Project 2 Report: Trong-Hieu Nguyen, Rajat Soni, Jordan Bell

Limitations of Current Design

In the original design, it is clear that the main game class, *LuckyThirteen* (or LT for short), controls the entire logic of the game: rendering GUI actors, handling players' behaviour at each turn, and scoring each player at the end of the game. This creates some problems for future extension, which we shall elaborate with regards to GRASP patterns.

1. Players' attributes stored separately

In the original LT design, player's attributes (scores, hands, auto movement) are stored in separate arrays. This is a violation of several GRASP patterns:

- **High Cohesion:** The main game class now becomes responsible for managing player attributes, which is not related to the game logic itself but rather to the state and behaviour of individual players. This results in significantly reduced cohesion.
- **Low Coupling:** Storing player attributes in the main game class increases coupling since any change in player-related functionality (drawing a card, discarding a card, assigning scores, etc) will require changes in the main LT class. This leads to tightly-coupled code that is very difficult to maintain and extend in the future.
- **Information Expert:** The information required to manage a player's hand, calculate their score, identify which card to discard, etc. is intrinsic to the player. The LT class is not the expert on these player-specific details, and therefore it should not handle these attributes directly.

-> Solution: Delegate handling player logic and attributes to a separate Player class for better encapsulation.

2. Overloading of Responsibilities in main LT class

In the initial design, the LT main class controls the entire logic of the game. This monolithic approach is a violation of several GRASP patterns:

- **Information Expert:** Main LT class is not the expert on handling scoring logic, controlling players' behaviour at each turn, rendering GUI actors, yet it has those responsibilities in current design.
- **Creator:** The current design has no mechanism for creating instances of Player, instead choosing to store their attributes separately.
- **High Cohesion:** LT's responsibilities span multiple domains, including player management, GUI rendering, game logic, and scoring. This lack of focus makes the class harder to understand and maintain.

Our new proposed design would involve delegating these responsibilities to separate classes: RenderManager for rendering GUI actors, ScoreStrategy for handling scoring and determining winners, Player for handling player logic, SumRule for handling summation to 13 logic. This would make our code easier to maintain, extend without interfering with other functionalities. In addition, identifying and isolating bugs would be much easier.

3. Lack of Adaptability

- The current LT design only has support for 2 different types of players: Random and Human players, with Player 0 by default human player. This presents a problem as player logic is handled exclusively in the playGame() function with if-statements to check whether the current player is human or not. If more player types are introduced, it would mean more if-statements in the main playGame() function. This leads to High Coupling between game logic and specific player types, making the codebase fragile and hard to extend. Also, every time a new player type or scoring strategy is added, the main playGame() method would have to be modified, violating the Open/Closed principle since the function's not closed to modification.
- Solution: Use polymorphism to handle variations in behaviour through abstractions/interfaces.

<u>Proposed Changes to LT Design</u>

1. **Support for Different Types of Players (Player superclass)**

- As mentioned previously, the new design would see a separate base Player superclass responsible for handling player logic at each turn, and 4 subclasses: Human, Random, Basic and Clever. Each turn, a player would draw a card and choose a card to discard. Since choosing which card to discard depends on player type, we utilised the Template pattern for this: the drawACard() method would be concrete implementation in the superclass, while the discard() method would be implemented concrete at a subclass level. This adheres to the Information Expert and High Cohesion principles, since now the Player class would handle player logic and behaviour instead of the main LT class, as well as ensuring that the Player class is focused on handling player logic only. Using the Template pattern allows us great flexibility, since a new type of player can just extend from the base Player and implements its own discard() method.
- Another change we made is that players now store all combinations that can sum up to 13 in a private List<List<Card>> object inside the Player superclass. This makes it easier to apply a scoring strategy, since we just have to check whether a card is private or public, instead of re-finding all combinations that can sum up to 13 for a given player and aggregating the max score of each combination.
- In addition, the responsibility of creating players was delegated to a separate singleton factory PlayerFactory. This also helps achieve Low Coupling as well, since the main LT class does not have to know the details of how to create different types of players, only PlayerFactory.createPlayer(). This method would be called in the initGame() method. This results in a more modular and decoupled system, making it easier to manage and extend (i.e. more player types).
- One other requirement was to implement how the clever computer player chooses which card to discard from their hand. We decided that out of the 3 cards in their hand, the card with the least probability of summing up to 13 with the non-visible cards would be discarded. At every turn, the clever computer player aggregates the frequency of each non-visible cards' possible sum values, then for each card in their hand, assigns a score based on how many ways it can form a sum of 13 with the remaining cards yet to be played. The card with the lowest score would be discarded.

2. Sum Rules Changes

- The new design accepts a third way to sum to thirteen, the two cards in hand and the two public cards. In order to improve modularity from the original design the code has been refactored and the code for the sum rules has been divided over multiple classes and interfaces. The SumRule interface has been added to define a common interface for the sum rules with a common method definition canSumTo13(Player) to check that the sum equals thirteen. This forces all sum rule implementations to adhere to the standard method for checking the sum. Next the BaseSumRule, an abstract class that provides the core functionality and constants used by the rule classes (SumRule1, SumRule2, SumRule3). Each sum rule implements the SumRule interface allowing them to be used interchangeably and allowing for more flexible modifications and changes thereby adhering to polymorphism.
- Each concrete SumRule implementation is not used in the main LT class, but instead used in the SumRuleComposite object. Our new LT class would hold a SumRuleComposite instance, and invoke sumRuleComposite.canSumTo13() at the end of round 4 to find the number of players whose hands can sum to 13.
- The SumRuleComposite class utilises the composite pattern and implements the common method canSumTo13() of the base SumRule interface to manage the sum rules while keeping the code flexible and easy to extend. We chose to use the Composite pattern here because it can aggregate results from a collection of classes implementing the SumRule interface. In this case, for each player, we call sumRuleComposite.canSumTo13() in the main LT class to aggregate results from each sumRule.canSumTo13(), if any of the results are true, then that player has cards that can sum to 13.
- The sumRuleComposite acts as both a creator, by creating and managing the sum rules and as the controller by delegating the sum checks among the rule classes. The sumRuleComposite also adheres to pure fabrication as it does not represent a problem domain but instead is made to achieve better modularity and minimise dependencies therefore improving low coupling. The design also adheres to the information expert principle by having each class be experts at what they designed for like each sum rule is an expert at its respective rule which promotes high cohesion making the design easier to understand and change.
- New sum rules can be added simply by creating a new class that implements the sum rule interface and adding it to SumRuleComposite. Due to the separation of rules into classes, changing the existing rules is simple. The changes to the sum rule implementation in the LT game create a modular and moldable code base. By utilising interfaces, abstract classes, composite pattern and GRASP principles the design is easily extendable and simpler to maintain and debug.

3. Scoring System

- We have added a common interface: ScoringStrategy, and a Singleton Factory object ScoringStrategyFactory responsible for choosing the appropriate scoring strategy at runtime. Our current ScoringStrategies all implement the base ScoringStrategy interface, allowing our Factory object to select the appropriate strategy at runtime. This use of polymorphism achieves high flexibility and ease of extension. In the future, all new Scoring Strategies can implement its own applyScore(Player[]) method from the base interface, without any modifications in the main LT class itself. The factory method can be extended to include these new strategies, adhering to the Open/Closed principle.

- This also decouples the main game logic from the specifics of scoring, as now main LT class only has to call scoringStrategy.applyScore(players) and scoringStrategy.findWinners() without knowing the implementation of all strategies, adhering to Low Coupling.

4. Use of Singletons

- In our new design, we have also decided to make DiscardPile (cardsPlayed) and PlayingArea (publicCards) singleton objects, since we want to have access to them at different stages of the game without passing them as parameters to our functions. Instances where access to these singletons is required include: clever computer player needs access to both singletons to decide which card to discard, or add a card to discard pile at the end of each player's turn, or check PlayingArea for public cards for summing to 13, etc.
- Using Singletons like this allows us to minimise dependencies between classes, thereby achieving Low Coupling. In addition, each singleton class encapsulates its own state and behaviour by providing public methods relevant to managing the state of its instance, keeping related functionality within the same class, thereby also achieving High Cohesion.