SWEN30006 Project 2 Report

Monday Workshop, Team #9
Min Thu Han, Arja Das, Yiyan Zhang

There are many weaknesses of the initial design of ‘Lucky Thirteen’, the weakness we have identified as of the most concern are:

High Coupling:

- ‘LuckyThirdteen’ currently handles every aspect of the game, including game logic, UI handling, and game state management. This violates the low coupling principle of GRASP, as classes should be independent of each other, as changes made to aspects of the class will likely lead to changes being made elsewhere. This can be seen particularly with the methods ‘initGame()’ and ‘playGame()’, as both are responsible for handling UI logic, event handling, and game logic, the high coupling makes it difficult to adapt and maintain the game in its current state.

Low Cohesion:

- Likewise, to the high coupling problem, the current design faces the problem of low cohesin. This is due to the class handling multiple responsibilities, such as setting up the game and handling its logic to updating the UI and dealing with the scoring logic. This obviously contradicts the GRASP principle of high cohesion, which advocates for classes with specific and narrowed responsibilities, ideally such responsibilities should be split up into their own respective classes.

Improper use of Creator:

- According to the creator principle of GRASP, a class should only create an instance of an object if the class is closely tied to the object, uses it frequently, or has the information needed to create it. In the current design, the ‘LuckyThirteen’ class handles the creation and configuration of ‘Card’ and ‘Hand’ instances, whereas the creation should be done in separate, specific classes.

No Polymorphism:

- The current design does not employ the use of polymorphism for handling different types of cards or game rules, further increasing the difficulty of extending the code for other types of cards or game rules.

Lack of encapsulation:

- Several attributes and methods that could be private are left as public, exposing the internal details of the code. Methods such as ‘dealACardToHand’ and ‘initScore’ could be made private as they’re only needed for the internal workings of the game.

In refactoring the ‘Luckythirdteen’ as depicted in the design class diagram below, we have aligned the current implementation to adhere to the GRASP principles and implement some of the GOF design patterns. These were done to tackle the issues that the prior implementation had, including its improper use of creator, lack of polymorphism and encapsulation, and high coupling with low cohesion.

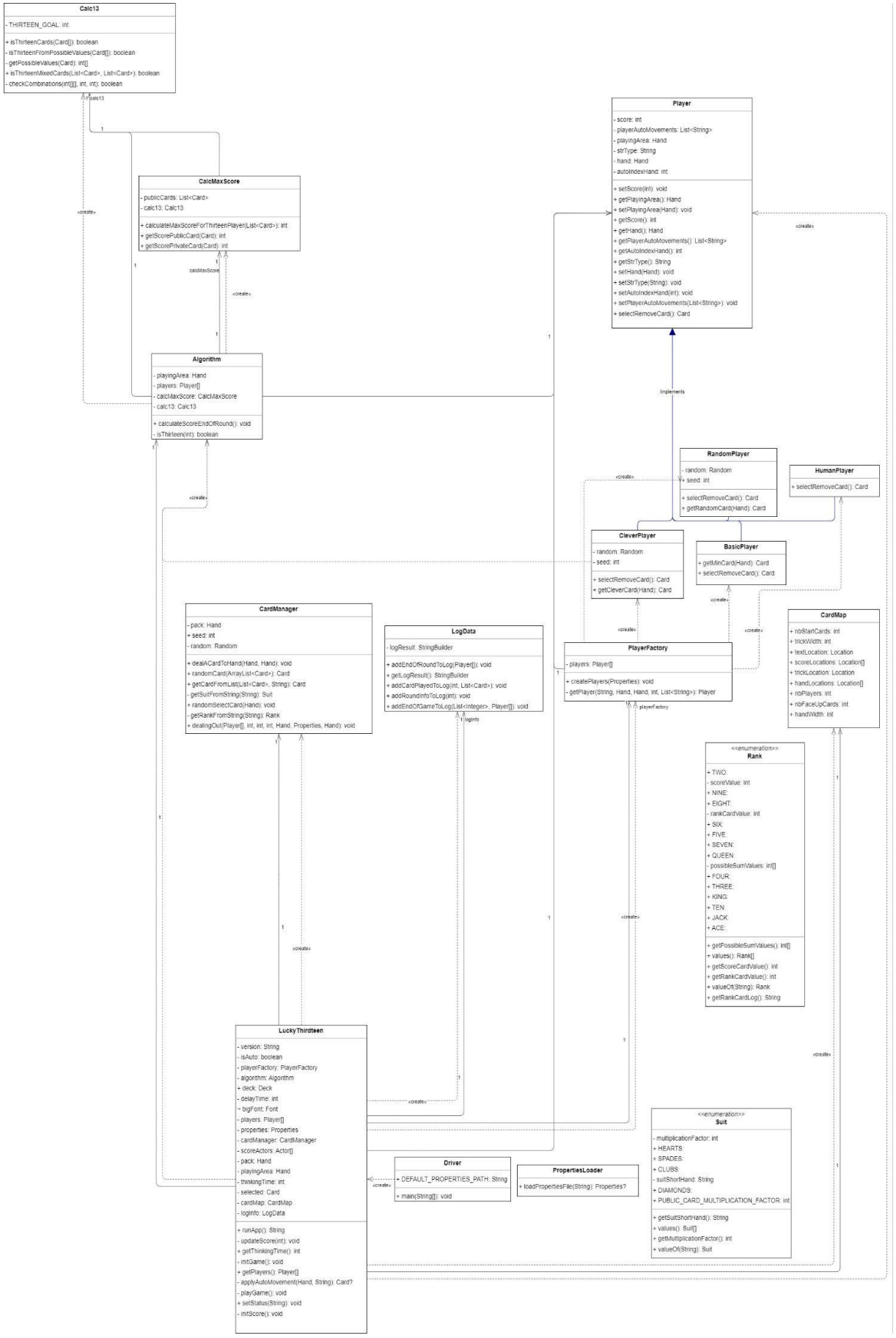
Calc13
- THIRTEEN_GOAL: int
+ isThirteenCards(Card[]): boolean
- isThirteenFromPossibleValues(Card[]): boolean
- getPossibleValues(Card): int[]
+ isThirteenMixedCards(List<Card>, List<Card>): boolean
- checkCombinations(int[][], int, int): boolean
calc13
CalcMaxScore
- publicCards: List<Card>
- calc13: Calc13
+ calculateMaxScoreForThirteenPlayer(List<Card>): int
+ getScorePublicCard(Card): int
+ getScorePrivateCard(Card): int
calcMaxScore
«create»
Player
- score: int
- playerAutoMovements: List<String>
- playingArea: Hand
- strType: String
- hand: Hand
- autoIndexHand: int
+ setScore(int): void
+ getPlayingArea(): Hand
+ setPlayingArea(Hand): void
+ getScore(): int
+ getHand(): Hand
+ getPlayerAutoMovements(): List<String>
+ getAutoIndexHand(): int
+ getStrType(): String
+ setHand(Hand): void
+ setStrType(String): void
+ setAutoIndexHand(int): void
+ setPlayerAutoMovements(List<String>): void
+ selectRemoveCard(): Card
Algorithm
- playingArea: Hand
- players: Player[]
- calcMaxScore: CalcMaxScore
- calc13: Calc13
+ calculateScoreEndOfRound(): void
- isThirteen(int): boolean
implements
RandomPlayer
- random: Random
- seed: int
+ selectRemoveCard(): Card
+ getRandomCard(Hand): Card
HumanPlayer
+ selectRemoveCard(): Card
CleverPlayer
- random: Random
- seed: int
+ selectRemoveCard(): Card
+ getCleverCard(Hand): Card
BasicPlayer
+ getMinCard(Hand): Card
+ selectRemoveCard(): Card
CardManager
- pack: Hand
+ seed: int
- random: Random
+ dealACardToHand(Hand, Hand): void
+ randomCard(ArrayList<Card>): Card
+ getCardFromList(List<Card>, String): Card
- getSuitFromString(String): Suit
+ randomSelectCard(Hand): void
- getRankFromString(String): Rank
+ dealingOut(Player[], int, int, int, Hand, Properties, Hand): void
LogData
- logResult: StringBuilder
+ addEndOfRoundToLog(Player[]): void
+ getLogResult(): StringBuilder
+ addCardPlayedToLog(int, List<Card>): void
+ addRoundInfoToLog(int): void
+ addEndOfGameToLog(List<Integer>, Player[]): void
logInfo
PlayerFactory
- players: Player[]
+ createPlayers(Properties): void
- getPlayer(String, Hand, Hand, int, List<String>): Player
playerFactory
CardMap
+ nbStartCards: int
+ trickWidth: int
+ textLocation: Location
+ scoreLocations: Location[]
+ trickLocation: Location
+ handLocations: Location[]
+ nbPlayers: int
+ nbFaceUpCards: int
+ handWidth: int
<<enumeration>>
Rank
+ TWO:
- scoreValue: int
+ NINE:
+ EIGHT:
- rankCardValue: int
+ SIX:
+ FIVE:
+ SEVEN:
+ QUEEN:
- possibleSumValues: int[]
+ FOUR:
+ THREE:
+ KING:
+ TEN:
+ JACK:
+ ACE:
+ getPossibleSumValues(): int[]
+ values(): Rank[]
+ getScoreCardValue(): int
+ getRankCardValue(): int
+ valueOf(String): Rank
+ getRankCardLog(): String
LuckyThirdteen
- version: String
- isAuto: boolean
- playerFactory: PlayerFactory
- algorithm: Algorithm
+ deck: Deck
- delayTime: int
~ bigFont: Font
- players: Player[]
- properties: Properties
- cardManager: CardManager
- scoreActors: Actor[]
- pack: Hand
- playingArea: Hand
- thinkingTime: int
- selected: Card
- cardMap: CardMap
- logInfo: LogData
+ runApp(): String
- updateScore(int): void
+ getThinkingTime(): int
- initGame(): void
+ getPlayers(): Player[]
- applyAutoMovement(Hand, String): Card?
- playGame(): void
+ setStatus(String): void
- initScore(): void
Driver
+ DEFAULT_PROPERTIES_PATH: String
+ main(String[]): void
PropertiesLoader
+ loadPropertiesFile(String): Properties?
<<enumeration>>
Suit
- multiplicationFactor: int
+ HEARTS:
+ SPADES:
+ CLUBS:
- suitShortHand: String
+ DIAMONDS:
+ PUBLIC_CARD_MULTIPLICATION_FACTOR: int
+ getSuitShortHand(): String
+ values(): Suit[]
+ getMultiplicationFactor(): int
+ valueOf(String): Suit

Figure 1

One of the initial changes we decided to implement was the use of polymorphism and inheritance, as shown in figures 2, we have introduced a new abstract class named ‘Player’, which the four different types of players, ‘HumanPlayer’, ‘RandomPlayer’, ‘BasicPlayer, and ‘CleverPlayer’ inherit from, as they all share the same core functionality. The use of inheritance allowed for code reuse, as methods such as discarding cards are identical regardless of player type, and thus shouldn’t have the same implementation in multiple classes.

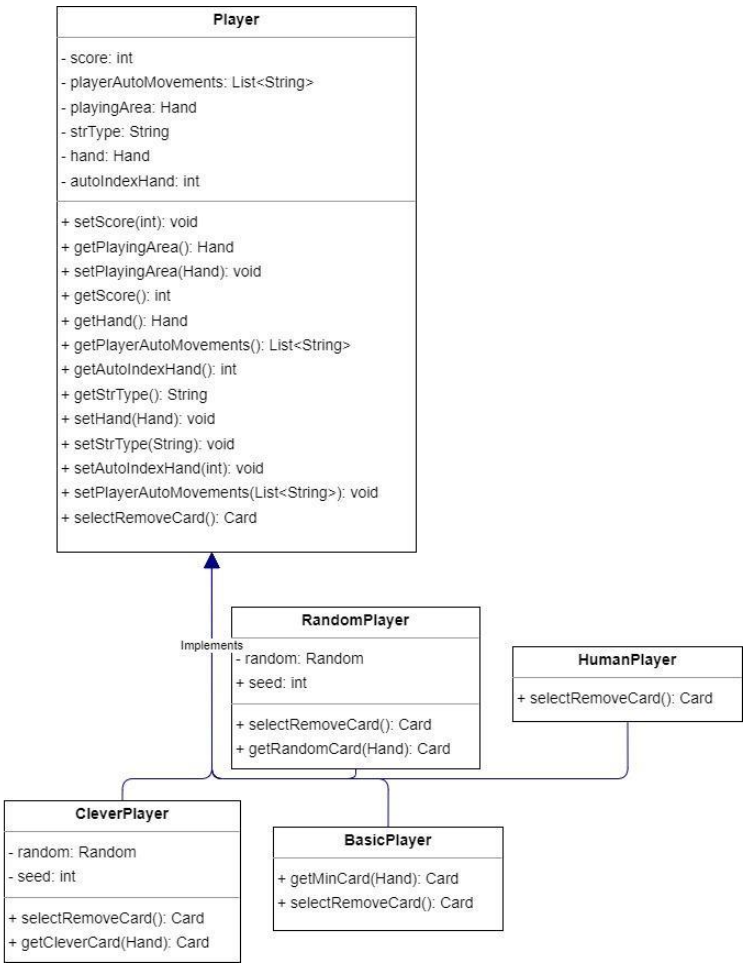

Figure 2

Another change we have implemented was using an indirection factory to create the players, although we decided not to utilise a singleton factory as multiple players may be of the same type. The use of a factory allows for player instantiation logic to be centralised; thus the logic isn’t scattered around the code base, but instead maintained in ‘PlayerFactory’, allowing for easier maintenance and extensibility.

Figure 3

A ‘CardManager’ class was also implemented for handling the card logic; we decided to further separate the card and deck management responsibility from the ‘LuckyThirdteen’ class to increase cohesion and lower coupling. The class is responsible for the handling of the deck and cards, keeping track of the current deck, dealing cards to players, and providing information of the cards.This naked ‘CardManager’ the information expert of card based operations, which simplifies modifications and maintenance of card based logic.

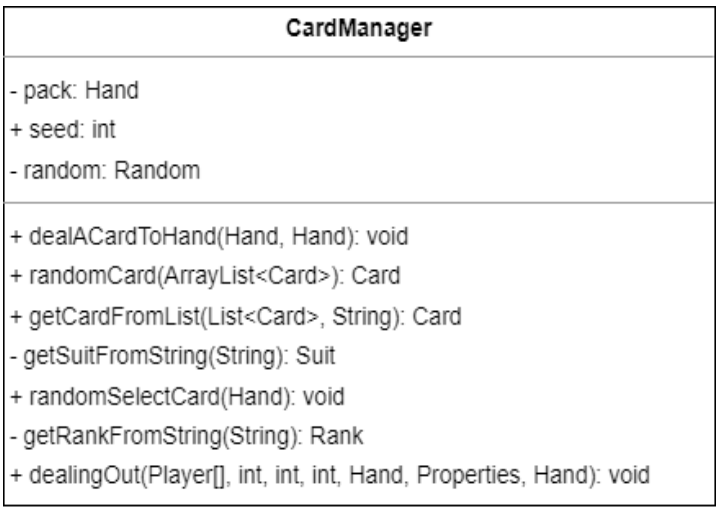

Figure 4

We also decided on creating a new ‘algorithm’ class for handling and evaluating all the different scoring algorithms, by doing so we increase cohesion, as all the scoring logic is centralised in one class, and thus maintaining and extending the logic is simplified, it also has the further benefit of being easier to understand, as the different scoring logics are not scattered throughout the code base. The new ‘algorithm’ class also works with the Facade pattern, as it provides a simple interface to calculate the score at the end of the game through calculateScoreEndOfRound(), with the complexities abstracted away, promoting ease of use and reducing dependencies. It also follows the Information expert principle as it houses all the required information and holds the responsibility of calculating the scores.

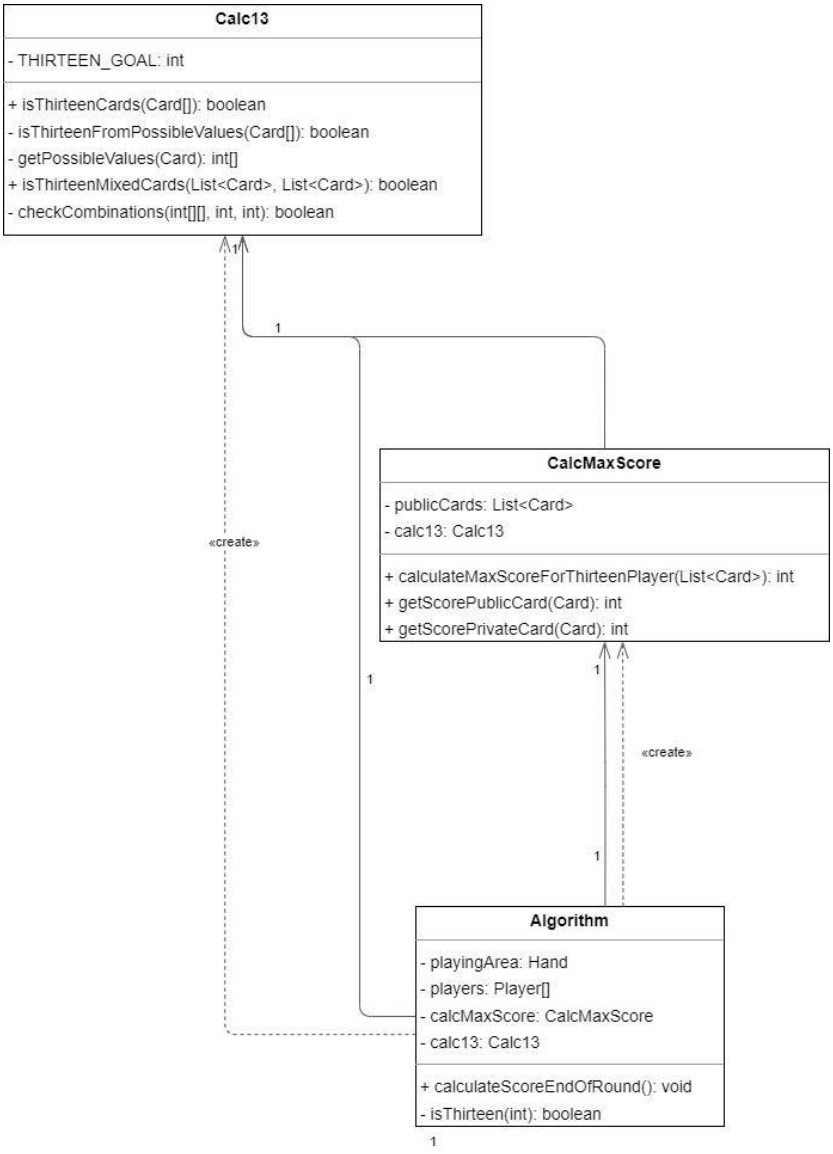

Figure 5

To further enhance cohesion maintain clear responsibilities, we have also introduced a ‘drawer’ class, as it further heightens cohesion by taking responsibility for the rendering of the game state, thus we can separate the visual and logical components of the simulation into their own specialised classes, additionally a ‘logData’ class is also introduced as it takes responsibility for keeping track of the logs of the game, separating the logging logic from the game logic.

Clever Bot Implementation:

For our implementation of the ‘Clever Bot’, we are going with a methodical approach that attempts to anticipate future game states and make decisions based on that to optimise its chances of success.

1) Card Matching and Specific Discarding

Summary: The ‘Clever Bot’ begins by checking if the drawn card matches the rank of any card in the bot’s current hand. If there is a match, the bot then compares the suits of both cards, and keeps the card with the higher value, discarding the lower. This ensures that the bot maintains high-value cards in terms of suits, prioritising getting a higher score during the scoring phase rather than going for 13.

Unlike the ‘Basic Bot’, which always discards the lowest value card in terms of rank, our ‘Clever Bot’ implementation selects card based on both rank and suit, maximising the potential scoring opportunities, as retaining higher suits could leverage a better score through the multiplier effects in the scoring algorithms, something that the ‘Basic Bot’ fails to capitalise on.

```
// Rule 1: Check if the drawn card matches any card in the hand
if (lastRank.getRankCardValue() == rank1.getRankCardValue()) {
    if (lastSuit.getMultiplicationFactor() > suit1.getMultiplicationFactor()) {
        return privateCard1;
    } else {
        return lastCard;
    }
}
if (lastRank.getRankCardValue() == rank2.getRankCardValue()) {
    if (lastSuit.getMultiplicationFactor() > suit2.getMultiplicationFactor()) {
        return privateCard2;
    } else {
        return lastCard;
    }
}
```

2) Evaluating Combinations

Summary: The ‘Clever Bot’ also keeps track of the current combination of public and private cards, constantly checking if there’s a possible 13 value combination. If there’s no immediate combination, the bot considers the possible chance of the next drawn card providing the 13 value combination.

The specific combinations are:

1) Sum of hand card 1 & hand card 2
2) Sum of hand card 1 & public card 1
3) Sum of hand card 1 & public card 2
4) Sum of hand card 2 & public card 1
5) Sum of hand card 2 & public card 2

If any combination meets the target number of 13, we play the combination. If none of the five combinations give a value of 13, we will decide based on the drawn card and hand/public cards.

```java
// Rule 2: Save combinations of hand cards and public cards
List<Card> publicCards = this.getPlayingArea().getCardList();
Rank publicRank1 = (Rank) publicCards.get(0).getRank();
Rank publicRank2 = (Rank) publicCards.get(1).getRank();
List<Integer> sumList = new ArrayList<>();

sumList.add(rank1.getRankCardValue() + rank2.getRankCardValue());
sumList.add(rank1.getRankCardValue() + publicRank1.getRankCardValue());
sumList.add(rank1.getRankCardValue() + publicRank2.getRankCardValue());
sumList.add(rank2.getRankCardValue() + publicRank1.getRankCardValue());
sumList.add(rank2.getRankCardValue() + publicRank2.getRankCardValue());

// Check if any combination equals 13
for (Integer sum : sumList) {
    if (sum == 13) {
        return lastCard;
    }
}
```

This is yet another difference between the ‘Clever Bot’ and the ‘Basic Bot’, as the basic implementation does not check if any combination is equal to 13 and thus holds onto the cards to maximize its end of game score; this however was implemented in the ‘Clever Bot’.

3) Optimization against the Target Number (13)

Summary: If there’s no current combination of card value 13, the clever bot calculates the new potential sums with drawn cards, comparing it against the previous combinations.

We save the new combinations of the drawn cards and public cards; the specific combinations are:

1) Sum of drawn card and hand card 1
2) Sum of drawn card and hand card 2
3) Sum of drawn card and public card 1
4) Sum of drawn card and public card 2

Compare the absolute values of the sums minus the target number for all combinations, if any of the four combinations have a smaller absolute value than the previous five combinations, we find that the drawn card is more valuable. If the absolute value of combination 1 is less than the values of previous combinations, we discard hand card 2, and vice versa for combination 2. If the absolute value of combination 3 or 4 is less than the previous five combinations, we randomly discard a hand card.

```java
// Rule 3: If no card combinations equal 13
List<Integer> newSumList = new ArrayList<>();
newSumList.add(lastRank.getRankCardValue() + rank1.getRankCardValue());
newSumList.add(lastRank.getRankCardValue() + rank2.getRankCardValue());
newSumList.add(lastRank.getRankCardValue() + publicRank1.getRankCardValue());
newSumList.add(lastRank.getRankCardValue() + publicRank2.getRankCardValue());

// Compare absolute values of sums minus 13
int minDiff = Integer.MAX_VALUE;
Card cardToDiscard = null;
for (int i = 0; i < newSumList.size(); i++) {
    int diff = Math.abs(newSumList.get(i) - 13);
    if (diff < minDiff) {
        minDiff = diff;
        if (i == 0) cardToDiscard = privateCard2;
        else if (i == 1) cardToDiscard = privateCard1;
        else cardToDiscard = random.nextInt( bound: 2) == 0 ? privateCard1 : privateCard2;
    }
}

return cardToDiscard != null ? cardToDiscard : lastCard;
```

This is similar to the explanation above in section (2) as the ‘Clever Bot’ implementation is able to determine if it wants to play for the target number of 13 to get 100 points or to play for the highest score if it deems that getting the target number is unlikely. This is in contrast to ‘Basic Bot’ which only plays for getting the highest end of game score.