. This can be fixed fairly easily by re-writing parts of the observeDraw method in the AI. Unfortunately, writing this method can actually be reasonably complicated.

4.2 Incomplete Features

- **Tutorial:** The intention was to have a short walkthrough of the basic game rules, we didn't have enough time to implement this, but only because we ended up spending more time in other sections that we didn't know would take as long to get working. It would actually be fairly quick to set up now that we have the basics working.
- **Player Notes:** We weren't able to implement the ability to allow a player to keep some personal notes about the game (current deductions and whatnot). Unlike the tutorial, this would have been a lot of additional work.
- **Sound:** We weren't able to implement any sound with the software.
- **Menu Integration:** We weren't able to connect the main menu to a game screen.
- **Rainbow Cards:** We were unable to implement the rainbow cards into the AI. It can currently only play the game with the standard deck and rules.

4.3 Workarounds

- **NaN Bug:** This is simply resolved by setting the discard EV of any card that has a NaN value to 1.0. The impact of this is that some hints that may be strong won't be given by the AI occasionally.

- **Menu Integration:** This is currently solved by just launching the game from the command line with the appropriate arguments, the title screen is skipped entirely, and the player is put directly into a game.

5 Highlights

I think the key strength of our project was our architecture. We went through many, many revisions of the architecture during spring break, and as a result we developed a strong, simple architecture that nicely captured the goals of the project and made the implementation phase fairly smooth and straight forward.

Another strength of our project is the interactivity of the view. Our decision was to avoid creating a 'radio-button simulator' by allowing the user to drag and drop cards, and give hints by interacting with the cards of their team-mates directly, rather than having the user wade through a series of pop-up dialogs. This ties in nicely with the general strength of the view, not only from a strictly visual sense, but from an organizational sense. We aimed to imitate the act of actually sitting around a table and playing cards.

The AI is fairly strong for using almost entirely a statistical approach in how it plays. It has the ability to give reasonable, relevant hints to other players. It also plays slightly risky, taking calculated risks by treating the fuses as a resource, rather than just a counter which tells you how close you are to losing. As it is now, the AI averages 14.15 per game when playing games with 4 other AIs of the same type.