CSCI 5448

OBJECT ORIENTED ANALYSIS AND DESIGN SEMESTER PROJECT REPORT

NAME OF THE PROJECT:

Blackjack simulation with GUI.

TEAM MEMBERS:

- Shruthi Sridharan - shsr7296@colorado.edu
- Sanika Dongre - sado9543@colorado.edu
- Preethi Vijai Lilly - prvi9376@colorado.edu

FINAL STATE OF THE SYSTEM:

Our final system has a Graphical User Interface which will display the option to Play, Exit or Help for the game BlackJack. Once the play button is clicked, the next page is displayed, which has the option to place a bet using the Chip Button. The bet can be 25$ to 100$. Then two cards for the player is displayed. Dealer gets his own set of cards. The player needs to decide whether he wants to Hit, Stand or Double the bet. Based on the player's action, the scores are calculated. We also have a SQL connection in which the values are inserted to the database and retrieved once the game is over.

Language:

Java 1.8

Tools:

- ➢ SQL Workbench
- ➢ Eclipse
- ➢ Xampp to start the server

Features / Use cases Implemented:

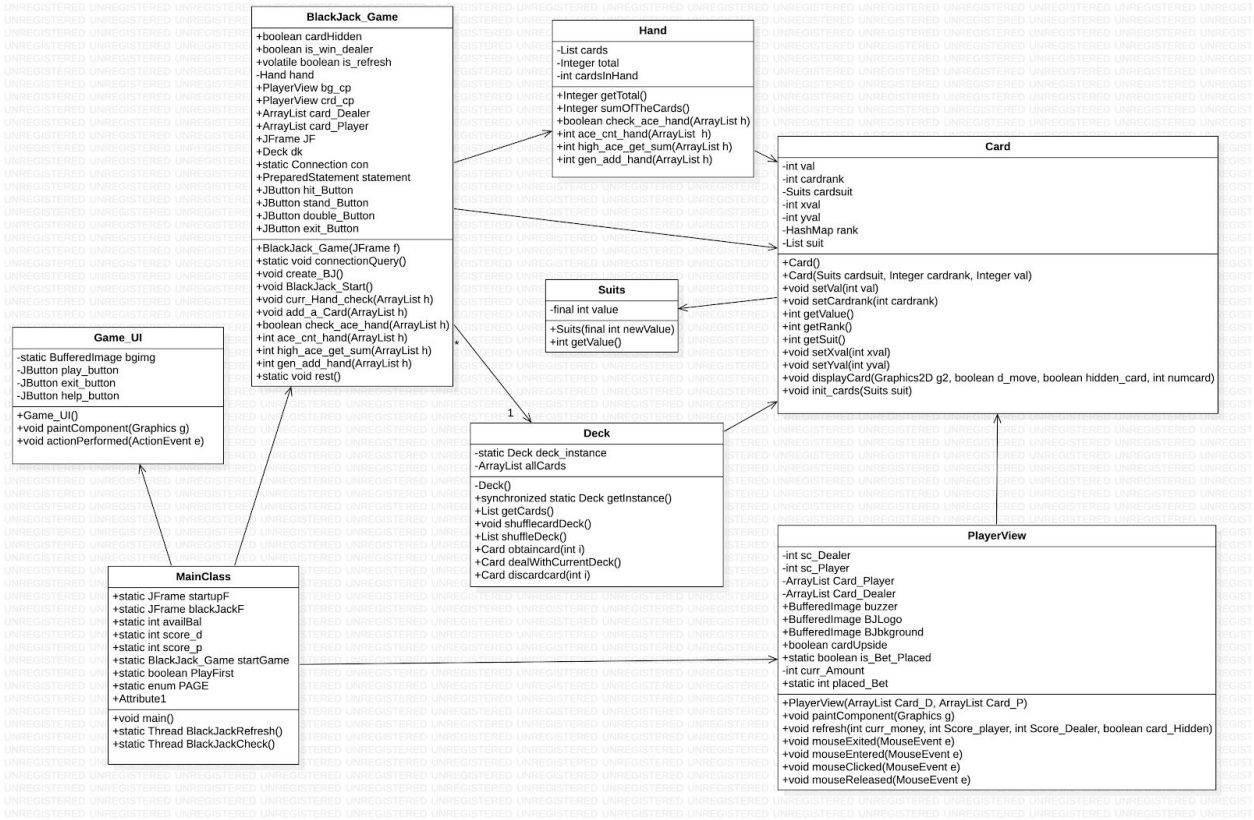
- ➢ Settling the shuffled deck of cards.
- ➢ Tracking the number of games won by the player and the dealer.
- ➢ Tracking the current balance amount of the player.
- ➢ Settling the winner based on whether the opponent is busted or the player got a blackjack or the player has a better hand than the opponent.
- ➢ Checking the Soft -17 rule i.e., containing one Ace and some other cards that adds up to six wherein Ace will be considered with a value of '11'.
- ➢ Graphical User Interface to ease the access of the project.
- ➢ SQL database to store the winner and the reason to retrieve it later.
- ➢ JUnit test-cases to test the sub-tasks.

Features Unimplemented:

Multiplayer implementation of the game - This feature is not implemented because the GUI implementation took more time than we anticipated.

FINAL CLASS DIAGRAM AND COMPARISON STATEMENT:

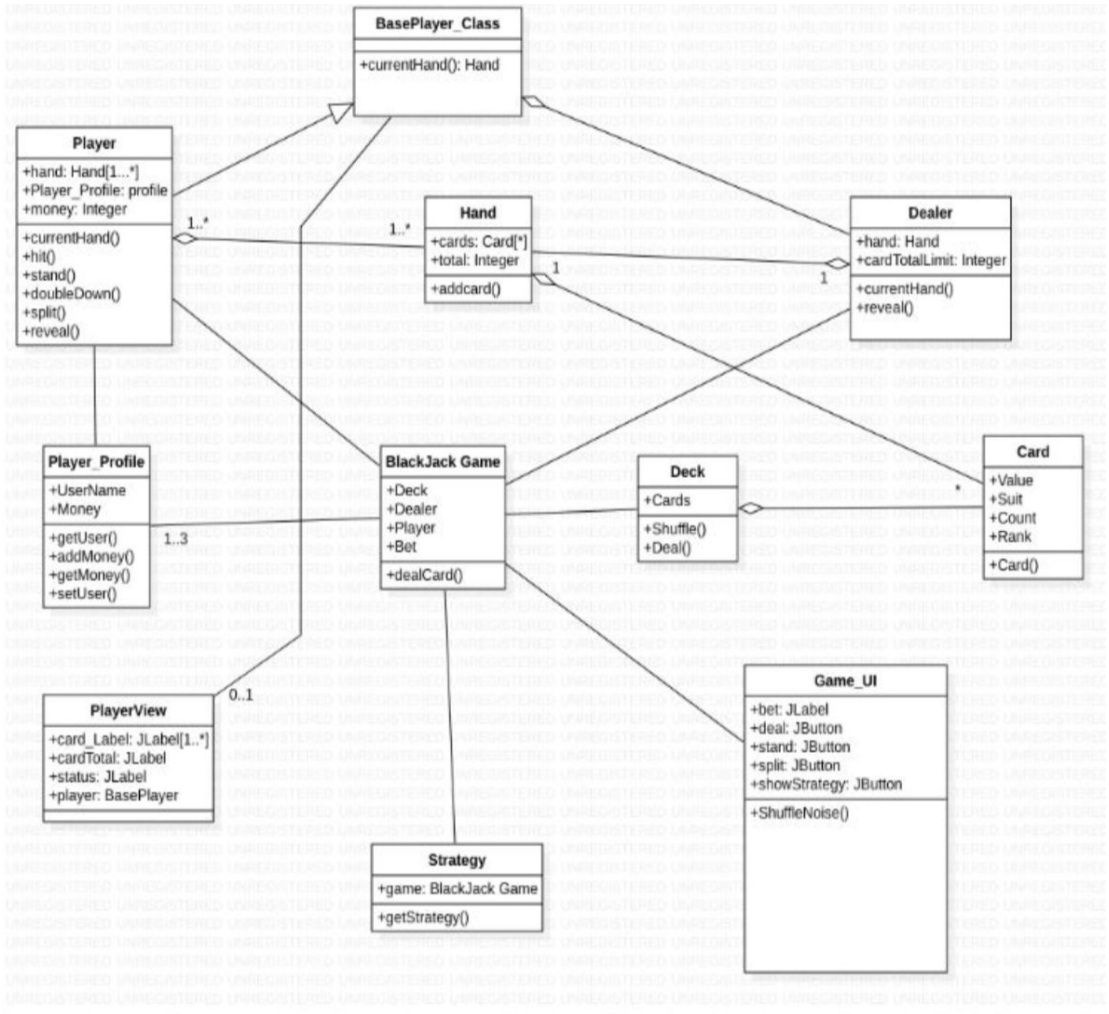
The final class diagram of our project looks like,



Patterns included:

- ➢ Iterator Pattern : Iterator pattern is used to iterate through the ArrayList card objects
- ➢ Singleton Pattern: Singleton pattern is used to create a single instance for deck so that the deck variables are commonly used by the dealer and player.
- ➢ Facade Pattern: java.sql.connections implemented for JDBC connections to the database is an example for facade pattern, we as users or clients create connection using the "java.sql.Connection" interface, the implementation of which we are not concerned about.

Project 4 Class Diagram:



Changes between Project 4 and Project 6:

- New Classes to implement JDBC connections to store values in the database.
- Implemented single player rather than multiplayer, so that we could refine use cases and corner cases for a single player implementation.
- Since single player is implemented rather than multiplayer, there is no inheritance hierarchy in project 6.

Key Changes in our project:

- Used Singleton pattern for Deck Class so that the object is commonly used by the dealer and the player.
- Iterator pattern is used in ArrayList and List type objects to remove the redundancies.
- Transitioned from a multi player model to single player model.

THIRD PARTY VS ORIGINAL CODE:

We created all the classes and its functions on our own. We implemented the conditions to check and decide the winner based on the properties of the blackjack game. Feature to maintain the scores and current balance amount of the player is also implemented. Writing a logger to export all print statements to a text file was done on our own. However, some third party code used in this project are as follows:

➢ We used the link https://www.tutorialspoint.com/jdbc/jdbc-insert-records.htm and stackoverflow for connecting jdbc to sql and creating the database.

➢ For implementing the GUI we took the reference from the Github repository: https://github.com/uzaymacar/blackjack-with-gui

OVERALL PROCESS FLOW OF THE SEMESTER PROJECT:

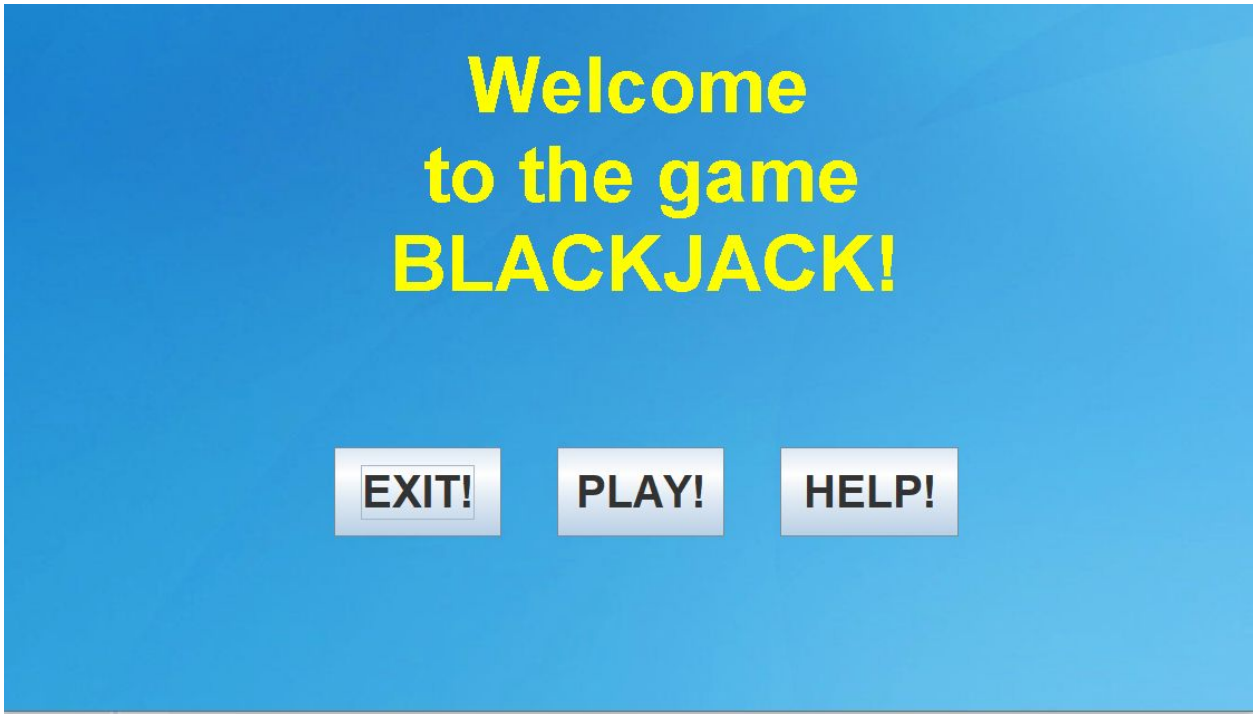
➢ Planning: We planned the project prior hand and started early so that we finish it on time.

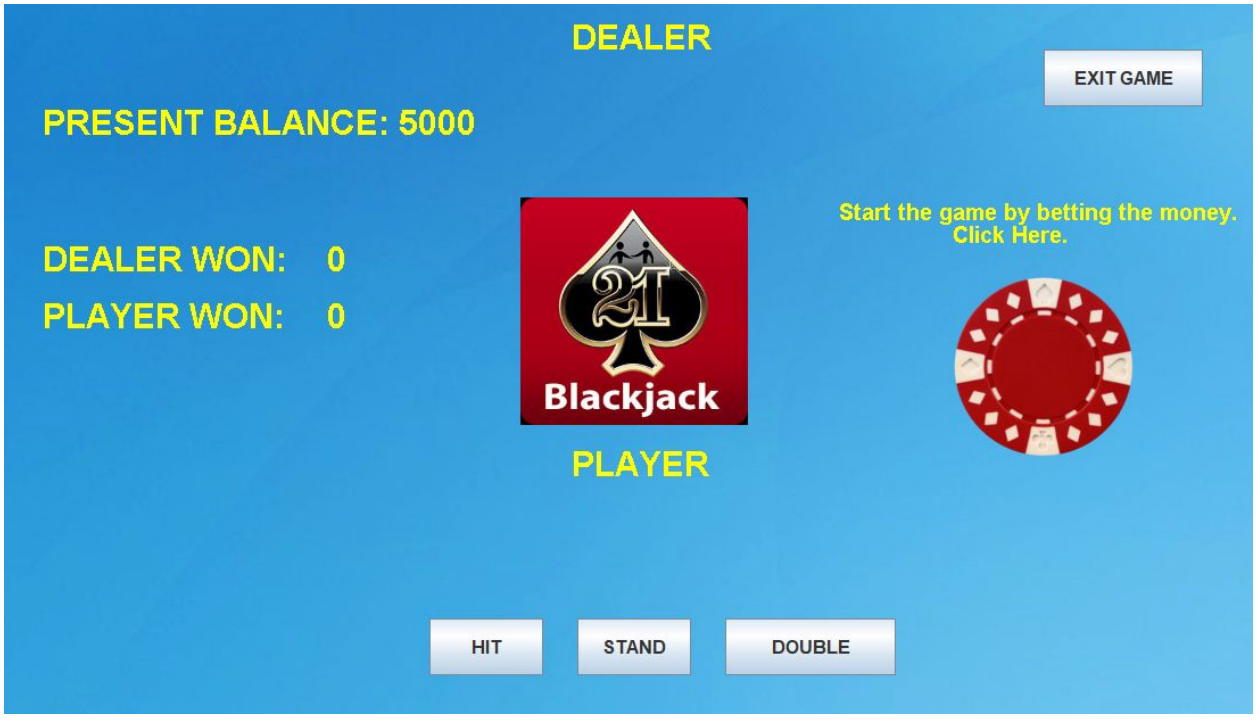
➢ Design: We identified which patterns would be appropriate to be implemented based on the project design to make it more optimal.

➢ Analysis and Implementation: We analysed all the modules of our project and implemented it. However due to the lack of proper analysis, multiplayer implementation was not developed in our project.

➢ Validation: We have written unit test cases to test the project components (end-to-end).

SCREENSHOTS:






DEALER
EXIT GAME
PRESENT BALANCE: 5000
DEALER WON: 0
PLAYER WON: 0
Start the game by betting the money.
Click Here.
Message
BUSTED! Dealer Won !!
OK
HIT
STAND
DOUBLE


Welcome
to the game
BLACKJACK GAME RULES
The goal of the player in the blackjack game is to get a sum of card values as close to 21, without going over.
The player and the dealer receive two cards from a shuffled deck.
If the sum is 21, its a blackjack, you won and the Game is over. If the sum is over 21, the player is busted and the opponent won.
The number cards (2 through 10) are worth the number displayed, face cards are worth 10, and an Ace can be worth either 1 or 11.
Getting another card is called 'Hitting'. Holding the total and ending your turn is 'Standing'. The player can stop hitting at any point.
The dealer must keep hitting until they get to 17. If they get above 17 without busting, they can stay.
OK
EXIT!
PLAY!
HELP!

TEST CASES:

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class BlackJack_Test {
		@Test
		void test1() throws Exception{
			Card card1 = new Card(Suits.Diamonds, 1, 1);
		Card card2 = new Card(Suits.Hearts, 11, 11);
		Card card3 = new Card(Suits.Hearts, 13, 13);

		assertEquals(card1.getRank(), 1); //Ace
		assertEquals(card2.getRank(), 10); //JAck
		assertEquals(card3.getRank(), 10); //King

		/*
		 * assertEquals(card1.getValue(), 1); assertEquals(card2.getValue(), 11); //
		 * Jack : card value is set as 10 by default assertEquals(card3.getValue(), 13);
		 * // King: card value is set as 10 by default
		 */		}
		@Test
		void test2() throws Exception{
			Deck deck = Deck.getInstance();
		assertEquals(deck.getCards().size(), 52); //create all 52 cards
		}
		@Test
		void test3() throws Exception{
			Deck deck = Deck.getInstance();
		assertEquals(deck.getCards().size(), 52); //create all 52 cards
		Card card = deck.getCards().get(0);
		deck.shufflecardDeck();
		assertNotSame(card, deck.getCards().get(0)); //shuffledDeck
		assertEquals(deck.getCards().size(), 52); //shuffled all 52 cards
		}
		@Test
		void test4() throws Exception{
		  Suits c = Suits.Clubs;
		  assertEquals(c.getValue(), 0);

		}

}
```