SMD Project 2 - Report

Alex de Vryer, Aidan Butler, Mithun Rithvik Ayyasamy Sivakumar

Analysis of Current Design:

When considering the current design of LuckyThirdteen, it is clear that it does not follow the core principles of object oriented design, which includes GRASP principles. We will highlight the numerous design faults below. The main weakness of the current design of the project is the absurdly low cohesion due to the LuckyThirdteen class being responsible for many, if not most, actions, those of which being completely unrelated to the LuckyThirdteen class itself. This goes hand in hand with the issue of the LuckyThirdteen class having extremely high coupling. This is because the class interacts with and handles all functions and responsibilities that it should be delegating to other, currently non-existent classes. As a result, it makes the overall program less modular and severely lacks protected variation. In summary, the main and overshadowing issue with the current design is the fact that all the program's code and responsibility handling functionality is contained in a single file.

The next glaring issue with the current design of the program is the lack of polymorphism and other GRASP principles being implemented. This can be observed in the dilemma of there being no “player” class in a game that has a specified number of constant players, and multiple different types of players. All the code in the LuckyThirdteen class, simply handled every responsibility for human and random players in a procedural program, therefore lacking cohesion and not making use of a situation where polymorphism would be considered extremely useful. Clearly, this is an obvious fault in the project design, having extremely low cohesion and a highly complex class. Hence, the current design is missing key classes which overall encapsulate the Lucky Thirteen environment, shown through the inability to delegate responsibilities to the respective classes. This current design does not follow the typical “object-oriented” nature that the code should and instead rather follows more of a sequential code structure, which is exactly what object-oriented programming attempts to mitigate. A key example of this is the fact that the logic for logging the data in the game was stored within the LuckyThirdteen class.

Finally, another major design flaw with the original program is that code was heavily repeated such that there were many functions missing for blocks of code that were fundamentally important for many different responsibilities. Instead of repeating the exact same blocks of code using the same conditionals, there should be functions made in order to mitigate the code repetition within the same responsibility.

To address the above issues, we will highlight how we modify the current version and add additional features to incorporate the new functionality required for the LuckyThirteen game in the next section of the report.

Modifications to Design:

Throughout our project we made a multitude of changes to the original source code in order to correct the design mistakes mentioned above. This began by separating the logic in LuckyThirdteen into separate classes and subclasses to delegate responsibility to the respective actions. This was done mainly to decrease the complexity of the current LuckyThirdteen class and to increase its overall cohesion. Through creating a domain class diagram for the project, it allowed us to form the initial new classes and gave us an insight into how we could delegate responsibilities to these new classes, as a result, decluttering LuckyThirdteen. This diagram is shown below.

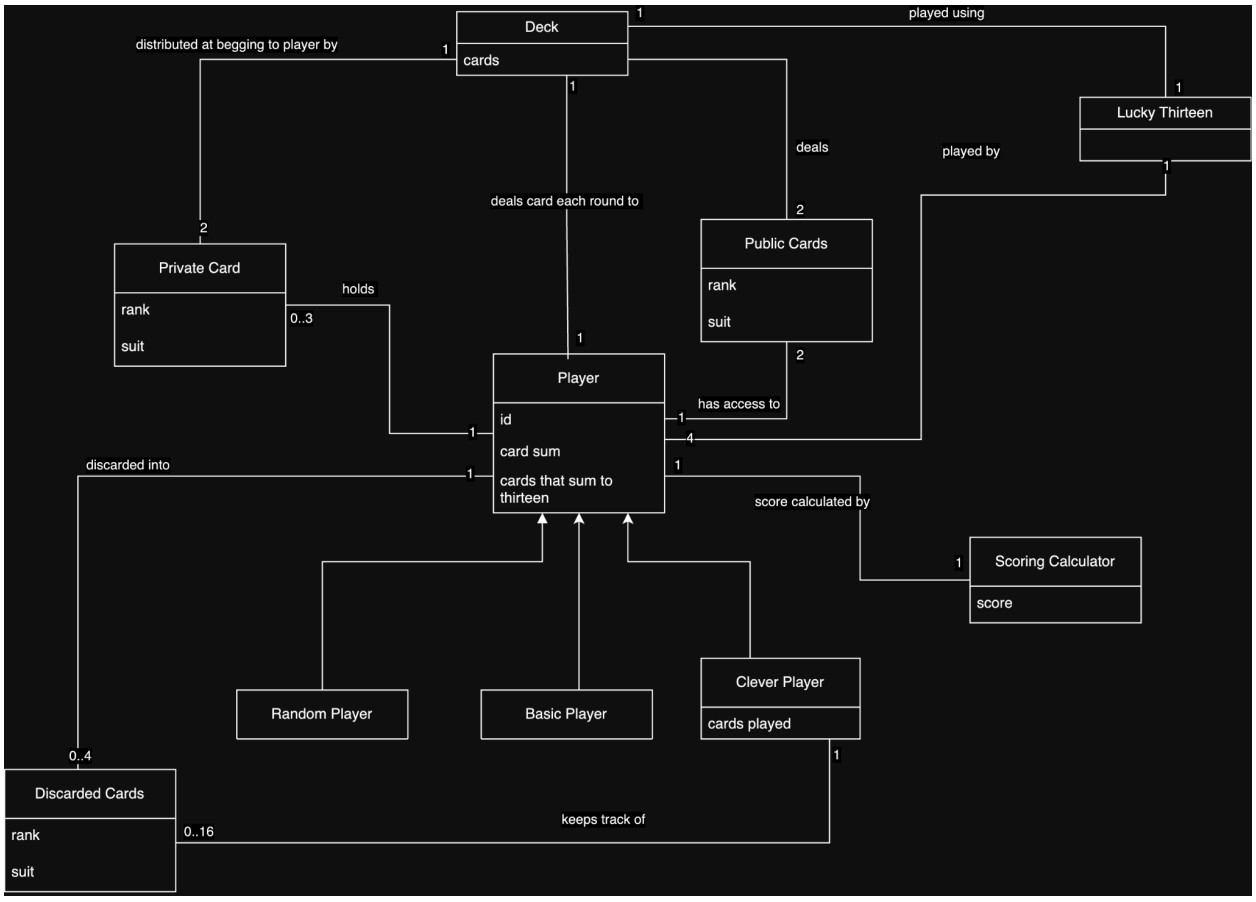

Figure 1. Domain Class Diagram in relation with computer players

After this, we created a Design Class Diagram for our final design for the project. This is also shown below (also available in the submission folder for easier readability).

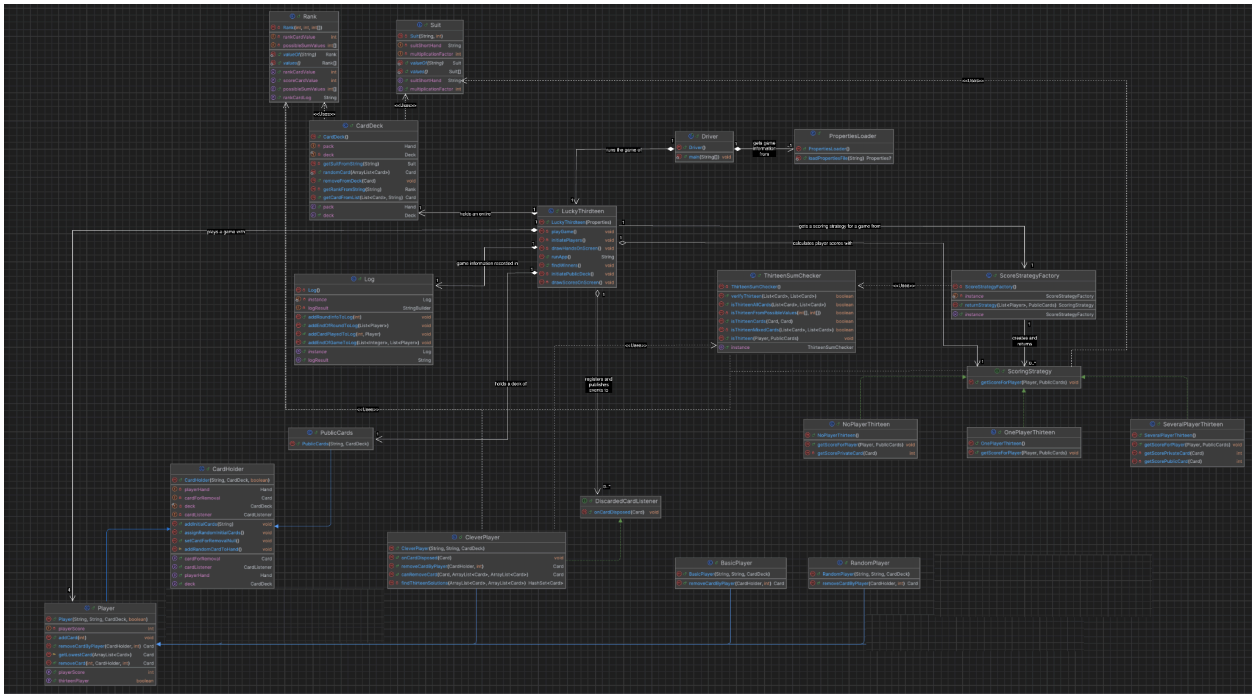

Figure 2. Design Class Diagram in relation with computer players

As shown in the Design Class Diagram, we created the Player Class which would encapsulate all the logic of Players and then applied the concept of polymorphism to enable the creation of its current variations. We made the Player class abstract, through which all other types of players could inherit the methods required to allow future extensions without the need for heavy modification following the Open-Closed principle.

As we had to change how the adding/removing card logic worked for the players compared to the original game, we have created a Design Sequence Diagram to demonstrate the new process to run all four rounds of the game. If a player has a pre-defined set of movements or initial cards, these are assigned to them at the start of the game before any random cards are drawn. If a player has a pre-defined movement for the current round, the game will add/remove the specified card. If not, a random card from the Pack will be added to the player's Hand, and the player will remove a card according to their removal strategy (i.e. random player removes a random card, human player clicks on a card to remove etc). This process is illustrated further below.

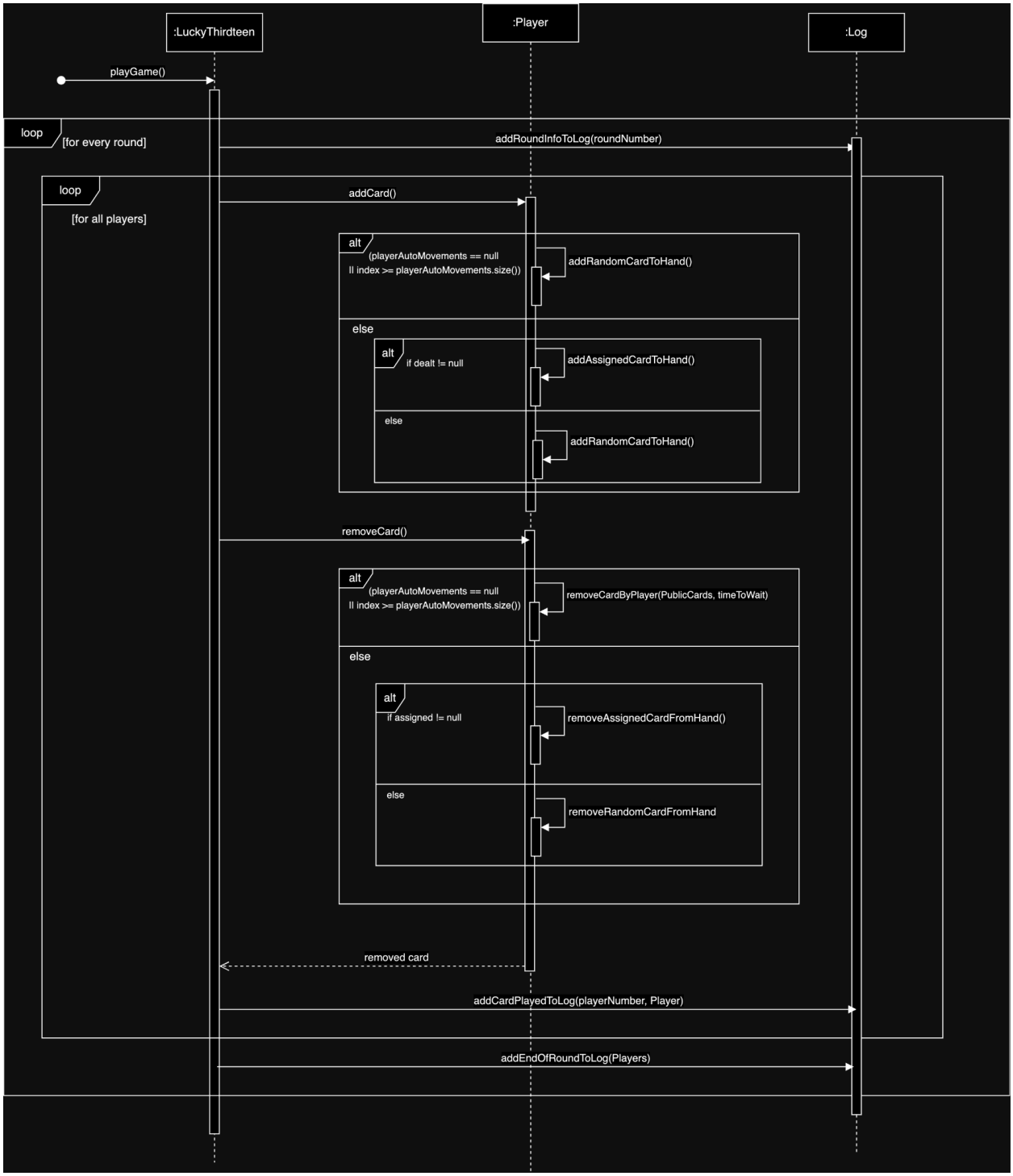

Figure 3. Dynamic Design Model - Game Playing Logic

Originally for this idea, we thought of not only making Player an abstract class, but also creating a new abstract class named CardHard such that Player would compositely aggregate CardHand. In the end we decided to stick with only making Player an abstract class as the new

CardHand class would've limited cohesion through making it more difficult to create a separate PublicCards class.

We then also created an abstract class which is inherited by both the player subclasses and the public cards in order to make use of polymorphism. The CardHolder class which we added holds the player's card hand as well as the public cards shown in the Class Diagram in Figure 2. The class also contains responsibilities such as assigning random initial cards, adding predefined cards from the properties file and adding a random card to a player's hand which are all used by the public cards and players.

Once we had increased cohesion in LuckyThirteen by creating new classes, our next task was to look further into how we could now improve these newly made classes. Another major change we made was to introduce polymorphism into the scoring aspects of the game through the use of the Gang of Four Strategy pattern.

Through the assignment specifications, there were three distinct ways in which we could score the game depending on how many players scored thirteen. Therefore, applying the Strategy Pattern such that instead of these three outcomes being contained within a single class, we could separate these behaviors of the scoring class from the actual scoring class itself. We encapsulated the different scoring behaviors into three different classes called NoPlayerThirteen, OnePlayerThirteen and SeveraPlayerThirteen, each containing its respective implementation.

The three classes implement the Scoring Strategy interface and interact with it through this common interface. This allows for future extension where more scoring strategies could be implemented by creating new classes and not having to undergo major modifications. These can also be seen above in the Design Class Diagram in Figure 2.

We also used more Gang of Four strategy patterns by using a Singleton Factory being responsible for the creation of the scoring strategies. The Singleton Factory class will check the number of players which have achieved 13 and will return the appropriate strategy. We also created another Singleton class called ThirteenChecker, which holds the various logic for summation calculations. Since all conditions for the various sum calculations need to be checked every time, we disregarded the possibility of potentially utilizing the strategy pattern with ThirteenChecker as it would have increased code complexity for very limited benefit (as all strategies would need to be used each time). We have created a Dynamic Design Model to further highlight the flow and sequence of this method which is shown below.

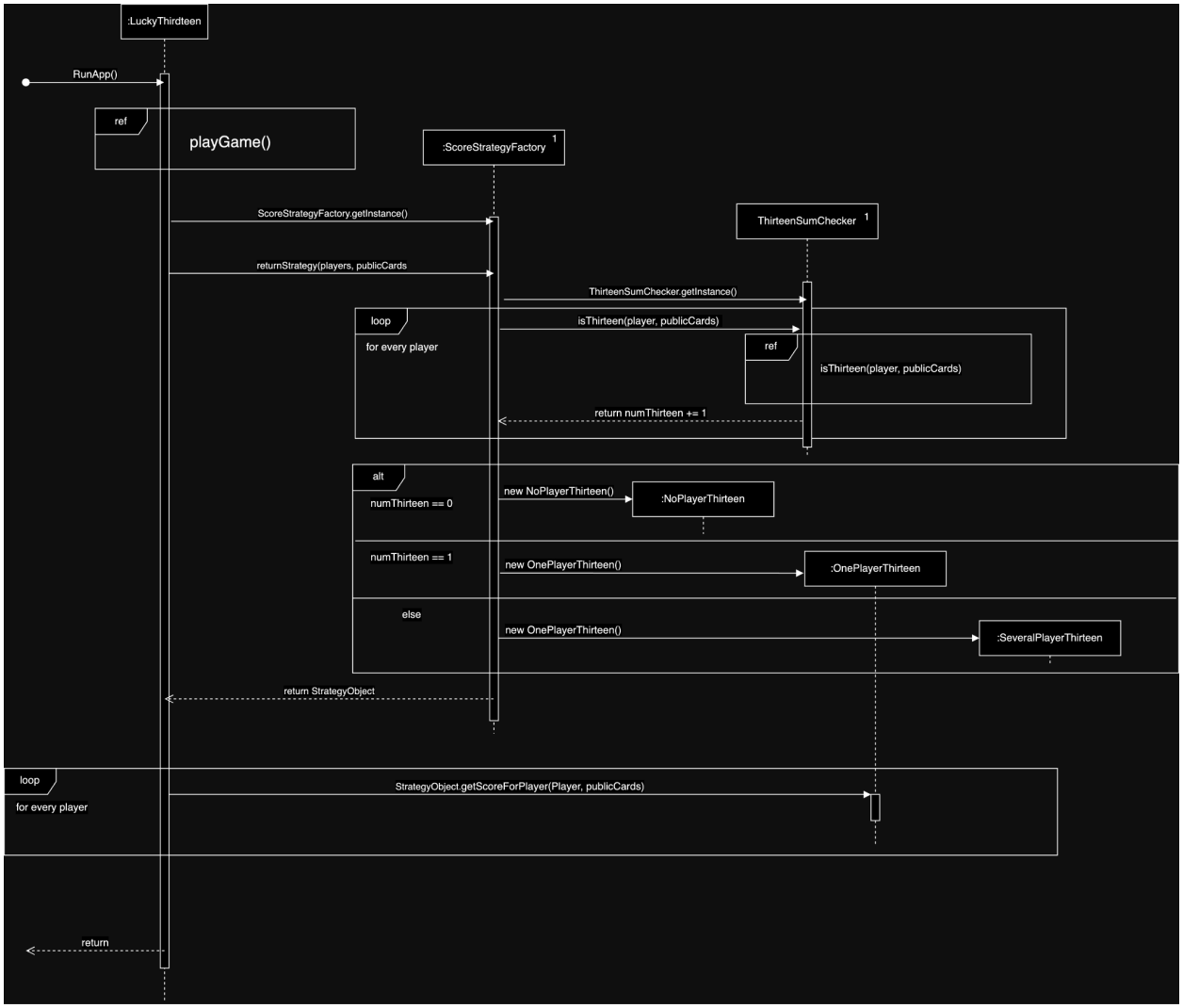

Figure 4. Dynamic Design Model - Strategy Creation with Factory (Case: 1 player sum to 13)

Other smaller changes we made that are notable include removing the log from LuckyThirdteen class in order to increase its cohesion. This was made into a separate Log class which is also a Singleton class due to their only being one Log within the game.

We also pulled out the enumerations for Rank and Suit for each card which was located within the original LuckyThirdteen class to further increase cohesion. Coupled with this we created a CardDeck class which manages all the game actions related to the pack of cards (i.e. removing a card from the deck). For the CardDeck class we originally planned to make it a singleton class such that only a single instance of it would exist. Although this seems logical and ideal, when implemented, it caused issues within our program such that LuckyThirdteen would crash during multiple rounds of testing. Therefore we made the decision that the LuckyThirdteen class would compositely aggregate the CardDeck and distribute all card objects to the players, hence fixing the previous bug. These changes increased the cohesion of the project and also made use of

the GRASP patterns of indirection and pure fabrication in order to delegate responsibilities to the CardDeck class.

Clever Player:

The clever player uses a more advanced card removal strategy compared to both the basic and random players. Compared to the basic player, the Clever Player checks if the cards in its hand (the player has three cards in its hand during deliberation time) can add up to 13 using any of the three possible summation strategies. It also observes the cards which have been disposed of and uses this information during play to inform its decisions and keeps track of this information by using a HashMap of each possible card value (0 to 13) and recording how many cards of each value have been discarded from play. Before removing any cards, a HashSet is created containing the cards which are in the player's hand which are a part of any 13 summation. Depending on the number of cards in the HashSet, the Clever Player will do the following actions:

0 cards in the hand contribute to 13:
The clever player will check all of the private cards and determine the difference between the target score (13) and all possible sum values which that particular card can take. It then checks the discarded card HashMap to see if three or more cards of that particular value have been discarded (out of the four cards in the deck for that value). If this is true, this card is a candidate for removal. Once all cards have been checked, if there are any which have been marked as a candidate for removal, the lowest valued card out of this subset is discarded. If none have been marked for removal, the lowest value card in the player’s deck is discarded instead.

1 card in the hand contributes to 13:
The clever player will check the two other cards which do not contribute to the 13 summations and will discard the card which has the lowest score value.

2 cards in the hand contribute to 13:
The clever player will remove the card which does not contribute to any of the 13 summations.

3 cards in the hand contribute to 13:
The clever player will go through all of the cards in the hand starting with the card which has the lowest score value. If this card can be removed without removing our player’s ability to reach 13 with the other cards in its hand, this card will be removed. If not, the player checks the other two higher scoring cards and sees which card can be removed out of those two. For example, the player has 5 Clubs and 8 Hearts and is dealt 8 Diamonds. None of the public cards help the player reach 13. If the 5 Clubs is removed (the lowest card), the player can no longer reach 13, so the 8 Diamonds will be removed instead).