Introduction:
A video game based on a popular card game called Oh Heaven which is developed by The Games division of New Exciting Realtime Discoveries Inc. (NERDI).

As a simple card game, Oh Heaven uses a standard 52-card deck with 4 suits and 13 ranks in each suit. The game is played independently by four players, and the one with the highest score wins. There are three rounds in total, each round randomly selects a trump card, and the player with the highest card in each round gets one point. At the end of the round, any player who made exactly the number of tricks that they bid will receive an additional 10 points.

Currently, the game can run well in original mode, the code is concise and easy to understand is a clear advantage. However, there are a few drawbacks in the current model. The first is the limitation of the current mode, which only supports one player and three NPCs who do not obey the rules, while legal NPCs and smart NPCs are required for playability in our game. Furthermore, there are some bad designs in the current build of this game. These designs and the lack of documentation lower the efficiency to implement new features and maintain the code. Almost everything is added in a class resulting in poor design of high coupling and low cohesion. In order to achieve a well-extensible or extended design based on GRASP and GoF patterns, help prevent subtle issues that can lead to major problems and improve code readability for coders and architects familiar with the patterns, we proceed improvement on the current basis.

Overall, the current build of Oh_Heaven does not have a quality design, and our goal is to change that and add some new functions to this application.

Oh_heaven:
In this application, Oh_heaven, as a class that can represent the entire system, is a use case controller, responsible for accepting and handling system events. In our design, Oh_heaven is well designed as a controller. After the user interface, it is the first object to receive and process system actions. It dispatches the task of handling system events and coordinates global information. Thus it's reasonable to make it the controller of this application, and anything else including functions and attributes that are not responsible by the controller has been extracted out of the Oh_heaven class and is put in other classes. Currently, the Oh_Heaven is accountable for reading properties from the designated properties file, using CardFactory class(a factory that is responsible for dealing out cards, details will be revealed later in this report) to deal out cards into players' hands before each round starts, managing player’s score, generating trump cards with Randomizer class and initializing bids for each player.

Player:
Our task is to implement a legal NPC who plays legal cards, and a smart NPC who is a legal NPC but somewhat smaller. The challenge in completing this class is that the current build does not differentiate between NPCs and human players, and basically has the same behavior. So we introduced the Player class. As shown in *figure 1*, Player class is extended by Human and NPC, representing two types of players in this game, and a composite class

Allplayer, that is responsible for taking care of each player's play in a single round. The player class has an abstract method of playhand, for classes that extend this class, this is where they specify what strategy is used to play a card. The parameter of the playhand method is Oh_heaven class because some NPC needs information about what cards the other players played and Oh_heaven is the only class with that information, and the composite class Allplayer also needs to interact with the Oh_heaven class. We've decided to use a composite pattern rather than a strategy pattern because we think that each player has its unique behavior when playing a card and can not switch to other behaviors, for example, a human can not automatically play a random card. Also, there are players that hold unique information and methods for the child class to use, and thus it makes more sense to use a composite class.

Allplayer:
Allplayer is the composite class in this application, it is an array with a length of 4, containing all players on the table. Its playHand method, when called, goes through all players and calls their playHand method one by one until all players run out of cards for this round. After each player plays their card, it gets transferred into a trick, which is a collection of cards that are already played by other players. Taking advantage of the composite pattern, every player has a playHand method, thus to play a card, Allplayer does not need to know what type of player this is, it just needs to call the playHand method. As shown in the sequence diagram, after the program reads what type of player each player is(smart/legal/human/random), there is no "if" statement to check which type of player each is when card, it just call the playHand method in each type of player and a card is played.

NPC:
An NPC is a type of player that plays automatically, it overrides the playHand abstract method with three steps, first, it delays the game by a preset amount of time to simulate a human player thinking before making the actual move, then it proceeds to calls the think method, which is the actual process of picking the card, after picking the right card in think method, it must set it to the selected attribute, which is the card that the playHand method finally returns. We've decided to override playHand method in the NPC class rather than in each specific type of NPC class because every NPC has a thinking time, so to avoid repeating the delay function in each subtype of NPCs, we moved it to the NPC class.

RandomNPC:
The randomNPC has a very basic strategy, it plays a random card generated using the Randomizer class. The procedure of getting a random card is written into a method rather than just included in the think method because legal npc and smart npc that comes after this type also need to play a random card in certain situations.

LegalNPC:
A legal NPC is a type of NPC that plays a legal card and is a randomNPC. A legal card, by definition, is a card that is the same suit as the lead card. The playLegal method is used to generate a legal card, it first checks whether there are other cards on the table, if there is, it returns a card with the same suit as the lead card, it is the first player to play, it returns a random card using the getRandomCard method defined in randomNPC.

SmartNPC:
A smartNPC is a legalNPC but a bit smarter, its playSmart method is used for it to decide which card to play. It first checks whether its trick is equal to the bid, if it is equal, then the smart player will try to lose this round by playing the smallest legal card that it has in order to get the bonus scores when the round ends. If the trick is either larger or smaller than the bid, then the smartNPC will try to win this round. To try its best to win a round, it will first find the largest card on the table with findLargestTrick method, then it finds the largest card on its hand with findLargestLegal, if its largest legal card is larger than the card on the table, then it will play that card if not, it will just play a legal card using playLegal method defined in legalNPC. Note that if smartNPC will play a random card when it is the first player to play or has no legal card on its hand.

Human:
Human is a type of player other than NPC, a human requires direct operations from the user, thus it has a card listener, when its playHand is called, it will check constantly if the user has double-clicked on a card, if a card has been double-clicked, then playHand will be ended by returning that card.

CardFactory:
CardFactory is a pure fabricated class with the responsibility of dealing out cards to players, it is also a singleton. We've decided to make cardFactory a singleton because it holds basically no information, and every class created is basically the same. It's dealing out method can operate basically on itself with the input parameters. CardFactory is a class with only one useful method and there can be an option to include it in other classes like Oh_Heaven, but we think that dealing out cards is not really Oh_Heaven's responsibility, and it will decrease the cohesiveness.

Randomizer:
Similar to cardFactory, the randomizer is also a pure fabricated class, it can be considered as a sort of utility class that can either generate a random card or a random enum based on the seed.

Properties Loader
Properties Loader is also a pure fabricated class dedicated to reading properties from properties files, different from the other two pure fabricated classes, propertiesLoader is not a singleton, because it can read from different files based on the file path passed into the constructor. However, all of its methods are static, we decided to represent it as a singleton on the sequence diagram. The PropertiesLoader will locate the assigned property file in runmode.properties, currently, the properties loader only supports reading the seed, nbStartCards, rounds, enforeceRule, and all 4 player type, but more properties can be read through adding more methods in the ProertiesLoader class. Because all of the methods are static in PropertiesLoader, an instance does not need to be created in order to use any methods.

How the program works
When an instance is created for Oh_Heaven, it will first read the properties from the properties file, and then create four players corresponding to the type inside the property file.

For the first round, the program then will initialize scores tricks, and bids for all players, and all players, one by one, players start to play their cards until all players' hands are empty. This process will repeat will for the number of rounds set in the property file.

Task 3 - Comments on how to accomplish smarter NPC bids:

The 'Oh Haven' game consists of 52 cards and each suit consists of 13 cards exclusively. As there are 4 players in total each player holds 13 cards. The design can be extended to make smarter bids by following a basic logic where the more suits one player holds, with higher the number, the more likely the player will have more wins. Whilst the player with less suits and smaller the number, the player will be more likely to have less win.

At the start of each round of the game, the initial bid should be set at 3 wins which is the average win divide amongst 4 players. The chance of a player to have a win number equal to 0 is rare - an individual with 1 suit will either have 13 wins if that person is randomly selected for leading and 0 win if that person is not selected for leading. 13 is impossible due to the game rule. As a result, 0 and 13 wins will not be considered by the AI except in rare cases.

In the majority of circumstances, it is better to have more suits (4 at maximum), which individual will likely have different suits and value of number, individuals who have 4 suits and each suit with 3 cards with higher number cards (any number that greater than 7) will have a higher winning score.

For instance, players who have the 4 different suits will add 1 score to the initial set of wins (3) and AI will predict win for 4, players who have different suits and relatively higher numbers will predict win for greater than 4. Vice versa, players who have less than 4 suits with a smaller number card (any number less than 7) will predict win for below 3 but greater than 0. This method will give a current score with consideration of combination of suits and numbers.

Additionally, this current score can be further extended by considering getting the same number card of different suits - where one player may have 4 cards of K from 4 suits, this considerably increase the likelihood of winning and should be considered toward changing the current score, likewise where one player may have 4 cards of 1, 4 cards of 3, this player is likely to loss many games as most of the smaller cards are withheld and larger values of cards are dispersed to other players, this player might have a winning score 1 or 0.

Since a card with a large number would be more likely to win over other cards which are in the same suit, thus if a certain number of large number cards coexists in a deck of cards, the expected bid should be increased, vice versa. The best possible deck which has a 100%-win rate would be a deck of cards including every highest valued card which belongs to each suit plus a random card. The suit variety could impact a deck's performance; however, the value of cards is still the largest factor which should be taken into consideration when bidding.

As a result, the logic which is mentioned above could be implemented in the score expectation system and apply to the situations/examples below:

- Player with 13 cards consist of 3 or more cards of one particular large number (add 1 score toward current score)

- Player with 13 cards consist of 3 or more cards of 2 or more large numbers (add 2-3 score toward current score)

- Player with 13 cards consist of 3 or more cards of 2 or more small numbers (minus 1-2 to current score)

  Unless extreme conditions are met, the variety of suits in a deck of cards should not be taken into serious consideration. Although several cards which do not belong to the leading card's suit might be played to be "wasted", however, since all the cards will be played eventually at the end of the game, the resulting number of tricks done should not differ by large amount compared to decks with every suit.

In the above example card which has value less than or equal to 5 is considered as a "small number" card, card which has value larger than or equal to 9 is considered as a "large number" card.

Please note that the example above is just a demonstration of how the score expectation system should operate, certain actions such as adjustments to the value of "small number" and "large number" may be required for the system to perform accurately.