HiFive Game Project Report

SWEN30006 Project 2

Team 11

Miles Li	1450710	`yuemingl3@student.unimelb.edu.au`
Skylar Khant	1450754	`kyishink@student.unimelb.edu.au`
Ngoc Thanh Lam Nguyen	1450800	`ngocthanhlam@student.unimelb.edu.au`

Workshop: Wed 11 a.m.
Tutors: Harry Wang, Khang Vo

Abstract

This report presents the design, implementation, and refactoring process of the HiFive game, a card game originally implemented with significant software design flaws. The refactoring efforts involved applying object-oriented principles, design patterns, and the GRASP principles to achieve a more modular, maintainable, and extendable system. Key improvements include the use of factory patterns for player entity creation, strategy and composite patterns for scoring calculations, and the decorator pattern for implementing wild card functionality. Additionally, a Clever Computer Player was developed, utilizing a greedy algorithm to maximize game scores within the limitations of the HiFive gameplay structure. This report aims to demonstrate how these enhancements have improved code readability, reduced complexity, and made future extensions easier.

Contents

Chapter 1

Introduction

The original implementation of the HiFive game demonstrated several critical issues related to software design, including high coupling, lack of encapsulation, and poor object-oriented design. All game functionalities were implemented in a monolithic class, leading to an unmanageable codebase. The aim of this project was to address these problems by refactoring the code to adhere to solid software engineering principles. We divided the HiFive class into smaller specialized components and applied various design patterns, ensuring that the system was both modular and maintainable. This report outlines the key shortcomings of the original design, the refactoring steps taken, and the improvements made to the game, including the introduction of a Clever Computer Player designed to optimize gameplay using an efficient strategy.

Chapter 2

Original Design Analysis

The original version of the HiFive game code is a classic example of poor design in software engineering. Below are the main problems identified in the original design:

2.1 Lack of Object-Oriented Approach

1. The class HiFive takes on nearly all the responsibilities in the game, including game initialization, player management, score calculation, and graphical rendering. This type of class, violates the Single Responsibility Principle of Oriented Object Programming and results in a massive and unmaintainable class.

2. Combining all the functionalities in a single class makes the code highly unreadable and difficult to maintain.

2.2 Lack of Proper Abstraction and Encapsulation

1. Each game entity (such as player, card, and scoring system) should have been implemented as separate classes with distinct responsibilities. However, in the original design, everything is bundled within a single class, leading to very high coupling.

2. For example, player behaviors, card management, and game state tracking are all handled within the same class. The absence of encapsulation makes the system difficult to extend and maintain. Adding or removing features becomes extremely challenging.

2.3 Violation of GRASP Principles

1. According to the Information Expert Principle, the responsibility for handling data should reside with the class that has the information needed. In the current code, methods such as 'initScore', 'calculateScoreEndOfRound', and 'updateScore' should be in a "Player" class but are instead placed in the main game class.

2. This leads to poor cohesion and an illogical coupling between responsibilities, making the code harder to maintain.

2.4 Lack of Polymorphism and Extendibility

1. The current implementation lacks any use of polymorphism or extendibility, especially when handling player behaviors. All player types, such as human and computer players, are hardcoded within the class.

2. For instance, the ’playGame’ method includes hardcoded logic for handling human and computer player turns, making it difficult to add new player types or modify existing behaviors without rewriting parts of this massive method.

2.5 Global Variables and High Coupling

1. The class relies heavily on global variables, such as ’scoreActors’, ’scores’, and ’hands’, which increases coupling between methods and makes them dependent on each other.

2. Global variables can be modified by multiple methods, leading to complex dependencies and increased potential for errors.

Chapter 3

Refactoring, Improvement, and Extension

To enhance the design of the HiFive game, we refactored the original monolithic HiFive class by dividing it into several specialized classes, each representing a distinct game entity in accordance with the Single Responsibility Principle. Additionally, to ensure that the project is extendable and adheres to the GRASP principles, we incorporated design patterns to create a more modular, maintainable, and cohesive architecture. This approach allows for clearer separation of concerns and improves the overall flexibility and scalability of the system.

3.1 Player Entities Functionalities Implementation and Improvements

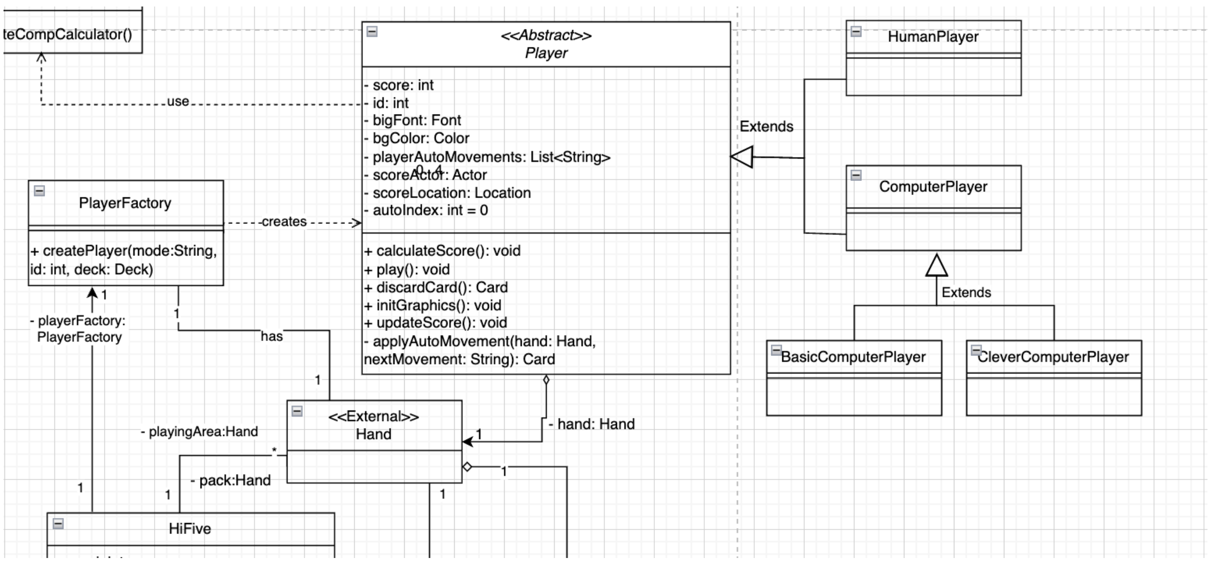

Figure 3.1: Player Entities implementation

3.1.1 Use of Factory Pattern

To improve the initialisation process of player entities, we applied the Factory Pattern through the introduction of a '**PlayerFactory**' class. The Factory Pattern facilitates the creation of different player types such as '**HumanPlayer**', '**BasicComputerPlayer**', and '**CleverComputerPlayer**'. By using a '**PlayerFactory**', we encapsulate the instantiation logic and keep it separate from the core gameplay code, promoting modularity and reducing code duplication. The '**PlayerFactory**' class contains a '**createPlayer()**' method that takes the player's mode, ID, and the deck to instantiate the corresponding player entity. This allows the code to be extendable with new player types while ensuring a uniform way of creating them.

3.1.2 Pure Fabrication and Information Expert Principles

The '**PlayerFactory**' class is a good example of a pure fabrication, meaning that it doesn't represent a concept from the problem domain but is rather a utility class introduced to achieve better modularity. The factory method is also in line with the Information Expert principle of GRASP, where the responsibility for creating player entities is assigned to a dedicated class. This makes the HiFive class simpler and focuses on game control logic instead of also handling player creation.

3.1.3 Inheritance on Player Classes

We refactored the player-related functionality into an abstract class, 'Player', which provides a shared set of properties and methods applicable to all players. The Player class serves as a base, and different player behaviors are implemented through inheritance in derived classes such as '**HumanPlayer**' and '**ComputerPlayer**'. The '**ComputerPlayer**' class is further extended to provide specialized player types like '**BasicComputerPlayer**' and '**CleverComputerPlayer**'.

This use of inheritance not only reduces code duplication but also enables polymorphism, where each player type can implement its play() method in a way that best fits its strategy. This results in better code organization, where player-specific behaviors are encapsulated within their respective classes.

3.1.4 Separation of Player Logic and Gameplay

In the refactored version, the responsibility for actions related to player decisions and behavior (such as scoring, discarding cards, and taking turns) is now fully encapsulated within the '**Player**' classes. Previously, all such logic was cramped within a monolithic '**HiFive**' class, leading to high coupling and poor readability. By breaking these responsibilities into separate classes, each class now has a single responsibility, aligning with the Single Responsibility Principle. This refactoring also ensures that adding or modifying player behavior is much simpler, as changes are limited to a single class without unintended side effects elsewhere in the code.

3.2 Scoring Algorithms Handling

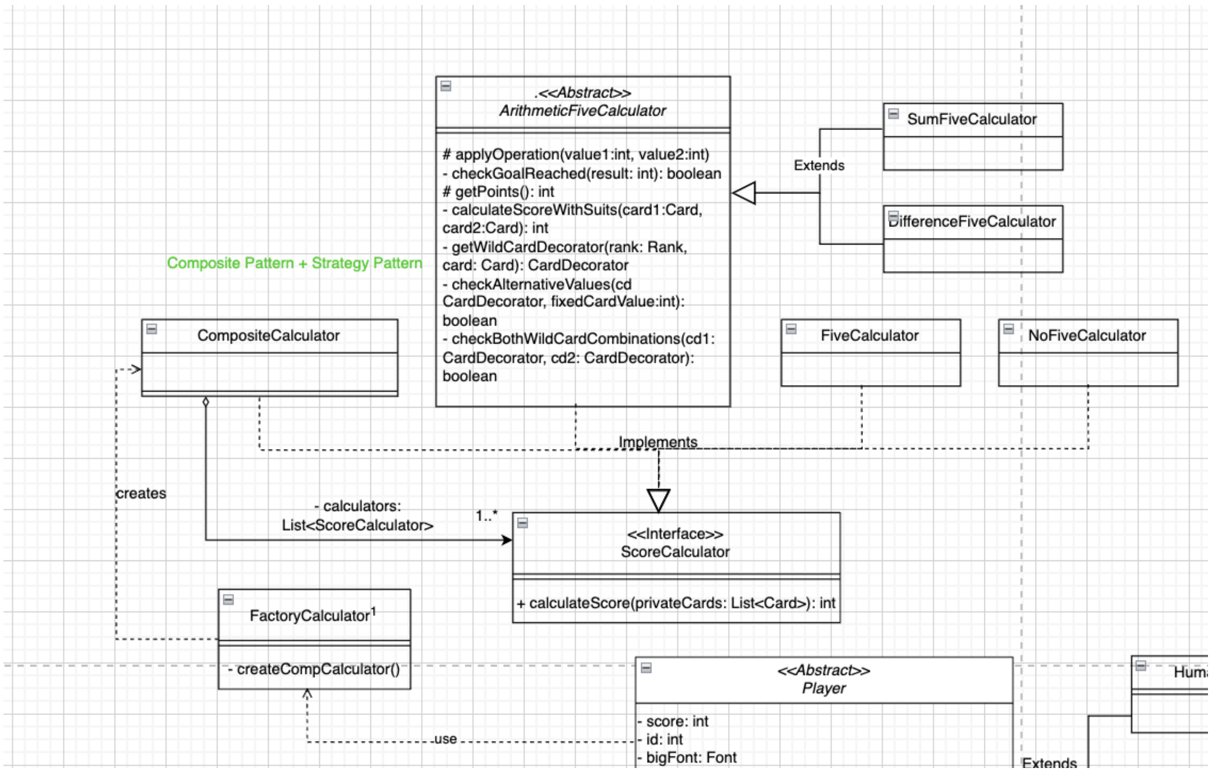
.<<Abstract>>
ArithmeticFiveCalculator
# applyOperation(value1:int, value2:int)
- checkGoalReached(result: int): boolean
# getPoints(): int
- calculateScoreWithSuits(card1:Card, card2:Card): int
- getWildCardDecorator(rank: Rank, card: Card): CardDecorator
- checkAlternativeValues(cd CardDecorator, fixedCardValue:int): boolean
- checkBothWildCardCombinations(cd1: CardDecorator, cd2: CardDecorator): boolean
SumFiveCalculator
Extends
DifferenceFiveCalculator
Composite Pattern + Strategy Pattern
CompositeCalculator
FiveCalculator
NoFiveCalculator
Implements
creates
- calculators: List<ScoreCalculator>
1..*
<<Interface>>
ScoreCalculator
+ calculateScore(privateCards: List<Card>): int
FactoryCalculator1
- createCompCalculator()
<<Abstract>>
Player
- score: int
- id: int
- bigFont: Font
use
Extends
Hum

Figure 3.2: Scoring Algorithm Entities

3.2.1 Use of Strategy Pattern

To handle different scoring rules, we applied the Strategy Pattern to encapsulate four different scoring algorithms: **FiveCalculator**, **SumFiveCalculator**, **DifferenceFiveCalculator**, and **NoFiveCalculator**. Each of these classes implements the **ScoreCalculator** interface, which defines a common method for calculating scores. The Strategy Pattern allows us to easily switch between different scoring rules based on the current game situation, making the system more flexible and adaptable. This approach also makes it easy to add new scoring rules in the future, as new strategies can simply be added by implementing the **ScoreCalculator** interface.

3.2.2 Use of Composite Pattern

In order to determine the highest possible score for each player in a round, we need to apply all four scoring algorithms and select the maximum score. For this purpose, we used the Composite Pattern to create a **CompositeCalculator** class that contains a list of **ScoreCalculator** objects, including all individual scoring calculators. The **CompositeCalculator** is responsible for delegating the score calculation to each individual calculator and determining the highest score.

The Composite Pattern helps us to manage multiple score calculators as a single entity, which makes it easier to add or remove scoring algorithms in the future. Moreover, it decouples the scoring logic from the **Player** classes, resulting in a more modular and reusable design. By delegating the scoring calculation to the **CompositeCalculator**, the **Player** class is simplified, as it only needs to interact with the composite to get the final score, rather than managing multiple scoring mechanisms directly.

3.2.3 Integration with ArithmeticFiveCalculator

We introduced an abstract class, **ArithmeticFiveCalculator**, that provides common functionality for scoring calculations involving arithmetic operations, such as sums and differences of card values. The **SumFiveCalculator** and **DifferenceFiveCalculator** extend **ArithmeticFiveCalculator**, inheriting the shared methods for calculating and verifying card values, while providing specific implementations for their respective scoring rules. This use of inheritance promotes code reuse and helps to avoid duplication of common logic across different score calculators.

3.2.4 Singleton on FactoryCalculator

In order to ensure consistency between calculators and the calculator factory, and reduce memory usage, we applied the Singleton Pattern to the **FactoryCalculator** class, which is responsible for creating calculators. By implementing the Singleton Pattern, we ensure that there is only one instance of the **FactoryCalculator** throughout the application, which guarantees that all calculators are created consistently and avoids unnecessary memory consumption.

3.3 Wild Card Functionalities Implementation

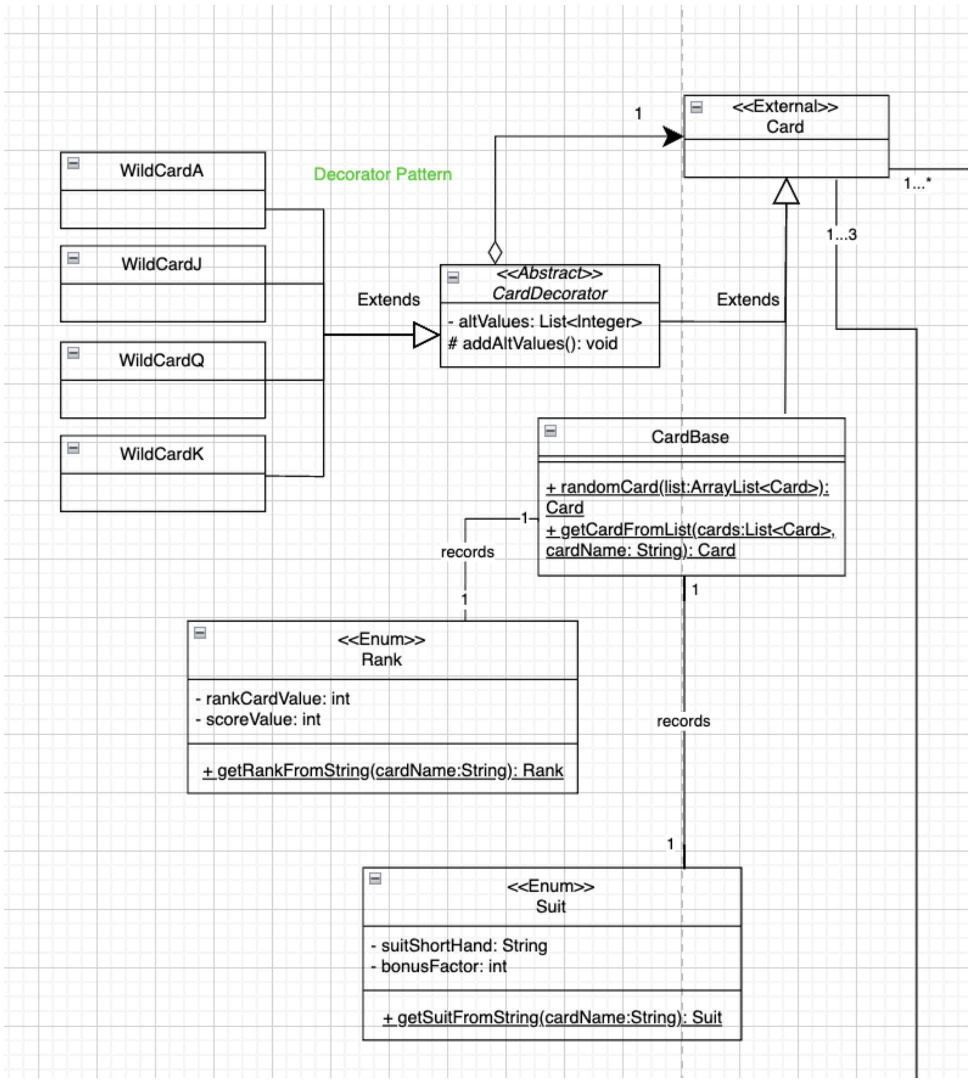

Figure 3.3: Design Model Diagram for Wild Cards Part

3.3.1 Use of Decorator Pattern

To implement the Wild Card functionalities in the HiFive game, we used the Decorator Pattern. The Decorator Pattern allows us to add new behaviors to existing card objects without modifying their original code, adhering to the *open-closed principle* of software design. By applying this pattern, we were able to enhance regular card instances with new characteristics that represent wild cards, thereby adding alternative values to the cards.

The **CardDecorator** class serves as the abstract decorator, extending the base Card class. It introduces a new property, **altValues**, which stores the list of possible alternative values for a decorated card. This enables the cards to take on multiple values based on game rules, making them more versatile in different scoring situations.

The specific wild cards, such as **WildCardA**, **WildCardJ**, **WildCardQ**, and **WildCardK**, extend from **CardDecorator** and override methods to provide additional value configurations based on game requirements.

3.3.2 Integration with CardBase and Enum Classes

The **CardBase** class contains utility methods to interact with card objects. These methods facilitate the selection of cards during the game, regardless of whether they are decorated as wild cards or not.

To further standardize card values, the Rank and Suit classes are implemented as Enums. They help define specific properties like rankCardValue and bonusFactor for card suits, which are used for calculating scores. The Decorator Pattern works seamlessly with these enumerations to provide additional values when calculating scores with wild cards.

Chapter 4

Clever Computer Player Implementation

The Clever Computer Player in the HiFive game aims to make intelligent decisions that maximize the overall score in each turn, following a strategic and computationally efficient approach.

4.1 Approach Overview

4.1.1 Limited Scope of Probability Consideration

The HiFive game only lasts for four rounds, and each player initially receives two cards, leading to a relatively small number of cards being played in total. Specifically, there are 51 cards available in a deck after removing the jokers, and at most, 16 cards are discarded over the course of four rounds (four players discarding one card each per round).

Given the limited rounds and high variability in cards dealt, we chose not to consider the cards played by all players when determining the best strategy. The cards in each player's hand are dealt randomly, and attempting to track and predict the remaining cards would significantly increase the game's complexity without adding much value to the player's decision-making. Thus, the Clever Computer Player uses the cards available in its hand to make decisions, without considering broader probability calculations.

4.1.2 Greedy Algorithm for Card Selection

We adopted a **greedy algorithm** to maximize the score in the current round based on the cards held by the player. Specifically, the Clever Computer Player uses the following strategy:

1. From the three cards in its hand, the player selects the best two cards that produce the highest score when combined. The third card is discarded.
2. The approach aims to ensure that in each round, the Clever Computer Player retains the combination of cards most likely to yield the highest score, given the rules of the HiFive game.

This greedy strategy allows the player to make optimal local decisions without trying to plan across multiple rounds, which aligns well with the limited scope and random nature of card distribution in the game.

4.2 Implementation Details

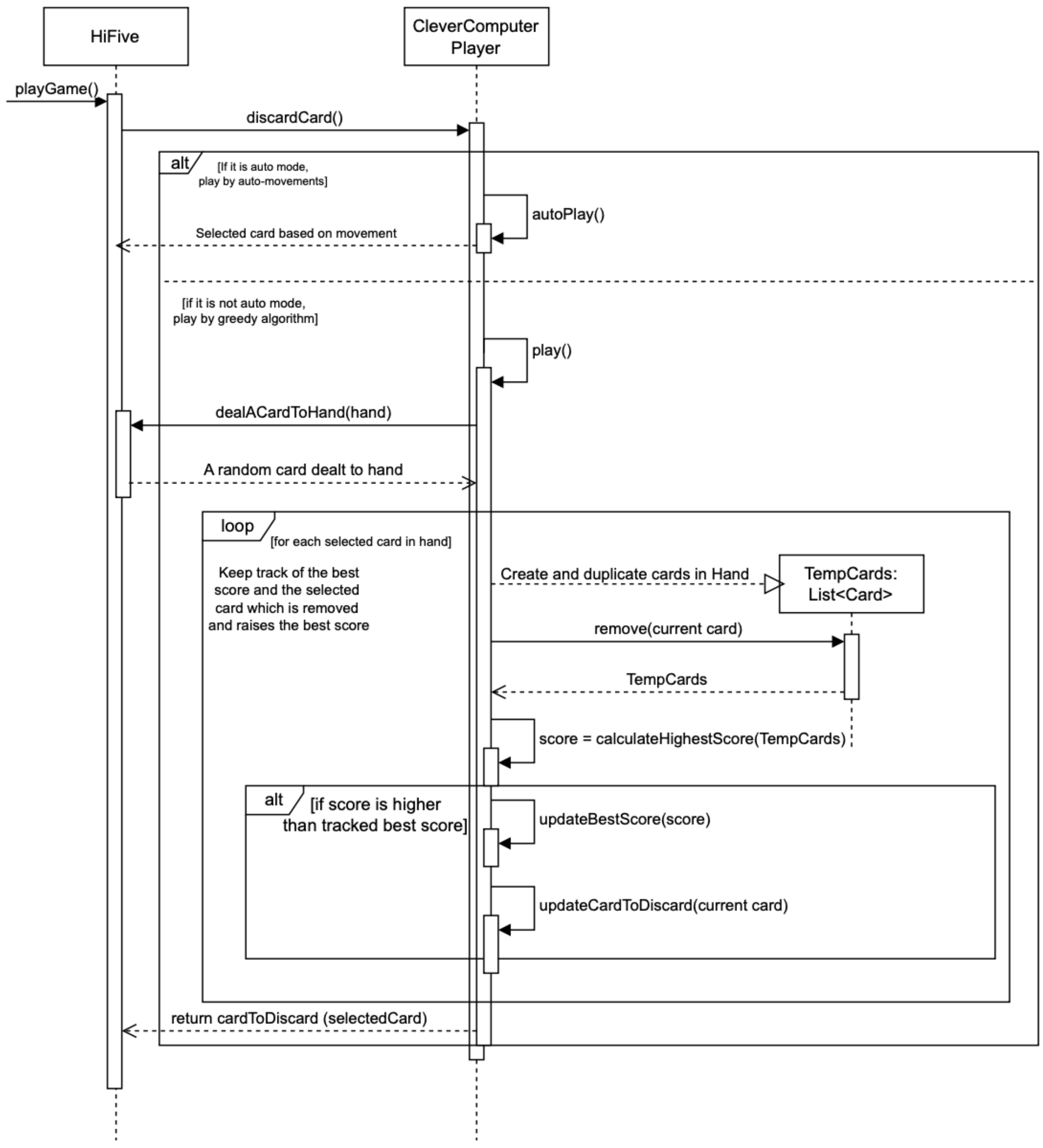
HiFive
CleverComputer Player
playGame()
discardCard()
alt
[If it is auto mode, play by auto-movements]
autoPlay()
Selected card based on movement
[if it is not auto mode, play by greedy algorithm]
play()
dealACardToHand(hand)
A random card dealt to hand
loop
[for each selected card in hand]
Keep track of the best score and the selected card which is removed and raises the best score
Create and duplicate cards in Hand
TempCards: List<Card>
remove(current card)
TempCards
score = calculateHighestScore(TempCards)
alt
[if score is higher than tracked best score]
updateBestScore(score)
updateCardToDiscard(current card)
return cardToDiscard (selectedCard)

Figure 4.1: Design Sequence Model for Clever Computer Players

The Clever Computer Player is implemented as a subclass of the **ComputerPlayer** class, inheriting the basic properties and behavior while extending its logic with a custom **play()** method that defines the player's decision-making process.

4.2.1 Simulating Card Discard and Score Calculation

1. The **play()** method begins by dealing a card from the deck and adding it to the player's hand.

2. For each card in the hand, the player simulates discarding the card and calculates the potential score that can be achieved using the remaining cards.

3. The score is calculated using the **CompositeCalculator**, which applies all four scoring algorithms and determines the maximum score possible with the given combination of cards.

4.3 Benefits of the Approach

4.3.1 Maximization of Current Score

By using the greedy algorithm, the Clever Computer Player ensures that it always discards the card that is least beneficial for the current round, thereby maximizing the potential score. This approach is well-suited to the short, four-round structure of the game, where each decision has a significant impact on the final outcome.

4.3.2 Simplicity and Efficiency

The algorithm focuses on making decisions based solely on the current hand, without considering the broader game state. This reduces computational complexity, allowing the player to make decisions quickly and efficiently. Given the random nature of card dealing, this approach strikes a balance between strategic decision-making and computational feasibility.

Chapter 5

Conclusion

The refactoring and extension of the HiFive game code have significantly enhanced the maintainability, flexibility, and overall design quality of the game. By breaking down the original monolithic class into smaller, specialized classes and applying appropriate design patterns, the code is now more modular, adheres to object-oriented principles, and is easier to extend. The introduction of the Clever Computer Player showcases the effectiveness of applying strategic algorithms to maximize game performance in a simplified yet efficient manner. Overall, the improvements have transformed the HiFive game from a poorly structured implementation to a well-designed, flexible, and more engaging system. Based on the robust design choices made during this refactoring process, future enhancements can be more easily integrated.