Uno Game Design Document

Authors:

Jhon Nunez

Christian Dominguez

Joseph Canning

Project Purpose/Information:

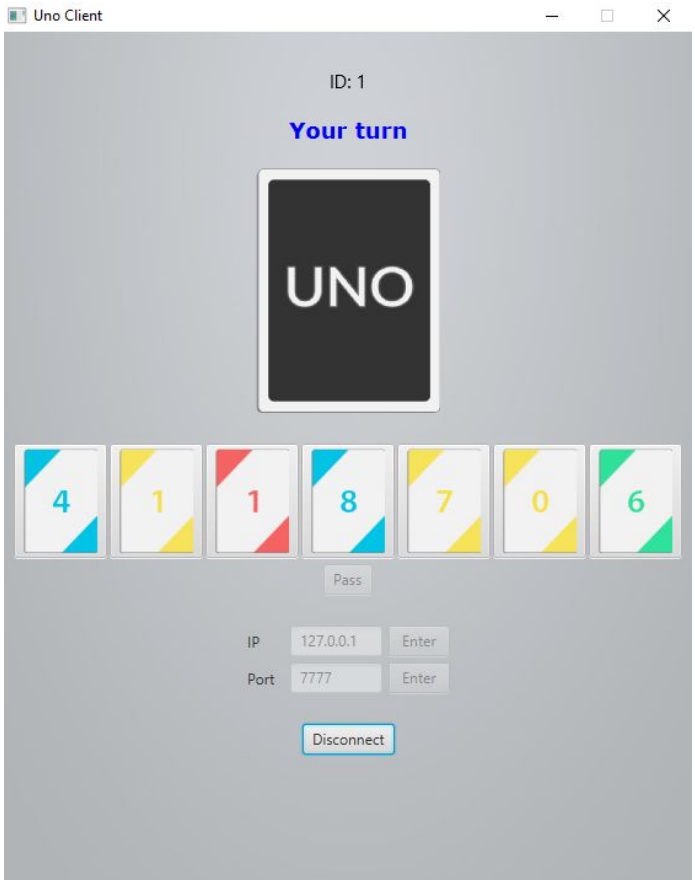
The purpose of our project was to create a version of Mattel's popular card game Uno playable as a desktop application. All files are written in Java, and the implementation is fairly simple with the game running in a small window at a fixed resolution. We provide two separate programs: one for the server and one for the client. The server program's purpose is to allow a number of players (running the client program) to connect to this server on a chosen port and play a game of Uno together. The server controls the game flow by tracking whose turn it is and player scores. The client program displays a UI to the player, showing them their hand, the card last played, server information, and game controls. Immediate game logic is handled in the client such as determining which cards are valid to be played. Programming separate client and server programs was done so because it was easy.

Uno Server

The server is composed of a javaFX UI and backend java code. The UI has the ability to enter a port number through a TextField. Upon port input, the user can press the “Create” and “Turn On” buttons to create a Server Socket that listens for clients in an infinite while loop. Our Server Socket will listen infinitely until the It is disconnected with our “Turn Off” button. The server also let’s the user know the amount of clients connected to the selected port number at the bottom. When a client connects or disconnects, this number will update.

On the backend of the server code, we create an InputListener object for every client that connects. This object has the ability to listen to messages sent by the Server. When 4 clients connect to the server, the server calls the startGame function that runs the game. The server receives and sends messages according to the game being played.The Server also uses an unoDeck object to deal the cards to each client. As the game begins, each client is dealt 7 cards. Once the cards have been dealt, the game continues until the Server receives a “win” or “tie” signal. If one of these signals is received, the game stops and a winner is declared.

Uno Client



The client is composed of a javaFX UI and backend java code. The UI has many components that let the user know about the game.

The first component is the ID. The ID is a TextField that lets the user know which client they are.

The second component is a TextField that displays messages of the game status to each client.

The third component is an ImageView which displays the cards being played by each client.

The fourth component are card buttons that allow the user to select which card they would like to play. Only the cards that they are allowed to play will be enabled through each round. If the user has no card to play, they can only select the “Pass” button.

The last component is an IP and Port TextField in which the user can enter the ip and port to connect to the server.

cardImages

This class is used to assign ID's and image paths to the card objects. It does so by creating a Hash Map with the ID as the key. This can be used to easily access a card's corresponding image and display them.

cardPic

This class is used to determine if a card in a player's hand has been played. It does so by using the boolean data member: "played". If that is the case, the card image will be displayed on the field and the button corresponding to that card will be disabled.

unoCard

This class is used to create card objects with its corresponding: id, card number, card color, and image name. These objects are used throughout the program in conjunction with other objects, such as cardPic and unoDeck.

unoDeck

As mentioned above, this class is used to store the unoCard objects in an ArrayList and it also has a Hash Map similar to the one in the cardImages class. This class initializes all the cards, stores them in the deck, and shuffles the deck at the beginning of the game. In addition, this class deals each player their hand at the beginning of the game.

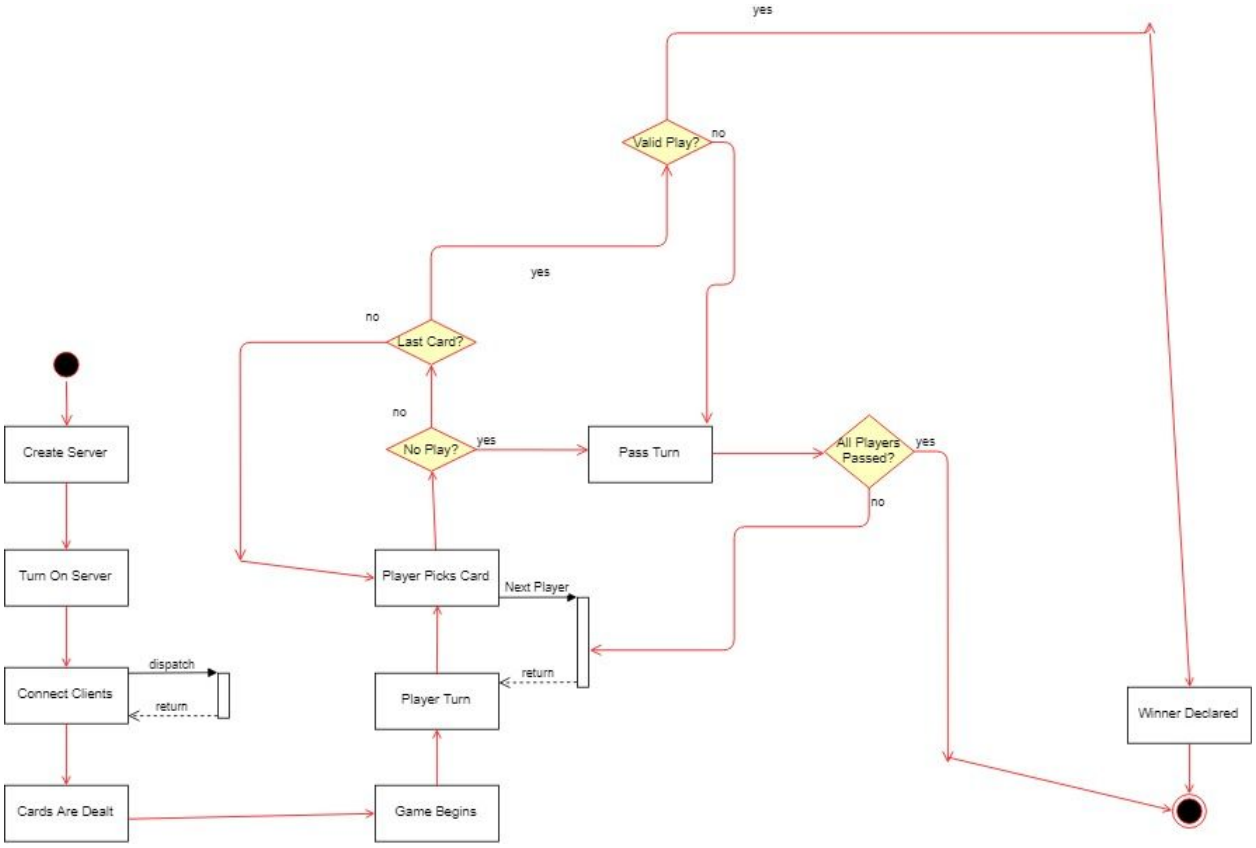
Images Package

This package is used to store all the images used throughout the project and its provides efficient access to them.

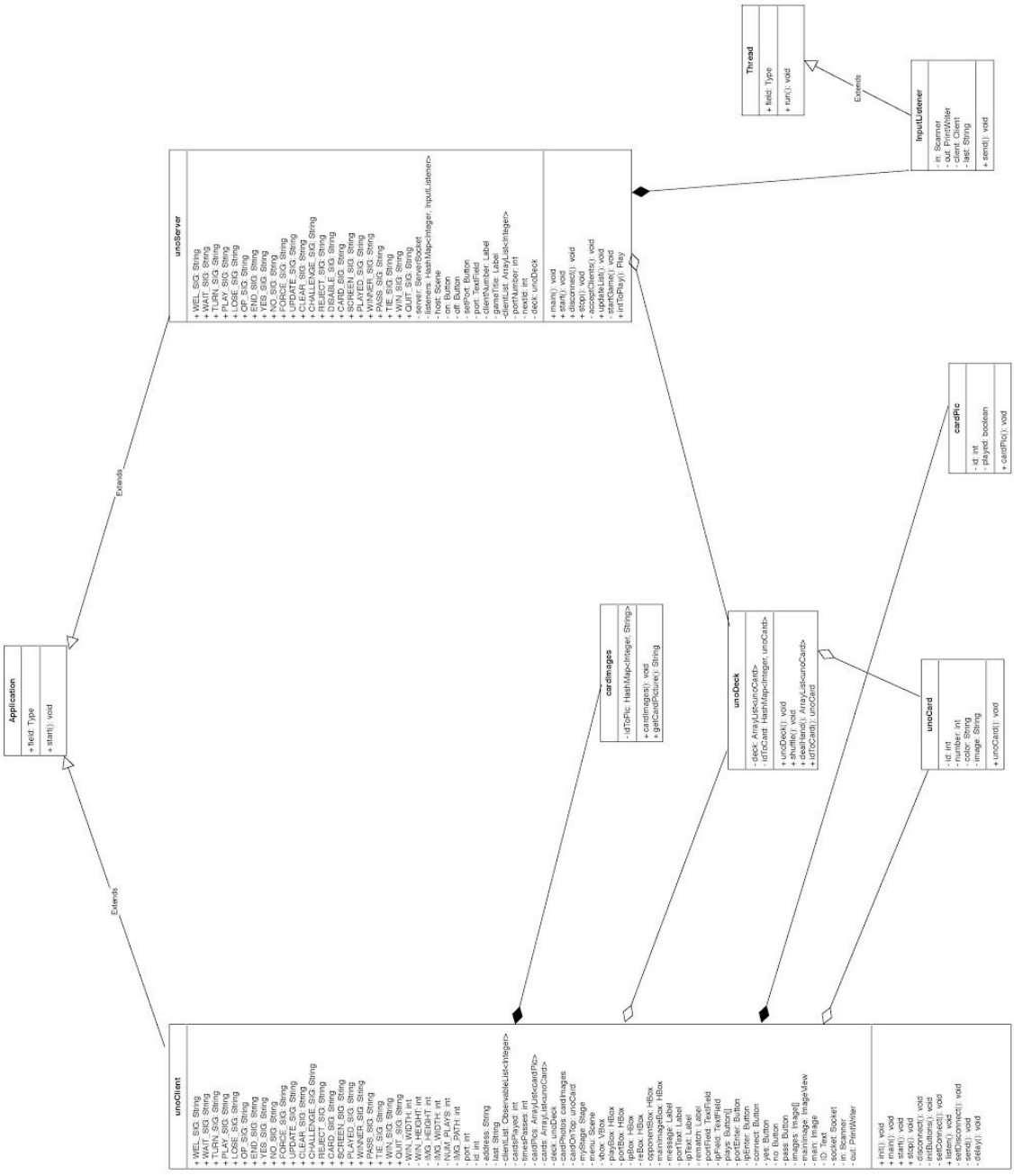
Benefits/Risks/Issues/Assumptions

- Our project cleverly uses predefined signals to communicate between the server and clients and are aptly named to understand their purpose. This is beneficial because it prevents confusion between several signals that need to be sent.
- One risks/issue that we had with this project was that it would occasionally not pass the turn to the next player after they made a move. The client's cards would not be enabled and their pass button would also not be enabled.
- This project assumes that the user is attempting to run it in an environment that supports Java and that the user knows how to play UNO at the most basic level.

Activity Diagram



UML


Application
+ field: Type
+ start(): void
Extends
Extends
unoClient
+ WEL_SIG: String
+ WAIT_SIG: String
+ TOO_SIG: String
+ PLAY_SIG: String
+ LOSE_SIG: String
+ OP_SIG: String
+ END_SIG: String
+ NO_SIG: String
+ FORCE_SIG: String
+ UPDATE_SIG: String
+ CLEAR_SIG: String
+ CHALLENGE_SIG: String
+ REJECT_SIG: String
+ CARD_SIG: String
+ DISABLE_SIG: String
+ SCREEN_SIG: String
+ PLAYED_SIG: String
+ WINNER_SIG: String
+ PASS_SIG: String
+ TIE_SIG: String
+ WIN_SIG: String
+ QUIT_SIG: String
- WIN_WIDTH: int
- WIN_HEIGHT: int
- IMG_HEIGHT: int
- IMG_WIDTH: int
- IMG_PATH: int
- NUM_PLAYS: int
- in: Scanner
- id: int
- address: String
- last: String
- clientList: ObservableList<Integer>
- cardsPlayed: int
- timesPasses: int
- cardPics: ArrayList<cardPic>
- cards: ArrayList<unoCard>
- mydeck: unoDeck
- cardPhotos: cardImages
- cardOnTop: unoCard
- myStage: Stage
- mScene: Scene
- vbox: VBox
- playBox: HBox
- portBox: HBox
- idBox: HBox
- ipBox: HBox
- opponentBox: HBox
- mainImageBox: HBox
- messageLabel: Label
- portText: Label
- ipText: Label
- rematch: Label
- portField: TextField
- ipField: TextField
- plays: Button[]
- portEnter: Button
- ipEnter: Button
- connect: Button
- yes: Button
- no: Button
- pass: Button
- images: Image[][]
- mainImage: ImageView
- main: Image
- ID: Text
- socket: Socket
- in: Scanner
- out: PrintWriter
+ init(): void
+ main(): void
+ start(): void
+ stop(): void
+ connect(): void
+ initButtons(): void
- setConnect(): void
+ listen(): void
- connect(): void
- send(): void
- delay(): void
Extends
unoServer
+ WEL_SIG: String
+ WAIT_SIG: String
+ TOO_SIG: String
+ PLAY_SIG: String
+ LOSE_SIG: String
+ OP_SIG: String
+ END_SIG: String
+ NO_SIG: String
+ FORCE_SIG: String
+ UPDATE_SIG: String
+ CLEAR_SIG: String
+ CHALLENGE_SIG: String
+ REJECT_SIG: String
+ DISABLE_SIG: String
+ CARD_SIG: String
+ SCREEN_SIG: String
+ PLAYED_SIG: String
+ WINNER_SIG: String
+ PASS_SIG: String
+ TIE_SIG: String
+ WIN_SIG: String
+ QUIT_SIG: String
- server: ServerSocket
- listeners: HashMap<Integer, InputListener>
- host: Scene
- on: Button
- off: Button
- enable: Button
- port: TextField
- clientNumber: Label
- gameTitle: Label
- playerList: ObservableList<Integer>
- portNumber: int
- next: int
- deck: unoDeck
+ main(): void
+ start(): void
+ disconnect(): void
+ stop(): void
+ sendLists(): void
+ updateList(): void
+ startGame(): void
+ initToPlay(): Play
Thread
+ field: Type
+ run(): void
Extends
InputListener
- in: Scanner
- out: PrintWriter
- client: Socket
- last: String
+ send(): void
cardImages
- idToPic: HashMap<Integer, String>
+ cardImages(): void
+ getCardPicture(): String
unoDeck
- deck: ArrayList<unoCard>
- idToCard: HashMap<Integer, unoCard>
+ unoDeck(): void
+ shuffle(): void
+ dealHand(): ArrayList<unoCard>
+ idToCard(): unoCard
cardPic
- id: int
- played: boolean
+ cardPic(): void
unoCard
- id: int
- number: int
- color: String
- image: String
+ unoCard(): void