IS442

Object Oriented Programming

G1T06

Austin Woon Quan

Tan Chin Hoong

Kelvin Chia

1. Analysis and Design

1.1 Initial Analysis

While developing our application, we strived to achieve “Loose coupling, High cohesion” in a bid to allow our application to stay as object-oriented as possible.

To do this, we ensured that every class only does what it is required to do. We wanted the Player Class to only be incharge of what he is required to do, which is to place bids, store his hand, track his bids and tricks. This allowed us to plan and develop classes that are truly independent from each other. This also allowed us the flexibility of not having to depend much on other classes, allowing for much more efficient programming.

In addition, we also wanted to have one “Master class” to handle all mutations to the game and keep track of the current state of the game (for example, player’s current points, bids, hand, current deck of cards). This would help us manage the flow of the game easily through only one class.

1.2 Singleton Design Pattern

After doing some research, we followed the Singleton Design Pattern to handle game logic. Our singleton class is `GameState.` We followed this pattern as it was simple and made sense to us due to the fact that each game can only have one state and all updates for the game can be done through accessing one instance (the state).

With this pattern, each time a game starts, we will only have one instance of the GameState in charge of keeping track of the state. It helps us provide a global access point to the instance and hence, manage game information like the current players trick wins, bids, hands and points easily.

Examples in Project

Our Singleton instance is called in our AppMain.java file where it is initialized when we call GameState.getInstance(). This is our global state for the new game that has started and all operations to mutate a Player’s tricks, points, etc. are updated using the constant variable `GAMESTATE.` All mutations are only made through the one instance of GAMESTATE.

This illustrates our usage of the Singleton design pattern where we have one instance of a class to access and manipulate shared resources (e.g Player’s Hand, Player’s Points, Players Tricks, Current Deck of cards, etc.).

1.2 Design Principles

1.2.1 Single responsibility principle (Excluding Singleton Class GameState)

In our application design, every class (except for our Singleton class) is responsible over a single part of the functionality. For example, the Card Class is in charge of solely defining what a card should have: a suit and a rank value. In cases where we need to compare different cards, the Check Comparator Class handles such tasks instead.

Another example is the Player class This class is responsible for all logic a player should do, like making a bid and choosing a card. Player class also tracks the necessary player statistics like bids, hand, points and trick wins.

However, one aspect to note is that our GameState Singleton Class violates this single responsibility principle. The Singleton class manages the entire game flow like getting possible bids for round, dealing the hand of cards to players, setting the trump cards, prompting player bids etc. As such, it has multiple responsibilities and does not have one single responsibility.

1.2.2 Open/closed principle

The application ensures that classes are open for extension, but closed for modification. For example, in the Player Class, the Player is overloaded with an additional parameter to provide an “Computer” identity. It retains the original “Player” identify and allows for more identities to be added in the future.

2. Approach on tackling GUI Design

The app was first implemented on the CLI to ensure that all functions behaves as expected. The main game logic was then migrated to Java Swing GUI. Prior to the migration, we brainstormed on the various GUI sketches using Figma. After settling on our final wireframe, we implemented it with the GUI Designer on Intellij.

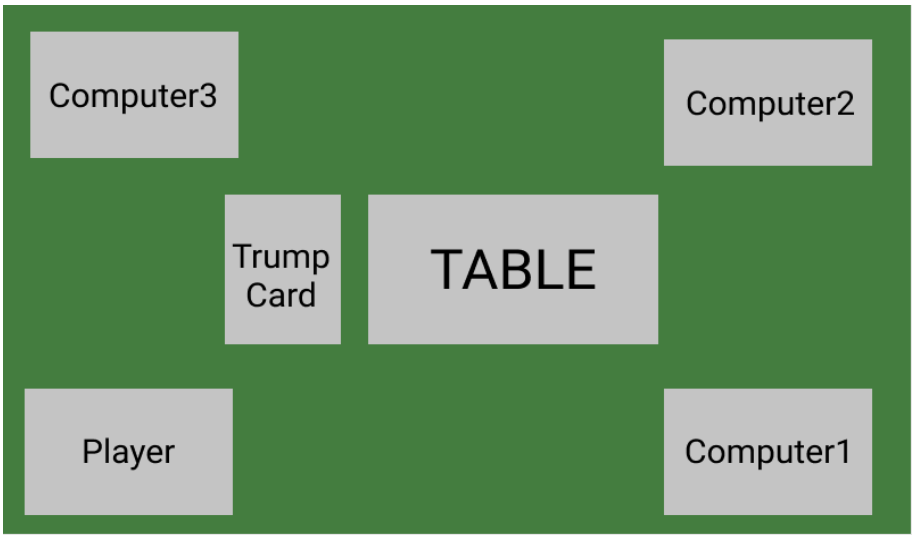


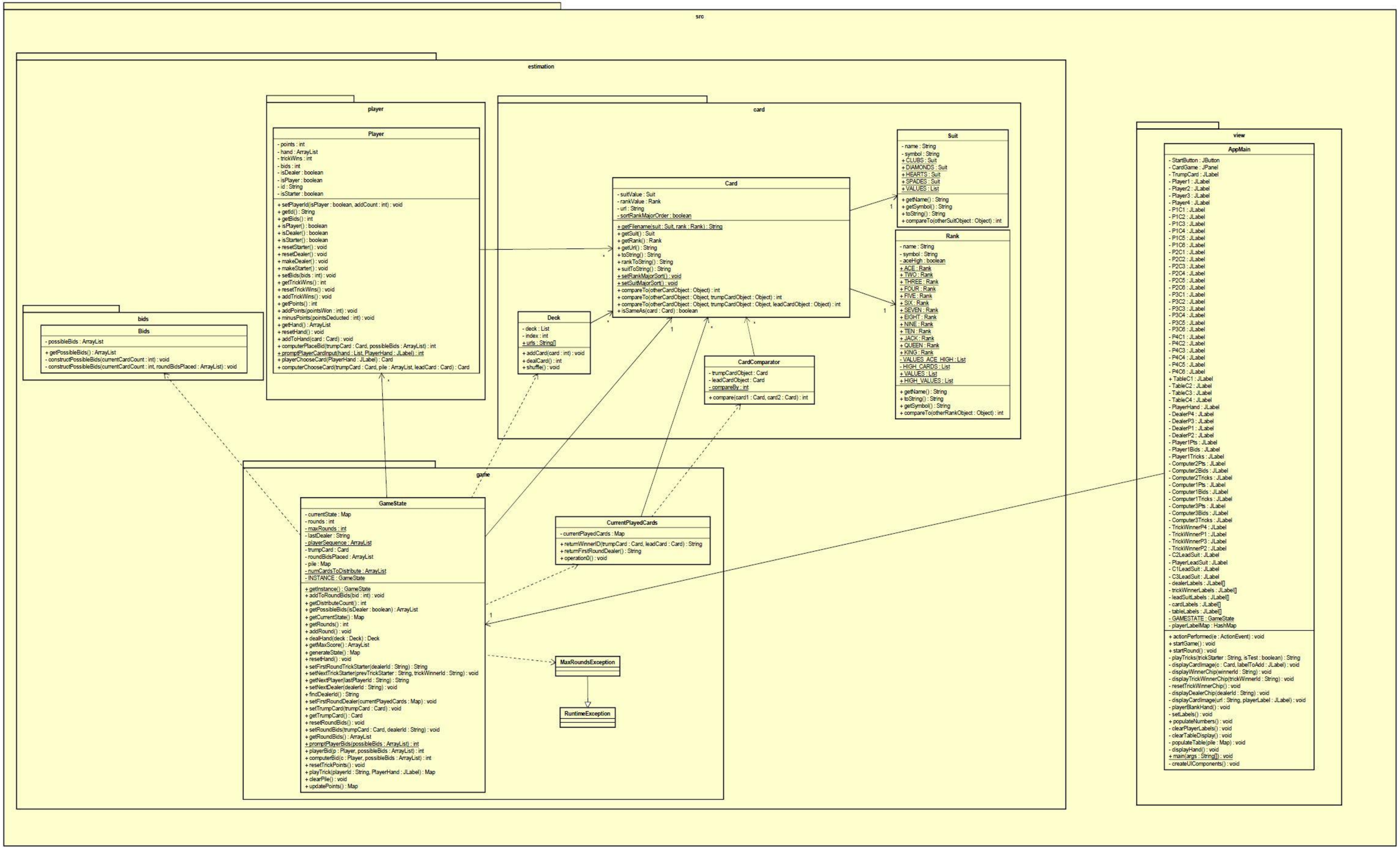
Illustration 1: Final Figma Wireframe

Illustration 2: Actual Implementation

These GUI design principles were considered:

- Green background to portray a casino table impression
- “Dealer” and “Trick Starter” gambling chips to visualise their roles
- Grid System to ensure consistent layout
- Default full window size when app is launched

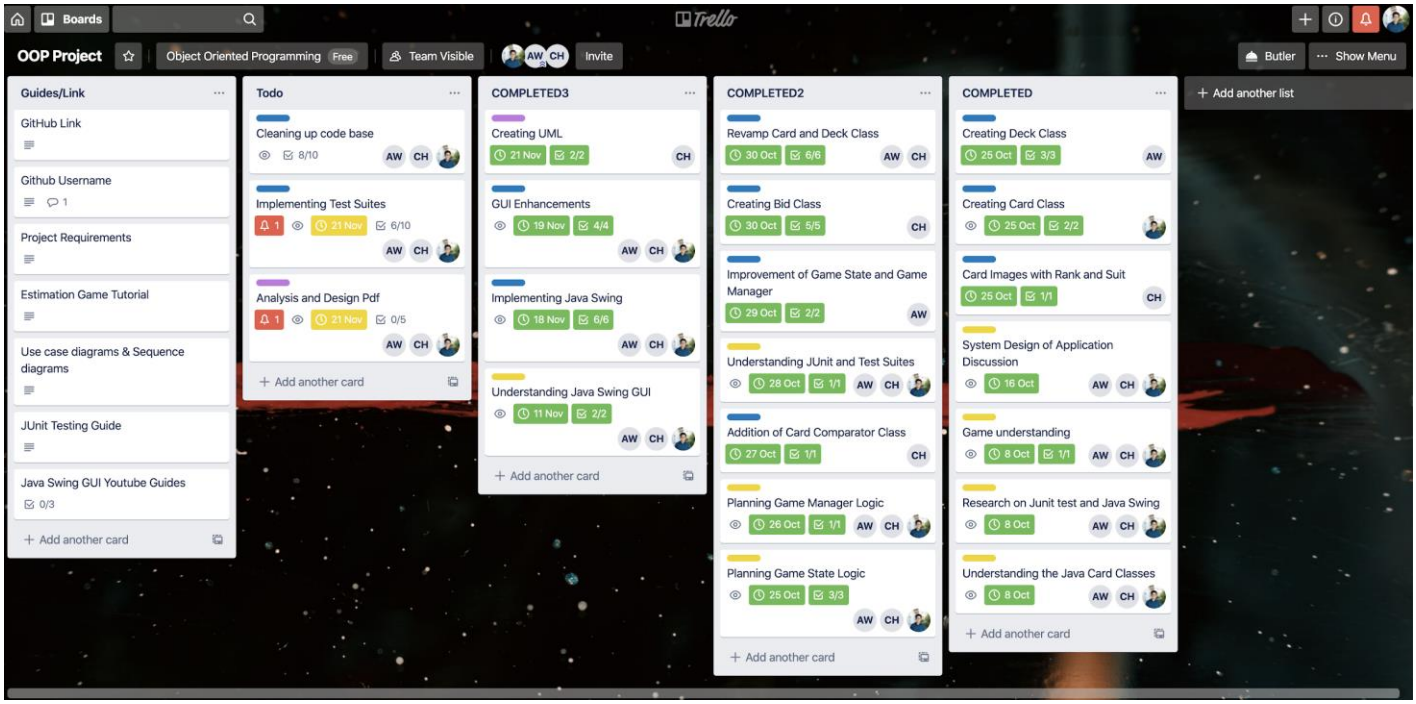
3. Design Class Diagram



4. Project Management Principles and Programming Approach

4.1 Managing Tasks - Kanban Board

In order to manage the project effectively, we applied the agile framework whereby it allows the team to visualise the critical tasks on a kanban board. We are also able to define the project stages clearly, from base code to GUI to testing implementation. Each team member is aware of the tasks and requirements of a Class and it allows for flexibility of other members to update them.



4.2 Programming Approach - Mob Programming

After programming individual java files, we did mob programming to handle the entire game logic. Each time, we had one driver writing code and two navigators reviewing and advising on game logic.

We found that the mob programming approach worked well as teammates who handled different logic for different scenarios (e.g Tricks, Card Sorting, Bid Logic) could advise the driver what to do. The driver was fully in charge of writing readable code and did not need to worry about logic whilst the navigators could fully focus on the logic - ensuring both readable and functional code. This approach worked well for us and we stuck through it for the entirety of our project.