SWEN30006 - Software Modelling and Design Project 2

The card game Oh Heaven was presented in project 2 for the subject. Our team was tasked to modify and enhance existing code to satisfy the requirements and enhance code quality. Several requirements were presented in the specification of the project. This includes: support for loading parameters through a property file, and addition of different types of NPC - including a player that plays random legal moves and another that plays legal moves in a more intelligent manner. Certain design changes have been implemented in order to facilitate the requirements, and to enhance the code structure to accommodate future extensions. GRASP principles and GoF principles have been applied in making such changes, as demonstrated in the paragraphs below.

Overview

The following changes have been implemented in the Java code of Oh Heaven to achieve the requirements presented. Namely, the addition of a PropertyLoader class, Player class (and subsequent subclasses) and a BotStrategy interface (and subsequent subclasses) were introduced to support immediate addition to code functionality. Other changes were introduced as well, to improve the modularity, clairity, and extensibility of the code.

Encapsulation of Oh_heaven

The Oh_heaven class is the entry point of the whole program. Before modifications, it was noticeable that Oh_heaven included a large amount of code to handle runtime functions, such as the Oh_heaven constructor. This greatly reduced the readability of the codes and cohesion therefore. To counter this issue, it was necessary to apply GRASP principle of high cohesion and indirection. Runtime functionalities of Oh_heaven class were migrated so that a unified interface is presented in the main function - a constructor call for GameManager. By removing large amounts of code from the entry point of the program and directing them elsewhere, the GRASP principle of indirection is achieved, that Oh_heaven class delegates its responsibilities to other classes, therefore making the code much more readable. This also guarantees that the GRASP principle of high cohesion is satisfied, that Oh_heaven is dedicated to the purpose of being an entry point for the program, instead of a coupled entity that takes up various tasks.

Introduction of Property Loader

Loading parameters from a property file was specified as part of immediately deliverable requirements of the project. To satisfy this requirement, a new class of PropertyLoader was created. The PropertyLoader class loads a selected property file from disk, and allows method calls of the PropertyLoader class to get different attributes within the class. PropertyLoader is accessed via static methods, and is not instantiated during the entire duration of program execution. Property Loader also keeps a static attribute of java.unil.properties, which is accessible via methods only.

The main design principles used in creating PropertyLoader class are GRASP principles of Pure Fabrication and GoF principle of Adaptor - more specifically, Object Adaptor. Prior to creation of the class in requirement analysis, it was identified that the existing code base of Oh_heaven does not consider I/O access of a property file, and would need additional components to do so. It was also highlighted that java.util.properties supports reading and parsing of property files into Properties class, which is not directly readable by existing code in Oh_heaven. An adaptor of these two functions - PropertyLoader class, is hence fabricated to allow access of java.util.Properties class. By introducing an adapter, it was possible to maximise reuse of existing code in java.util, whilst not introducing coupling between Properties class and Oh_heaven gameplay. The game functions (generally in GameManager class) can access property files via method call, without explicit coupling of the two classes. This also allows the code to be open to extension, so that future developers can write another PropertyLoader with the same methods to support different property files.

Creation of GameManager

As shown above in Encapsulation of Oh_heaven, functionalities of game control have been migrated from Oh_heaven to other classes. These functions are mostly rebased to a new class called GameManager. GameManager is a newly-fabricated class that controls the overall game mechanics and data structure, in which most gameplay logic is implemented. The creation of GameManager is directed by GRASP Principle of Indirection and Pure Fabrication. By increasing the level of abstraction within the Oh Heaven game, the addition of GameManager allows lower coupling between classes and higher cohesion. The game mechanics are also protected from modifications of PropertyLoader and/or Oh_heaven class - as directed by the principle of protected variation.

Modularisation of GameManager

GameManager, upon its first creation, included over 200 lines of code of various functionalities. Upon closer examination, it was revealed that this is in direct violation of various GRASP principles, including but not limited to: high cohesion, low coupling, protected variation, and indirection. The GameManager class, given these violations, was then separated into different methods and classes. Apart from the separation of procedural logic into different functions (to achieve higher cohesion), the introduction of classes of GraphicsManager, RandomManager and Player in particular, is worth noting given relevant applications of various design principles. Introduction and modification of these classes will be discussed in more detail below.

Separation of Graphics and The Creation of GraphicsManager

Prior to modifying the original source code, the graphics of the Oh Heaven game were handled by several methods scattered throughout the original Oh_Heaven class. In its previous state, the code achieved low cohesion, readability and reusability and as such, we have separated the graphics logic and the in-game mechanisms by creating a new class through pure fabrication. The new GraphicsManager class encapsulates only the logic involved in displaying the necessary user-interface when the game is run, and thus presents a solution that provides low coupling and high cohesion. It accomplishes this by possessing a highly cohesive set of methods only relating to graphics that can be easily reused and extensibile for future design. The low coupling is achieved by our GraphicsManager class having a singular bidirectional relationship with the GameManager class (as per the design class diagram). By creating a new class through pure fabrication, the design pattern of Information Expert has been inadvertently applied as our GraphicsManager class now has the relevant information to fulfil all the intended responsibilities that we initially desired.

BotStrategy Interface and Subclasses

To address the task of implementing a legal and smart player, we have decided to define each of the NPC types in a separate class of its own and have additionally created a common BotStrategy interface as the point of contact for these newly created and varied bot types. In doing so, we have followed the GoF strategy pattern where the smart, legal and random strategy classes implement the BotStrategy interface in

which the Bot class uses. As a result, a specific bot strategy can be passed as a parameter when we create a new bot, and the logic of adding a certain card to a trick is delegated to that specific bot strategy class. A high level of abstraction is provided through BotStrategy having overridden methods in each of its subclasses with varied logic inside each of them. We have implemented our solution this way so that our code stays open for extension with ease, via inheritance that would just extend our existing BotStrategy class. Furthermore, our code is easily modifiable as the use of method overriding allows changes to be easily implemented. The separation of specific lead and follow methods enables cohesion to be maintained at a high level as unique functionalities are separated.

The Creation of RandomHandler

Upon identifying the many instances of obtaining random numbers and objects in the original Oh_Heaven class, it was apparent that there was a highly inefficient level of coupling between the Oh_Heaven class and the Random.random class. To combat this issue, we have decided to delegate this responsibility to a new class of its own. Through pure fabrication, we have created a singleton RandomHandler class that oversees the responsibility of returning a random trump suit, player number, card and seed. This achieves an appropriate level of abstraction by allowing our GameManager to call specific methods using the singular RandomHandler instance. By utilising pure fabrication and the separation of responsibilities, we have not only kept our code understandable and manageable through high cohesion, but we have also kept coupling to a minimum. There now remains no reference to the Random.random class in GameManager as our randomHandler sorts out all the logic relating to that. After our modifications, our RandomHandler class is open for easy extensibility for any future methods requiring randomness.

Creation of the Player class and subclasses - Player Factory, BotStrategy interface, Inheritance/ Polymorphism

The project specification dictates that players may not have access to each other's information. In order to meet this requirement as well as improve code readability, extensibility, and modularity, a new abstract class of Player was created. The creation of Player class serves as an self-evident representation of its real-world counterpart. Such creation therefore contributes greatly to lowering the representational gap between real-world entities and software entities.

Player is created as an abstract class. This allows the extension of player functionalities via inheritance, while the base abstract class is closed to modification, demonstrating the GRASP principle of protected variation. In the submitted version of project code, the Player class has two derived classes - Human and Bots. Humans are Player entities that require UI interaction, which ideally comes from a real world user. Bots are Player entities that implement a BotStrategy interface, which then guides the Bot to make a move, as explained in the BotStrategy section.

The fabricated Player class now encapsulates the player's hand, player number, score, tricks won and bid. Originally these attributes were delegated to the Oh_Heaven class to track for all players, making it highly coupled with other entities. By delegating those existing responsibilities to the player class via the design principle of information expert, our code is now highly cohesive in that our GameManager class does not keep records of each player's personal attributes, rather just updates an array of Players using methods in the Player class.

Players are generated by a fabricated class of PlayerFactory. The PlayerFactory class is created following the GoF principles of Factory pattern, as well as Strategy and Composite for bot generation. Parameter of type String is read by PlayerFactory, which would create and return a Player of required type. A bot type player will be initialised with a newly created BotStrategy Instance, which is then coupled to the bot. Whilst slightly sacrificing low coupling principle, such design for Player creation, especially Bot creation, yields substantial cohesion boost and reduces code simplicity. This is also in line with the GRASP principle of polymorphism, that different types of players can be created via the same method call.

Introduction of Trick Class

In the original implementation of Oh Heaven, round playing is coded within the playRound() function entirely, for close to 100 lines. After analysing objects that are involved in the game, it was decided as a design decision that a Trick class should be separated from the playRound method, in order to improve code readability as well as modularity.

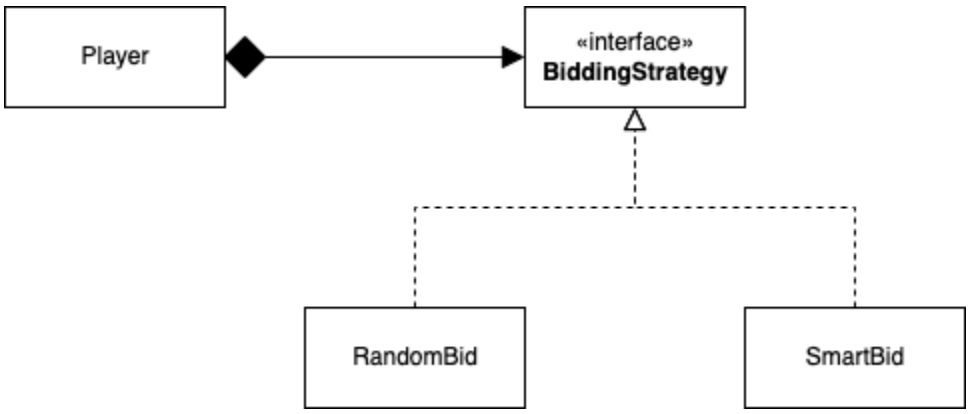
Tricks were originally represented by a Hand() class variable within what is now GameManager. Upon the decision to introduce a new class, a new Trick class was created. Following the GRASP principle of indirection and information expert, the new class is created and maintained by GameManager. Compared to the previous implementation, the novel Trick class also provides a unified interface for GameManager to handle gameplaying during each player turn. The GRASP principle of high cohesion is also now

satisfied, that all trick-related functions are packed in the same dedicated class. The implementation of Trick class also lowers the representational gap between the design model and real world entities, allowing stakeholders to better understand the application design where needed.

Extension to Bidding Strategies

In our current implementation of the game, there is a makeBids method in GameManager that loops through all players and calls the method makeBid for each Player which makes a random bid for each Player just like the original specification. The makeBids method then modifies the first player's bid if all the current bids add up to the number of starting cards just like the original implementation of the game.

The way we would extend our design to allow for bidding strategies would be to delegate the responsibility of bidding to a new interface BiddingStrategy which then has various implementations for specific strategies, which therefore makes this design an example of the Strategy Pattern. The basic design of our bidding strategy is shown below.



This allows any bidding strategy to be contained within its own class, increasing the overall cohesion of our design. In addition, the strategy interface provides a unified interface for the Player class which allows us to easily add new bidding strategies without modifying any of the existing code. We would add the bidding strategy to the properties file and have a particular bidding strategy as a variable to be passed into the Player constructor, meaning only the properties file needs to be modified for any new strategies and no existing code.