SWEN30006 Project 2: Oh Heaven!

Workshop Friday 13:00, Team 8

Andrew Nguyen-Dang	913711
Sean Maher	1079800
Elizabeth Wong	1082634

UML Design Diagram:

The UML design diagram (see Figure 1) only includes classes that have been created outside of the original Oh_heaven file as well as additional functions and attributes implemented. The UML design diagram is useful in illustrating a visual representation of how classes interact with one another as well as displaying the design patterns and principles used within our Project.

UML Sequence Diagram:

The UML sequence diagram (See Figure 2) illustrates how and in what order our classes work together. It models the logic of legalNPC and depicts how objects and components within our Project interact with each other to complete the process of performing legal moves.

Refactoring Oh Heaven.java

Split Oh Heaven to make it more modular with additional classes whilst maintaining the same rules and game state as per the requirements. This decentralises control of the game as well as improving the readability of the code, allowing for future additions and improvements to be easily made by other team members.

Other classes will need to access Oh_Heaven, and since there will be only one game running at any time, we stand to benefit by employing a singleton pattern, which provides global accessibility to any other class via a single access point. We also utilise an observer pattern which observes the ‘table’, being every card played onto the table. This is used to collect data on which cards have been played by which player, under which leading suit (if any). We can use this data later through our Smart NPC to enable it to make better decisions based on the gameplay information. Since the observed information will be the same for any player on the table, we also employ the singleton pattern for this observer. The information observed from playing on the table should be the same for every player, however, the way in which they digest and strategize around this data should be independent for every player.

Additional NPC Types

All NPC types are implemented via the adapter pattern through playerAdapter and each NPC type's adapter respectively. Similarly the human player has its own adapter. The adapter is a structural pattern which allows these other classes (eg. legalNPC, smartNPC, Human, randomNPC) to work together, passing different arguments to the concrete NPC class. All adaptors are implementing the Player Adapter which is an interface that is used by the Player class. Player is its own class but also has a factory pattern, however the pattern has not been made into its own new class eg. PlayerFactory. Adapter classes are applied inside our redesign of Oh Heaven to take into account any additional extendible behavioural functions that a client may request besides the original playCard function.

Legal NPC

For the legalNPC which is only allowed to perform legal moves, the behaviour of randomNPC was used as a base. Firstly, it checks if the card it selects is a legal play. I.e does the NPC have a card of the same suit as the lead and if so, are they playing it? If this is not the case, we then re-randomise the draw until the card selected by the NPC is determined legal by Oh Heaven's game rules.

Smart NPC

The specifications for the assignment requires that the smart NPC performs better than the legal NPC overall, which is true for our smart NPC given that the legal NPC does not receive an overwhelmingly better hand. Our smart NPCs' strategy pattern revolves around the game being in 2 different states. The first state is that our NPCs number of trick wins does not equal the bid it made at the start of the game. When this is the case, it employs our 'winning strategy' where, based on the logic we provided and if the NPC is the leading player or following, it will do its best to win every trick. The second state is when the number of tricks won is equal to the bid made at the start of the game. In this state, the smart NPC will employ the 'losing strategy' whereby it will do its best to lose according to the logic we provided taking into account if they are leading or following.

A core part of our smart NPCs winning strategy makes use of the observer pattern via the TableObserver class which monitors which cards have been played and also which players have exhausted a given suit. This information is used by our smart NPC to decide whether the card they play can be beaten, if so it will hold onto the card and play a lower card, otherwise if the card they have is the highest of the suit that hasn't been played, it will play that card as it is most likely going to win, given that all other players haven't exhausted that suit. When leading, the smart NPC checks whether it has the best card of a given suit, if it does not, it will play the highest card it has of the suit which it has the least of. This allows the NPC to attempt to exhaust a suit allowing them to play a trump card in earlier rounds in response to a non-trump lead suit.

The loss strategy which we implemented has the NPC, if they are leading, playing their lowest card of the suit which they have least of. If they are following another lead, they will play the lowest card they can play legally. Due to the possibility of the NPC going over their bid, the NPC will continue the win strategy if the bid is exceeded.

To clarify any confusion the pseudocode logic behind the smart NPC:
The 'win strategy' is as follows:

- If the player is leading:
    - If they have the highest unplayed trump card, play it.
    - If they have the highest unplayed card of other suits, play it.
    - Otherwise, play a card of the suit in their hand with the least cards, playing the highest of that suit. This is so that they are able to play their trump cards more easily on rounds when they do not have a card of the lead suit.
- If the player is not leading:
    - If they have the leading suit:
        - If they own the highest unplayed card of the suit, play it.
        - Otherwise, play the lowest. (Don't want to waste your King on a trick if the Ace hasn't been played yet).
    - If they don't have the leading suit:
        - Play the highest trump card, if they have one
        - Otherwise play the card of a suit with the least cards, to get rid of the suit, prioritising the lowest, since they won't be winning this bout.

Extremely Good NPC

Currently, our smart NPC is quite capable of making good plays in this card game. However, we've built some of the framework for a better version to be made, should one desire to do so.

The TableObserver pattern examines each card being played, and keeps track of which players have run out of certain suits using the unexhaustedSuits attribute. This is determined when a player (legally) plays a non-lead suit. An even smarter NPC could use this information to make better decisions, perhaps if they know another player has none of a suit, they could avoid leading with it, to prevent them playing a card of the trump suit.

Another addition we could make to the NPC to make it better is to plan for wins they will make in the future. This would prevent the NPC from reaching its bid amount and still having cards that would result in a win. For example, if it knows it has the highest trump card remaining, it could count that as a win already meaning that it is less likely to go over its bid.

Parameters (PropertiesLoader)

We constructed a PropertiesLoader class to configure the initialization of the game based on a properties file. The default properties included from the properties files we were given as follows:

- seed
- nbStartCards
- rounds
- enforceRules
- Behaviour for each player (human, random, legal, smart)

This could be extended with additional optional properties which may include, but are not limited to:

- showCards (true/false), in order to keep each players' hands secret
- madeBidBonus, default would be 10
- nbPlayers, default would be 4
- trump: if left blank, which would default to a random suit each round, otherwise would have a set trump suit.

Further player behavioural types could be configured within the player property, which may include new bidding behaviours

Bidding Behavior

The bidding behaviour in the current version of the game is very basic and gives the player a bid of either 3 or 4 regardless of the cards in their hand. This isn't optimal as a player may get a really good hand, meaning such a bid is too low, or they might receive a terrible hand and the bid is too high. If the player has the highest trump card for example, this can be counted as an automatic trick win and their bid should take that into account. Additionally, if a player has a larger number of trump cards than expected or they have a few aces, their hand can be considered better and should bid higher than the current bidding system allows. The opposite situation where the player receives cards with low values or they receive very little or no trump cards should result in the player making a lower bid. Based on our current smart NPCs strategy, if we were to implement a bidding strategy, it would consist of taking into account how many trump cards are in the players hand; whether or not the player has the ace of the trump suit or aces of other suits; and whether they have a low number of cards of any suit that isn't the trump suit which would allow them to use trump cards earlier to win tricks when given suit is exhausted.

Testing

Testing was completed after each big refactor in order to make sure that our changes preserved the original Oh Heaven game behaviour. The framework was thoroughly tested and run on different seeds, with the thinkingTime of each player reduced to 1 in order to allow for the most efficient testing. We ran the code multiple times in order to verify the behaviours of our smartNPC were being performed as intended.

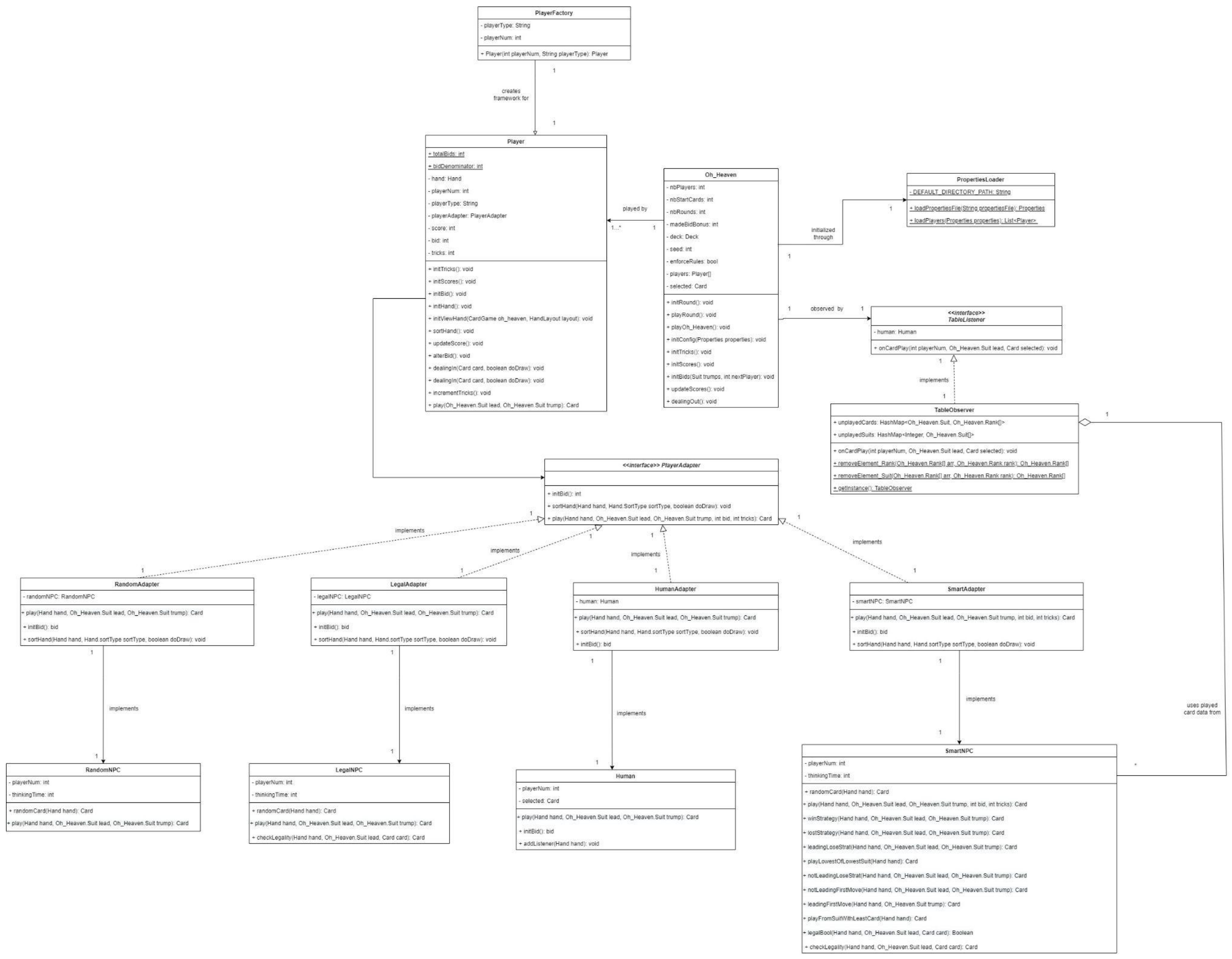
PlayerFactory
- playerType: String
- playerNum: int
+ Player(int playerNum, String playerType): Player
creates framework for
Player
+ totalBids: int
+ bidDenominator: int
- hand: Hand
- playerNum: int
- playerType: String
- playerAdapter: PlayerAdapter
- score: int
- bid: int
- tricks: int
+ initTricks(): void
+ initScores(): void
+ initBid(): void
+ initHand(): void
+ initViewHand(CardGame oh_heaven, HandLayout layout): void
+ sortHand(): void
+ updateScore(): void
+ alterBid(): void
+ dealingIn(Card card, boolean doDraw): void
+ dealingIn(Card card, boolean doDraw): void
+ incrementTricks(): void
+ play(Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
played by
1...*
Oh_Heaven
- nbPlayers: int
- nbStartCards: int
- nbRounds: int
- madeBidBonus: int
- deck: Deck
- seed: int
- enforceRules: bool
- players: Player[]
- selected: Card
+ initRound(): void
+ playRound(): void
+ playOh_Heaven(): void
+ initConfig(Properties properties): void
+ initTricks(): void
+ initScores(): void
+ initBids(Suit trumps, int nextPlayer): void
+ updateScores(): void
+ dealingOut(): void
initialized through
PropertiesLoader
- DEFAULT_DIRECTORY_PATH: String
+ loadPropertiesFile(String propertiesFile): Properties
+ loadPlayers(Properties properties): List<Player>
observed by
<<interface>> TableListener
- human: Human
+ onCardPlay(int playerNum, Oh_Heaven.Suit lead, Card selected): void
implements
TableObserver
+ unplayedCards: HashMap<Oh_Heaven.Suit, Oh_Heaven.Rank[]>
+ unplayedSuits: HashMap<Integer, Oh_Heaven.Suit[]>
+ onCardPlay(int playerNum, Oh_Heaven.Suit lead, Card selected): void
+ removeElement_Rank(Oh_Heaven.Rank[] arr, Oh_Heaven.Rank rank): Oh_Heaven.Rank[]
+ removeElement_Suit(Oh_Heaven.Rank[] arr, Oh_Heaven.Rank rank): Oh_Heaven.Rank[]
+ getInstance(): TableObserver
<<interface>> PlayerAdapter
+ initBid(): int
+ sortHand(Hand hand, Hand.SortType sortType, boolean doDraw): void
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump, int bid, int tricks): Card
implements
RandomAdapter
- randomNPC: RandomNPC
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ initBid(): bid
+ sortHand(Hand hand, Hand.sortType sortType, boolean doDraw): void
LegalAdapter
- legalNPC: LegalNPC
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ initBid(): bid
+ sortHand(Hand hand, Hand.sortType sortType, boolean doDraw): void
HumanAdapter
- human: Human
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ sortHand(Hand hand, Hand.sortType sortType, boolean doDraw): void
+ initBid(): bid
SmartAdapter
- smartNPC: SmartNPC
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump, int bid, int tricks): Card
+ initBid(): bid
+ sortHand(Hand hand, Hand.sortType sortType, boolean doDraw): void
uses played card data from
RandomNPC
- playerNum: int
- thinkingTime: int
+ randomCard(Hand hand): Card
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
LegalNPC
- playerNum: int
- thinkingTime: int
+ randomCard(Hand hand): Card
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ checkLegality(Hand hand, Oh_Heaven.Suit lead, Card card): Card
Human
- playerNum: int
- selected: Card
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ initBid(): bid
+ addListener(Hand hand): void
SmartNPC
- playerNum: int
- thinkingTime: int
+ randomCard(Hand hand): Card
+ play(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump, int bid, int tricks): Card
+ winStrategy(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ lostStrategy(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ leadingLoseStrat(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ playLowestOfLowestSuit(Hand hand): Card
+ notLeadingLoseStrat(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ notLeadingFirstMove(Hand hand, Oh_Heaven.Suit lead, Oh_Heaven.Suit trump): Card
+ leadingFirstMove(Hand hand, Oh_Heaven.Suit trump): Card
+ playFromSuitWithLeastCard(Hand hand): Card
+ legalBool(Hand hand, Oh_Heaven.Suit lead, Card card): Boolean
+ checkLegality(Hand hand, Oh_Heaven.Suit lead, Card card): Card

Figure 1: UML Design Diagram

(For clearing viewing see UML Design Diagram JPEG within the submission folder)

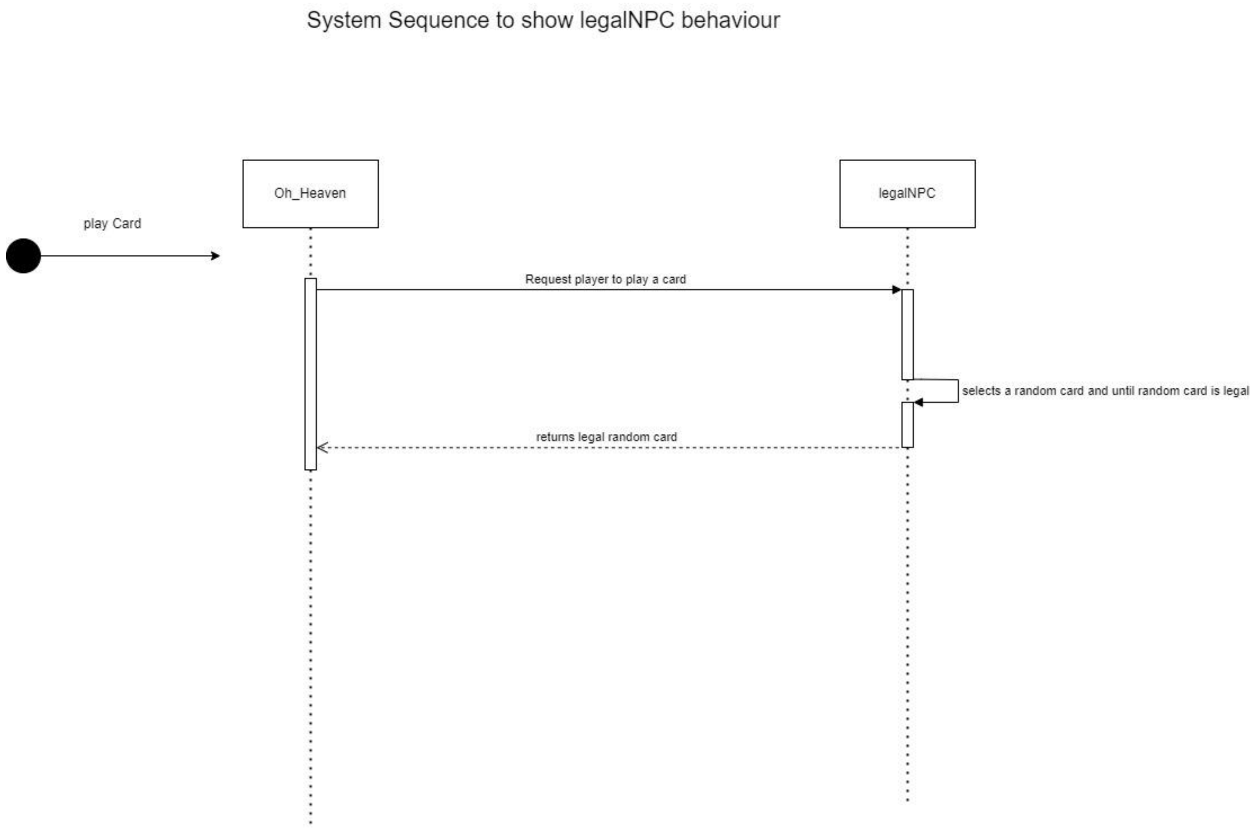
System Sequence to show legalNPC behaviour
Oh_Heaven
legalNPC
play Card
Request player to play a card
selects a random card and until random card is legal
returns legal random card

Figure 2: UML Sequence Diagram

(For clearing viewing see UML Sequence Diagram JPEG within the submission folder)