BlackJack Game

Lucy Van Kleumen
Lucy.VanKleunen@colorado.edu

Bryce Ikeda
Bryce.Ikeda@colorado.edu

Aman Satya
aman.satya@colorado.edu

Final State of the Project

For our project, we implemented a BlackJack game in Java with a GUI and a database connection. In Project 4, we proposed 9 use cases for our BlackJack game: (1) user logs into their account, (2) user can access stats about their previous games, (3) user can access a tutorial about how to play BlackJack, (4) user can set game play settings - number of players to play against, skill level, and starting number of chips, (5) user can play a game of blackjack and take actions during game play, (6) user can access a cheatsheet for best move during gameplay, (7) user can toggle a display to see the current card count, (8) user can leave the game, and (9) user can see how well they just played the game when they leave. We have implemented all of these use cases except for (2), due to some challenges with implementing a database request for user statistics.

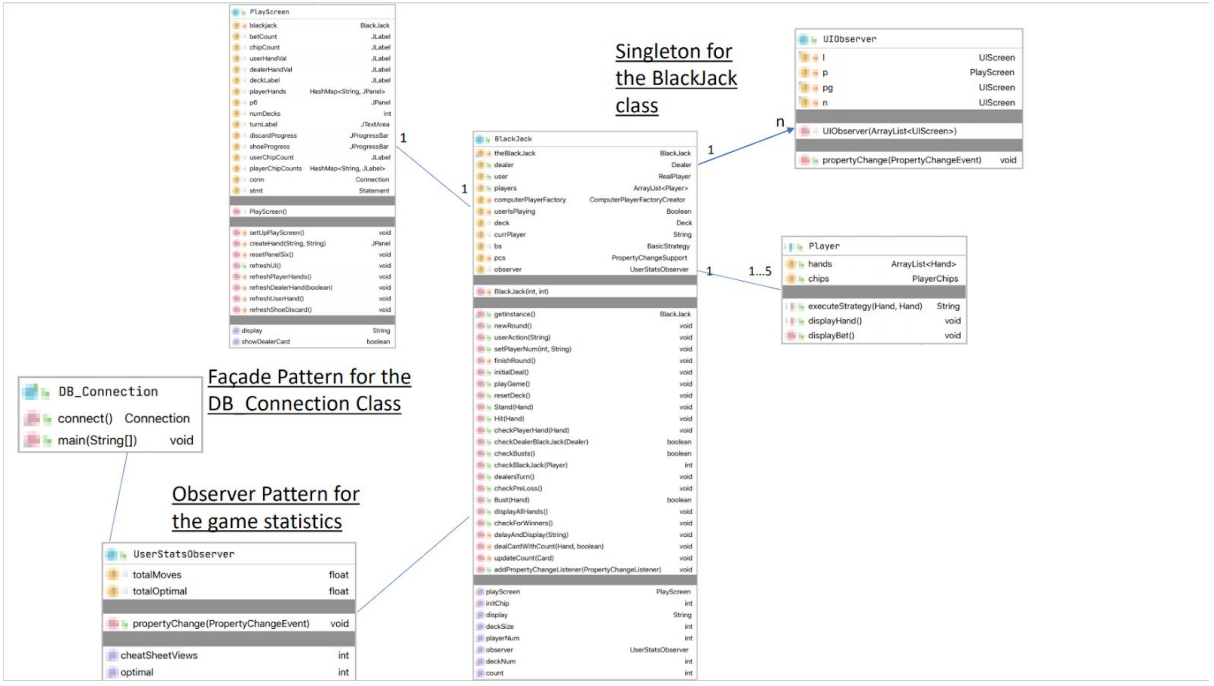
Since the Project 5 check in, we primarily worked on hooking up gameplay logic with the UI and adding additional features. In addition to completing these use cases, we have created a GUI that the user can play with which offers a more intuitive experience than playing a game through a non-graphical user interface. To play, simply Log in and start a New Game with the desired settings. Then you can play a round. The Computer players automatically play, then you can play (cheat if you need to!), then the dealer plays. You can see if you win and your updated chip count and then click “New Round” to play again.

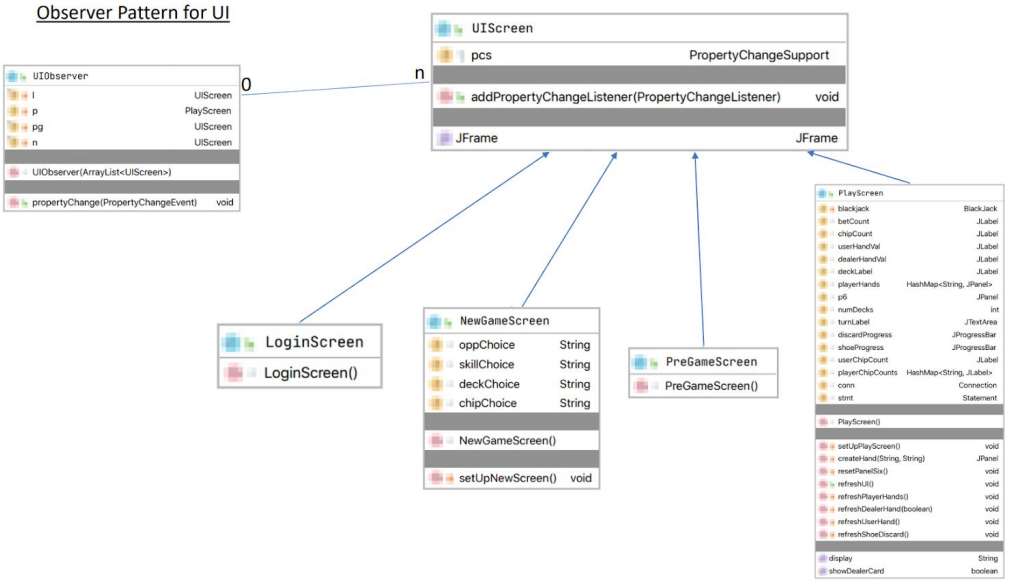
The deck automatically refreshes when low on cards so play as long as you’d like. Then you can “Leave” the table and see how many chips you won or lost, how often you made the optimal move, and how often you checked the cheatsheet.

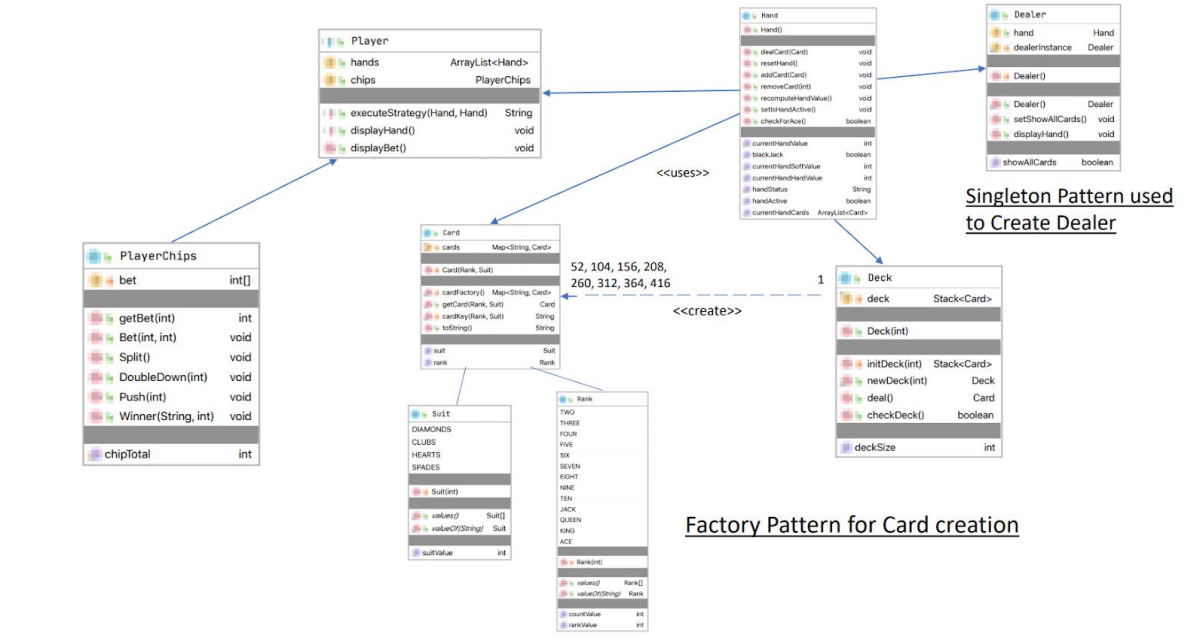
The code on Github is set up with the database connection (for saving log in information) commented out so you can play without it and we have provided information on how to set up the database.

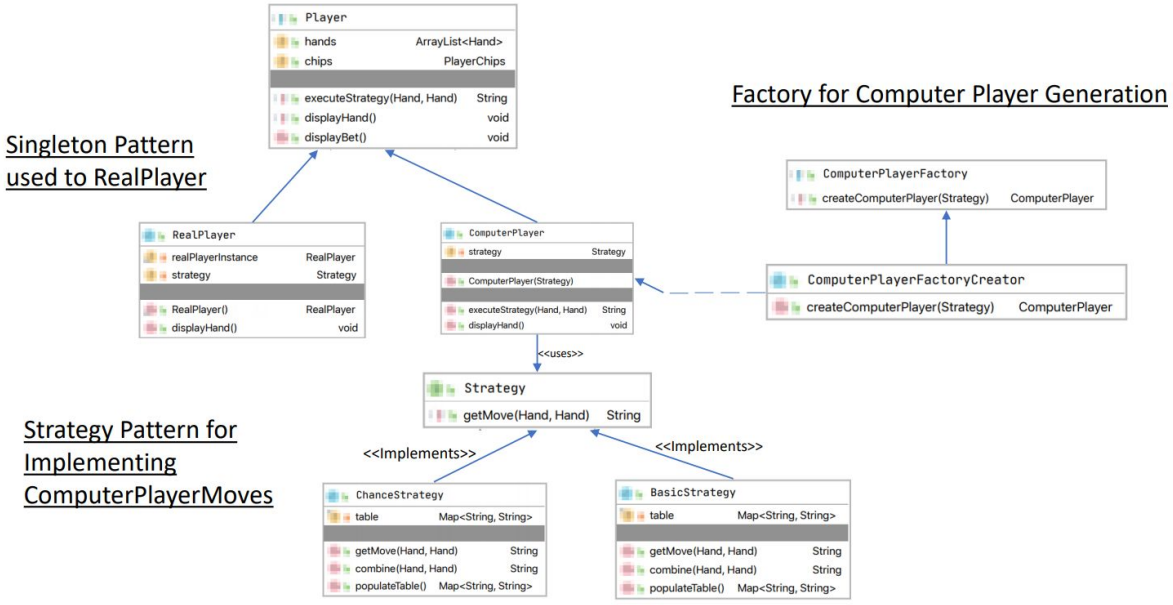
Class Diagrams & Comparison

This is what our UML diagram looks like now:









Design patterns that we used

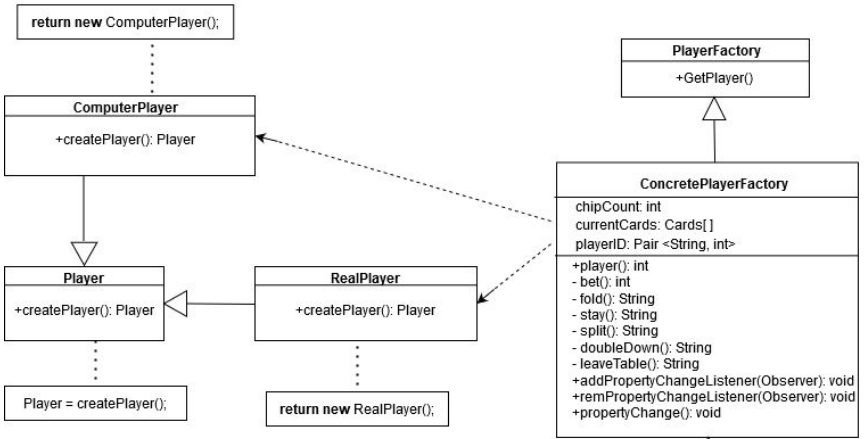
- We used an Observer pattern in two places. We use an Observer to keep track of when we need to switch between UI screens and take care of that switch. We also use an Observer to keep track of user game play statistics (cheatsheet checks, etc.)
- We used a Factory pattern for creating the computer players
- We used a Strategy pattern for computer player automatic gameplay strategies. The basic strategy uses a BlackJack table with the optimal moves given a player's hand and the dealer's up card. The chance strategy takes the basic strategy but adds the

component of when the user's hand is less than 17, and if their move is mapped to be a stand, they have a 50% chance of hitting instead.

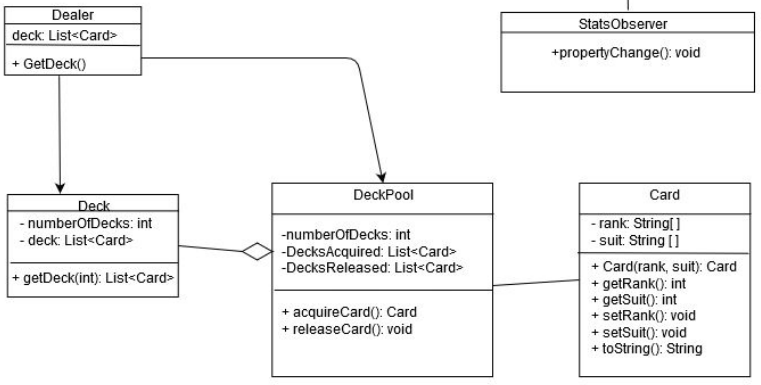
- We used a Singleton design pattern to ensure that we did not mistakenly instantiate more than one logical BlackJack game play object when referencing this object from different UI files
- We also used a Singleton pattern for the Dealer and the User because we only need to instantiate one of those in our game
- We use ideas taken from Model View Controller to split up our logical code from our view code. So, we use the BlackJack class for example as our "model" and then the PlayScreen class as the "view". This decouples the logic from the view and the view simply periodically refreshes based on the game play state. We don't include an explicit controller class, but the PlayScreen itself sort of acts as a controller.
- We used a factory pattern to build the cards and to put them into a deck
- We sort of use a Facade pattern for the database access, by creating a standalone class DB_Connection that handles making database requests - this class encapsulates the complexities of database requests so that simpler calls can be made in other files. The DB_Connection helps in connecting with databases by using sqlite jdbc drivers.

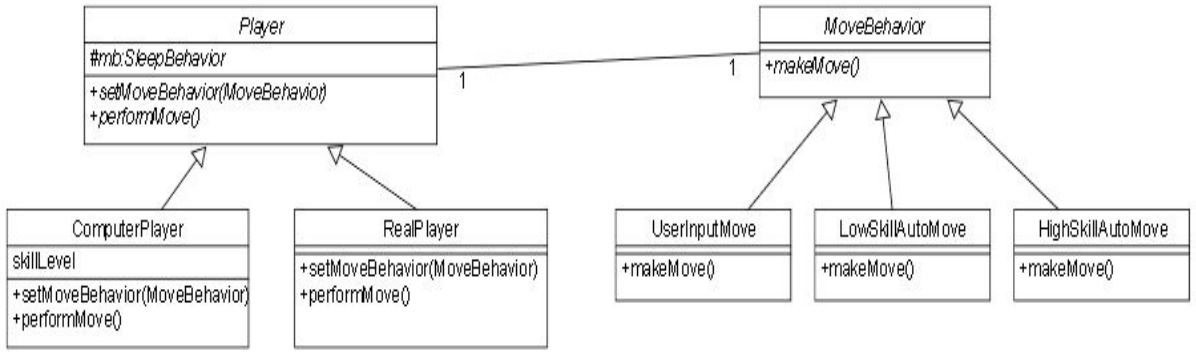
This is what our UML diagrams looked like in the Project 4 proposal

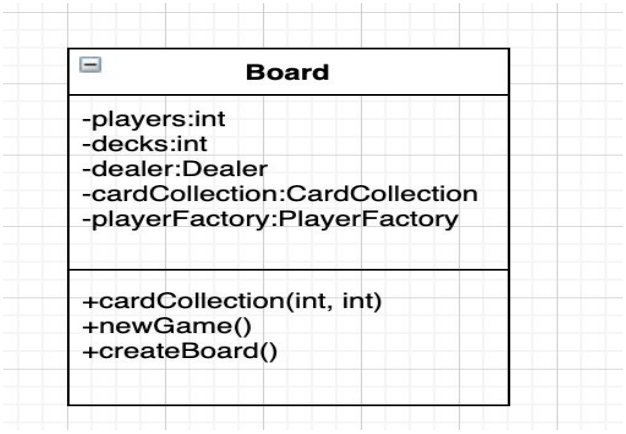
Player Creation Factory Pattern UML

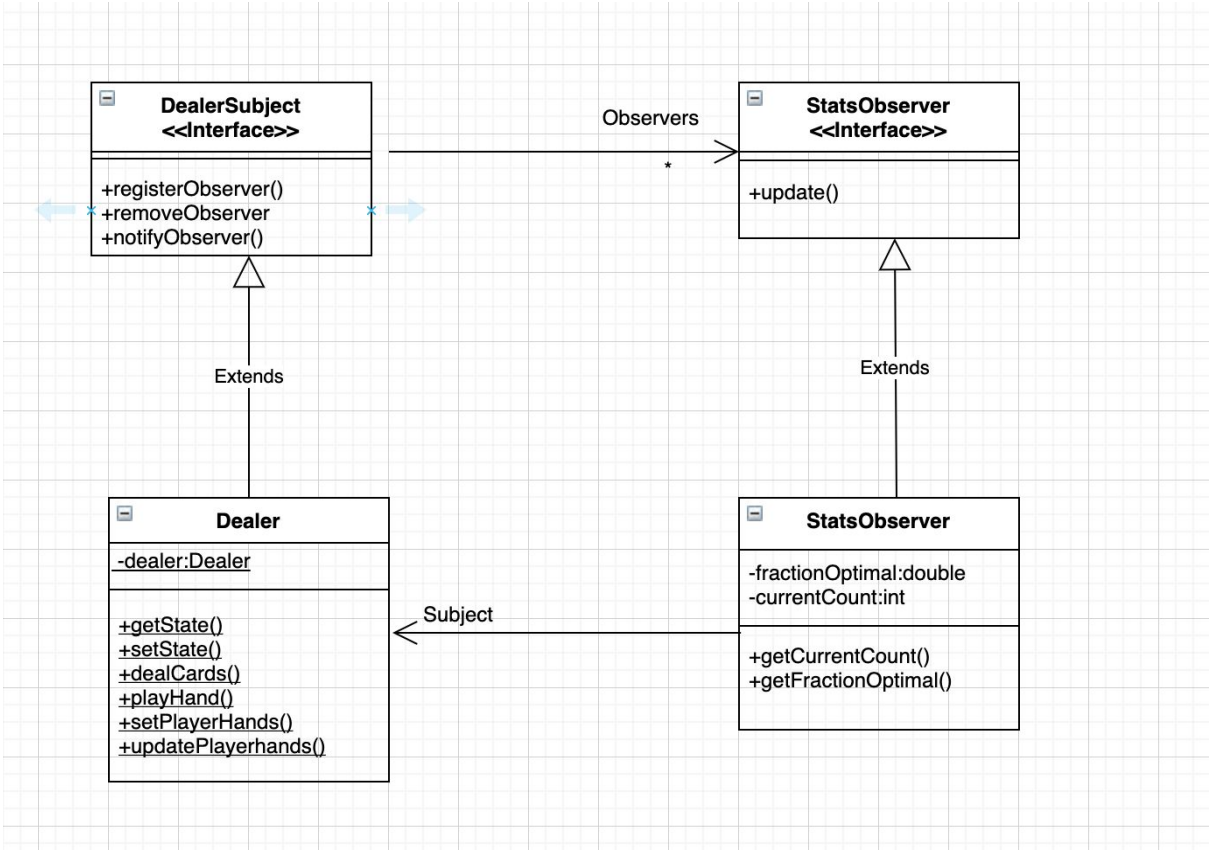


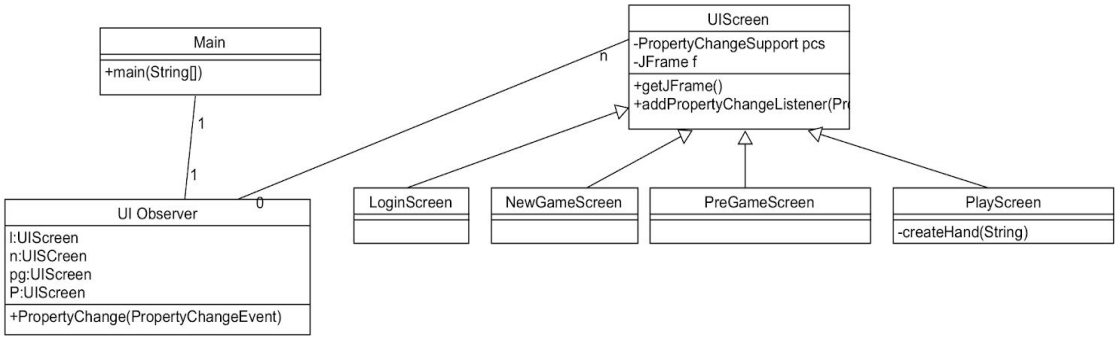
Card Deck Object Pool UML




Player
#mb:SleepBehavior
+setMoveBehavior(MoveBehavior)
+performMove()
1
1
MoveBehavior
+makeMove()
ComputerPlayer
skillLevel
+setMoveBehavior(MoveBehavior)
+performMove()
RealPlayer
+setMoveBehavior(MoveBehavior)
+performMove()
UserInputMove
+makeMove()
LowSkillAutoMove
+makeMove()
HighSkillAutoMove
+makeMove()


Board
-players:int
-decks:int
-dealer:Dealer
-cardCollection:CardCollection
-playerFactory:PlayerFactory
+cardCollection(int, int)
+newGame()
+createBoard()


DealerSubject
<<Interface>>
+registerObserver()
+removeObserver
+notifyObserver()
Observers
*
StatsObserver
<<Interface>>
+update()
Extends
Extends
Dealer
-dealer:Dealer
+getState()
+setState()
+dealCards()
+playHand()
+setPlayerHands()
+updatePlayerhands()
Subject
StatsObserver
-fractionOptimal:double
-currentCount:int
+getCurrentCount()
+getFractionOptimal()



What changed from Project 4 to the final implementation?

There are a few key changes to our system since Project 4. One key change is Card Deck. Rather than building an Object Pool for our card deck, we use a factory pattern to build the deck. Another change we made is with the observer. Rather than having an observer wait for input from individual player classes such as the dealer, we have it connected to the gameplay itself. Then, we check if an optimal move occurs during gameplay and send that data to the stats observer. Another area we changed was with the UI. We found that using a UI observer between the game logic and the UI was another level of abstraction that made the code more complex. Rather than doing that, we trigger the UI based off of BlackJack logical events. Another piece we changed was having a board object. Instead, we had a statistics observer that observed the Blackjack object for moves to update its database.

Third- Party code vs Original Code Statement

We used the https://www.youtube.com/watch?v=5QyU35ct6M0 tutorial as a guideline for how to set up a deck of cards and how to access it. The rest of the code for this project is original.. Our work was accomplished via the use of standard Java libraries: we used the java.beans.PropertyChangeEvent and java.beans.PropertyChangeListener to implement the Observer pattern, we used javax.swing components for setting up the UI, we used java.sql.Connection for connecting to the database.

For setting up the database and connecting the database with our application required us to understand the sqlite database and how it can be connected to the database. To better understand we looked up for the couple of tutorials online. One was to understand the setting up of databases in Intellij and another was to understand connecting the database with jdbc: sqlite driver. Below is the tutorial that we referred:
https://www.jetbrains.com/help/idea/connecting-to-a-database.html
https://www.sqlitetutorial.net/sqlite-java/sqlite-jdbc-driver/

Statement on OOAD process for semester project

1. Key Issue 1 - Database connection
    a. The database connection required JDBC: sqlite driver. The platform where we executed our application is Intellij. Intellij has the built in sqlite plugin which helps in creating a local database, called " sqlite-master" . The issue that I faced was when I connected the database with my code, it was not recognizing the database. The error was " No database found". After doing extensive research we found out that the "sqlite-JDBC-3.7.2.jar" was missing. The location of the jar file was incorrect, after putting the jar file inside the project environment, we were able to access the database and it's tables.

2. Key Issue 2 - UI / Logic connection
    a. Another challenge was in hooking up the blackjack playing logic with the UI, particularly in figuring out how to best wait for and respond to user actions. This took a lot of communication and working together to get right. There was one user action we thought about implementing (letting a player split their hand) that was reasonably okay to set up logically but introduced a lot of complexity into the UI so we ended up not implementing for the final product. Decoupling the View (UI display) from the actual blackjack gameplay logic helped in this process because once we had the logic correct, then we just had to make sure the UI was refreshed at the right times, we didn't really make any changes to the gameplay state from the view class except for receiving user input and passing it to the model.
3. Key Issue 3 - Handling edge cases
    a. Another challenge was handling edge cases. Because we allow for so much flexibility in what the BlackJack board state can be, there are many possible configurations for user / player / dealer hands, chip counts, etc. We handled this by constantly testing during implementation, both user testing (actually playing with the application) and via unit testing