Matt McGauley, Bea Casey, Jack Otto, Sami Wurm

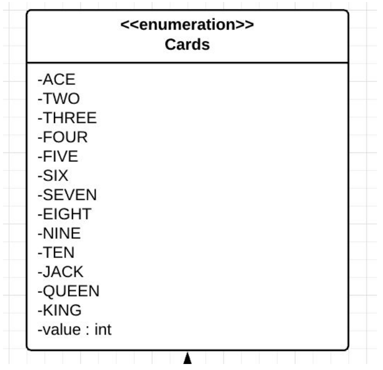
CSCI 205 - King

BlackJack with AI Design Manual

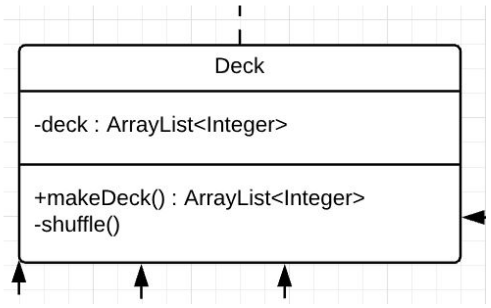
First we started by making a consul version of blackjack allowing us to have a starting point for our project and use the functions that were written for the consul version in the GUI version. Our GUI includes multiple scenes for better functionality. First being a landing page to start at and allow the user to choose what they want to do with our project. Choosing between the Blackjack game or a graph scene that allows the user to determine whether they want to simulate a game with a random player or with a smart player. This third scene was the bulk of our project, and in order to simulate each type of game we had to first build a random player for the random game which played the game choosing hit and stand without consequence, this player is given a set number of games to simulate from a user and then every time the game is played the outcome of the game is recorded within an array in order to track the success of the random player. And in the final scene it is displayed on a graphing chart how the random player did, by graphing its success rate throughout the set number of games. In order to do this same type of graph with a smart player, we found a neural net with random weight assigned to the neurons for importance. From there, we attached the neural net to our game having it learn how to play blackjack over the set number of games specified by the user. The smart player then does the same thing as the random in the sense that it will add each game's result into an array and return the array allowing another method to determine the success rate. Finally once the smart player is graphed its success rate over time will be displayed on the graph just like the random player's was.

We have completed 9 user stories, a.k.a. requests from the user for how the game should be working and how they want the game to work. These include: "We want to make the game generic so that we can use multiple solving algorithms and still be able to play the game", "We want to be able to use a neural net AI that can learn how to play blackjack so that we can show how a more advanced player could succeed but also show the limitations of the game", "We want to be able to graph the success rate for a specific number of hands of the game so that we can display it in a GUI", "We want to be able to play the game in an interactive GUI", "We want to be able to make a bot that will make a completely random moves to show how a beginner might play", "We want to be able to track the success of the games", "We want to be able to simulate betting with the dealer so that there is a real life aspect to it", "We want to be able to shuffle the deck and hit/stand", and "We want to also be able to make bets/play with money and teach the AI to play in a way that maximizes winnings". While we have completed the main idea behind all of these user stories, the only one that we failed to complete in its entirety is making the AI play in a way to maximize the earnings of the game, for now it just plays to win the game. While this does mean we did not attach the AI to the money had we been given more time, then just by making the AI place an arbitrary bet at the start of each round, the goal of the AI would remain the same, because the neural net was designed to make the AI win and by winning the AI is making the most money. Other than this one failure to complete a client request we have completed the rest of their desires and have a finished project that can both be played by the client along with show the client that it is better to have some sort of strategy when playing blackjack. Shown by the fact that the random player, who was created to simulate the playing of a new comer with no clue how to play the game, does significantly worse than the smart player, who was given instructions to attempt to win and does a fairly good job at it, by winning around 50-55% of the time compared to the random player who sits around 30-40%. While this game does only play with one deck, which is when the odds of victory for the player is the highest against the house a.k.a. the dealer, the game still shows how they neural net holds up against a dealer with strict rules on how to play the game.

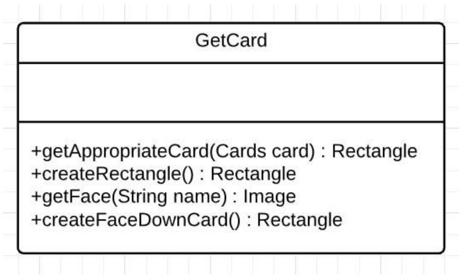
Our game is split into 5 separate packages that include many classes within each, we will start with the most import, the Blackjack package which is what the entire project is built off of. The most important piece is our card enumeration,



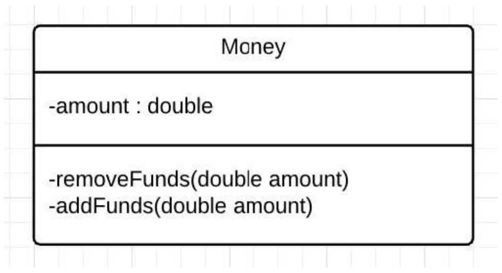
which holds each form of the card and assigned to each is the value of the card, with an ace starting with a value of an 11. However within both the player and dealer classes, there is a method in each called aceSwitch() that goes through the hand and if the player or dealer has busted (gone over 21) then this method is called and it changes the value of the ace from 11 to 1, allowing each to continue playing without having lost the game. The other most important piece to playing blackjack is the actual deck of cards being used to play the game. We created a class shown here,



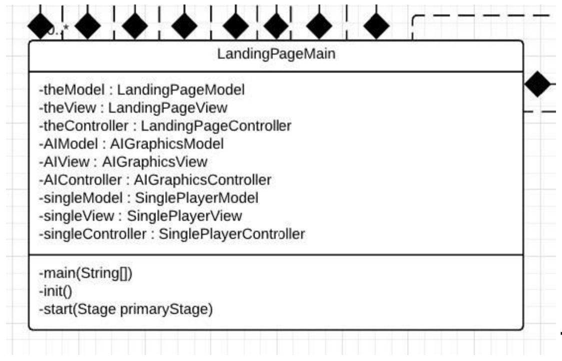
that will build a deck of cards in numerical order, and the most important function in this class is the ability to shuffle the deck, because this means that the deck does not need to be rebuilt for every game, but instead can just be shuffled. Outside of the Blackjack package we made a package DeckOfCards that adds the pictures to each card in the deck of cards.

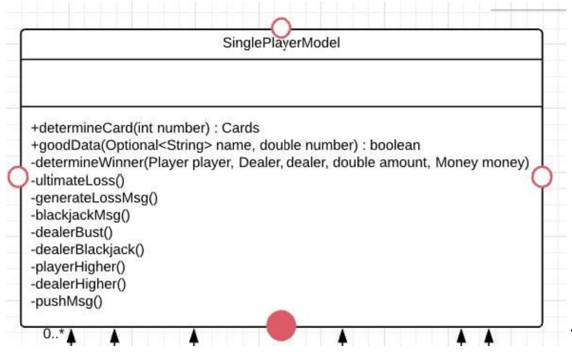

GetCard
+getAppropriateCard(Cards card) : Rectangle
+createRectangle() : Rectangle
+getFace(String name) : Image
+createFaceDownCard() : Rectangle

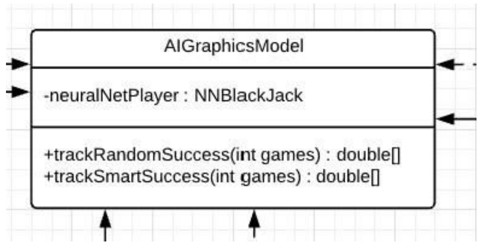
The functions within this class will get the right number to use for the card and then will build a rectangle that is filled with an image of the card being used. This functionality is what makes the game playable for the user, by showing them what cards are in their hand at the time so they know how to play their hand and whether they should hit or stand. Lastly at the end of our project, the user asked for the ability to make bets during the game to in a way track their own success while playing the game. In order to do this, we created another class within the blackjack package that would allow for the user to place a bet and get returns based on the outcome of the singular game that was played.

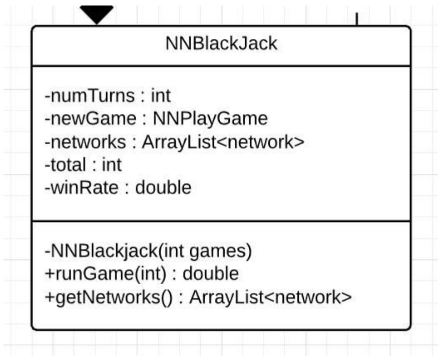


This class makes it easy for the game to take the bet of the user by removing the bet initially from their total and then later returning it with winnings after each game should the user have won, otherwise keeping the money if they lost. For the GUI we had three separate model, view, controllers, a.k.a. MVC's, one for each scene of the game. However the most important piece of the GUI is the main function that builds the game for the user to play.


0..*
LandingPageMain
-theModel : LandingPageModel
-theView : LandingPageView
-theController : LandingPageController
-AIModel : AIGraphicsModel
-AIView : AIGraphicsView
-AIController : AIGraphicsController
-singleModel : SinglePlayerModel
-singleView : SinglePlayerView
-singleController : SinglePlayerController
-main(String[])
-init()
-start(Stage primaryStage)
This class takes in all three parts of each MVC so that the computer can combine them all into one singular game and then run the whole program with the start function that sets the stage to be the opening scene allowing the user to pick what they want to do. For the single player portion of the game, the view makes all the nodes for where each piece of the game will go when it is being played, the controller reacts to the actions of the user by completing the next task of the game when the user initiates it by clicking on certain user friendly areas. However the model of the single player game is the most important part,


SinglePlayerModel
+determineCard(int number) : Cards
+goodData(Optional<String> name, double number) : boolean
-determineWinner(Player player, Dealer, dealer, double amount, Money money)
-ultimateLoss()
-generateLossMsg()
-blackjackMsg()
-dealerBust()
-dealerBlackjack()
-playerHigher()
-dealerHigher()
-pushMsg()
0..*
this class makes the hands for both the player and the dealer, takes in the amount bet by the user, and most importantly generates all of the end of game messages by using the determineWinner() function in order to find who won that individual hand and from there displaying the correct alert on the screen. For the other half of the game, the most important part of the AI side is also the model, for the same reasons as the single player, the view builds the physical game and the controller reacts to the touches, but the model does the grunt work.


AIGraphicsModel
-neuralNetPlayer : NNBlackJack
+trackRandomSuccess(int games) : double[]
+trackSmartSuccess(int games) : double[]
The model builds a neural net and also builds a random player when necessary depending on which chart the user wants to see. However the most important functions are the trackSuccess() ones for each type of

player, this runs each type of player with a set number of games and returns an array of their success against the house that can be graphed by the views generateSuccessChart() function that will display the array as it was played from start to finish. For a more in depth look at how we tied the neural net into our game, we created another package called NeuralNetPlay that takes the neural net we got from github and attaches its decisions to the hit or stand option a player had when they are playing blackjack. 

This neural net uses the blackjack consul version we made and runs a singular game, once that game has been completed with each decision of hitting or standing being made by the neural net the outcome is added to the array and then another game is played until the total number of games played is equal to the number specified by the user. This was the breakdown of our object oriented game of blackjack, that included 5 packages to help make a simple game a blackjack to fulfil the requests of our client.