Report P2

The HiFive game has undergone a significant redesign, effectively addressing several issues present in the original implementation. The initial design suffered from a lack of modularity, tight coupling between components, limited extensibility, and poor separation of concerns, making the codebase difficult to maintain and extend. The new design introduces a modular architecture that employs interface-based design principles to achieve loose coupling between components. By separating different aspects of the game into distinct modules—such as card management, game engine, UI, and scoring strategies—the redesign incorporates several GRASP (General Responsibility Assignment Software Patterns) principles and Gang of Four (GoF) design patterns, including Strategy, Observer, Factory, Singleton, and Template Method. The application of these patterns significantly improves code organization, enhances extensibility, and facilitates easier maintenance. Additionally, the redesign centralizes game configurations, simplifying the modification of game parameters. These improvements not only resolve the issues present in the original implementation but also provide a solid foundation for future enhancements and variations of the game.

1. Code Structure and Organization

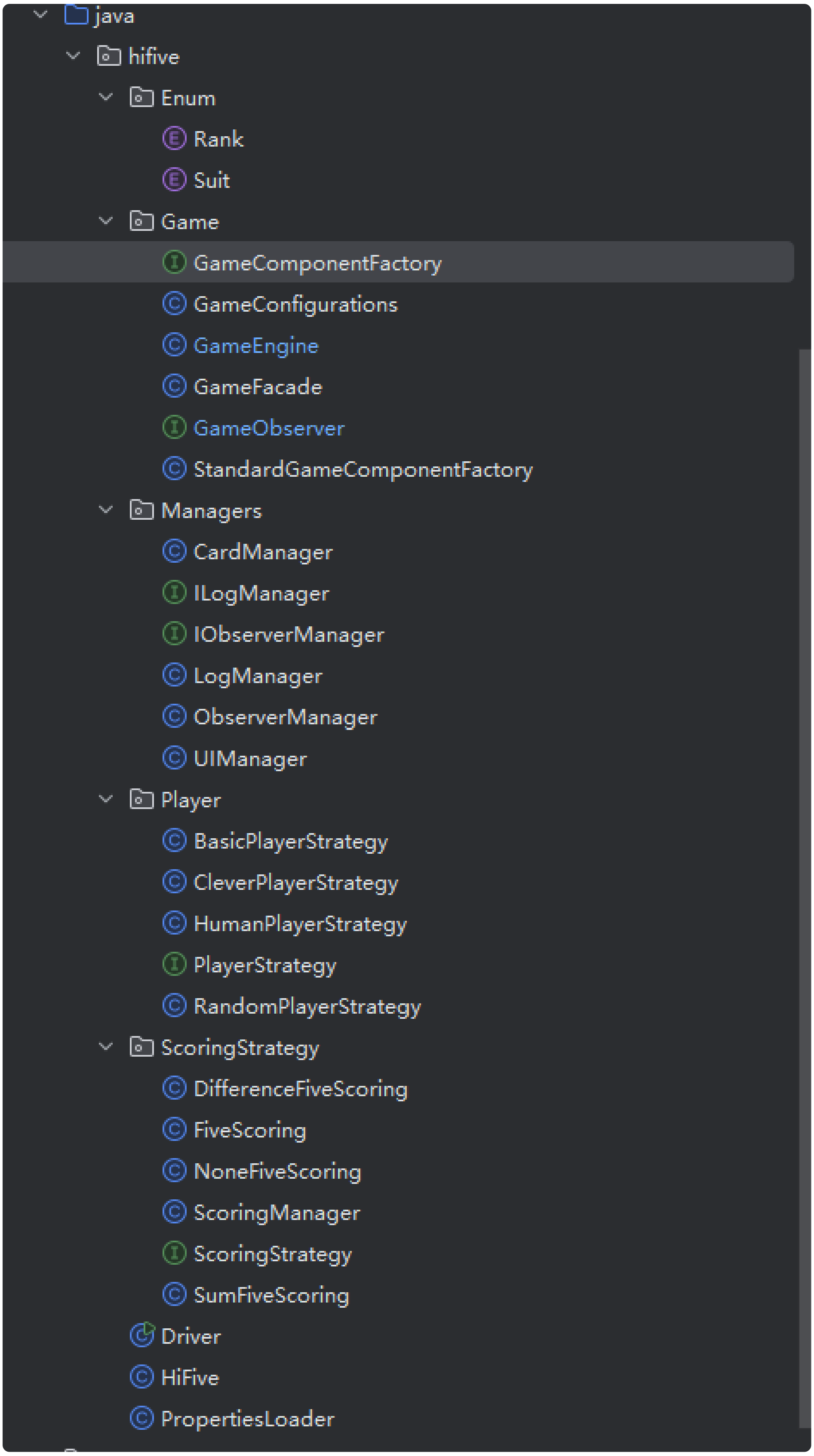
The updated HiFive codebase exhibits significant improvements in structure and organization compared to the original single-file implementation. Classes and interfaces are now separated based on their functional areas into packages such as `Game`, `Enum`, `Managers`, `Player`, and `ScoringStrategy`. This separation aligns with several GRASP patterns and promotes better modularity.

The primary packages and their contents are:

- `Enum`: Contains enumerations for `Rank` and `Suit`, defining the properties and behaviors of card ranks and suits, including wild card handling.

- **`Game`**: Houses the core game logic classes such as `GameEngine`, `GameFacade`, `GameConfigurations`, `GameObserver`, and `StandardGameComponentFactory`.
- **`Managers`**: Includes manager classes responsible for specific aspects of the game, like `CardManager`, `UIManager`, `LogManager`, and `ObserverManager`.
- **`Player`**: Defines the `PlayerStrategy` interface and its concrete implementations (`HumanPlayerStrategy`, `RandomPlayerStrategy`, `BasicPlayerStrategy`, `CleverPlayerStrategy`), representing different player behaviors.
- **`ScoringStrategy`**: Contains the `ScoringStrategy` interface and its implementations (`FiveScoring`, `SumFiveScoring`, `DifferenceFiveScoring`, `NoneFiveScoring`) for calculating player scores based on different criteria.
- **`hifive`**: The root package containing the `HiFive` main class, which initializes and starts the game.
  This structured organization promotes separation of concerns and enhances the maintainability and extensibility of the

codebase.



```
java
  hifive
    Enum
      Rank
      Suit
    Game
      GameComponentFactory
      GameConfigurations
      GameEngine
      GameFacade
      GameObserver
      StandardGameComponentFactory
    Managers
      CardManager
      ILogManager
      IObserverManager
      LogManager
      ObserverManager
      UIManager
    Player
      BasicPlayerStrategy
      CleverPlayerStrategy
      HumanPlayerStrategy
      PlayerStrategy
      RandomPlayerStrategy
    ScoringStrategy
      DifferenceFiveScoring
      FiveScoring
      NoneFiveScoring
      ScoringManager
      ScoringStrategy
      SumFiveScoring
    Driver
    HiFive
    PropertiesLoader
```

2. Application of GRASP Principles

Several GRASP (General Responsibility Assignment Software Patterns) principles have been applied in the updated design:

Information Expert

Assign responsibilities to the class that has the necessary information to fulfill them.

- `CardManager`: Responsible for operations related to cards, such as dealing, retrieval, and automatic movements, since it has the most information about the cards.

```
private final GameConfigurations gameConfig;
private final Random random;
private final Deck deck;
private final Hand pack;
```

- `Rank` and `Suit` Enums: Encapsulate properties and behaviors related to card ranks and suits. For example, `Rank` knows its `wildValues` and can determine if it's a wild card.
- `ScoringStrategy` Implementations: Each scoring strategy class (e.g., `FiveScoring`, `SumFiveScoring`) contains the logic necessary to calculate scores based on the cards, leveraging the information they hold.

Creator

Assign the responsibility of creating an instance of class A to class B if B aggregates, contains, or closely uses A.

- `GameEngine`: Responsible for creating and managing game-related objects, such as scoring strategies and player strategies. It also initializes scores, sets up player movements, and manages the overall game flow.

```
public GameEngine(GameConfigurations config, Deck deck, CardManager
cardManager, UIManager gameUI,
                  ILogManager logManager, IObserverManager
observerManager, ScoringManager scoringManager,
                  PlayerStrategy[] playerStrategies, int[] scores) {
    this.config = config;
    this.deck = deck;
    this.cardManager = cardManager;
    this.gameUI = gameUI;
    this.logManager = logManager;
    this.observerManager = observerManager;
    this.scoringManager = scoringManager;
    this.playerStrategies = playerStrategies;
    this.scores = scores;
    this.playerAutoMovements = new ArrayList<>();
    this.hands = new Hand[GameConfigurations.NB_PLAYERS];
    this.autoIndexHands = new int[GameConfigurations.NB_PLAYERS];
        // Initialize the game as part of the constructor
    initializeGame();
}
```

- `StandardGameComponentFactory`: Creates game components like scoring strategies and player strategies, adhering to the Factory design pattern.
- `CardManager`: Creates and initializes player hands, as it manages the deck and deals cards to players.

Low Coupling

Aims to reduce dependencies between classes to enhance modularity and flexibility.

- Use of Interfaces: Extensive use of interfaces (e.g., `ICardManager`, `IUIManager`, `ILogManager`, `ScoringStrategy`) promotes low coupling, allowing for easy substitution of implementations without affecting other parts of the code.
- Dependency Injection: Classes like `GameEngine` receive dependencies through constructors, facilitating easier substitution and testing.
- Separate Manager Classes: Responsibilities are divided among manager classes (`CardManager`, `UIManager`, `LogManager`), each handling a specific aspect of the game and minimizing interdependencies.

High Cohesion

Ensures that classes are focused on a single task or closely related tasks.

- `ScoringStrategy` Implementations: Each scoring strategy class focuses solely on calculating scores based on a specific rule.
- `PlayerStrategy` Implementations: Each player strategy class encapsulates the logic for a specific player behavior (e.g., random, basic, clever).
- `GameEngine`: Manages the overall game flow, delegating tasks to appropriate managers and strategies without being burdened with unrelated responsibilities.

Controller

Assigns the responsibility of handling system events to a non-UI class that represents the system or a use-case scenario.

- `GameEngine`: Acts as the primary controller of the game, managing the sequence of play, interactions between players and the system, and coordinating other components.
- `GameFacade`: Provides a simplified interface to start the game, hiding the complexity of the underlying system and acting as a controller for initiating gameplay.

3. Application of Gang of Four Design Patterns

Several Gang of Four design patterns have been implemented in the updated design:

Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

- `PlayerStrategy` Interface and Implementations: Defines different algorithms for player behavior (`RandomPlayerStrategy`, `BasicPlayerStrategy`, `CleverPlayerStrategy`, `HumanPlayerStrategy`), allowing the `GameEngine` to use them interchangeably.
- `ScoringStrategy` Interface and Implementations: Encapsulates different scoring algorithms, enabling the `ScoringManager` to

select the appropriate strategy based on the game's configuration.

```
public class FiveScoring implements ScoringStrategy {
    private final int fiveGoal;
    private final int fivePoints;

    public FiveScoring(int fiveGoal, int fivePoints) {
        this.fiveGoal = fiveGoal;
        this.fivePoints = fivePoints;
    }

    @Override
    public int calculateScore(List<Card> cards) {
        int maxScore = 0;
        for(Card card : cards) {
            Rank rank = (Rank)card.getRank();
            Suit suit = (Suit)card.getSuit();
            if(rank.getRankCardValue() == fiveGoal || (rank.isWildCard()
&& rank.getWildValues().contains(fiveGoal))) {
                int score = fivePoints + suit.getBonusFactor();
                if(score > maxScore) {
                    maxScore = score;
                }
            }
        }
        return maxScore;
    }
}
```

Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

- `IObserverManager` and `ObserverManager`: Manage observers that implement the `GameObserver` interface, notifying them of game events like round starts, card plays, score updates, and game over.

```
public interface GameObserver {
    // Notifies when a new round starts
    void onRoundStart(int roundNumber);
```

```java
    // Notifies when a player plays a card
    void onCardPlayed(int player, Card card);
    // Notifies when a player's score is updated
    void onScoreUpdate(int player, int newScore);
    // Notifies when the game is over, providing final scores and winners
    void onGameOver(int[] finalScores, List<Integer> winners);
}
```

- Usage in `GameEngine`: The `GameEngine` notifies observers of game events through the `ObserverManager`, allowing decoupled components to respond to changes.

```java
@Override
public void notifyRoundStart(int roundNumber) {
    for (GameObserver observer : observers) {
        observer.onRoundStart(roundNumber);
    }
}

@Override
public void notifyCardPlayed(int player, Card card) {
    for (GameObserver observer : observers) {
        observer.onCardPlayed(player, card);
    }
}

@Override
public void notifyScoreUpdate(int player, int newScore) {
    for (GameObserver observer : observers) {
        observer.onScoreUpdate(player, newScore);
    }
}

@Override
public void notifyGameOver(int[] finalScores, List<Integer> winners) {
    for (GameObserver observer : observers) {
        observer.onGameOver(finalScores, winners);
    }
}
```

Factory Pattern

Provides an interface for creating objects without specifying their concrete classes.

- `GameComponentFactory` Interface and `StandardGameComponentFactory`: Define methods to create game components like scoring strategies and player strategies, allowing for flexibility in creating different game configurations without changing the client code.

```java
public interface GameComponentFactory {
    // Creates and returns a list of scoring strategies based on the given game configuration
    List<ScoringStrategy> createScoringStrategies(GameConfigurations config);

    // Creates and returns an array of player strategies based on the given game configuration
    PlayerStrategy[] createPlayerStrategies(GameConfigurations config);
}
```

Template Method Pattern

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

- While the `PlayerStrategy` and `ScoringStrategy` interfaces define methods implemented by subclasses, the Template Method pattern is not directly applied in the code, as there isn't a base class providing a common algorithm with steps overridden by subclasses.

4. Potential Future Extensibilities

The new design allows for several potential future enhancements:

New Player Strategies

- The Strategy pattern facilitates the addition of new AI behaviors. Implementing a new `PlayerStrategy` and adding it to the `StandardGameComponentFactory` allows for easy expansion of player types.

New Scoring Strategies

- Similar to player strategies, new scoring rules can be added by implementing the `ScoringStrategy` interface and integrating them into the `StandardGameComponentFactory`.

UI Customization

- The separation of UI logic into the `UIManager` class enables potential future UI enhancements. New methods could be added to support different visual styles or layouts.

Game Variants

- The modular design of the `GameEngine` and its components makes it easier to implement variations of the HiFive game. New game rules could be added by extending the `GameEngine` class or creating new `ScoringStrategy` implementations.

Networking Capabilities

- The Observer pattern, implemented through `GameObserver`, could be extended to support networked multiplayer gameplay. A new `NetworkGameObserver` could be created to handle network communication.

Persistence

- The separation of game state (in `GameEngine`) and game logic simplifies the implementation of save/load functionality. A new `PersistenceManager` could be created to handle saving and loading game states.

In conclusion, the updated HiFive codebase demonstrates a significant improvement over the original design through the extensive application of GRASP and Gang of Four design patterns. The new structure is highly modular, extensible, and maintainable, providing a solid foundation for future enhancements and variations of the game. By adopting interface-based design principles and separating concerns across well-defined packages and classes, the redesign not only resolves previous issues but also positions the codebase for scalability and adaptability to evolving requirements.

Use case diagram

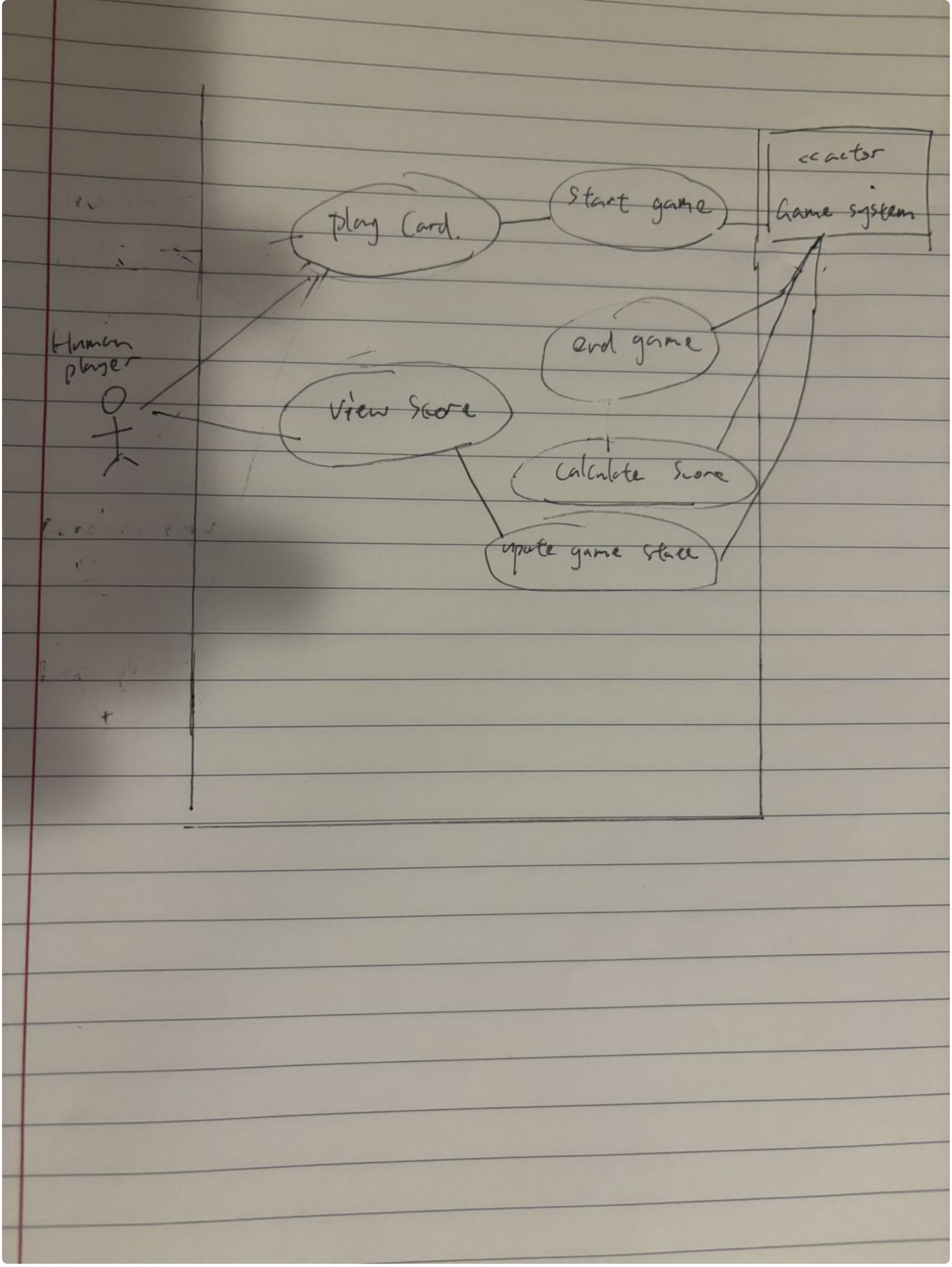


Figure 1 : *Use Case Diagram*

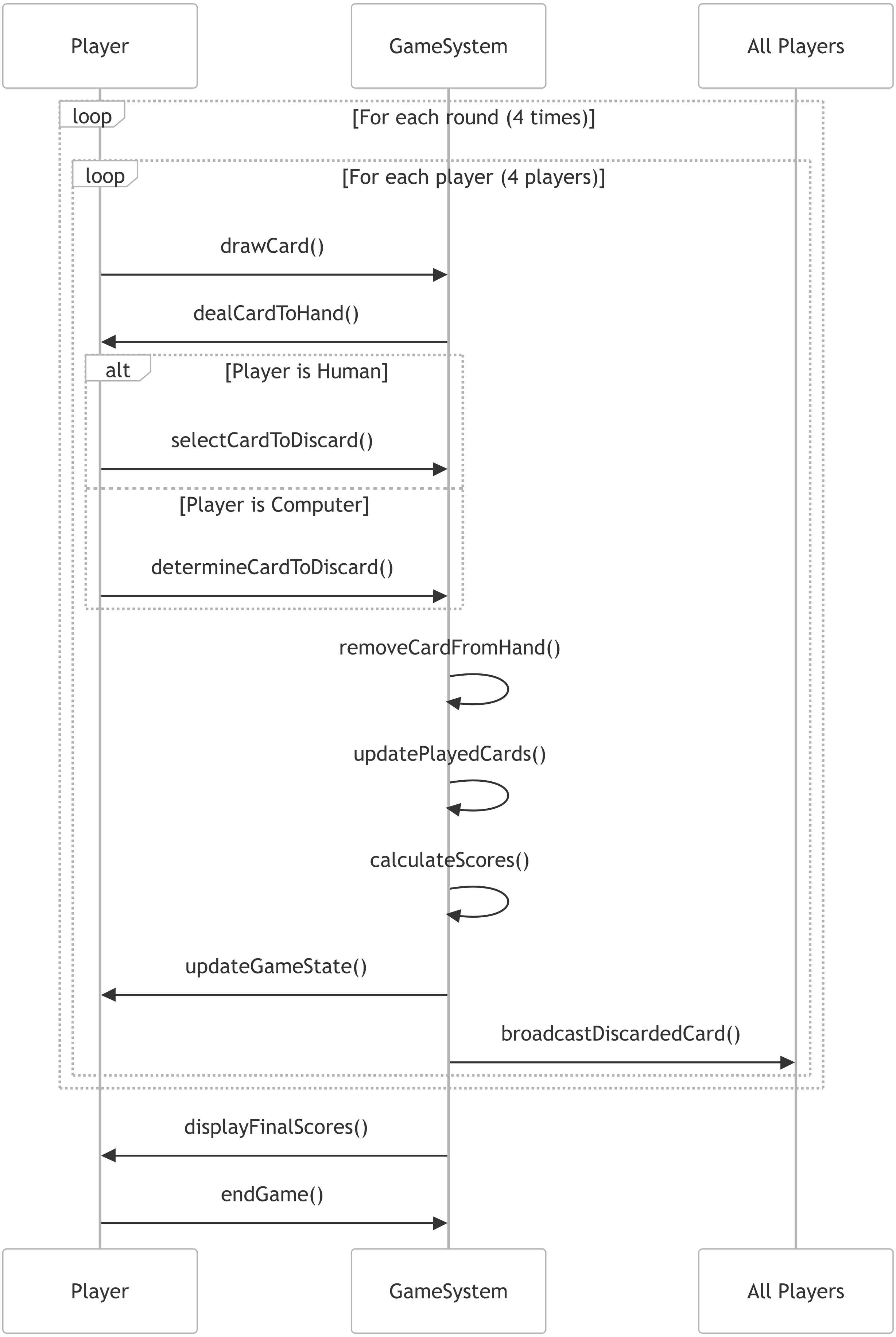
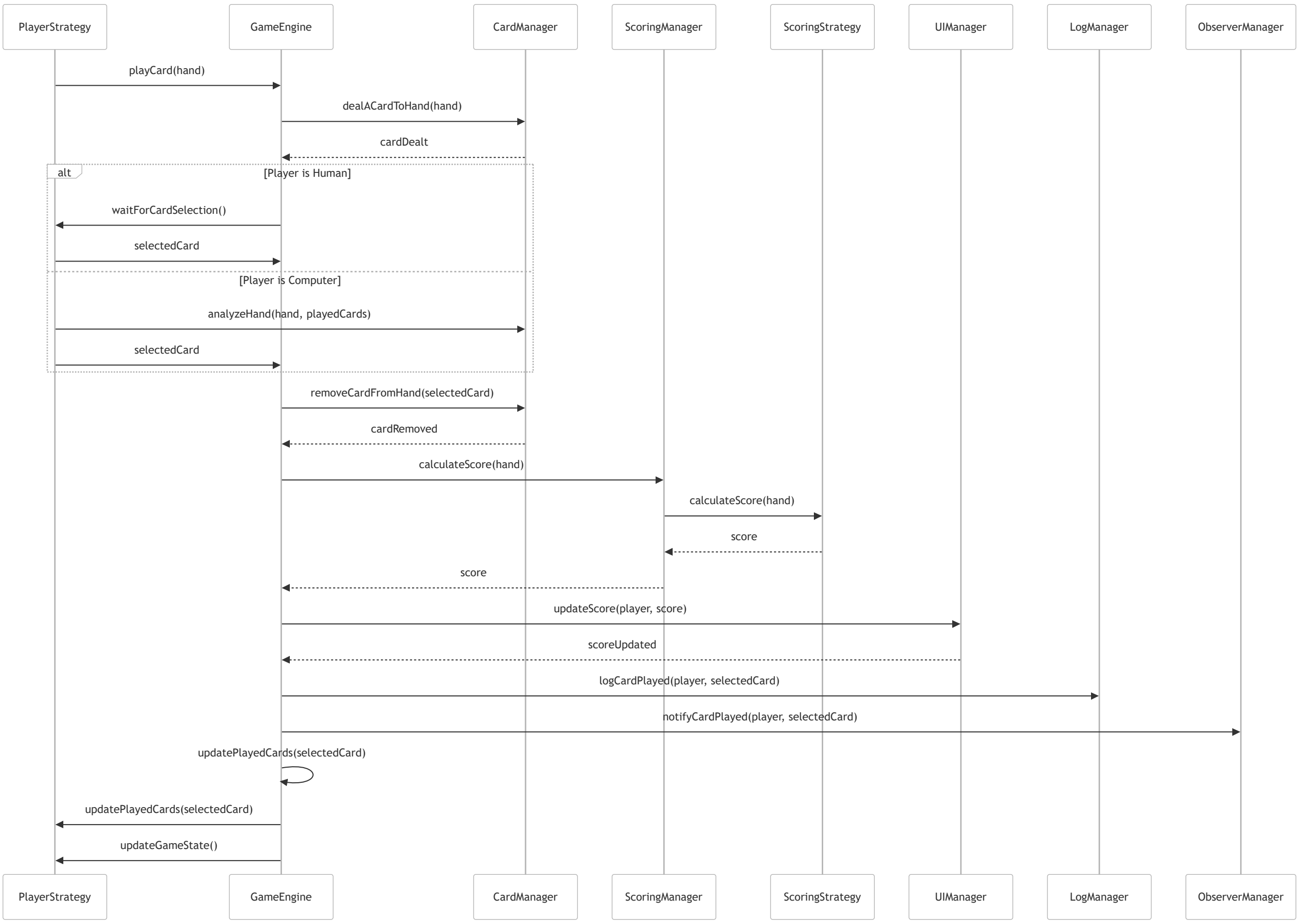
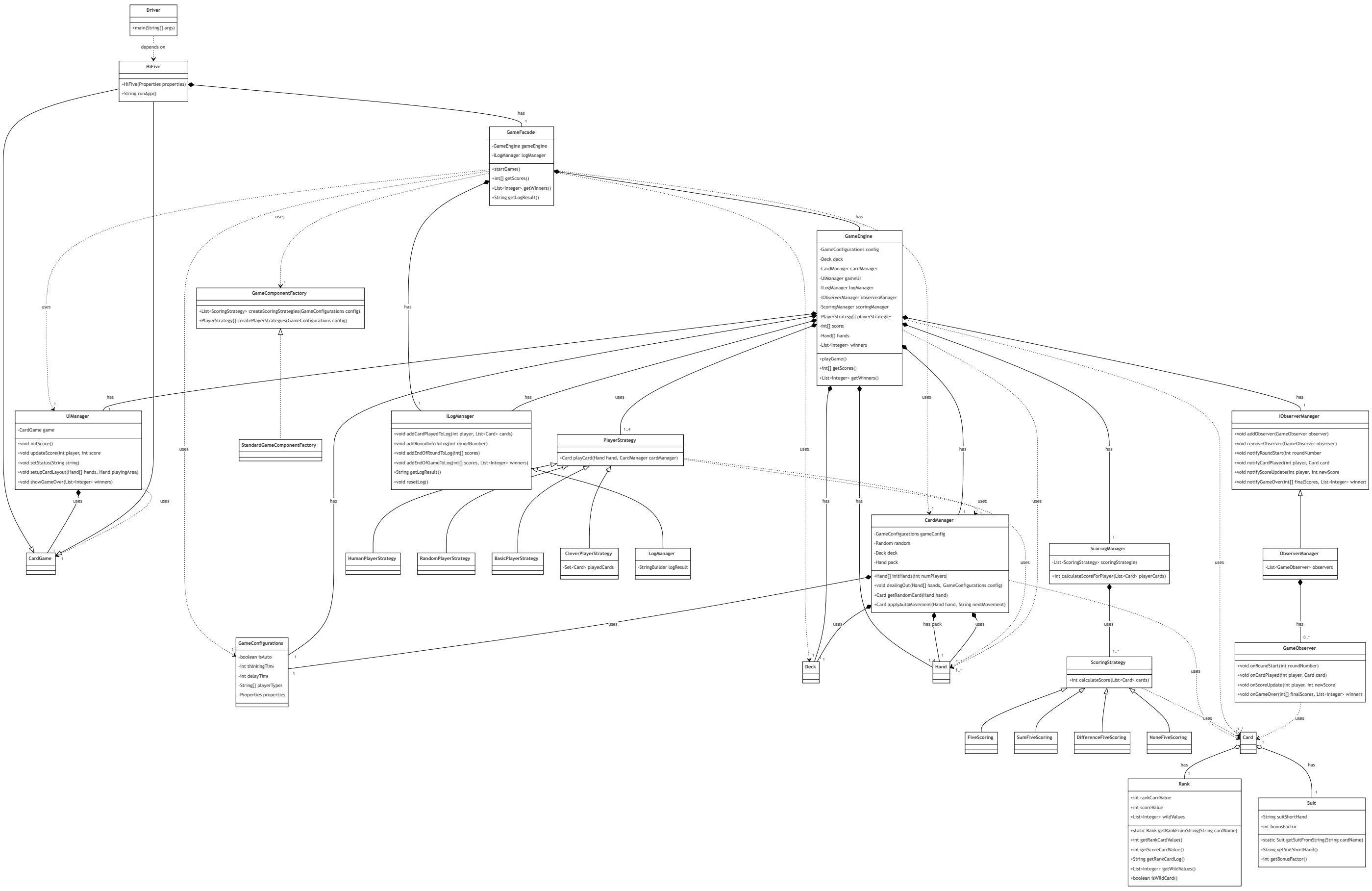


Figure 2 : Design Sequence Diagram



$Figure\, 3: \ Design\ Class\ Diagram$


Driver
+main(String[] args)
depends on
HiFive
+HiFive(Properties properties)
+String runApp()
has
GameFacade
-GameEngine gameEngine
-ILogManager logManager
+startGame()
+int[] getScores()
+List<Integer> getWinners()
+String getLogResult()
uses
GameComponentFactory
+List<ScoringStrategy> createScoringStrategies(GameConfigurations config)
+PlayerStrategy[] createPlayerStrategies(GameConfigurations config)
StandardGameComponentFactory
UIManager
-CardGame game
+void initScore()
+void updateScore(int player, int score
+void setStatus(String string)
+void setupCardLayout(Hand[] hands, Hand playingArea)
+void showGameOver(List<Integer> winners)
CardGame
ILogManager
+void addCardPlayedToLog(int player, List<Card> cards)
+void addRoundInfoToLog(int roundNumber)
+void addEndOfRoundToLog(int[] scores)
+void addEndOfGameToLog(int[] scores, List<Integer> winners)
+String getLogResult()
+void resetLog()
PlayerStrategy
+Card playCard(Hand hand, CardManager cardManager)
HumanPlayerStrategy
RandomPlayerStrategy
BasicPlayerStrategy
CleverPlayerStrategy
-Set<Card> playedCards
LogManager
-StringBuilder logResult
GameConfigurations
-boolean isAuto
-int thinkingTime
-int delayTime
-String[] playerTypes
-Properties properties
GameEngine
-GameConfigurations config
-Deck deck
-CardManager cardManager
-UIManager gameUI
-ILogManager logManager
-IObserverManager observerManager
-ScoringManager scoringManager
-PlayerStrategy[] playerStrategies
-int[] scores
-Hand[] hands
-List<Integer> winners
+playGame()
+int[] getScores()
+List<Integer> getWinners()
CardManager
-GameConfigurations gameConfig
-Random random
-Deck deck
-Hand pack
+Hand[] initHands(int numPlayers)
+void dealingOut(Hand[] hands, GameConfigurations config)
+Card getRandomCard(Hand hand)
+Card applyAutoMovement(Hand hand, String nextMovement)
has pack
Deck
Hand
ScoringManager
-List<ScoringStrategy> scoringStrategies
+int calculateScoreForPlayer(List<Card> playerCards)
ScoringStrategy
+int calculateScore(List<Card> cards)
FiveScoring
SumFiveScoring
DifferenceFiveScoring
NoneFiveScoring
IObserverManager
+void addObserver(GameObserver observer)
+void removeObserver(GameObserver observer)
+void notifyRoundStart(int roundNumber
+void notifyCardPlayed(int player, Card card
+void notifyScoreUpdate(int player, int newScore
+void notifyGameOver(int[] finalScores, List<Integer> winners
ObserverManager
-List<GameObserver> observers
GameObserver
+void onRoundStart(int roundNumber)
+void onCardPlayed(int player, Card card)
+void onScoreUpdate(int player, int newScore)
+void onGameOver(int[] finalScores, List<Integer> winners
Card
Rank
+int rankCardValue
+int scoreValue
+List<Integer> wildValues
+static Rank getRankFromString(String cardName)
+int getRankCardValue()
+int getScoreCardValue()
+String getRankCardLog()
+List<Integer> getWildValues()
+boolean isWildCard()
Suit
+String suitShortHand
+int bonusFactor
+static Suit getSuitFromString(String cardName)
+String getSuitShortHand()
+int getBonusFactor()
1..4
1..*
0..*
1

$Figure\, 4: \ Domain\ Model\ Class\ Diagram$