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

1 Introduction

The original implementation of the HiFive game faced numerous critical software design issues, including high coupling, lack of encapsulation, and a complete absence of object-oriented principles. The entire game logic was contained within a monolithic class, which made it difficult to extend, maintain, and understand.

The goal of this project was to address these deficiencies by refactoring the code to apply sound software engineering principles, thereby enhancing its modularity, maintainability, and extendibility. Key improvements include splitting the monolithic class into specialized components, applying several design patterns (such as Factory, Strategy, Composite, and Decorator patterns), and introducing a Clever Computer Player that optimizes gameplay using strategic algorithms. This report highlights the shortcomings of the original design, the detailed refactoring steps, and the resulting improvements in both functionality and performance of the game.

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.

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

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.

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.

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.

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.

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.

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.

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.

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.

3.2 Wild Card Functionalities Implementation

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.

To streamline the creation of card decorators, we introduced a **CardDecoratorFactory** class. This factory class centralises the instantiation logic for different types of decorated cards, promoting consistency and reducing code duplication. The use of

CardDecoratorFactory makes it easier to introduce new types of decorated cards while maintaining a clean and modular approach to card creation.

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.

4 Clever Computer Player Implementation

The Clever Computer Player in the HiFive game is designed to make strategic decisions, aiming to maximize both immediate score and future potential outcomes.

4.1 Approach Details

The **CleverComputerPlayer** follows a two-step strategy:

The Clever Computer Player extends **ComputerPlayer** with a custom **play()** method for strategic card selection. The **play() Method** begins by drawing a new card to add to their hand.

In **Step 1**, the **findBestScoringCard()** method is called to determine which card, when discarded, would yield the highest score. It works by simulating the discard of each card in the player's hand and calculating the resulting score with the remaining two cards using the **FiveCalculator**, **SumFiveCalculator**, and **DifferenceFiveCalculator**. The card whose omission results in the highest score is chosen for discarding.

In **Step 2**, if none of the two-card combinations achieve the target score ('**FIVE_GOAL**'), the player proceeds to evaluate the cards using the **selectCardBasedOnProbabilities()** method rather than relying on the **NoFiveCalculator**.
Our strategy is as follows:

- **Keeping Picture Cards**: If we have picture cards, it's advantageous to keep them, as their alternative values increase the likelihood of achieving a score of 5 with other cards. For example, since the possible values of K are 1, 3, 7, 9, 11, and 13, we can expect any of 2, 4, 6, or 8 to form a **FIVE_GOAL**, either through addition or subtraction.

- **Discarding Picture Cards**: If all the cards are picture cards (A, J, Q, K), the one with the lowest suit value is discarded. This is because, with picture cards, we have many potential combinations to form a **FIVE_GOAL**, so we prioritize the suit bonus points over immediate score potential.

- **Evaluating Complements of Value Cards**: If the cards are not just value cards, we assess how many of their complement cards, needed to form a **FIVE_GOAL**, have already been discarded. We then calculate the probability of obtaining such complement cards in future rounds:

$$P(complementCardsOfValueN) = \frac{no.ofDiscardedCardsOfValueN}{no.ofTotalCardsOfValueNInOriginalDeck}$$

If 25% or higher of the complement cards are already in the discarded stack, we consider it unlikely to receive the complement cards and add the current card to the list of **potentialDiscards**. For example, if we have a 7 card, its complement card would be 2 (since 7 - 2 = 5). In a normal deck, there are 8 cards that provide a value of 2 (4 two-cards and 4 jack-cards). If 2 two-cards have already been discarded in previous rounds, giving a probability of $2/8 = 1/4$, we will discard the 7 card. We use a threshold of 25%, which reflects high sensitivity, because the game is played over only four rounds. This means we need to make decisions quickly and can't afford to wait for low-probability events to occur.

- **Choosing Between Multiple Discards:** If there are multiple potential cards to discard, we discard the card with the higher rank value, as we are aiming to achieve a "Sum Five" rather than a "Difference Five" as it comes with higher bonus points. Even if all the potential cards are greater than 5, this strategy does not significantly impact the outcome because we maintain a high sensitivity with the 0.25 probability threshold.

4.2 Benefits of the Approach

Balanced Decision-Making and Efficiency

The Clever Computer Player combines immediate score maximization with strategic probability evaluation, ensuring it makes informed decisions in each round. This strategy balances immediate scoring opportunities with long-term game potential, enhancing the player's chances of success.

The approach maintains efficiency by focusing on the current hand, using simple probability calculations, and avoiding complex tracking of all played cards. This enables quick and effective decision-making, enhancing gameplay quality.

5 Conclusion

The refactoring of the HiFive game significantly improved its overall structure, maintainability, and flexibility by adopting a modular, object-oriented design. Key enhancements include the use of Factory, Strategy, Composite, and Decorator patterns to ensure modularity, adherence to the Single Responsibility Principle, and reduced coupling. The introduction of a Clever Computer Player demonstrates the effectiveness of strategic algorithms in enhancing gameplay by balancing immediate score maximization with consideration of future potential outcomes.

Through these efforts, the HiFive game has been transformed from a rigid, difficult-to-maintain implementation into a well-structured, extendable, and engaging system. The new design lays a solid foundation for future expansions, with features like the CardDecoratorFactory and a sophisticated player strategy that ensure scalability and ease of integration for any subsequent enhancements.