SWEN30006 Project 2 - Oh Heaven

Workshop 15, Team 14
James La Fontaine 1079860
Yun Keng Leong 1133704
(2 member team)

Our goal was to redesign the original program to simultaneously support the new functionality, more closely follow GRASP principles, and make use of GoF patterns while also preserving original behaviour. We implemented multiple new classes and refactored Oh_Heaven to accomplish this. The classes added include Round, NPCInitialiser, Player, PropertiesLoader and the NPC interface (RandomNPC, LegalNPC, SmartNPC)

Part 1: Problem Domain to Domain Model

We ensured that we had a sound understanding of the problem before beginning by constructing a partial domain model (see Figure 1) to cover all the aspects of the Snakes and Ladders Game.

Part 2: Domain Model to Design Model

Our design class diagram (see Figure 2) depicts the actual implementation alongside the changes that we made, with old / mostly unchanged attributes and methods being colour coded in red. The modifications and justifications for the design will be discussed in more detail below.

2.1 - Initial Refactorisation

We decided that the original design did not demonstrate high cohesion as all responsibilities and information were grouped up into one large and unfocused class (the Oh_Heaven class). We decided that cohesion could be greatly increased by breaking up the program into smaller subclasses and delegating the aforementioned responsibilities to the new subclasses. Through doing so, we aimed to improve the manageability, reusability and understandability of the program.

We began by creating a Round class which has the responsibility of storing the information and methods necessary to perform a round of Oh Heaven and therefore is the class which drives each Round of Oh Heaven. More specifically, the playRound() method was moved into the Round class alongside any other methods that are required for the playRound() method to function correctly (e.g. methods relating to bidding, dealing of cards, tricks, etc.). We also converted some of the local variables present in playRound() into class variables of Round to allow easier access to these values. We refactored duplicate code relating to card selection into a method and also added support for any player to be a human player instead of only player 0 being a human player in the original code.

We then decided that a Player class should be created to provide a location to store player specific information and also satisfy the requirement of each player having their own personal storage of game information. We considered the idea of a GameInformation class to store all the necessary information players need to inform their card choices, however we felt that this would increase the complexity of the program and introduce more coupling without benefitting the design greatly. We therefore settled on simply storing such information in the Player class. The game information now recorded in an instance of the Player class includes the player's number in relation to where they are on the table, their current hand for the trick, the cards in play for the current trick, the current lead and trump suit for the trick and round, and all player's current number of tricks, bids and their scores. This information is all accessible quite easily through the use of getters provided in the class and all values are updated at the appropriate times during the game. However, it could be argued that this method of updating information is a limitation of the design and could be slightly tedious for a developer to work around as, if a developer would like to store some new information in the Player class, they must determine where exactly in the code this information must be updated for each player. We considered moving hands from Round to be stored in the player instances instead, as the players logically possess these hands and the NPCs need to be able to see them to make decisions about which card to play. This also aimed to reduce coupling and improve cohesion by preventing the program from having to store hands in both the Round class and Player class and keep two arrays updated and synchronised, and instead leave the hands to be possessed solely by the player class. However, this would violate the restriction of players only being able to see their own hands so we had to compromise to satisfy this design rule.

We originally discussed whether we should leave the responsibility of score tracking and managing to the Oh_Heaven class or if we should create an entirely new class to handle this responsibility, therefore increasing cohesion and delegating responsibility away from Oh_Heaven at the cost of increasing design complexity. We decided that the small responsibility of the score handling methods didn't warrant an entirely new class to house them and instead decided to leave the responsibility of driving score updates with the Oh_Heaven class. It seems quite appropriate to have these methods in the Oh_Heaven class as the Oh_Heaven class drives the whole game, and the score is a value which must be recorded throughout the whole game rather than on a per-round basis. However, the Player class actually stores the copies of each player's scores in line with the specification.

Throughout the design of these subclasses, we utilised the principle of information expert to determine where variables and methods belong which allowed us to achieve higher cohesion than was present in the original design and also keep the design manageable while decomposing Oh_Heaven. While this decomposition into smaller subclasses has of course increased the coupling in the design, we believe that coupling is still at an acceptably low level and that the greatly increased cohesion is a more than sufficient tradeoff.

2.2 - Additional NPC Types

In regards to implementing additional NPC types and supporting future NPC types, we determined that the Strategy Pattern would be suitable for this task as the algorithms to select cards all vary in logic but share the common goal of selecting a card for a player to play. We created a common NPC interface for the new RandomNPC, LegalNPC and SmartNPC classes to implement. Contained within the NPC interface are the chooseCard() and getNPCName() methods. chooseCard() contains the logic which the particular NPC implementation uses to choose a card, while the getNPCName() method returns a string matching the one used to specify this NPC in a properties file (this will be further explained in Section 2.3). Through this use of polymorphism we achieved a design which upholds the principle of protected variation and is both highly cohesive and keeps coupling low between NPC implementations and the rest of the program by only allowing interaction between them through the chooseCard() method. The only class that NPCs depend on is the Player class due to the fact that a player instance allows an NPC implementation to easily access any game information they require as long as that information is stored in the Player class and is updated appropriately throughout the game. This should mean that any developer who wishes to develop an even smarter NPC in the future should find it quite easy to implement, and would only have to make additions to the Player class if there is some form of game information which is not already present or cannot be inferred with the currently provided information.

To handle the creation of NPC types in the program, the Factory Pattern and Singleton Patterns were implemented through the NPCInitialiser class which is a pure fabrication that simply serves the purpose of housing the initialisation code for implemented NPC types and providing a static method with which the rest of the program can access the single instances of these NPC types. This aimed to keep cohesion high and coupling low by providing a singular location in which developers can add their NPC types to the program while also instantly being able to access them anywhere in the program through the getNPCs() method.

Refer to Figures 3, 4, 5, and 6 to see how our specific NPC implementations interact with the different classes and function to select a card.

2.3 - Property File Specified Parameters

When deciding how to add functionality for adjustable parameters into Oh Heaven through a properties file, we considered also delegating game initialisation responsibilities originally assigned to Oh_Heaven to a GameInitialiser class. We decided that the responsibilities weren't great enough to merit their own class and increase design complexity and so we left these responsibilities in the Oh_Heaven class. The Oh_Heaven class therefore sets up its own values for important parameters if none are specified by a property file. We then created another pure fabrication called PropertiesLoader which has the role of loading in properties from the file into the appropriate variables within the game. It simply reads in properties from the specified file and, using static methods, sets the specified value for each parameter in the appropriate location. For the task of initialising each player with the specified NPC type, we decided to make the RandomNPC class - which stores the logic for random card selection - to maintain conformity with the Strategy pattern that was used to implement the other NPC types and support future NPC types. We then simply use the NPCInitialiser to retrieve an array of each NPC instance and iterate through this array until the getNPCName() method returns a match with the string provided in the properties file. A new Player instance is then created to be stored in the game with the appropriate NPC type. Human players are instantiated with a null NPC type and have an isHuman boolean set to true to denote that they are a human player and choose their cards on their own. If no player type is specified in the properties file then we default to a random NPC type for that player. We decided to go with the latter option by creating the aforementioned getNPCName() method which requires implementation with an appropriate string for any new NPC types.

Refer to Figure 7 to see how our PropertiesLoader handles this task in our design.

2.4 - Extending the Design to Support Smarter NPC Bidding

This design could be extended to support smarter NPC bidding simply by adding a makeBid() method to the NPC interface which each NPC would be required to implement. Developers could determine here how they want their NPC to decide on their bid amounts. This would utilise a similar approach to the solution of supporting NPC card selection variations through the Strategy Pattern and polymorphism, therefore upholding the principle of protected variation. Developers aiming to create a smart NPC could access the allPlayerBids array stored within the current player's instance and could decide on their bid based on what previous players have bid. The trump suit for the round could also be accessed from the same player instance and used to judge how strong the current player's hand is based on how many trump suit cards they have and how many high cards they have. The makeBid() method would be called for each player during the initBid() method at the start of a round in the Round class. It should be noted that our particular SmartNPC design would likely perform much better if smarter NPC bidding functionality was added to the design.

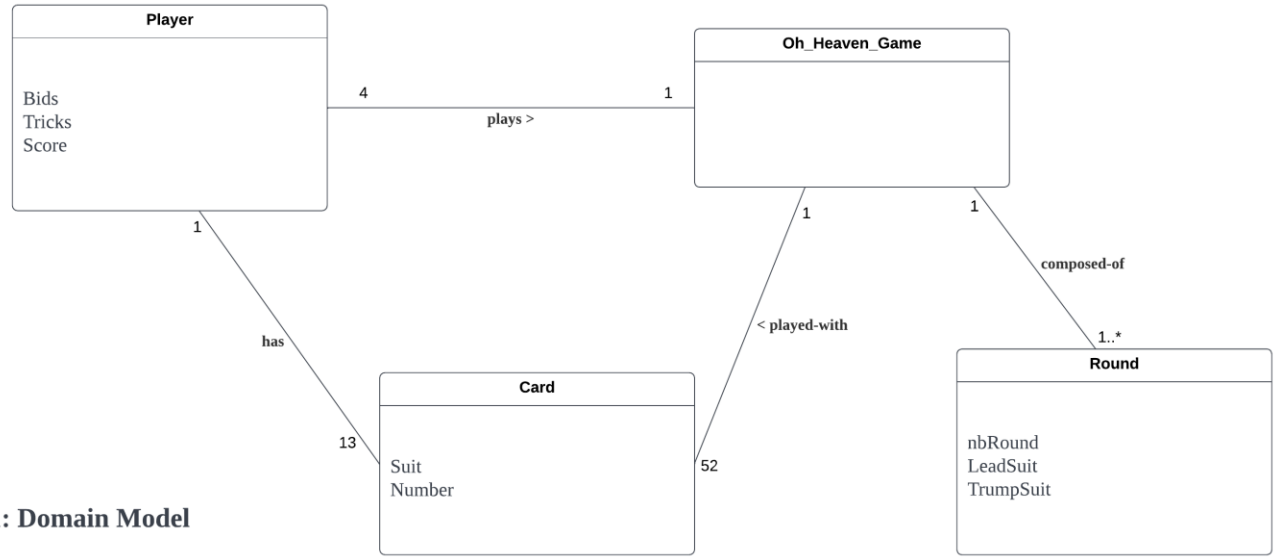

Figure 1: Domain Model

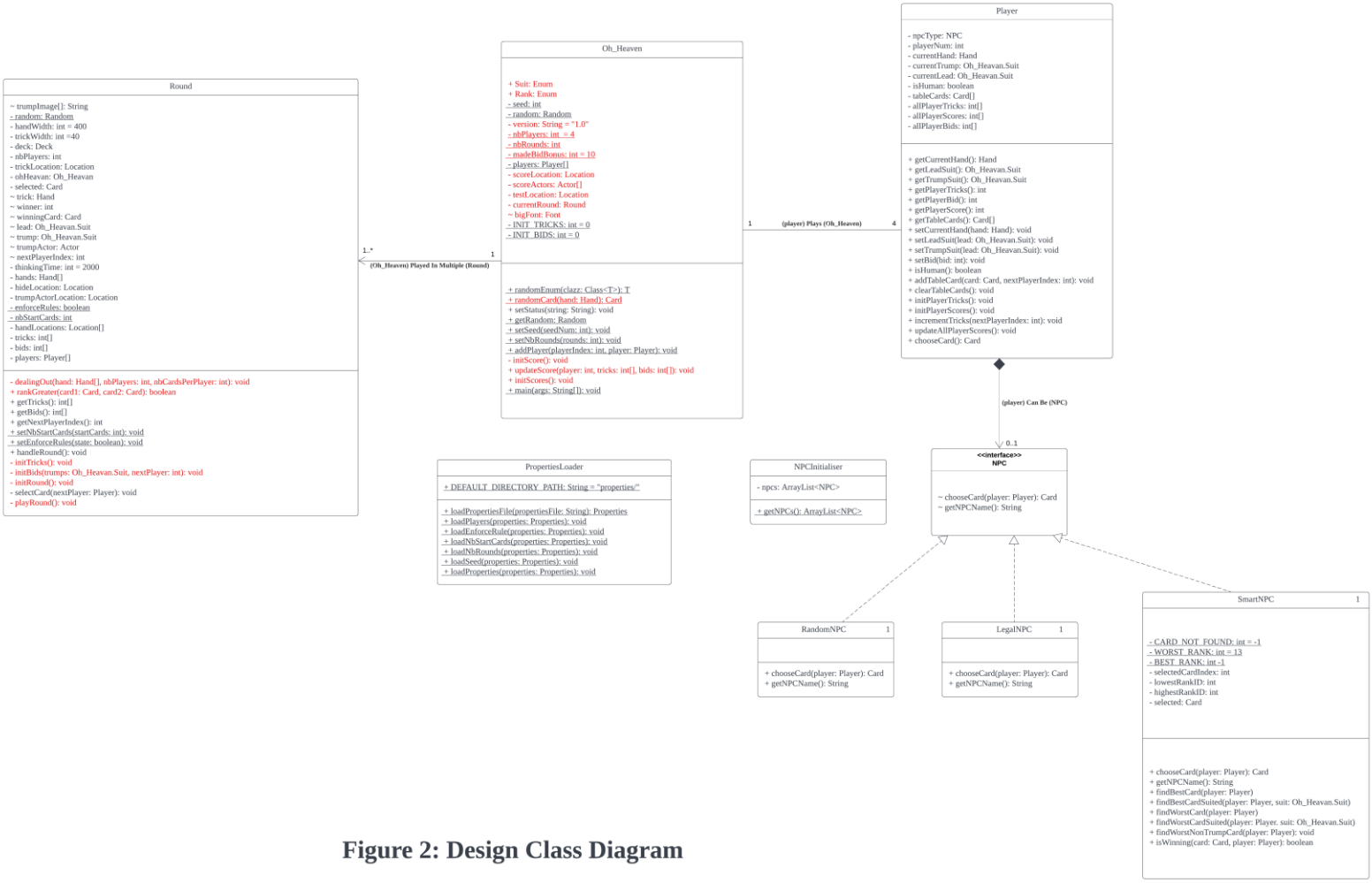

Figure 2: Design Class Diagram

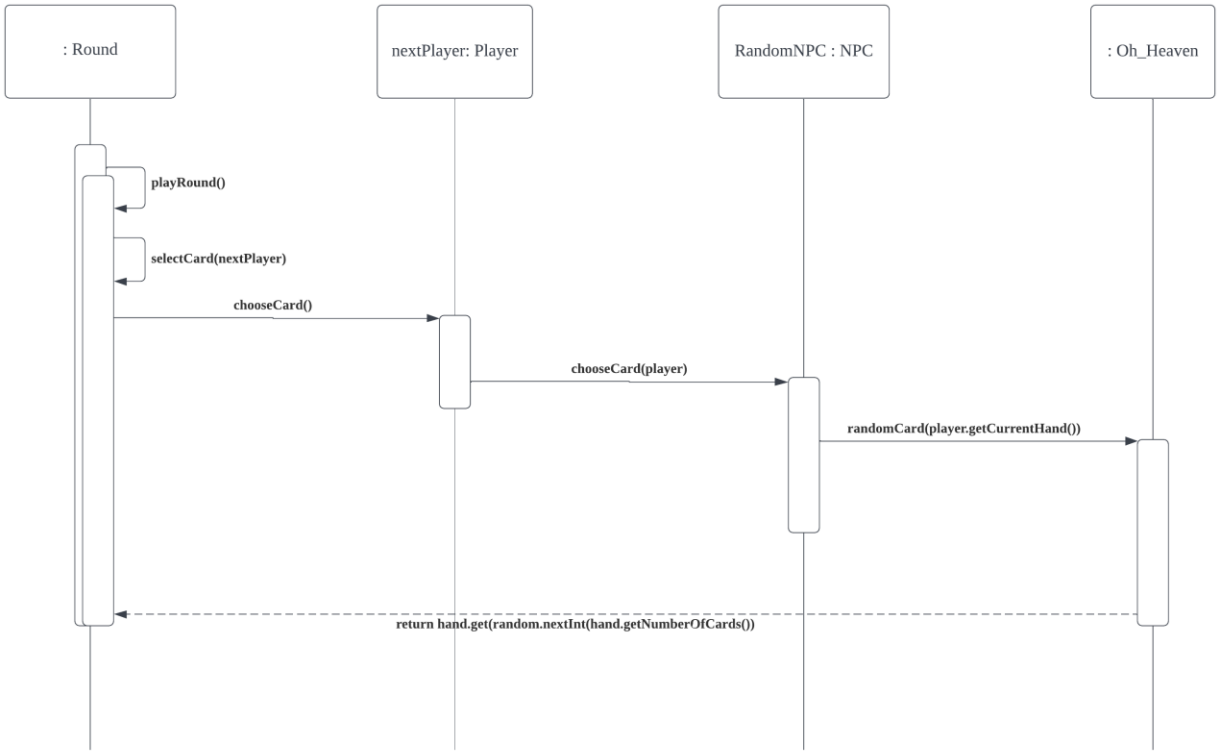

Figure 3: RandomNPC Sequence Diagram

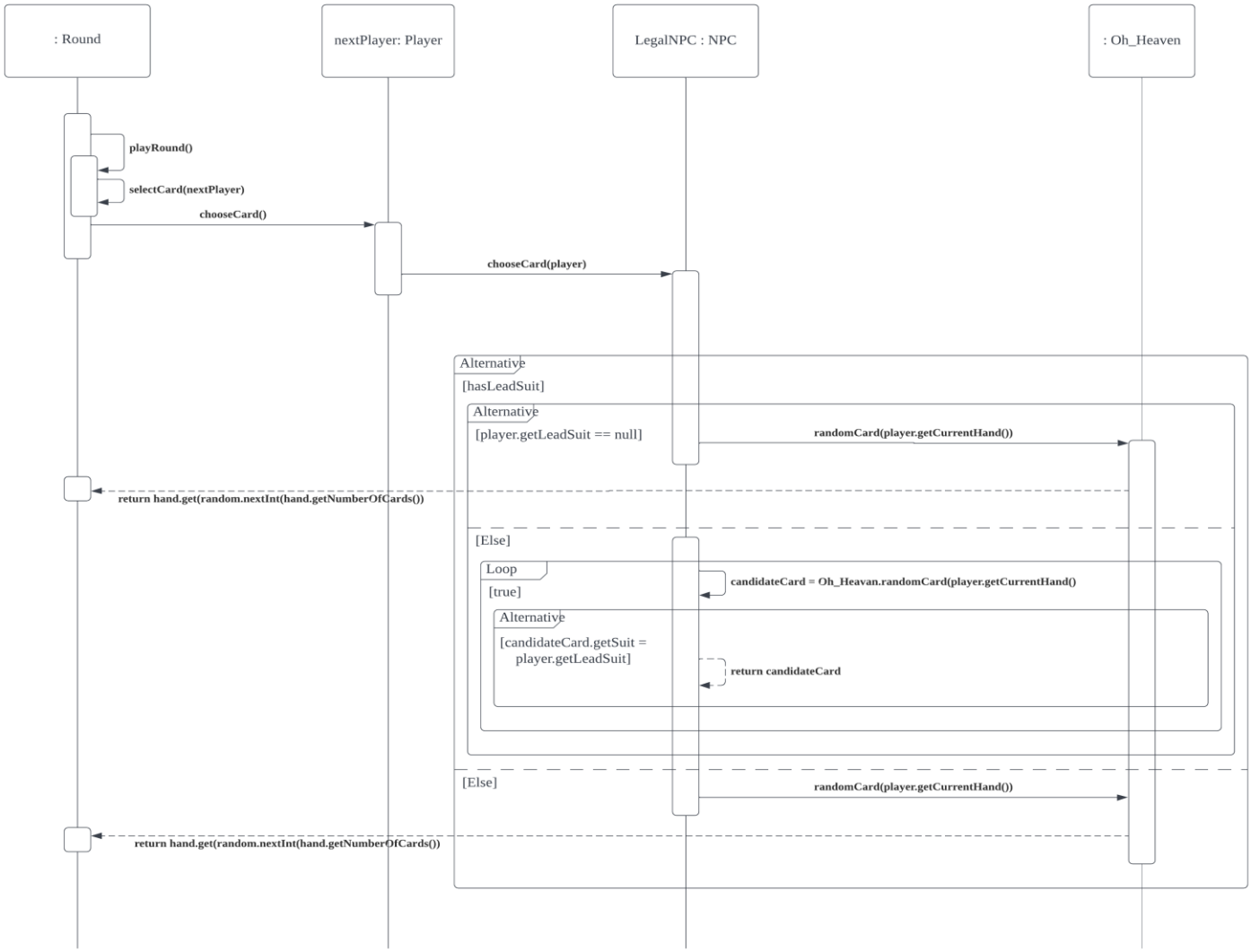

Figure 4: LegalNPC Sequence Diagram

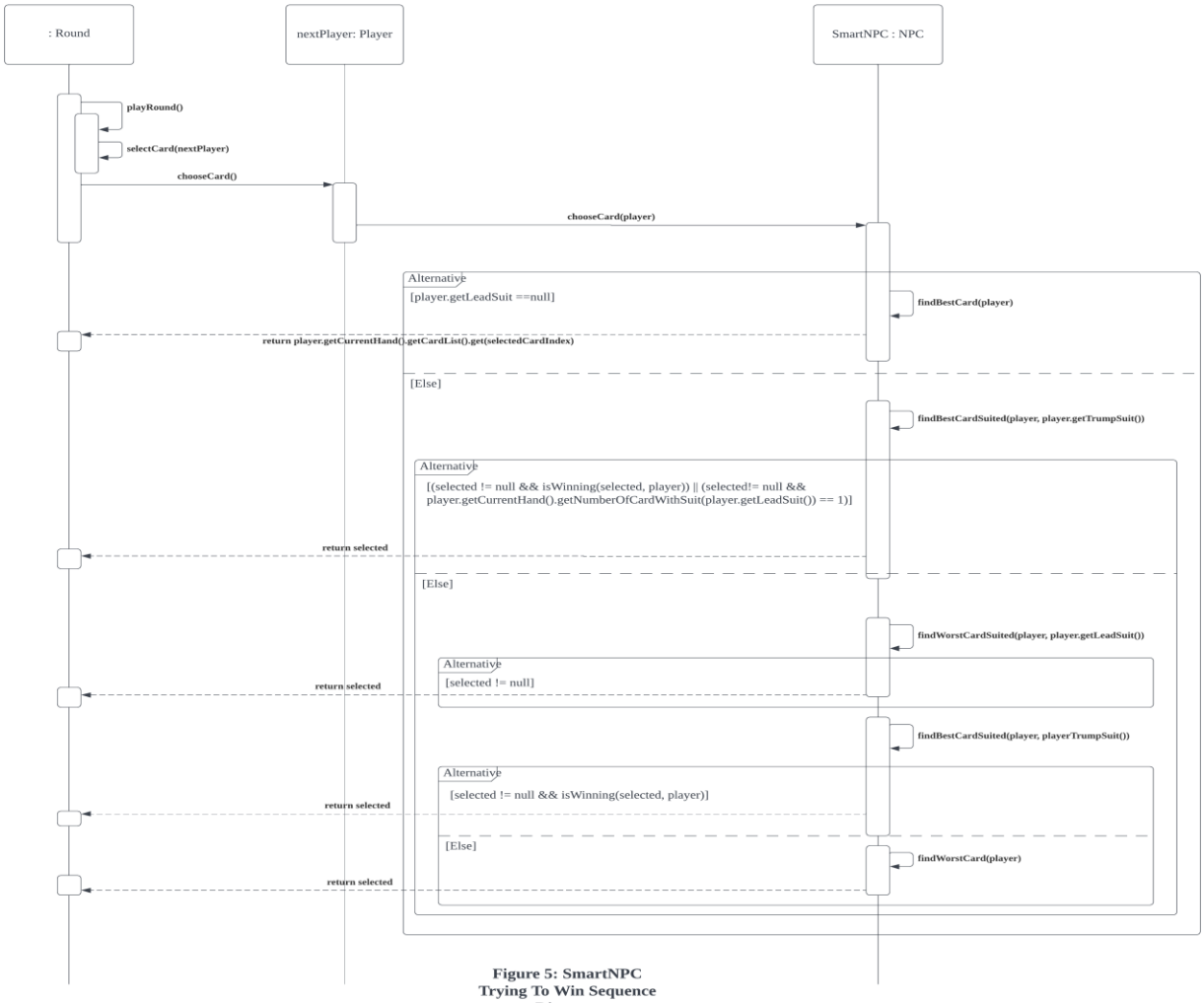

Figure 5: SmartNPC Trying To Win Sequence Diagram

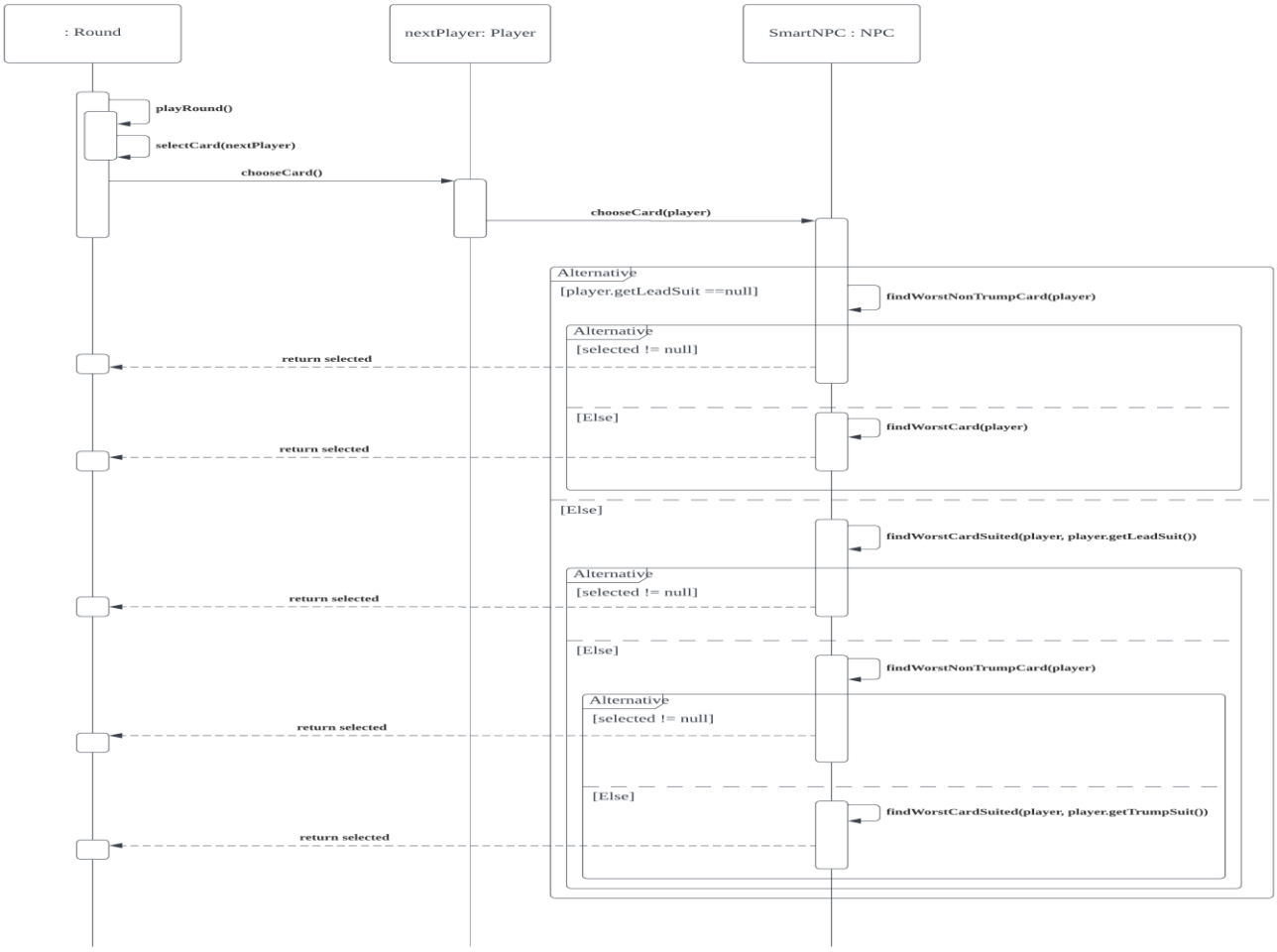

Figure 6: SmartNPC Trying To Lose Sequence Diagram

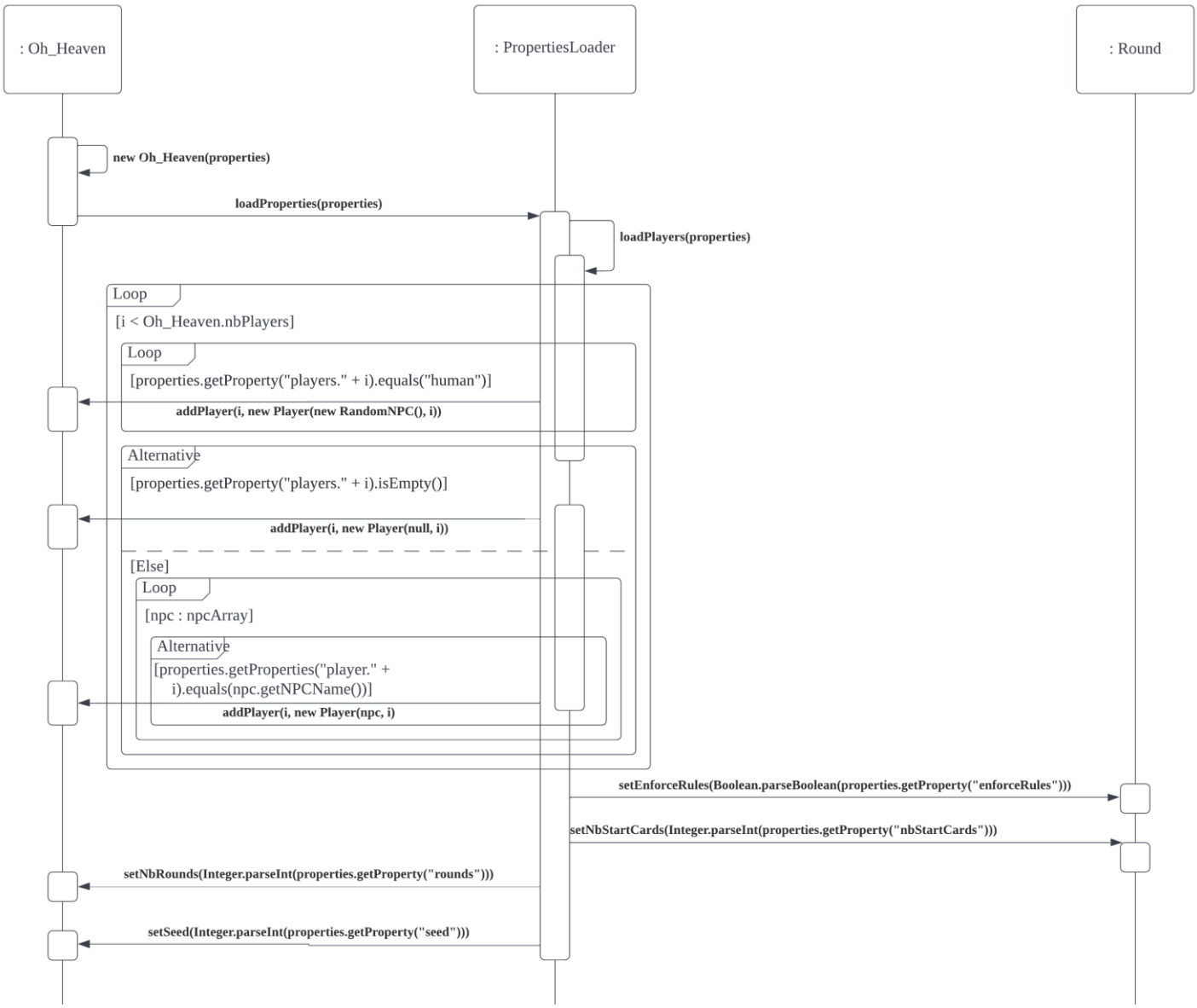

Figure 7: Properties Loader Sequence Diagram