SWEN30006 Project 2

Erich Wiguna 1389444

Jessica Aurelia Wijaya 1364255

Reagen Purnama 1389448

Introduction

This report provides a detailed analysis of the design changes made to the LuckyThirdteen project. The changes involve refactoring the existing codebase to improve maintainability, readability, and scalability. The report also identifies design alternatives and justifies the chosen design decisions with respect to design patterns and principles (GoF and GRASP). Additionally, it outlines the strategy for implementing a clever computer player.

Analysis of the Current Design

The original design of the LuckyThirdteen game was not modular, with the LuckyThirdteen class handling all aspects of the game, including card dealing, player actions, score calculations, and game logging. This design had several significant issues. Firstly, the lack of modularity meant that all functionalities were implemented within a single class, resulting in a large and complex class that was difficult to maintain and understand. The tight coupling of game logic with player logic made it challenging to add new player types or modify existing ones. The LuckyThirdteen class also violated the GRASP Principle - Responsibility by handling multiple responsibilities, such as game logic, card dealing, and logging, resulting in a bloated class. Consequently, testing and maintaining the code was challenging, as any changes in one part of the code could have unintended side effects on other parts. Lastly, the design's limited extensibility meant that adding new features required significant changes to the existing codebase.

Refactored Design

The refactored design follows a modular approach, where each functionality is encapsulated within specific classes. This approach promotes maintainability by reducing the size and complexity of individual classes, making the code easier to understand and modify. It also enhances scalability, as new features can be added by creating new classes or extending existing ones without affecting other parts of the system. The downside of this approach is that it requires a more complex initial setup and a good understanding of design patterns to implement effectively. However, the long-term benefits of reduced complexity and improved maintainability far outweigh the initial overhead.

Detailed Analysis

The refactored design incorporates several design patterns to achieve its goals. The Factory Pattern is implemented in the PlayerFactory class to simplify the creation of different player types based on configuration. This pattern adheres to the Polymorphism (Open Closed Principle) by allowing the system to be extended with new player types without modifying existing code. The PlayerFactory class centralizes the creation logic, making it easier to manage and update.

The Strategy Pattern is employed in the scoring mechanisms, using the CompositeScoreStrategy and individual score strategies (PrivateCardStrategy, PrivatePublicCardStrategy, TwoPrivateTwoPublicCardStrategy) to encapsulate different algorithms for score calculation. This pattern allows for flexible and interchangeable scoring rules, promoting the GRASP principles of Low Coupling and High Cohesion by separating the scoring logic from other parts of the game logic. It also adheres to the Protected Variations (Open/Closed) Principle by enabling the addition of new scoring strategies without modifying existing code.

The Singleton Pattern ensures that only one instance of the PlayerFactory exists, providing a global point of access and ensuring consistent player creation throughout the game. This pattern is used to manage the creation of player instances, centralizing the logic and making it easier to control and update.

Applying GRASP principles further enhances the design. High Cohesion is achieved by assigning each class a well-defined role, such as CardDealer handling all card-related operations, GameLogger managing logging, and LuckyThirdteen orchestrating the game flow. This makes each class responsible for a single aspect of the game's functionality, leading to clearer and more maintainable code.

Low Coupling is maintained by minimizing dependencies between classes, which interact through well-defined interfaces. This reduces the likelihood of changes in one part of the system affecting others, enhancing maintainability. For example, the game does not directly instantiate player objects; instead, it uses the PlayerFactory to create them, ensuring that player creation is decoupled from the game logic.

The Controller pattern is evident in the LuckyThirdteen class, which acts as a central point for managing the game flow. By centralizing control logic, this pattern makes it easier to manage the overall game state and flow, improving the readability and maintainability of the code.

Clever Computer Player Strategy

The initial strategy for the clever computer player was heuristic-based, using simple rules such as selecting the card with the highest or lowest value. While this approach provided a basic level of intelligence, it lacked the sophistication to make the player truly competitive.

The enhanced strategy involves tracking played cards to make more informed decisions about the remaining cards. By keeping track of the cards that have been played, the clever player can calculate the probabilities of achieving a score of thirteen with the remaining cards, making decisions based on this information. This probabilistic reasoning allows the player to make more strategic choices, increasing the chances of winning.

Additionally, the clever player implements a minimax algorithm with a limited depth to evaluate possible future states of the game and select the optimal move. The minimax algorithm is commonly used in game theory to minimize the possible loss for a worst-case scenario. By evaluating future states of the game, the clever player can anticipate the opponent's moves and make decisions that maximize the chances of winning. In this implementation, the clever player selects the card with the highest value to discard. This is a basic strategy that can be improved by incorporating more sophisticated decision-making algorithms, such as probabilistic reasoning and the minimax algorithm.

Software Models

1. Domain Class Diagram

   To facilitate the discussion of our design, we provide a domain class diagram capturing the domain concepts relating to computer players (Refer to Appendix 1). The Player class is a central entity with various subclasses representing different player types (HumanPlayer, BasicPlayer, CleverPlayer, etc.). The PlayerFactory class is responsible for creating instances of these players.

2. Static Design Model

   The static design model documents our design and implementation relating to computer players (Refer to Appendix 2). It includes the class structure and the relationships between the classes. The diagram highlights the use of the Factory Pattern in the PlayerFactory class, which creates different types of players. It also shows the Strategy Pattern in the CompositeScoreStrategy and its associated strategies for score calculation.

3. Sequence Diagram

   To further support our explanation, we've crafted a sequence diagram to illustrate the interactions between the main components during the game initialization and play phases (Refer to Appendix 3). The diagram shows how the LuckyThirdteen class interacts with other components such as CardDealer, PlayerFactory, and ScoreStrategy during the game's lifecycle. It provides a dynamic view of the system's behavior, complementing the static design model.

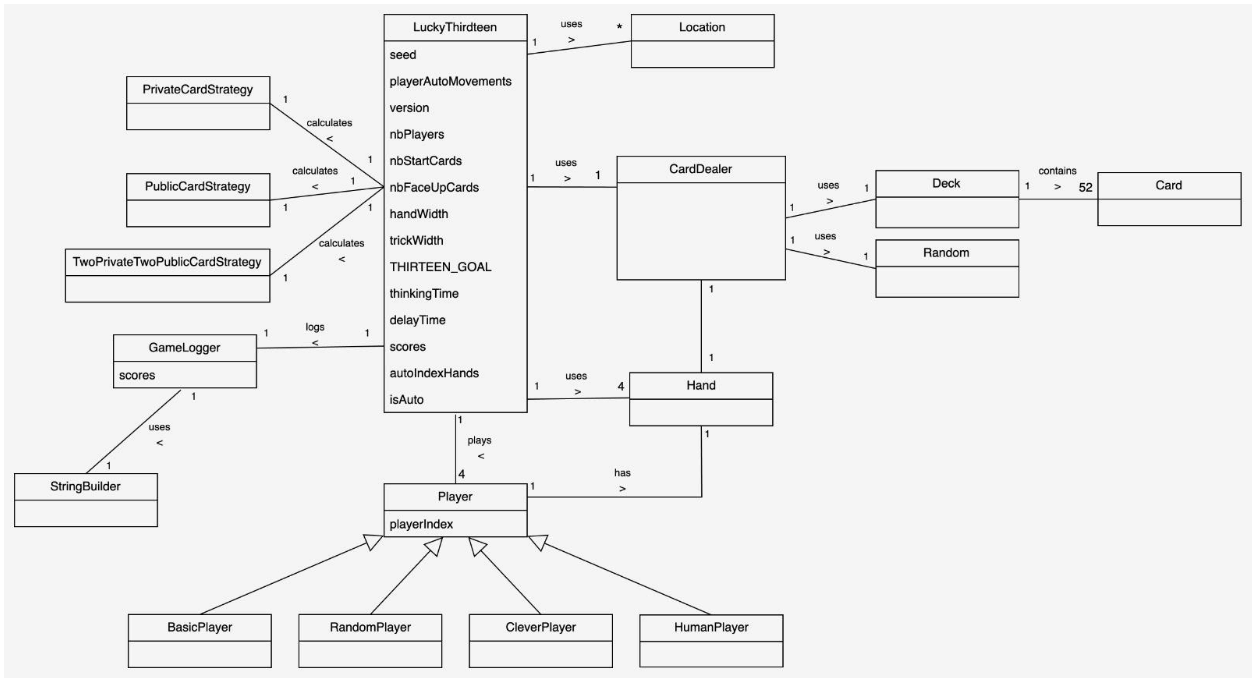
Conclusion

The refactoring of the LuckyThirdteen project has significantly improved the codebase's structure, making it more maintainable, readable, and extendable. The application of design patterns and principles has led to a cleaner separation of concerns and a more modular design. The enhanced strategy for the clever computer player promises to provide a more challenging and engaging gameplay experience. Future work includes further refining the clever player strategy and exploring additional design patterns to enhance other aspects of the game.

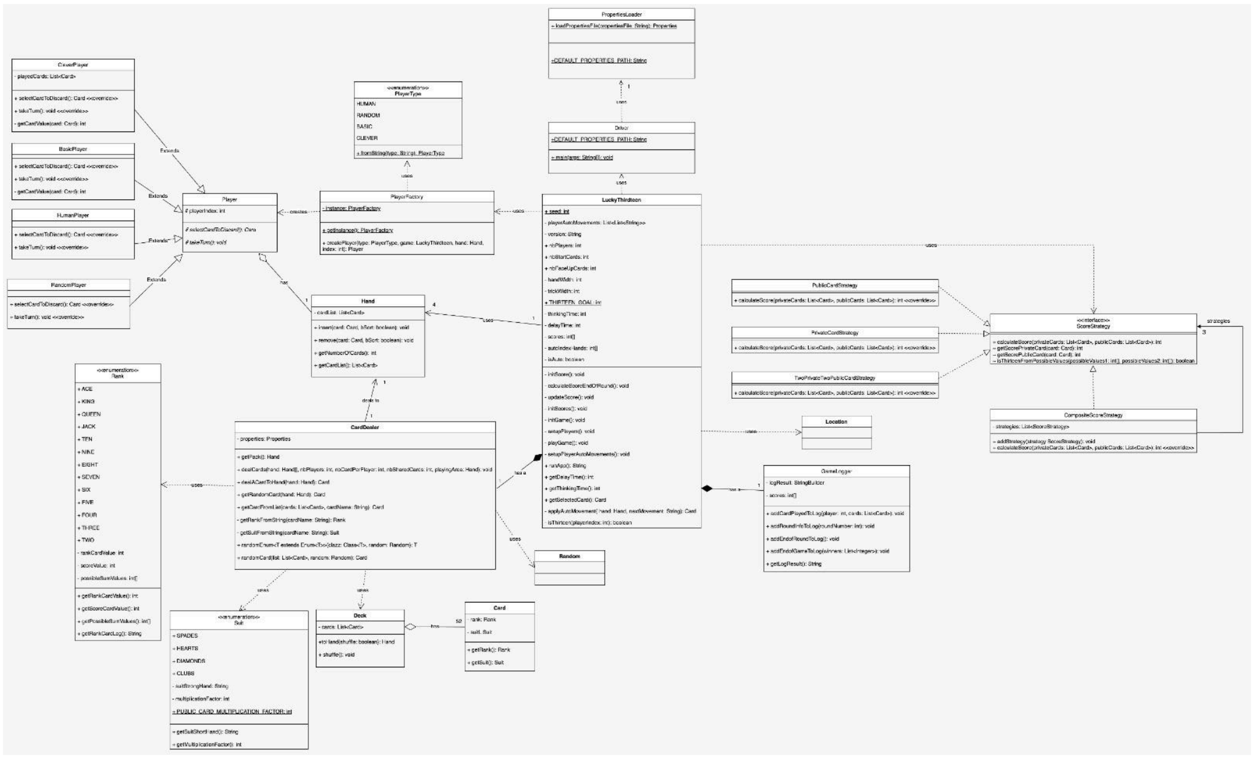
By implementing these changes, we have transformed a difficult-to-maintain codebase into a modular, maintainable, and scalable system. This new design not only improves the current implementation but also provides a solid foundation for future enhancements and extensions. The use of design patterns and principles has been instrumental in achieving these improvements, demonstrating the value of applying these concepts in software development.

Appendices:

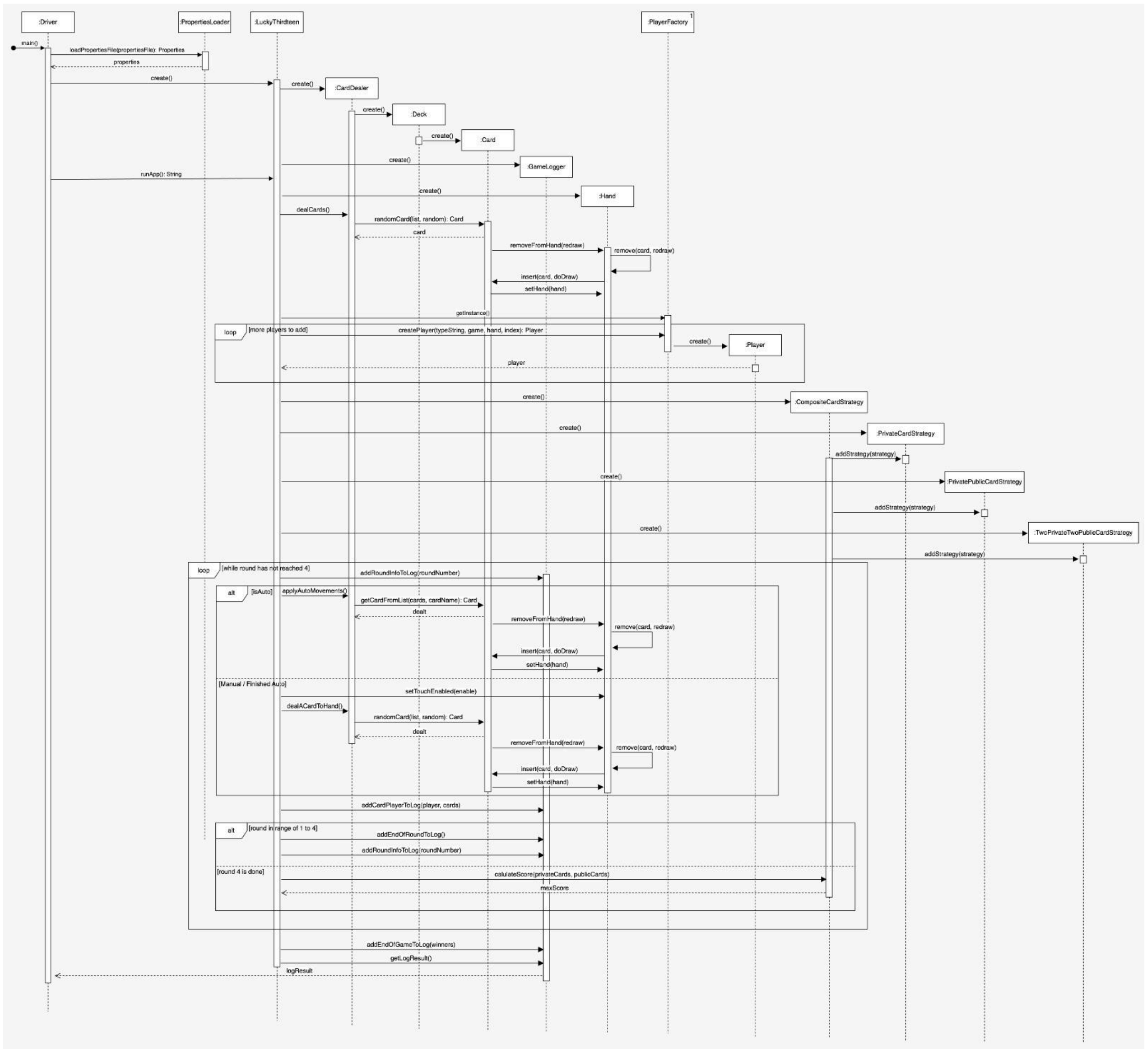
Appendix 1 - Domain Class Diagram



Appendix 2 - Design Class Diagram


PropertiesLoader
+ loadPropertiesFile(propertiesFile: String): Properties
+DEFAULT_PROPERTIES_PATH: String
uses
1
Driver
+DEFAULT_PROPERTIES_PATH: String
+ main(args: String[]): void
uses
CleverPlayer
- playedCards: List<Card>
+ selectCardToDiscard(): Card <<override>>
+ takeTurn(): void <<override>>
- getCardValue(card: Card): int
BasicPlayer
+ selectCardToDiscard(): Card <<override>>
+ takeTurn(): void <<override>>
- getCardValue(card: Card): int
HumanPlayer
+ selectCardToDiscard(): Card <<override>>
+ takeTurn(): void <<override>>
RandomPlayer
+ selectCardToDiscard(): Card <<override>>
+ takeTurn(): void <<override>>
Extends
Extends
Extends
Extends
<<enumeration>>
PlayerType
HUMAN
RANDOM
BASIC
CLEVER
+ fromString(type: String): PlayerType
uses
Player
# playerIndex: int
# selectCardToDiscard(): Card
# takeTurn(): void
creates
PlayerFactory
- instance: PlayerFactory
+ getInstance(): PlayerFactory
+ createPlayer(type: PlayerType, game: LuckyThirdteen, hand: Hand, index: int): Player
uses
LuckyThirdteen
+ seed: int
- playerAutoMovements: List<List<String>>
- version: String
+ nbPlayers: int
+ nbStartCards: int
+ nbFaceUpCards: int
- handWidth: int
- trickWidth: int
+ THIRTEEN_GOAL: int
- thinkingTime: int
- delayTime: int
- scores: int[]
- autoIndex-Hands: int[]
- isAuto: boolean
- initScore(): void
- calculateScoreEndOfRound(): void
- updateScore(): void
- initScores(): void
- initGame(): void
- setupPlayers(): void
- playGame(): void
- setupPlayerAutoMovements(): void
+ runApp(): String
+ getDelayTime(): int
+ getThinkingTime(): int
+ getSelectedCard(): Card
- applyAutoMovement(hand: Hand, nextMovement: String): Card
- isThirteen(playerIndex: int): boolean
has
1
Hand
- cardList: List<Card>
+ insert(card: Card, bSort: boolean): void
+ remove(card: Card, bSort: boolean): void
+ getNumberOfCards(): int
+ getCardList(): List<Card>
4
uses
1
deals to
1
1
uses
PublicCardStrategy
+ calculateScore(privateCards: List<Card>, publicCards: List<Card>): int <<override>>
PrivateCardStrategy
+ calculateScore(privateCards: List<Card>, publicCards: List<Card>): int <<override>>
TwoPrivateTwoPublicCardStrategy
+ calculateScore(privateCards: List<Card>, publicCards: List<Card>): int <<override>>
<<interface>>
ScoreStrategy
~ calculateScore(privateCards: List<Card>, publicCards: List<Card>): int
~ getScorePrivateCard(card: Card): int
~ getScorePublicCard(card: Card): int
~ isThirteenFromPossibleValues(possibleValues1: int[], possibleValues2: int[]): boolean
strategies
3
CompositeScoreStrategy
- strategies: List<ScoreStrategy>
~ addStrategy(strategy ScoreStrategy): void
~ calculateScore(privateCards: List<Card>, publicCards: List<Card>): int <<override>>
uses
Location
<<enumeration>>
Rank
+ ACE
+ KING
+ QUEEN
+ JACK
+ TEN
+ NINE
+ EIGHT
+ SEVEN
+ SIX
+ FIVE
+ FOUR
+ THREE
+ TWO
- rankCardValue: int
- scoreValue: int
- possibleSumValues: int[]
+ getRankCardValue(): int
+ getScoreCardValue(): int
+ getPossibleSumValues(): int[]
+ getRankCardLog(): String
uses
CardDealer
- properties: Properties
+ getPack(): Hand
+ dealCards(hand: Hand[], nbPlayers: int, nbCardPerPlayer: int, nbSharedCards: int, playingArea: Hand): void
+ dealACardToHand(hand: Hand): Card
+ getRandomCard(hand: Hand): Card
+ getCardFromList(cards: List<Card>, cardName: String): Card
- getRankFromString(cardName: String): Rank
- getSuitFromString(cardName: String): Suit
+ randomEnum<T extends Enum<T>>(clazz: Class<T>, random: Random): T
+ randomCard(list: List<Card>, random: Random): Card
has a
1
GameLogger
- logResult: StringBuilder
- scores: int[]
+ addCardPlayedToLog(player: int, cards: List<Card>): void
+ addRoundInfoToLog(roundNumber: int): void
+ addEndofRoundToLog(): void
+ addEndofGameToLog(winners: List<Integer>): void
+ getLogResult(): String
has a
1
uses
Random
uses
uses
<<enumeration>>
Suit
+ SPADES
+ HEARTS
+ DIAMONDS
+ CLUBS
- suitShortHand: String
- multiplicationFactor: int
+ PUBLIC_CARD_MULTIPLICATION_FACTOR: int
+ getSuitShortHand(): String
+ getMultiplicationFactor(): int
Deck
- cards: List<Card>
+toHand(shuffle: boolean): Hand
+ shuffle(): void
has
52
Card
- rank: Rank
- suit: Suit
+ getRank(): Rank
+ getSuit(): Suit

Appendix 3 - Dynamic Design Diagram


:Driver
:PropertiesLoader
:LuckyThirdteen
:PlayerFactory
1
main()
loadPropertiesFile(propertiesFile): Properties
properties
create()
create()
:CardDealer
create()
:Deck
create()
:Card
create()
:GameLogger
runApp(): String
create()
:Hand
dealCards()
randomCard(list, random): Card
card
removeFromHand(redraw)
remove(card, redraw)
insert(card, doDraw)
setHand(hand)
getInstance()
loop
[more players to add]
createPlayer(typeString, game, hand, index): Player
create()
:Player
player
create()
:CompositeCardStrategy
create()
:PrivateCardStrategy
addStrategy(strategy)
create()
:PrivatePublicCardStrategy
addStrategy(strategy)
create()
:TwoPrivateTwoPublicCardStrategy
addStrategy(strategy)
loop
[while round has not reached 4]
addRoundInfoToLog(roundNumber)
alt
[isAuto]
applyAutoMovements()
getCardFromList(cards, cardName): Card
dealt
removeFromHand(redraw)
remove(card, redraw)
insert(card, doDraw)
setHand(hand)
[Manual / Finished Auto]
setTouchEnabled(enable)
dealACardToHand()
randomCard(list, random): Card
dealt
removeFromHand(redraw)
remove(card, redraw)
insert(card, doDraw)
setHand(hand)
addCardPlayerToLog(player, cards)
alt
[round in range of 1 to 4]
addEndOfRoundToLog()
addRoundInfoToLog(roundNumber)
[round 4 is done]
calulateScore(privateCards, publicCards)
maxScore
addEndOfGameToLog(winners)
getLogResult()
logResult