SWEN30006 Project 2 Report

Workshop: Thursday 2.15pm

Team 4: Maheen Abdul Khaliq Khan (1193813), Mahamithra Sivagnanam (1225270), Alleena Haider Waseem (1204035)

Introduction

The CountingUpGame project represents a meticulous culmination of deliberate design choices, iterative evaluations, and the synthesis of design patterns, tailored to meet both the functional and non-functional requirements of the game. This report delves deep into the heart of these design choices, highlighting not only the eventual decisions made but also the alternatives, the logic behind these decisions, and a comprehensive breakdown of the strategy underpinning the Clever player's gameplay.

Design Changes

Throughout the project's lifecycle, several design amendments were incorporated to ensure alignment with evolving requirements and optimization of performance. The significant changes are:

2.1 Implementation of the Strategy Pattern

Rationale: The Strategy Pattern was employed to allow a flexible assignment of different gameplay strategies to players.

Implementation: The project design encapsulated card-playing strategies within four classes: `CleverCardStrategy`, `RandomCardStrategy`, `BasicCardStrategy` and `HumanCardStrategy`. Each class implements a unique strategy, rendering gameplay strategies interchangeable, this is done through the implementation of a strategy interface.

2.2 Addition of CountingUpRules Class

Rationale: A dedicated class was required to streamline game rule validation and maintain a connection with the main game.

Implementation: The `CountingUpRules` class derives crucial data from `CountingUpGame` and is instrumental in the Strategy classes. This class also shoulders the responsibility of validity checks, ensuring unwavering adherence to game rules. It ensures that the first card played is the Ace of Clubs (according to the project spec), and it also ensures that the card selected is one that is valid.

2.3 Utilization of Static Variables

Rationale: Consistency across gameplay instances was paramount.

Implementation: Variables such as `selected`, `keyPressed`, and `aceClubPlayed` were rendered static. This design choice ensured that all players and strategies accessed consistent data, minimizing potential gameplay inconsistencies. These variables were consistently checked by various classes to inform certain decisions, for example, each of the player classes would check to see if `aceClubPlayed` was True or False so that they could then choose the appropriate card. The `keyPressed` variable was most relevant to the Human player and was used to make sure that the human's behavior was being consistently monitored throughout the game.

Design Alternatives

Our project's journey involved the evaluation of multiple design alternatives. Despite the distinct advantages each alternative presented, the decisions we made were were grounded in the project's unique requirements:

3.1 Singleton Pattern

Although the Singleton Pattern can ensure unique class instances, it might introduce a restrictive global state. For a dynamic game like `CountingUpGame`, this pattern was not deemed optimal and so we chose not to use it. The Singleton pattern would limit the flexibility for game expansion in the future and it hinders testing.

3.2 Game Referencing

Directly referencing `CountingUpGame` within strategy classes was contemplated. However, such an approach would introduce tight coupling, thereby reducing flexibility and maintainability. The alternative of indirect referencing through `CountingUpRules` was consequently adopted, we felt this indirection would maintain the flexibility of our code, whilst still allowing us to add on new features (inspired by the open-close principle).

3.3 Key Interaction

Initially, we considered moving the `keyPressed()` method from the `CountingUpGame` class to the `HumanStrategy` class, however this plan was dismissed due to its reliance on keyboard interactions, which could hamper user experience and introduce lower cohesion as the `HumanCardStrategy` class would now be focusing on multiple tasks instead of its main purpose- carrying out its strategy. The introduction of static variables offered a more intuitive alternative.

3.4 Game Setup Class

While a dedicated setup class could centralize game initialization, we felt that its introduction might overcomplicate the design without adding any substantial benefits.

3.5 Factory Pattern

We contemplated using the Factory Pattern for creating players/ strategies. However, the distinct behavior exhibited by each player rendered this approach redundant as the factory pattern is typically used to create one type of object and we needed to create 4 different types of players.

3.6 Card Strategy Interface Changes

The initial proposition was to house both `playCardStrategy` and `playCard()` within the `CardStrategy` interface. However, due to constraints posed by `HumanStrategy`, particularly concerning user interactions, this design was reassessed. If our game only considered Basic, Random, and Clever players then we would have moved `playCardStrategy` into the `CardStrategy` interface as they all exhibit similar behavior (differing solely on actual strategy), but this wasn't possible as the Human also needs to consider user interaction (from the `CountingUpGame` class) in order to play a card.

3.7 Inheritance vs Association

Initially, we wanted `CountingUpRules` to be the parent class from which all of our strategy types inherited certain methods and attributes. This seemed like a good idea in terms of code encapsulation and code tidiness. However, the more we analyzed the requirements of our project, we realized that inheritance would not be the right choice for this class. Firstly, the different strategy types fail the "is a" test that you typically see with inheritance (i..e, `HumanCardStrategy` "is a" `CountingUpRules` - this does not make sense). Additionally, because `CountingUpRules` is implementing the indirection principle, we felt that this would work better in an associative relationship which is why we chose to link the strategy types with `CountingUpRules` through association instead. This decision was also further solidified as being the right one, as it reduced coupling between `CountingUpRules` and the strategy types, and allowed for more code flexibility (as with inheritance, implementing changes within `CountingUpRules` was hindered by dependencies).

Design Patterns and Principles

To achieve software robustness and maintainability, we utilized specific design patterns and principles:

4.1 GoF (Gang of Four) Design Patterns:

4.1.1 Strategy Pattern

Distinct card-playing strategies are encapsulated within classes like `BasicCardStrategy`, `CleverCardStrategy`, `HumanCardStrategy`, and `RandomCardStrategy`. The implementation of the `CardStrategy` interface ensures a consistent method (`playCard()`) across these strategies.

4.1.2 Composite-Strategy Pattern

In developing `CleverStrategy`, we were inspired by the Composite-Strategy Pattern. Drawing from the Composite Pattern's principle of treating individual objects and their composites uniformly, we integrated this pattern to provide `CleverStrategy` with the flexibility to encapsulate and interchangeably use advanced tactics and the `BasicCardStrategy` based on game dynamics. This was particularly evident in our decision-making process around reverting to a Basic play when all other clever logic was not appropriate.

4.2 GRASP (General Responsibility Assignment Software Patterns):

4.2.1 High Cohesion and Low Coupling

Strategy classes like `BasicCardStrategy`, `RandomCardStrategy`, `HumanCardStrategy` and `CleverCardStrategy` exhibit high cohesion by focusing solely on their specific card-selection strategies. The

`CardStrategy` interface fosters low coupling, facilitating easy interchange of strategies as the different strategy types are not dependent on each other.

4.2.2 Creator Principle

The `CountingUpGame` class shoulders the responsibility of creating and initializing players and strategies, centralizing the creation logic, in other words the `CountingUpGame` class has the responsibility of creating the game as it has the knowledge to do so (based on responsibility driven design, the `CountingUpGame` class has the knowledge and ability to create the game).

4.2.3 Controller Principle

`CountingUpGame` functions as a controller, managing game flow, user interactions, and delegating responsibilities like card validation, to the `CountingUpRules` class.

4.2.4 Indirection Principle

`CountingUpRules` functions as an implementation of the indirection principle, it establishes communication between `CountingUpGame` and the various player strategy types. We felt that this improved abstraction allowed for greater flexibility (as we were able to move validity checks to this class - illustrating its flexibility), and allowing for the player strategies to implement their own logic whilst still communicating and receiving relevant updates from the `CountingUpGame` class.

4.2.5 Polymorphism

As mentioned in 4.1.1, we implemented the Strategy design pattern as we believed it best suited our needs. The `CardStrategy` interface we use exhibits polymorphic behavior. The interface includes the `playCard` method which is implemented by each of the four players/strategies as each player's strategy is vastly different. This allowed us to treat all four players as behaving similarly despite the vast differences in logic. This also made our code cleaner and allows for more player/strategy types to be added in the future.

Strategy for Clever Computer Player

Our design strategy for the clever player was derived from our hands-on experience playing the game. We each played the game (as human players) several times in order to grasp the optimal, clever moves. One challenge we faced during our own playing for research - was pretending like we weren't privy to our opponents' cards. From this, we aimed to create a strategy ensuring our clever player would thrive even without knowledge of opponents' cards.

We initiated our plan by recording the opponents' played cards in a HashMap, providing our intelligent player with a database (of sorts) of past moves. This information was pivotal in shaping our player's tactics.

The strategy consists of four key components:

1. ***Prioritizing Aces:*** The first tactic was to rid our player of all Ace cards promptly. Despite being the lowest-ranked card, an Ace carries a hefty 10 points, which can significantly hamper the score when negative scores are tallied at the end of the game. Given Aces usually play at the round's outset or atop other Aces (based on their low ranking and because the Ace of Clubs always starts a game), our primary goal was their swift removal from the clever player's hand.

2. ***Luring with Jacks and Queens:*** In situations where the clever player has already played all Aces or doesn't have them, the next move is to play either a Jack or a Queen. Central to our strategy is the King

card's potency. The long-term objective is to ensure every opponent exhausts their King card. This way, our player's King card remains undefeated, if our player puts down a King then the other players are unable to top this and so our player will be undefeated in that round. To achieve this, we bait our opponents by putting down Jacks and Queens, compelling them to lay down their Kings.

3. ***Strategic Play of the King Card:*** If the player has spent all its Aces, Queens, and Jacks or cannot use them, the King card comes into play, contingent upon its availability in the hand. Before making a move, our player evaluates the HashMap. If conditions are favorable (e.g., if 3 out of 4 Kings are accounted for), the player will use its King. However, if this King-play isn't feasible, we proceed to the next tactic.

4. ***Lengthening Game Duration:*** This tactic aims to extend gameplay and minimize the clever player's negative score. Here, the player employs the Basic strategy, inspired by the Composite Strategy Design Pattern, by playing its lowest-ranked card. This move usually prompts opponents to either match the card or place a higher-ranked one, giving our player more chances to deplete its hand.

If none of the strategies fit the current scenario, the clever player opts to skip its turn. This offers a potential advantage in subsequent turns as opponents may play more cards, providing better opportunities for the clever player.

Notes on our Diagrams

6.1 Domain Model

A few things to note with our Domain Model are that our team made some assumptions/ decisions when it came to displaying some of the classes already given to us in the codebase. The project spec mentions the importance of loading in game properties correctly and so instead of including the provided `PropertiesLoader` class in our diagram (as we felt this would be too technical), we chose to replace it with a representative `PropertiesFile` instead. Additionally, we know that `CountingUpGame` implements the `GGKeyListener` interface, however we also felt that this was too technical and so decided to not include it in our diagram.

6.2 Design Class Diagram

Since this type of diagram has more detail than a Domain model, we did choose to include both the `GGKeyListener` interface as well as the `PropertiesLoader` class. In this diagram, having this level of technical knowledge seemed appropriate.

6.2 Design Sequence Diagram

Because of how detailed our diagram is, we decided to separate certain elements of it into their own blocks to allow for a less cluttered and more easily readable diagram. Hence, certain parts of the diagram will say `ref` and `sd` to both 'reference' and highlight the 'sequence diagram' for a specific part of our logic/code.

Conclusion

The Counting Up Game project has traversed a rigorous journey of refinement, optimization, and enhancement. At its core, the project demonstrates the use of tried-and-tested design patterns, primarily from the GoF and GRASP methodologies, with the innovative approaches uniquely tailored to the game's requirements. The meticulous choices made during the project's evolution not only serve to fortify the game's architecture but also accentuate its adaptability and user engagement. Moreover, the crafting of the Clever player's strategy showcases the depth of understanding and commitment to recreating a human-like gaming intuition and experience. In essence, we hoped to capture thoughtful design, structural integrity and considered user experience in our project.

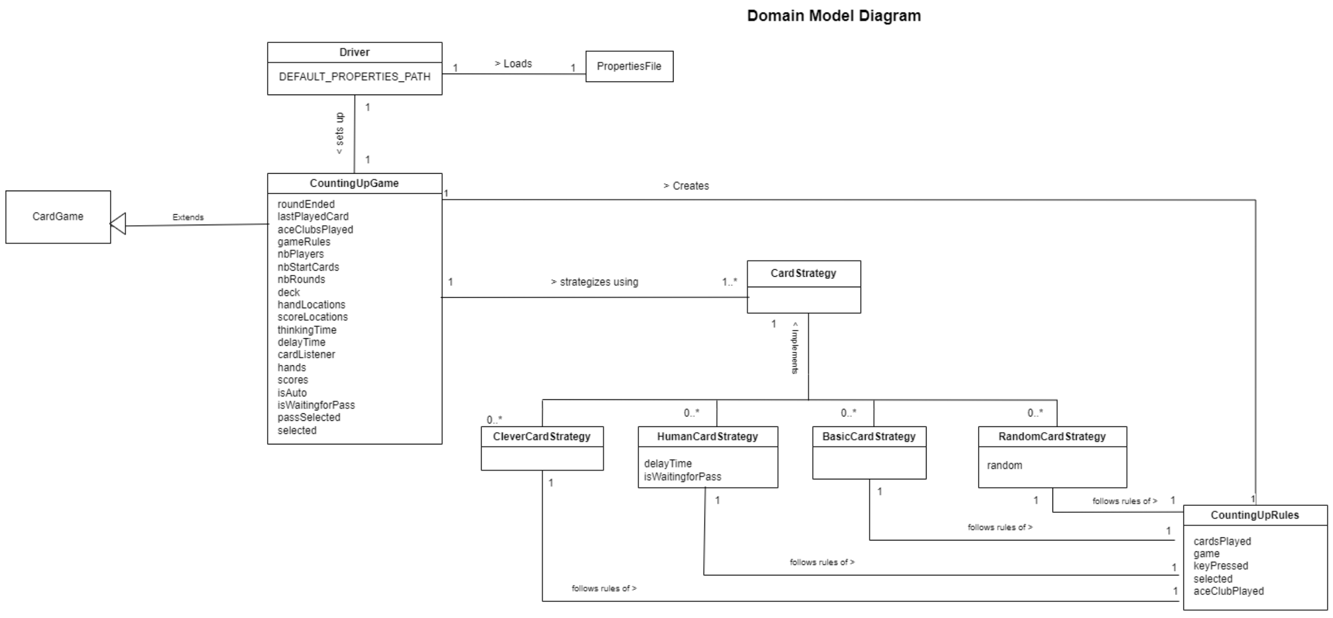
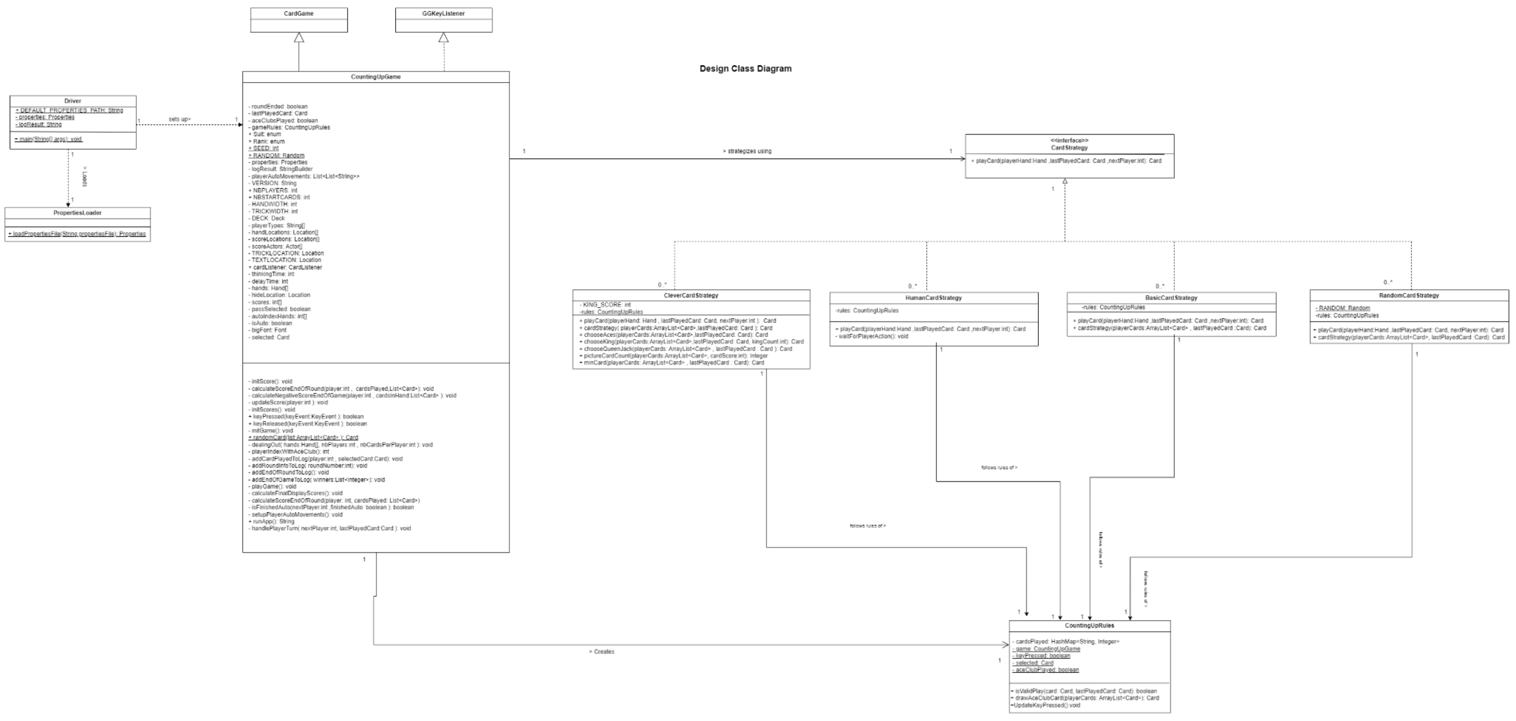


Figure 1: Domain Class Diagram



Design Class Diagram
CardGame
GGKeyListener
Driver
+ DEFAULT_PROPERTIES_PATH: String
- properties: Properties
- logResult: String
+ main(String[] args): void
sets up>
> Loads
PropertiesLoader
+ loadPropertiesFile(String propertiesFile): Properties
CountingUpGame
- roundEnded: boolean
- lastPlayedCard: Card
- aceClubsPlayed: boolean
- gameRules: CountingUpRules
+ Suit: enum
+ Rank: enum
+ SEED: int
+ RANDOM: Random
- properties: Properties
- logResult: StringBuilder
- playerAutoMovements: List<List<String>>
- VERSION: String
+ NBPLAYERS: int
+ NBSTARTCARDS: int
- HANDWIDTH: int
- TRICKWIDTH: int
- DECK: Deck
- playerTypes: String[]
- handLocations: Location[]
- scoreLocations: Location[]
- scoreActors: Actor[]
- TRICKLOCATION: Location
- TEXTLOCATION: Location
+ cardListener: CardListener
- thinkingTime: int
- delayTime: int
- hands: Hand[]
- hideLocation: Location
- scores: int[]
- passSelected: boolean
- autoIndexHands: int[]
- isAuto: boolean
- bigFont: Font
- selected: Card
- initScore(): void
- calculateScoreEndOfRound(player:int , cardsPlayed,List<Card>): void
- calculateNegativeScoreEndOfGame(player:int , cardsInHand:List<Card>): void
- updateScore(player:int): void
- initScores(): void
+ keyPressed(keyEvent:KeyEvent): boolean
+ keyReleased(keyEvent:KeyEvent): boolean
- initGame(): void
+ randomCard(list:ArrayList<Card>): Card
- dealingOut(hands:Hand[], nbPlayers:int , nbCardsPerPlayer:int): void
- playerIndexWithAceClub(): int
- addCardPlayedToLog(player:int , selectedCard:Card): void
- addRoundInfoToLog(roundNumber:int): void
- addEndOfRoundToLog(): void
- addEndOfGameToLog(winners:List<Integer>): void
- playGame(): void
- calculateFinalDisplayScores(): void
- calculateScoreEndOfRound(player: int, cardsPlayed: List<Card>)
- isFinishedAuto(nextPlayer:int ,finishedAuto: boolean): boolean
- setupPlayerAutoMovements(): void
+ runApp(): String
- handlePlayerTurn(nextPlayer:int, lastPlayedCard:Card): void
> strategizes using
<<interface>>
CardStrategy
+ playCard(playerHand:Hand ,lastPlayedCard: Card ,nextPlayer:int): Card
0..*
CleverCardStrategy
- KING_SCORE: int
-rules: CountingUpRules
+ playCard(playerHand: Hand , lastPlayedCard: Card, nextPlayer:int): Card
+ cardStrategy(playerCards:ArrayList<Card>,lastPlayedCard: Card): Card
+ chooseAces(playerCards:ArrayList<Card>,lastPlayedCard: Card): Card
+ chooseKing(playerCards:ArrayList<Card>,lastPlayedCard: Card, kingCount:int): Card
+ chooseQueenJack(playerCards: ArrayList<Card> , lastPlayedCard : Card): Card
+ pictureCardCount(playerCards:ArrayList<Card>, cardScore:int): Integer
+ minCard(playerCards: ArrayList<Card> , lastPlayedCard : Card): Card
HumanCardStrategy
-rules: CountingUpRules
+ playCard(playerHand:Hand ,lastPlayedCard: Card ,nextPlayer:int): Card
- waitForPlayerAction(): void
BasicCardStrategy
-rules: CountingUpRules
+ playCard(playerHand:Hand ,lastPlayedCard: Card ,nextPlayer:int): Card
+ cardStrategy(playerCards:ArrayList<Card> , lastPlayedCard :Card): Card
RandomCardStrategy
- RANDOM: Random
-rules: CountingUpRules
+ playCard(playerHand:Hand ,lastPlayedCard: Card, nextPlayer:int): Card
+ cardStrategy(playerCards:ArrayList<Card>, lastPlayedCard :Card): Card
follows rules of >
< Creates
CountingUpRules
- cardsPlayed: HashMap<String, Integer>
- game: CountingUpGame
- keyPressed: boolean
- selected: Card
- aceClubPlayed: boolean
+ isValidPlay(card: Card, lastPlayedCard: Card): boolean
+ drawAceClubCard(playerCards: ArrayList<Card>): Card
+UpdateKeyPressed():void
1

Figure 2: Design Class Diagram

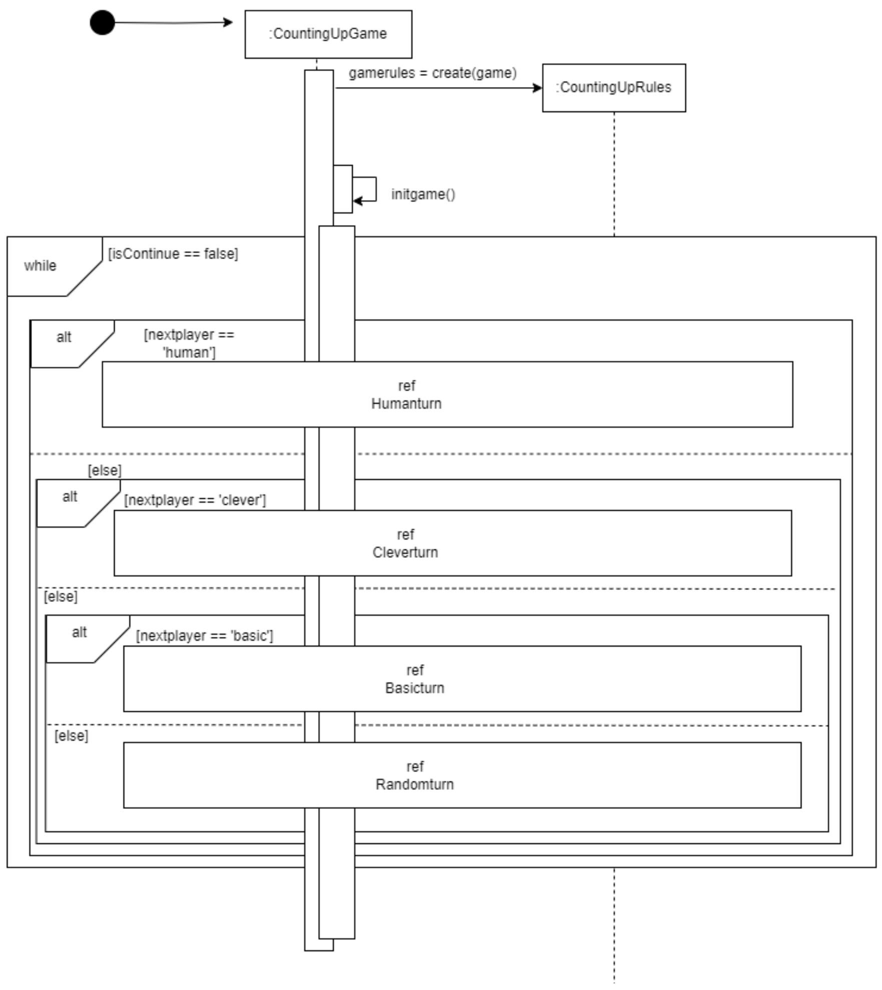


Figure 3.1: Design Sequence Diagram

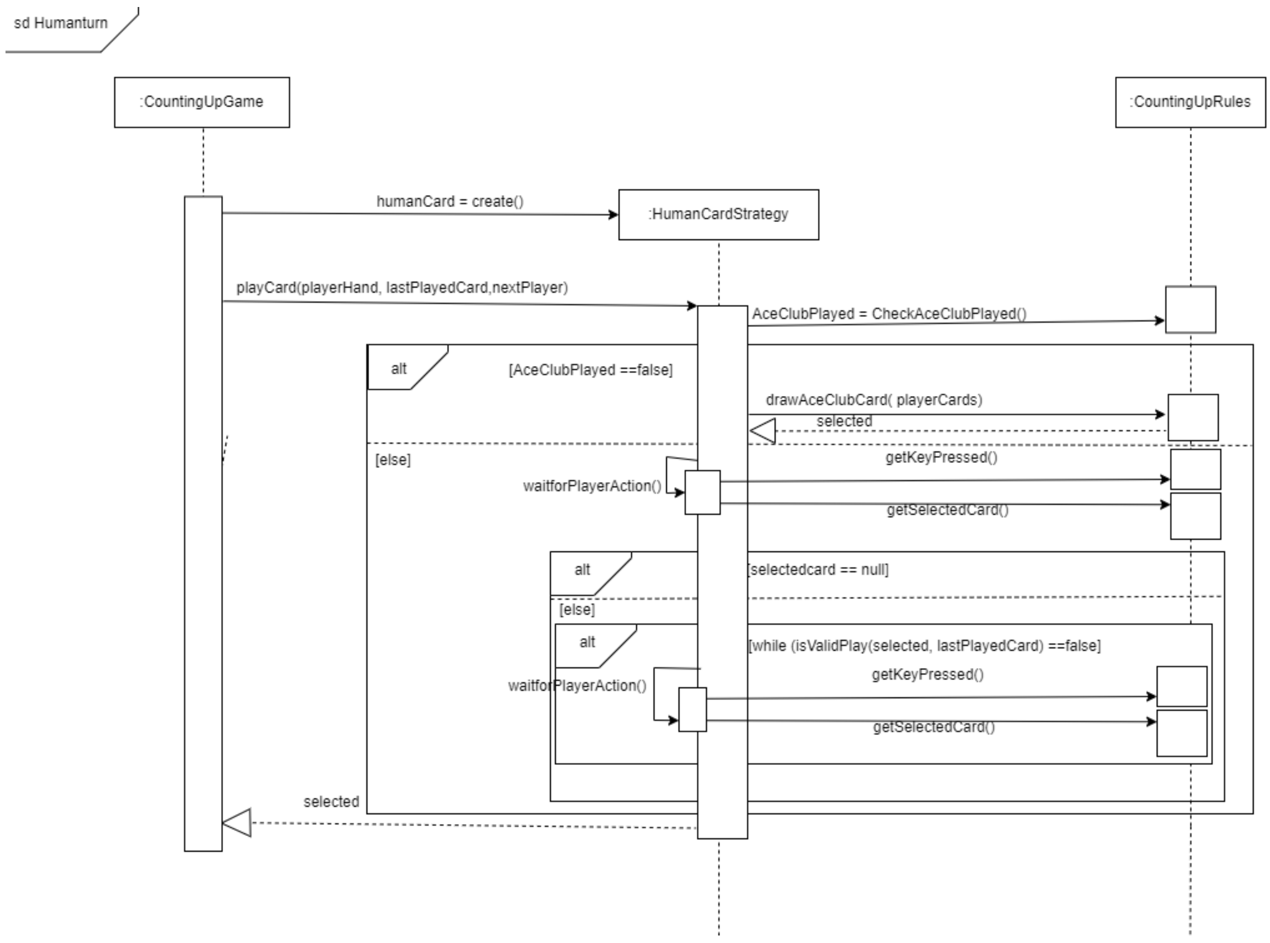


Figure 3.2: Design Sequence Diagram for Human Player

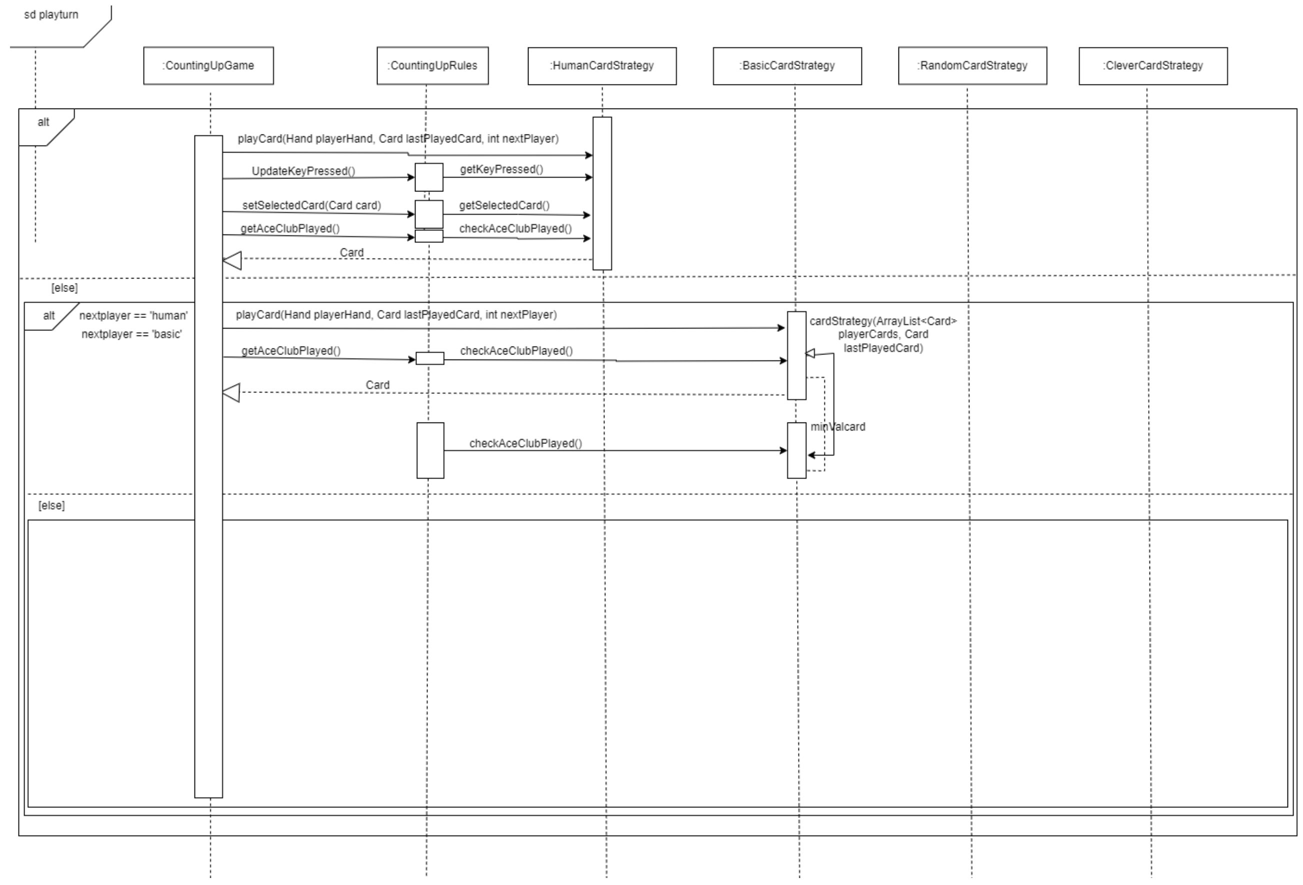


Figure 3.3: Design Sequence Diagram for Player Turn

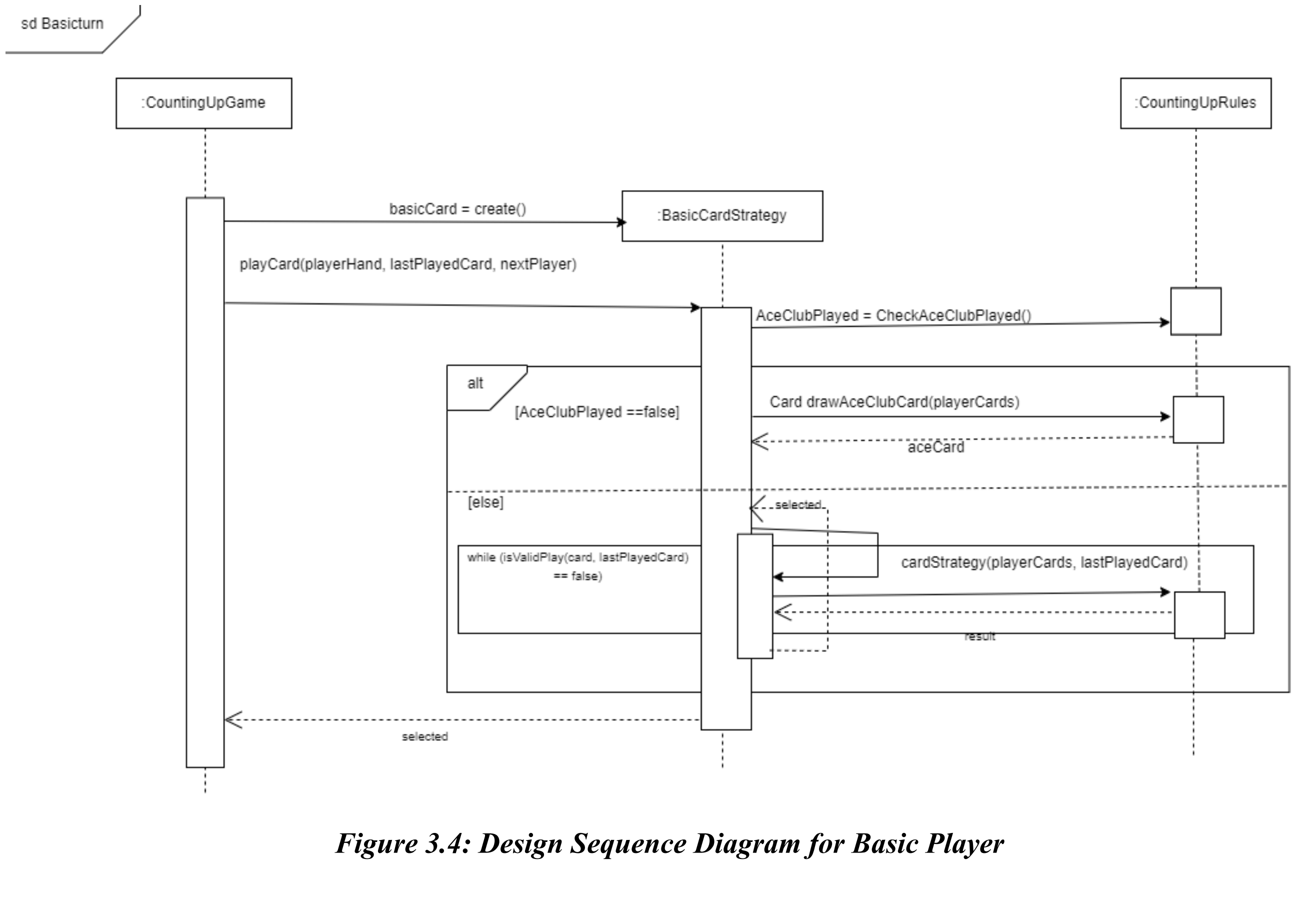

sd Basicturn
:CountingUpGame
:CountingUpRules
basicCard = create()
:BasicCardStrategy
playCard(playerHand, lastPlayedCard, nextPlayer)
AceClubPlayed = CheckAceClubPlayed()
alt
[AceClubPlayed ==false]
Card drawAceClubCard(playerCards)
aceCard
[else]
selected
while (isValidPlay(card, lastPlayedCard) == false)
cardStrategy(playerCards, lastPlayedCard)
result
selected

Figure 3.4: Design Sequence Diagram for Basic Player

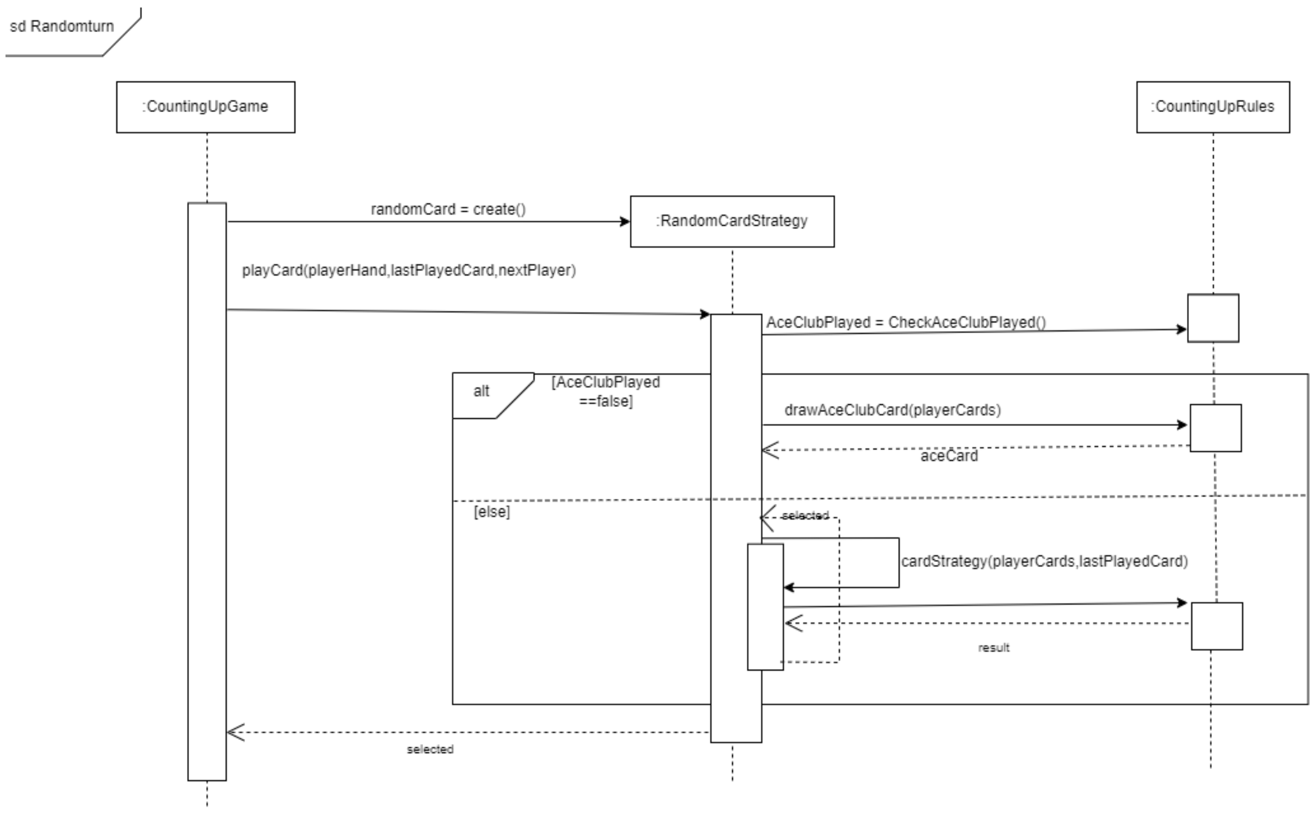

sd Randomturn
:CountingUpGame
:CountingUpRules
randomCard = create()
:RandomCardStrategy
playCard(playerHand,lastPlayedCard,nextPlayer)
AceClubPlayed = CheckAceClubPlayed()
alt
[AceClubPlayed ==false]
drawAceClubCard(playerCards)
aceCard
[else]
selected
cardStrategy(playerCards,lastPlayedCard)
result
selected

Figure 3.5: Design Sequence Diagram for Random Player

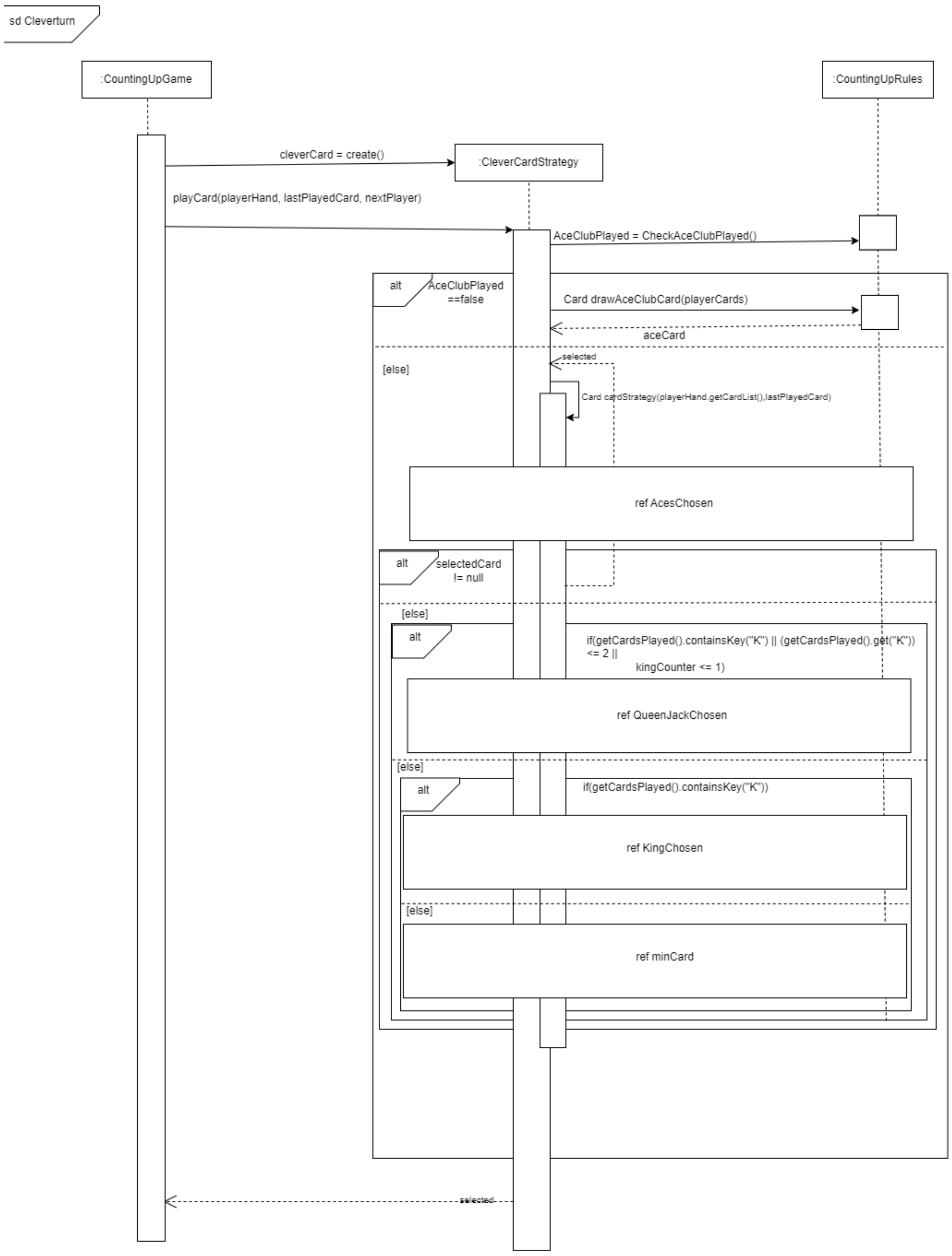
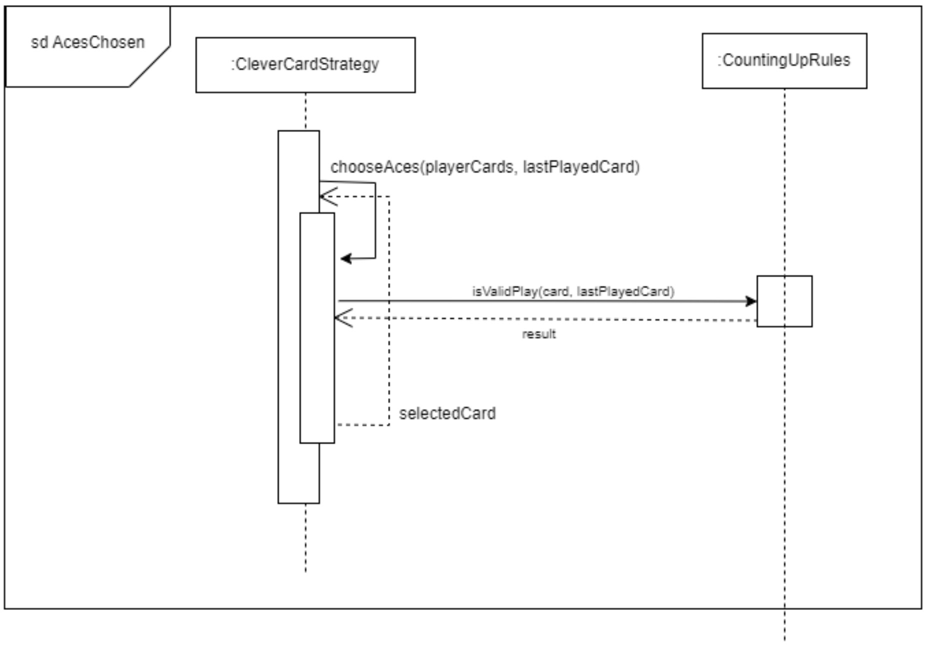

sd Cleverturn
:CountingUpGame
:CountingUpRules
cleverCard = create()
:CleverCardStrategy
playCard(playerHand, lastPlayedCard, nextPlayer)
AceClubPlayed = CheckAceClubPlayed()
alt
AceClubPlayed ==false
Card drawAceClubCard(playerCards)
aceCard
selected
[else]
Card cardStrategy(playerHand.getCardList(),lastPlayedCard)
ref AcesChosen
alt
selectedCard != null
[else]
alt
if(getCardsPlayed().containsKey("K") || (getCardsPlayed().get("K")) <= 2 || kingCounter <= 1)
ref QueenJackChosen
[else]
alt
if(getCardsPlayed().containsKey("K"))
ref KingChosen
[else]
ref minCard
selected

Figure 3.6: Design Sequence Diagram for Clever Player


sd AcesChosen
:CleverCardStrategy
:CountingUpRules
chooseAces(playerCards, lastPlayedCard)
isValidPlay(card, lastPlayedCard)
result
selectedCard

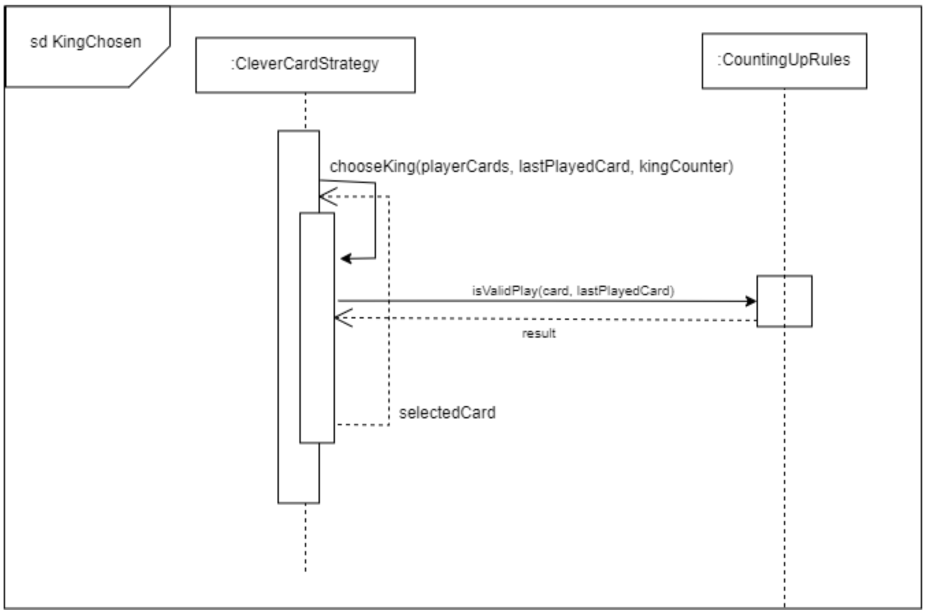
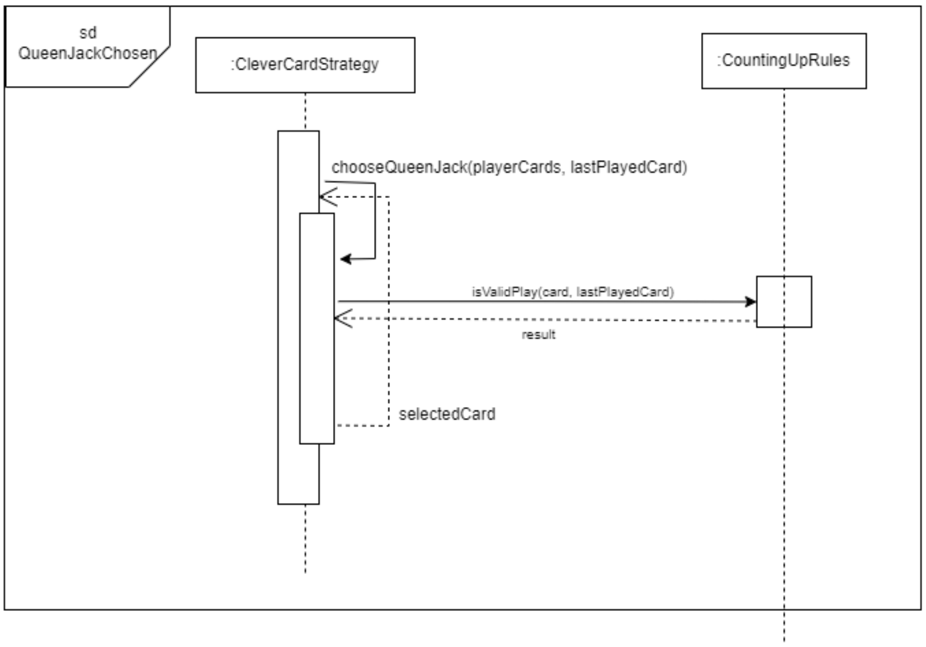

sd KingChosen
:CleverCardStrategy
:CountingUpRules
chooseKing(playerCards, lastPlayedCard, kingCounter)
isValidPlay(card, lastPlayedCard)
result
selectedCard

Figure 3.7: Design Sequence Diagram for chosen King cards and Aces


sd QueenJackChosen
:CleverCardStrategy
:CountingUpRules
chooseQueenJack(playerCards, lastPlayedCard)
isValidPlay(card, lastPlayedCard)
result
selectedCard

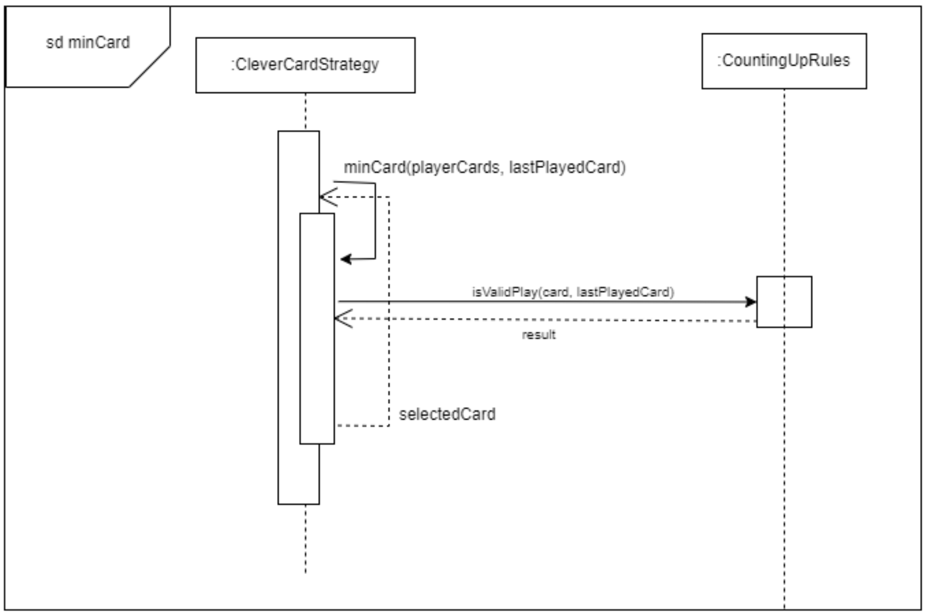

sd minCard
:CleverCardStrategy
:CountingUpRules
minCard(playerCards, lastPlayedCard)
isValidPlay(card, lastPlayedCard)
result
selectedCard

Figure 3.8: Design Sequence Diagram for chosen Queen cards and lowest value card