



SOFTWARE DESIGN DOCUMENT (SDD)

Blackjack Game

Volume 1 - Blackjack

Version 1

Oct 1, 2024

**Authored by:
Sumanth Reddy Bakkem
Raymundo Guerrero
William Schneider**

For more information on Blackjack games, go to the website at
<https://www.247blackjack.com/>.

Copyright © 2024 by CWRU

REVISION LOG

Version	Description of changes	Initials	Date
1.0	First Draft	SRB	Oct 1 2024
1.1	Created table of figures	SRB	Oct 7 2024
1.2	Inspection and Revision	RG	Oct 7 2024

Table of Contents

1	Introduction.....	4
1.1	Overview.....	4
1.2	Scope.....	4
2	System Architecture.....	5
2.1	High-Level Architecture.....	6
2.2	Component Design.....	8
3	Game Logic.....	10
3.1	Card Deck Management.....	10
3.2	Players Action (Hit & Stay).....	11
3.3	Card Scoring System.....	12
3.4	Win/Loss Conditions.....	13
4	User Interface.....	15
4.1	Login Screen.....	15
4.2	Sign-Up Screen.....	15
4.3	Card representations.....	16
4.4	Button Controls.....	16
5	Data Model.....	16
5.1	Player & Dealer.....	16
5.2	Cards.....	16
5.3	User Account.....	17
6	User data management.....	17
6.1	User Registration.....	17
6.2	User Authentication.....	17
6.3	User Statistics tracking.....	17
7	Database Data Persistence.....	17
7.1	Database Schema.....	17
8	Testing.....	17
8.1	Unit Testing.....	17
9	References.....	18
10	Glossary.....	18
11	Inspection Report.....	18

Table of Figures

Figure 1	Game basic Architecture flow.....	5
Figure 2	High Level Backend Data Flow Figure.....	6
Figure 3	Login/signup UI Design look.....	7
Figure 4	Game UI Design look.....	8
Figure 5	Interaction between different components.....	9
Figure 6	CardObject class details.....	11
Figure 7	DealerPlayFunction logic flow.....	12
Figure 8	CalculateCardsTotalValue logic flow.....	13
Figure 9	Game Flow.....	14
Figure 10	UI login/Signup logic Flow.....	16

1 Introduction

Blackjack is one of the most popular card games which is often played in a casino. In this document we will explain how the blackjack game works and how we can implement it using Java. In this design document, we will also explain how we can integrate different components to build this application. We are building this digital version of the game so that it might help any individual who wants to or interested in playing this game in a real casino. In addition, this application also provides a fun way to play blackjack at the user's convenience.

1.1 Overview

Overview of game rules:

- In this game, one player will compete against a dealer.
- The goal of the game is to get card values that are close to 21 (≤ 21).
- The player will lose the game if their hand value exceeds 21.

Game play:

- Initially both player and dealer get two cards; the dealer's first card will be hidden.
- The player can decide either to click the 'Hit' or 'Stand' buttons.
- If the player clicks on the 'Hit' button, then the player will get one card from the shuffled deck pile.
- If the player clicks on the 'Stand' button, then the player's turn will end, and the dealer's turn will begin.

1.2 Scope

In this game we will implement the GUI version of blackjack using Java, taking reference from the classic blackjack game rules. Here are a few details on what we will be implementing in this game application.

1. In this game a player will play against a dealer, here dealer is an automated computer-controlled entity. So essentially the user will play against a computer.
2. The user can track their win and loss statistics against a dealer right in the UI, once he/she is logged in.
3. User interface will be user friendly and will be easy to use and understand.
4. User data from the game will be stored in backend database permanently, to ensure reliability

2 System Architecture

Our game should be implemented in a straightforward structure, where we use different classes and objects to write different parts of the game. User data related code should be implemented in 'User' entity class and game flow will be implemented in blackjack. Class likewise we will be implementing the application code in different classes.

We will also have an interactive UI. Users only need to have access to UI elements to play the game, the backend code will encapsulate the flow and application of the game.

Our application also needs to handle the player's data, so that we can store their data in the database and calculate their statistics. In conjunction with player statistics, we will be also using a backend database to store the user login and signup details. To ensure reliability, we need to persist user statistics in the database without failure.

We will implement controller classes so that we can direct event traffic to appropriate classes upon user interaction. We will also use micro service architecture (MVC) to implement game controllers, game models, etc., this structure will be easy to maintain, test, and debug when there are any issues or bugs in the application.

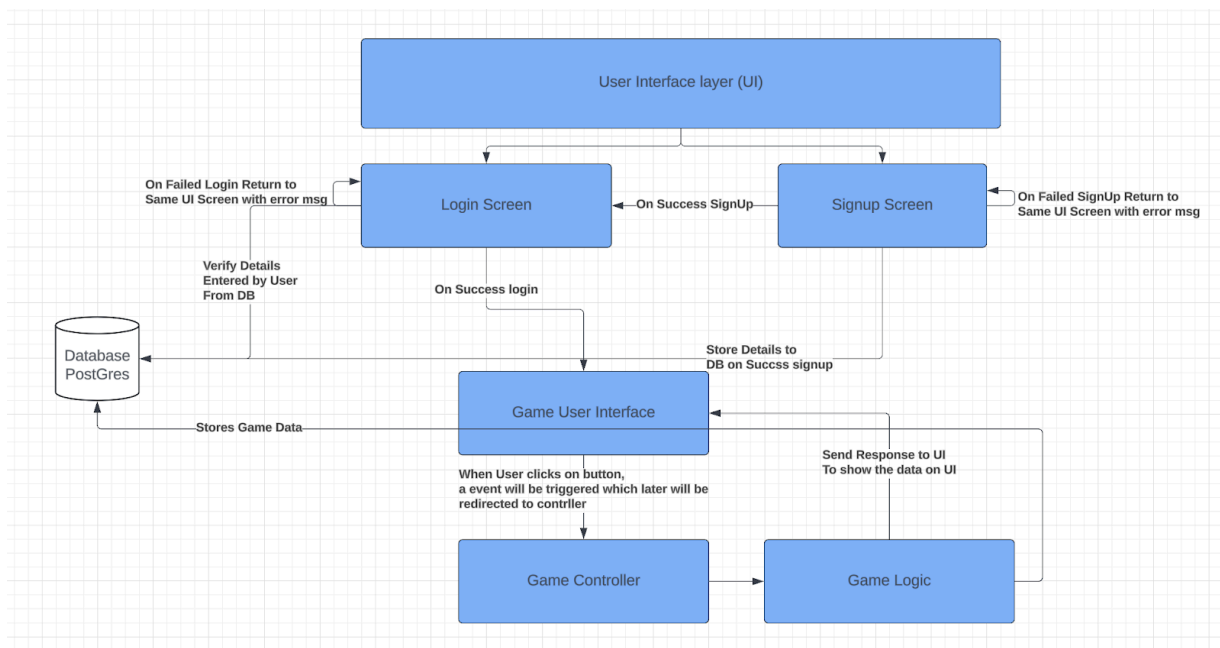


Figure 1 Game basic Architecture flow

Web link to above fig:

https://lucid.app/lucidchart/98645317-4a21-4ac4-8bd7-9258c4c6f9f6/edit?invitationId=inv_5497eec3-a492-4853-8474-0fc958ca3f3

2.1 High Level Architecture

Our game consists of two main high-level architecture components:

- Backend (Java Spring Boot to handle logical events and calculations)
- Frontend (Java Swing to handle UI)

Backend:

In our backend application will take care of handling game flow and user's data like login validations, Signup, games won, etc. It will be mostly built by using our Spring Boot application, and it will include controllers like `UserController.class` & `gameController.class`. These controllers will accommodate functionalities like user login, signup, game flow, and event handler's requests. These events will be redirected to their respective controllers and the controller will send a request to appropriate methods after validation.

Communications from the frontend to the backend will happen from events, where event request data will be processed by the backend. When the user interacts with the frontend, the backend will create a Postman request to interact with the database (Login, Signup, playGame, HitCards, HitStandButton etc.,). The events will hit REST API endpoints via Postman and communication in this case will happen via HTTP protocol, instead of event-based requests. Our backend application should be able to handle both HTTP requests and event-based requests. Any request coming to the backend via Postman should be authenticated for security reasons, only REST API endpoints like `/login` & `/signUp` should be allowed without authentication.

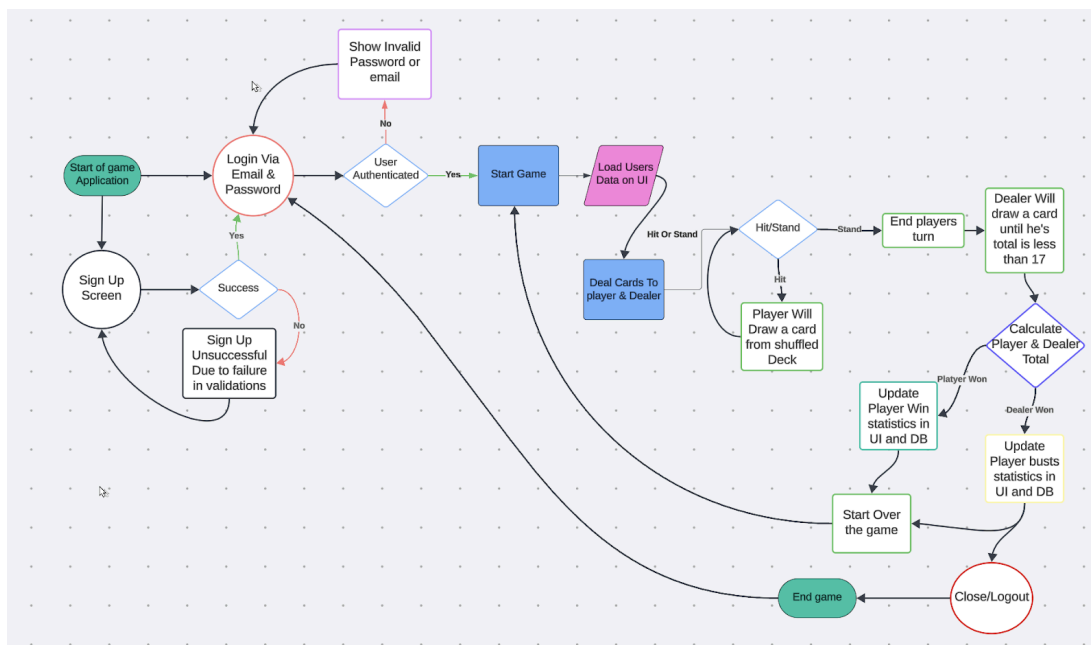


Figure 2: High Level Backend Data Flow Figure

Weblink for above figure:

https://lucid.app/lucidspark/ded50a11-c445-4be3-a5c1-689698f1acc3/edit?viewport_loc=1677%2C-3048%2C3552%2C1749%2C0_0&invitationId=inv_eea8c966-76e1-4e5a-b2a1-1ac

35652c43e

Frontend:

In our application frontend will be implemented using Java Swing. This framework is provided by Java itself. For frontend we will be using event-based design architecture, as we will be having buttons that will generate an event request to the backend upon click. Our frontend will contain buttons like (Login, Signup, StartGame, Hit, Stay, Start Over, Close/Logout).

Login Button: On click of this button, the UI needs to send an event request to the backend controller. The backend will validate the details entered by the user and send back the appropriate message. The log in event is valid if the username and password correspond to an existing entry in the database. Depending on the message, the UI will respond accordingly.

Sign up Button: Upon click of the sign-up button, the UI needs to send a sign-up event request to the backend controller. The backend will validate the details entered by the user. A sign up request is valid if the username is unique, and the password is not empty or comprised of spaces. It will store these details in the database when every validation passes. It will send back an appropriate message, and, based on the message, the UI will respond accordingly.

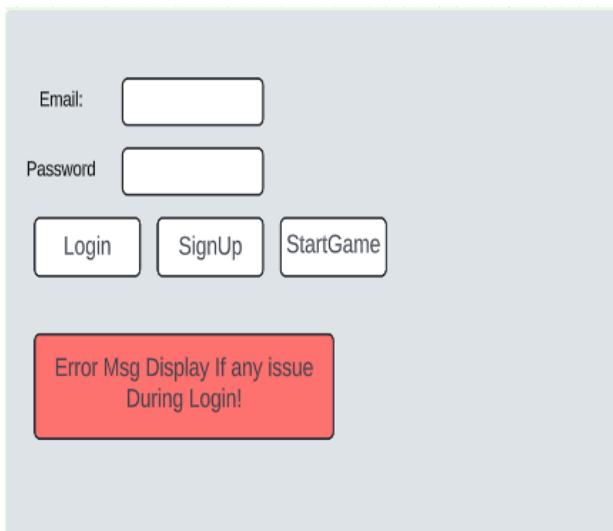
Start Game Button: Before log in, the start game button will be disabled. When the user logs in, this button will be enabled. This button allows the user to play the game. Upon click, the application will generate a new game screen and dispose of the login screen.

Hit Button: Upon click of this button, the user will get a new card from the shuffled deck.

Start Over Button: Upon click of this button, the game will restart. This doesn't affect the user's statistics.

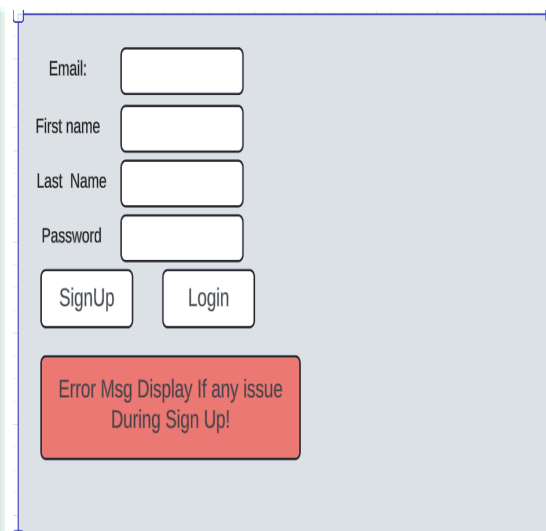
Close/Log out Button: Upon click of this button, the user will be logged out and the frontend will exit the blackjack game. The screen will be redirected to the login screen.

Login UI Design:

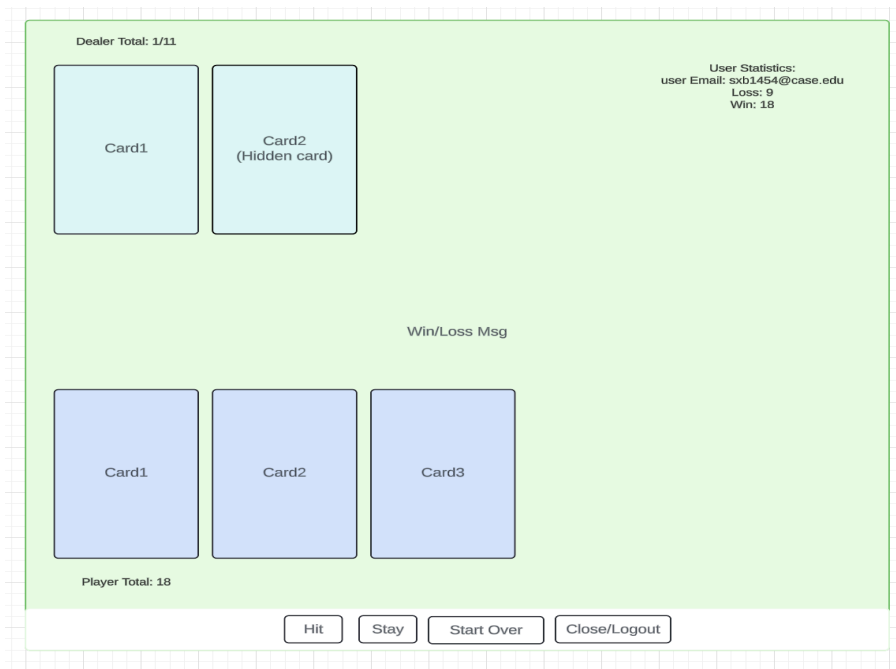


The Login UI Design shows a light blue rectangular area. At the top, there is an 'Email:' label followed by a white rectangular input field. Below it is a 'Password' label followed by another white rectangular input field. Under the password field are three white rectangular buttons labeled 'Login', 'SignUp', and 'StartGame' in sequence. At the bottom of the area is a red rectangular box with white text that reads 'Error Msg Display If any issue During Login!'.

Signup UI Design:



The Signup UI Design shows a light blue rectangular area. It contains four white rectangular input fields stacked vertically, labeled 'Email:', 'First name', 'Last Name', and 'Password'. Below these fields are two white rectangular buttons labeled 'SignUp' and 'Login' in sequence. At the bottom of the area is a red rectangular box with white text that reads 'Error Msg Display If any issue During Sign Up!'.

Figure 3: Login/signup UI Design look**Game UI Design:****Figure 4: Game UI Design look****Web link to above UI design figures:**

https://lucid.app/lucidchart/6af2d345-b306-447f-81bd-659db0ef46de/edit?invitationId=inv_b7b99898-8cc5-4ed7-a3bc-42c8ee9b1560

2.2 Component Design

Our blackjack application contains many key components. Each component will be responsible for its own functionality. We are doing this to reduce coupling and achieve high cohesion between different components. That means each component should only do what it's supposed to do, no other features, code, or unnecessary calls to these components should not happen.

User Controller: This controller will be used to handle all REST API's and event driven requests related to all user actions like login, signup, logout etc. User controller will interact with the **userService** class to validate the details entered by the user. The **userService** class will interact with the **userRepository** class to do database calls like (Delete, Update, Put etc.)

User Service: This component class will contain core business logic for login, signup, logout functionalities. This component class will also contain core business logic for retrieving user statistics data. This component user service acts as an intermediate level between the controller and the repository layer (Data layer), ensuring that user operations are processed correctly.

User Repository: This component will be used to get access to the database, where this repository interface will be used to do CRUD operations in Postgres database.

Game Controller: This controller component will be used to manage the game flow of the Blackjack game. It controls the events that are generated by user actions such as hitting cards, handling player moves ("hit",

"stay"), and calculating results (win/loss). The GameController also interacts with the Player component to update the status of each player object during the game.

Blackjack UI: In this component, we will write the logic related to creating blackjack game UI components. We will be using the Java Swing UI framework to build these UI components like buttons, event triggers, background images etc. This component displays the game board, shows cards, and provides buttons for the user to interact with (like "Hit", "Stay", or "Start Over"). The UI sends player event trigger actions to the Game Controller class, which updates the game state on the click of a button. The UI should be able to update dynamically every time there is change in game state.

Blackjack: In this component, we will write the logic in this class based on the blackjack rules. This class will send requests to the Game Controller class to call user Service & user repository class to save the user statistics and save the game state. This class defines all the methods which are required to run the game flow.

Player: The Player component class keeps track of individual player statistics, such as the cards held, total points, and win/loss status. Player will also contain some additional Boolean fields which can be used to enable or disable the buttons.

Card Object: This component stores the data related to cards that player and dealer possess. In this class there will be three fields: card Type, card value, hidden flag. Using these fields we will show the card in UI, and calculate the total hand values of the player and dealer.

Interaction between different components:

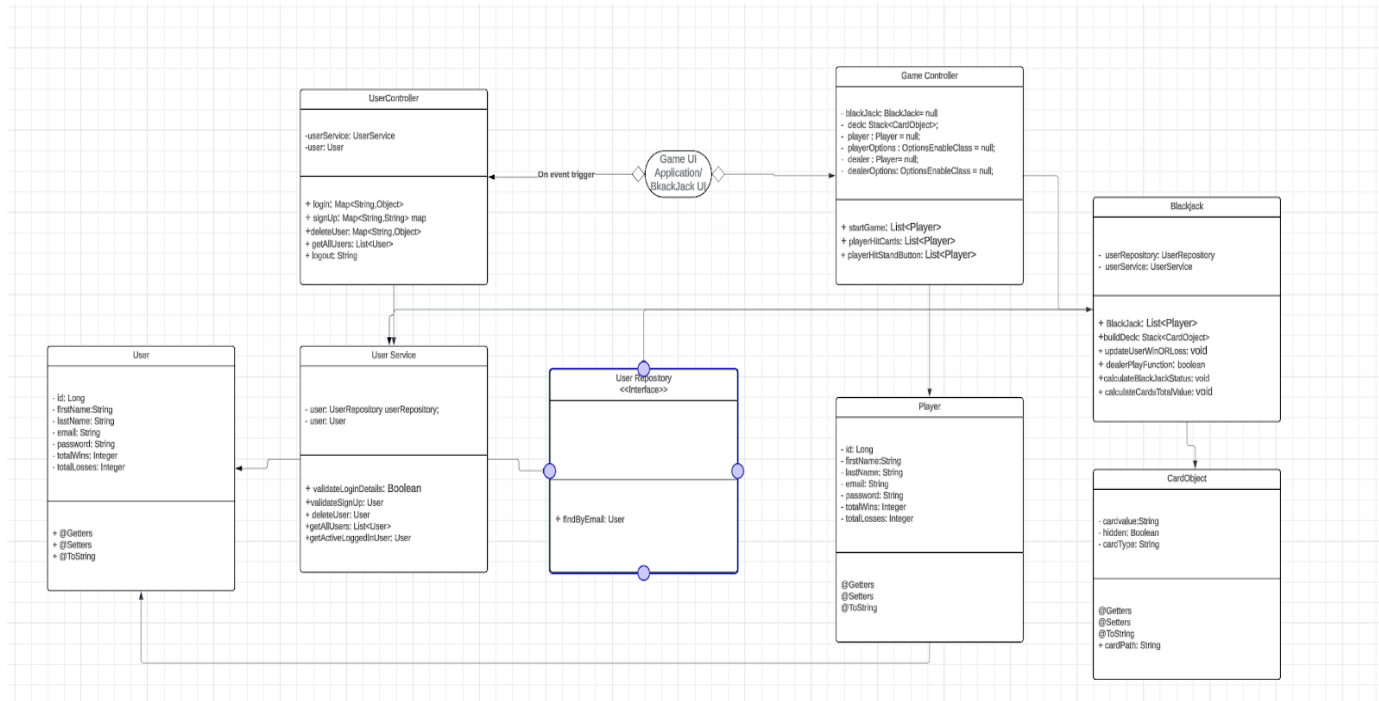


Figure 5: Interaction between different components

web link URL for above figure:

https://lucid.app/lucidchart/c9d9a812-52f5-4c0d-ae1-f999fa0f4597/edit?invitationId=inv_e0de1d27-0669-4d61-9337-b4dcbf054ef1

3 Game Logic

The game logic will be handled in the **Blackjack.java** class. The application will follow simple actions in sequential order during both player and dealer play. At the start of the game, both players will receive two cards, where one card of the dealer will be hidden, and the player's cards will be only shown to the player. The main objective of our game is to reach the hand value closest to 21, also in the process of doing that the player can't exceed his/her hand value greater than 21.

Method name: BlackJack(Player player, Player dealer, Stack<CardObject> deck)

This method will return list of Player Objects

Method logic flow:

This will be one of the main methods in our game. The flow of the method will be as follows:

- **Shuffling the deck:** Deck object will be passed as an argument to the method. We will implement logic that shuffles the deck. We can use many ways to do this like random functions or by using Collection libraries that are available in Java.
- **Drawing cards:** Once we are done with shuffling the deck, we need to distribute cards to both the player and the dealer. Where one card of the dealer will be hidden and the other one unhidden. We will need to implement logic that accounts for a hidden card for the dealer's first card (this can be a parameter or another nearly identical draw() method that outputs a hidden card).
- **Calculating hand value:** After distributing the cards, the next flow of the code should be calculating the total hand value of the dealer and the player. This should be stored in a variable which is later used to show the value in the UI. Here while implementing the code the developer needs to only calculate the value of cards that are unhidden.
- **Calculate Blackjack Status:** After doing card value calculation, one scenario for winning is when a player gets a total hand value of 21 they win directly. This will be discussed briefly in the Win/Loss conditions section. But for now we need to calculate the blackjack win status of the player.
- **Calculate Bust Status:** After doing card value calculation one scenario for Losing is when a player gets total hand value over 21, In that scenario he bust directly. This will be discussed briefly in the Win/Loss conditions section. But for now we need to calculate the bust Status of the player.
- **Return details:** After doing all above operations we need to store all the details of the current game state and return it. We need to return the details in List< Player>. Where this return object contains details regarding player and dealer details in a list.

3.1 Card Deck Management

In this section we will discuss the deck building process. This logic will be written in the build Deck method. This method will return Stack<CardObject>.

Method Logic flow:

To build the deck, we need to know a little bit about the cards. The deck contains a total of 52 cards. In this deck, each card will be unique. There will be four different types of cards: 'Diamonds,' 'Hearts,' 'Spades,' and 'Clubs.' In each type of card, for example the 'Diamond' type, there will be 13 cards: numbers 2 - 10, an Ace, a Jack, a Queen, and a King. Each card will have a value for calculating the total hand value. The numbered cards will have a value of their number. A two of diamonds will be worth two card value. The 'face' cards, which are the Jack, Queen, and King, are all worth 10 card value. Finally, the ace's card value will be worth either 1 or 11, whichever value brings the user closer to 21 without exceeding 21. The total deck should be implemented as a Stack of these cards. The deck should include all 52 of these unique cards.

CardObject Class Details:

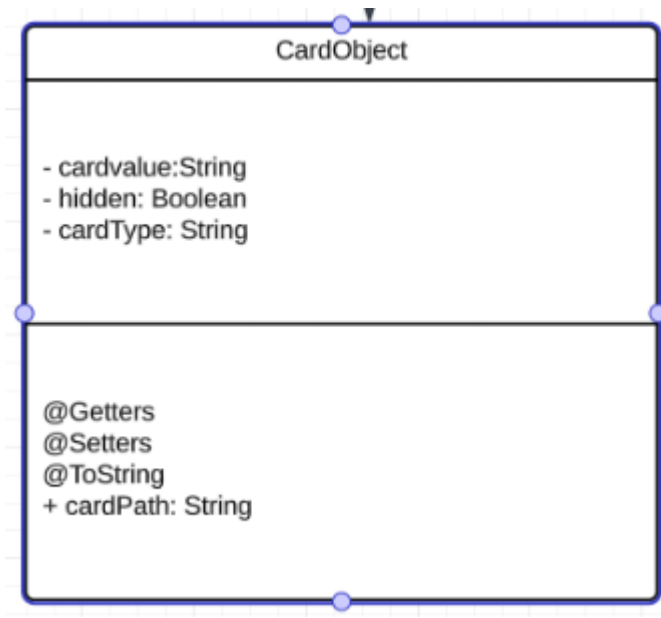


Figure 6: CardObject class details

3.2 Player Actions (Hit & Stay)

Hit Event Logic:

In this method we will implement the logic of when the player draws a card. In this part of the game, when a player clicks on the hit button in the UI (UI related design will be discussed in further sections), the player gets a new card from the deck.

We will implement all the required logic in the *playerHitCards* method.

Method structure: `List<Player> playerHitCards();`

Method logic flow:

- We will pop one card from the deck and add it to the player cards list.
- We will then calculate the total hand value
- Later we need to calculate the bust status of the player in case of hand value more than 21.

Stand Event logic:

In this method we will implement the logic of when the player clicks on the stand button. In this part of the game, when the player clicks on the stand button in the UI (UI related design will be discussed in further section), the player's turn will end, and the dealer's turn will begin. Once the stand button has been clicked, we also need to disable the Hit button in UI for the player (This UI changes will be discussed in further section).

We will implement all the required logic in the *dealerPlayFunction* method.

Method structure: `boolean dealerPlayFunction(Player player, Player dealer, Stack<CardObject> deck);`

Method logic flow:

- First we will hide the hit button, so the player can't draw another card.
- Then we will unhide the dealer's hidden card and calculate the total hand value.
- We need to calculate the status of the dealer's total hand value.
- If the dealer's hand value is less than 17 the dealer will hit. Otherwise the dealer will stand.
- We will then check for the Bust & Win status of the dealer by comparing the player total value.
- We need to run this logic in a while loop, until we meet win/ bust conditions.

Method flow chart:

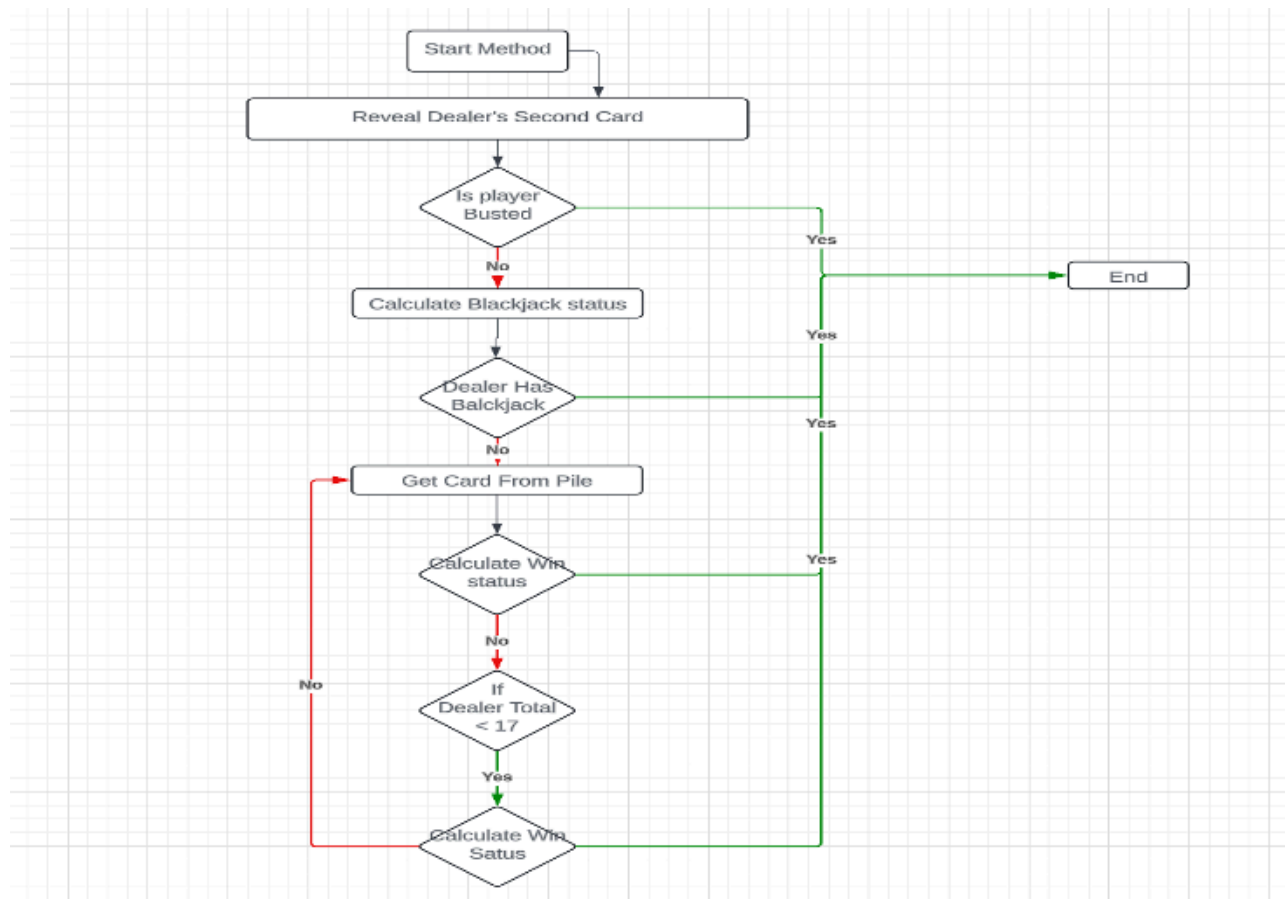


Figure 7: DealerPlayFunction logic flow

Web link to above figure:

https://lucid.app/lucidchart/6775820d-6fd0-4b82-a572-28c6c5ca2dfa/edit?invitationId=inv_f0f3bc25-f9c1-491b-9152-1f83093eb020

3.3 Card Scoring System

In this part of the section, we will discuss the score system. This part of the logic will be implemented in the method *updateUserWinORLoss*

Method structure: void updateUserWinORLoss(Player player, String action);

Method logic flow:

- In this method we will take the final result of the game between player and dealer. The result will be Win, Loss, or Draw.
- In case of win/loss we will make a connection to the backend DB, and update the player win/loss statistics.
- Player details will be passed to the method as an argument.
- We can use the repository interface in the spring boot library to persist data in the backend DB. That is one of the approaches but developers can choose any way to persist the data as long as the data is getting updated against the right user.

3.4 Win/Loss Conditions

In this part we discuss briefly about the win/loss conditions. We also need to incorporate these win/loss logic in *dealerPlayFunction* and *playerHitCards*.

For this purpose, we need to implement one public method which will calculate the total hand value of the cards for dealer and player.

Method Structure: void calculateCardsTotalValue(Player player);

Method logic flow:

- In this method we will get an argument of Player using which we can fetch the cards of the player and dealer.
- After fetching the cards we need to run a loop for all available cards of the individual player.
- This for loop should be run for all cardObjects, using these cardObject classes we can get the card type and card value.
- Using this card value variable we calculate the total values of the cards and update the total hand value available variable of the player.

Method flow chart:

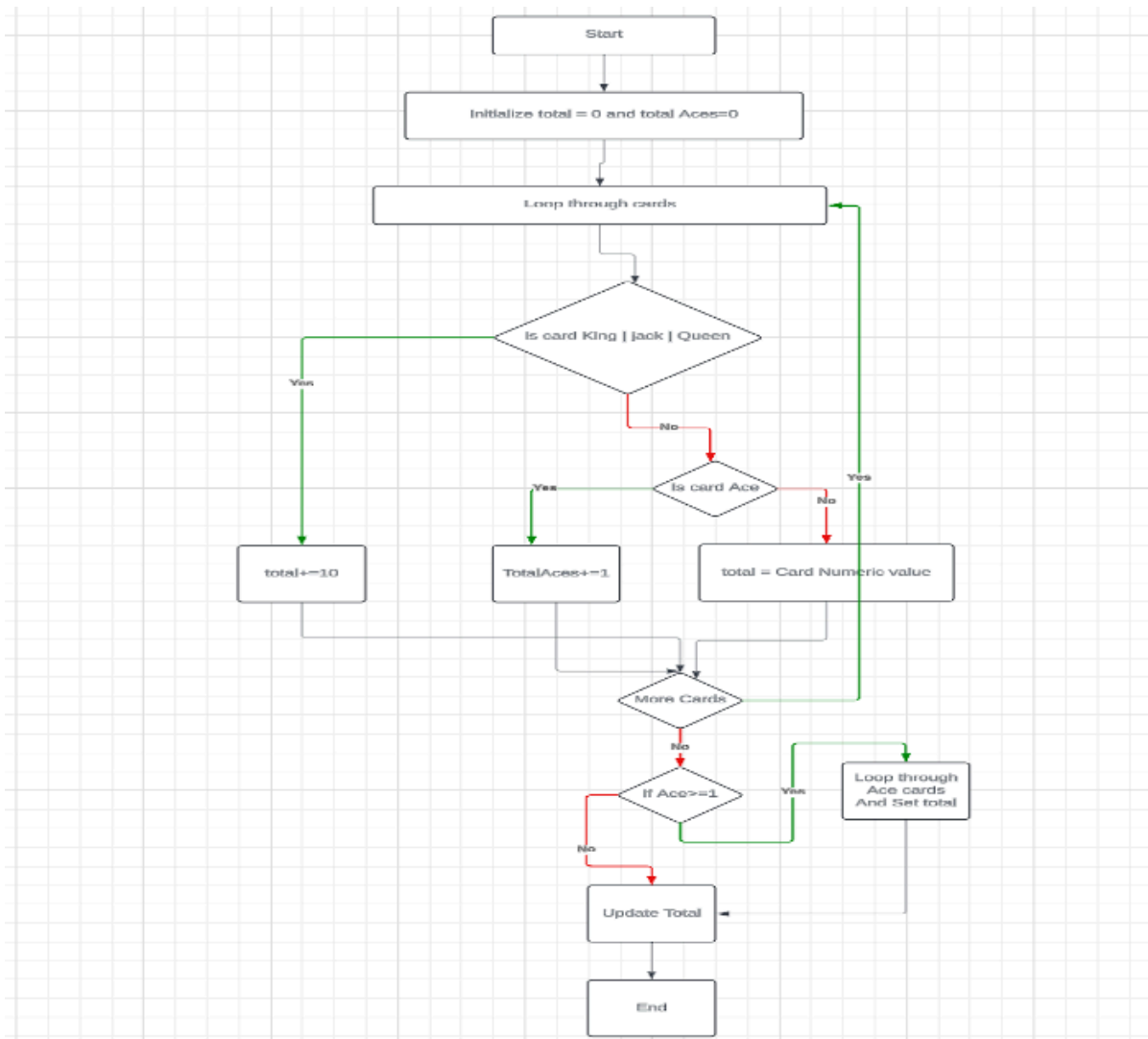


Figure 8: CalculateCardsTotalValue logic flow

Web link to above figure:

https://lucid.app/lucidchart/6775820d-6fd0-4b82-a572-28c6c5ca2dfa/edit?invitationId=inv_f0f3bc25-f9c1-491b-9152-1f83093eb020

After calculating the cards total, we will check for Win/loss conditions. The conditions are as follows:

Player Win/Loss Rules:

- If the player value==21 initially before hitting cards, the player will win blackjack.
- If player value == 21 after hitting cards he will win the game but not blackjack.
- If player value>21 he will bust.

Dealer Win/Loss Rules:

- Once player done with his turn and when dealer plays the game we will unhide his hidden card if his total ==21 in that scenario before hitting cards he will win blackjack
- If dealer value == 21 after hitting cards he will win the game but not blackjack.
- If dealer value>21 he will bust.
- If dealer value>17 he can't draw any more cards from the deck pile. Dealers can only draw cards when his total is less than 17.

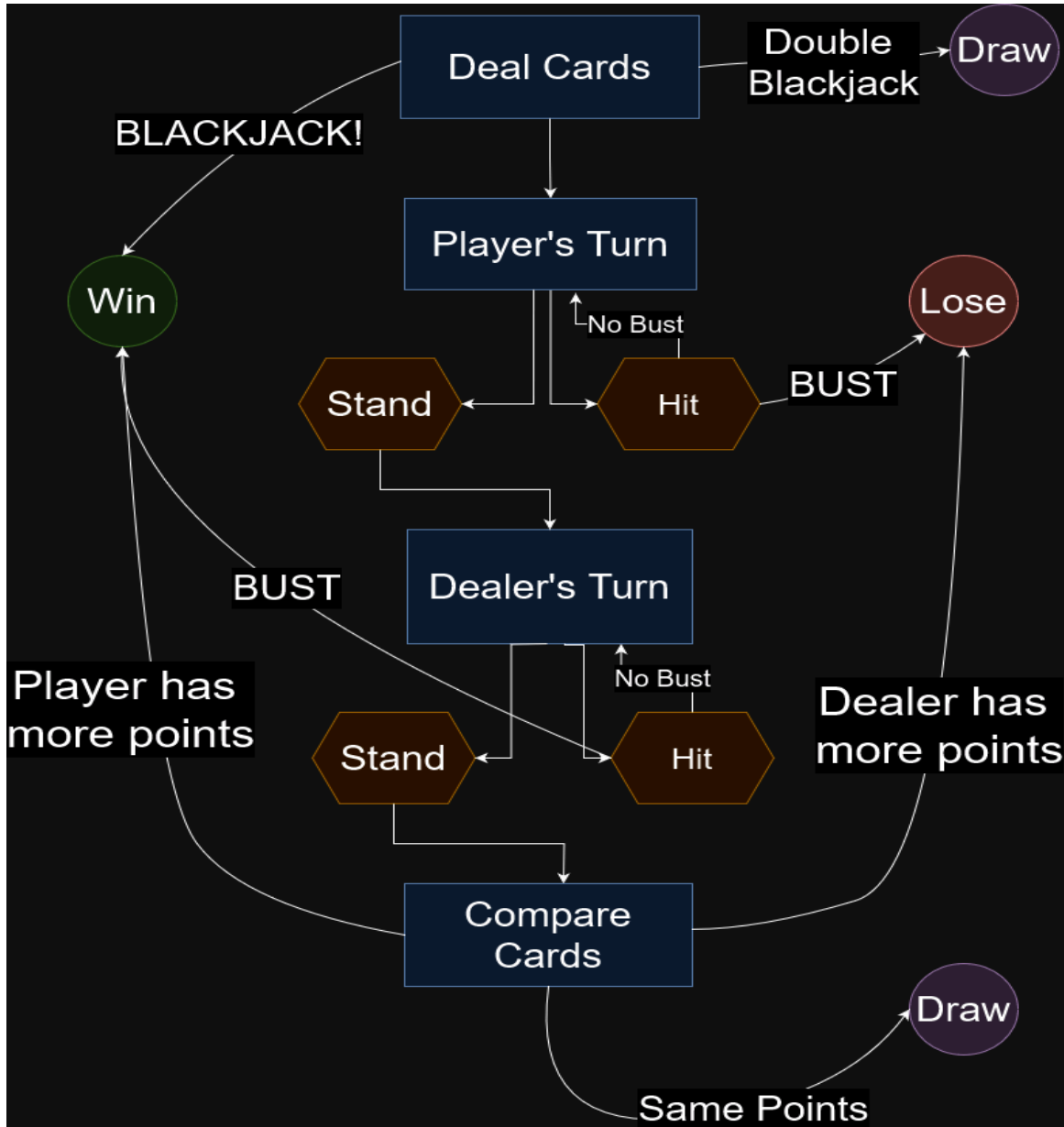


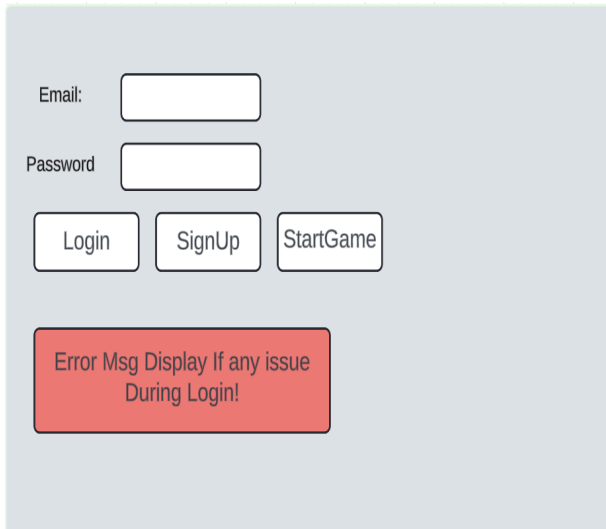
Figure 9: Game Flow

4 User Interface

4.1 Login Screen

The login screen will look like this:

This will be stored in the **Login.java** file. The 'Login' button will send a fetch request to the database and verify the provided login information exists in the database.

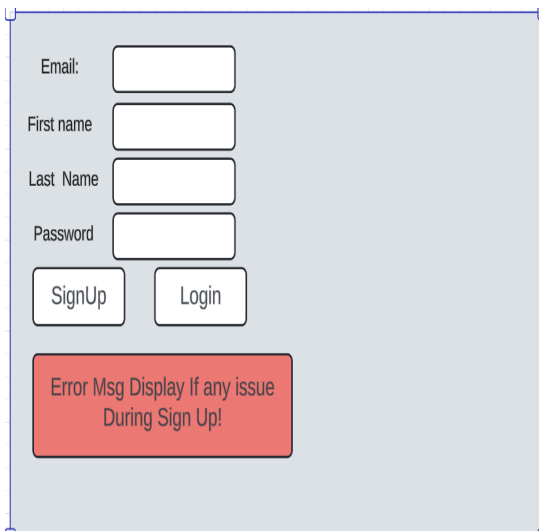


A UI mockup of a login screen on a light gray background. It features two input fields: 'Email:' and 'Password'. Below these fields are three buttons: 'Login', 'SignUp', and 'StartGame'. At the bottom, there is a red rectangular box with the text 'Error Msg Display If any issue During Login!'.

4.2 Sign up Screen

The sign up screen will look like this:

This will be stored in the **Login.java** file. It is managed using the UserService for account creation. It validates the input data, checks for existing users, and saves new users in the database using the **UserRepository.java**.



A UI mockup of a sign up screen on a light gray background. It features four input fields: 'Email:', 'First name', 'Last Name', and 'Password'. Below these fields are two buttons: 'SignUp' and 'Login'. At the bottom, there is a red rectangular box with the text 'Error Msg Display If any issue During Sign Up!'.

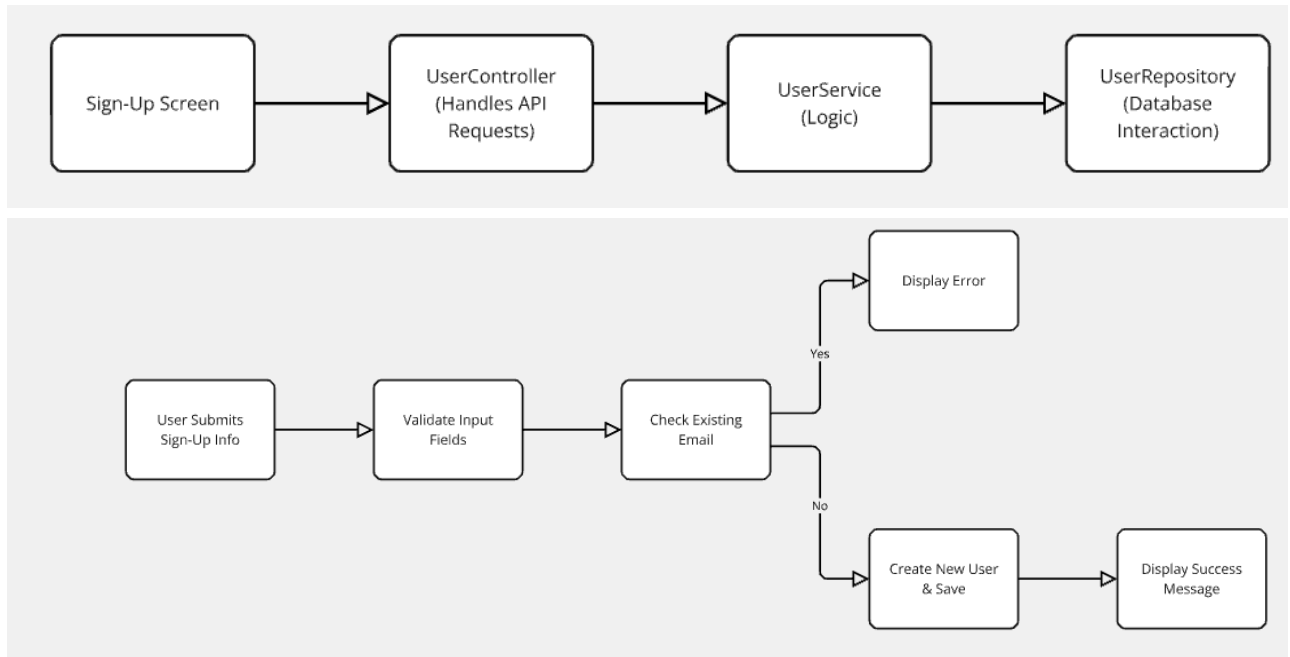


Figure 10: UI login/Signup logic Flow

4.3 Card Representations

The game uses a standard 52-card deck. A cardObject is created, where cardType represents the suit (Spades, Hearts, Diamonds, Clubs), and each card has a value. The cardObject provides a file path for the card images based on its suit and value, allowing the UI components to load and display the correct image when the card is shown on the screen. The card can also be hidden which displays the back side of the card.

4.4 Button controls

The button controls respond appropriately to user interactions and are integrated with the game logic to update the game state based on the user's choice. The button controls are managed through the **BlackJackUI** class. Each button is associated with a single function. Its modular design ensures that the buttons can be modified or expanded with the need to impact the entire game.

5 Data Model

5.1 Player & Dealer

The Player and Dealer are components that manage the data and actions of the participants in each blackjack game. Both have similar attributes such as scores, and cards, but differ in behavior and rules. The player is the user while the dealer is the computer. The BlackJack class manages the state and interaction between the player and dealer. In addition the BlackJackUI class updates the screen based on the player's and dealer actions to display the current state of the game. It does this by using the data provided by the player and dealer components to ensure accurate and real-time visual feedback

5.2 Cards

Each card is created by the CardObject and is central to managing the state of the deck and hands in the games. It holds the cardType(suit), the cardValue, and whether the card is hidden. The card manages all these attributes and provides image paths and data access, allowing for the CardObject to encapsulate all properties and methods needed to represent a playing card in the game.

5.3 User Account

The User account component manages the information of the player's who play the game. Thus allows for players to retrieve their data and view their profile information. The **BlackJackUI** class accesses the User class to display the user information such as win and losses.

6 User data management

6.1 User Registration

The user registration feature allows new users to create an account. The sign-up screen collects the user's information (email, password, first name, last name). It then checks if the registration request is validated, and then the **UserService** class checks if the email already exists. If the validation succeeds, a new user is created in the **UserRepository**.

6.2 User Authentication

The user authentication feature handles user login. As the Login screen collects the user's email and password. The **UserService** class validates the login credentials and, if it's successful, the user is logged in. The user's details are retrieved for further game interactions

6.3 User Statistics tracking

The System tracks user statistics to provide feedback. It tracks all games played, and the user's total wins and losses. After each game is played the games update the user's record in the database based on the outcome of the game. The statistics are displayed on the main game's screen.

7 Database data persistence

7.1 Database Schema

Users table - Stores information about each user who registers and logs into the game.

- ID (Int, Primary key): Provides a unique identifier for each user
- first_name (varchar, not null): User's first name
- last_name (varchar, not null): User's last name
- username (varchar, not null): User's username
- email (varchar, unique, not null): User's email address
- password (varchar, not null): User's password
- wins (int): Total number of wins by the user
- losses (int): Total number of losses by the user
- games_played (varchar): Total number of games played by user

8 Testing

8.1 Unit testing

The testing done on the application, test to confirm that event handling and GUI component updates in Java Swing work as expected in relation to the user interaction with the game. It tests to ensure that all actions performed such as a button click are updated to the game according to that button's functionality.

9 References

<https://www.247blackjack.com/> - Detailed Information about the game of BlackJack and its rules.

10 Glossary

Acronym	Definition
UI	User Interface
CRUD	Create, Read, Update and Delete
API	Application Programming Interface
GUI	Graphical User Interface
MVC	Model-View-Controller
HTTP	Hypertext Transfer Protocol

11 Inspection Report

Issue	Status	Solution
Glossary was omitted from the design document	Resolved	The Glossary was added to the design document detailing all acronyms
All models and figures were not properly formatted and labeled	Resolved	All Figures are now properly formatted and have detailed labels