SWEN30006 PROJECT 2 REPORT

Workshop [Thu13:00] Team 02

Jess Lee (1260948)

Bella Kwon (1346599)

Samuel Imanuel Gunawan (1371743)

Introduction

Lucky Thirteen (LT) is an upcoming card game by JQK Company. Our team was tasked with completing the development of the game, starting from the original LT game skeleton code, which was rushed and lacked proper design. Significant refactoring and improvements to both its design and functionality were implemented.

This report presents an in-depth analysis of the problems, considerations, and improvements implemented in the LT game.

The scope of our project includes the following:
Code Refactoring: Improving the code structure by applying GoF design patterns and GRASP principles, enhancing readability, and ensuring maintainability.
New Features: Implementing additional summing options for card values, introducing new player types with distinct behaviours, and supporting game configuration through property files.
Design Documentation: Creating domain class diagrams, static design models, and other necessary software models to document the revised design.

Part 1. Refactoring The Codebase

Skeleton Code Issues

The skeleton code given was horrible. It is the complete opposite of all GRASP patterns; high coupling and low cohesion. Everything was on a single God class “luckyThirdteen”, meaning it has all responsibilities for creation, game flow logic, scoring logic, UI visualisation. While this means that all variables needed by each method are always available, it is very hard to read and maintain for future development, a single fundamental change can lead to remaking lots of other parts, very fragile.
The skeleton code provided was poorly designed, it implemented the complete opposite of all GRASP patterns; high coupling and low cohesion. All responsibilities were concentrated in a single God class, luckyThirdteen, which handled object creation, game flow logic, scoring logic, and UI visualisation. This approach made the code difficult to read, maintain, and extend, as any fundamental change cascaded throughout the code and thus demands extensive modifications.

Refactor

Our first approach in fixing this is to first decouple the God class into separate packages and

Given that JGameGrid provided standard classes for a card game, we applied polymorphism to leverage these existing classes. This approach allowed us to avoid designing classes from scratch, instead focusing on adding extra functionality and methods through inheritance and method overriding. We also improved abstraction by introducing getters and setters, enhancing flexibility for future development, such as supporting different game modes or new rules.

Implementing GoF design patterns

To encourage each class having as specific relevant responsibility as possible while avoiding redundancy, GoF was used. One class would be responsible solely for creation of objects, another allows runtime algorithm change. Methods were also modified to encapsulate major processes, making it easy for future developers to build on top of it.

Detailed descriptions of each design pattern used, along with justifications, are provided below.

Part 2. Design Pattern Implementation

Facade

Each method in a class can handle specific responsibilities, and to run a specific process, it needs multiple methods that are run sequentially. This process details could be encapsulated and abstracted in one method call such that the developer only needs to know one method. This pattern was particularly used in running our game. Initially playerThirdteen's '**`runApp`**' method calls different methods to initialise each function group, another '**`playGame`**' to simulate the game, and explicit score counting which is prone to break after modifications if the methods are not called in the intended order. We then modified '**`playGame`**' to encapsulate the entire Game operation process, including the initialization of function groups (object, scoring, logic). This simplified interface allows sub-system and processes to be atomic and consistent while improving ease of development.

In the context of GRASP principles, this approach enhances high cohesion by grouping related functionalities and operating these within the Facade instead of related sub-systems. Similarly, low coupling is promoted as only interaction with the Facade is required, reducing dependencies between client code and individual components of the game flow. Additionally, the Facade aligns with the Controller principle, as it holds sole responsibility to coordinate the interactions between the client and program model.

Factory Method

The Factory Method was implemented to encapsulate the logic of creating different player types and to abstract away the implementation details from the client code, effectively centralising responsibility of similar object creation. This abstraction is achieved through a common interface ('**`createPlayer`**') that hides the specific instantiation details of the player from the client codebase. As a result, the client code is decoupled from the concrete player implementations ('**`randomPlayer`**', '**`basicPlayer`**', '**`cleverPlayer`**', '**`humanPlayer`**') and only interacts with the '**`PlayerFactory`**' class to create player object, promoting low coupling and a modular design. This centralisation not only makes it easier to manage changes to the player creation processes, but also upholds the Creator principle, as the code responsible for creating player objects is separate from the code that uses these objects.

For future enhancements, this design choice allows for easy addition of new player types by simply extending the '**`Player`**' class, promoting scalability and adaptability to the program. Additionally, the Factory Method is also utilised in '**`ScoreStrategyFactory`**' to create different types of scoring rules, further emphasising the flexibility and extensibility of the codebase.

Strategy & Composite

The coupled implementation of Strategy and Composite design patterns constitutes a flexible and modular scoring system. It allows the program to independently handle various scoring rules, while also having a unified interface to utilise them collectively.

The Strategy pattern is applied through the **`'ScoreStrategy'`** interface and its concrete implementations (**`'OptionOneScoreStrategy'`**, **`'OptionTwoScoreStrategy'`**, **`'OptionThreeScoreStrategy'`**), enabling different scoring options to be used interchangeably. This encapsulates the score calculation behaviour and enables convenient addition of new options without modifications to the existing code in the future, adhering to the open-close principle.

The **`'CompositeScoreStrategy'`** and **`'CompositeMaxStrategy'`** classes comprise the Composite pattern, operating as the common interface for combining the different scoring options and choosing the most adequate one. For the current program, it aggregates the three scoring options to determine the maximum score attainable for a player. In the future, multiple logics for deciding on the score can be added easily.

Such modified program design adheres to Low coupling and High Cohesion GRASP principles, as alterations to individual scoring options do not impact the overall structure and as each class serves a single, definite responsibility. Moreover, it promotes code reuse and simpler addition of scoring rules, facilitating easier maintenance and extension of the scoring system.

Singleton

The Singleton design pattern was implemented for the below classes to enhance efficiency and consistency across the program. This design choice ensured a more maintainable and scalable design, while adhering to the GRASP principles:

- **`GameDeck`**
  The use of Singleton prevents inconsistencies and manages resources efficiently by ensuring there is only one deck instance throughout the game. It adheres to Information Expert by centralising the responsibility of managing the deck within a single instance, and promotes low coupling by having a single point of access to the deck.
- **`LuckyThirdteen`**
  Ensures there is only one instance of the game, avoiding conflicts by managing the game state and flow in a consistent manner. This implementation groups the game state management and functionalities within a single instance, promoting high cohesion.
- **`GameController & ScoreActors`**

Having one instance of the GameController and ScoreActors avoids conflicting states and behaviours due to having multiple instances. The game control logic and score calculation logic encapsulated within a single instance also contributes to having consistent game behaviour and state transitions. This behaviour adheres to the Information Expert principle, as the knowledge to manage the game / scoring is centralised in individual classes.

- **`PlayerFactory & ScoreStrategyFactory`**
  Ensures there is a single factory instance for both the creation of players and score strategy objects. This is a crucial element in not only maintaining consistency in strategy instantiation, but also in simplifying the creation process to manage and update the logic as required. This encapsulation of object creation adheres to the Creator principle while decoupling the client code from concrete player and score strategy implementations.

Part 3. Clever Computer Player Implementation

The Clever player implements a more sophisticated strategy than the Random and Basic players in the game by considering both the previously discarded cards and the current cards in hand to maximise its score.

1. Tracking discarded cards
    - The Clever player maintains a list of all discarded cards with the `'discardedCards'` list.
    - *Justification: this allows the player to make informed decisions by considering which cards have already been played and are no longer available, or less likely to appear in the player's hand.*
2. Immediate Thirteen check
    - Before discarding a card, the player checks if it is possible to create thirteen with any of the 3 summing options.
    - If it is possible, then the player discards one of the remaining cards in the hand, preserving the cards that sum to thirteen for future use.
    - If summing to thirteen is not possible with the current cards, the player employs a strategic logic to identify the best card to discard (see below).
    - *Justification: the immediate check ensures the player does not miss opportunities to achieve a win condition immediately.*
3. Strategic discarding
    - The player discards a card that will least likely sum to thirteen in the future.
    - This is done by summing all possible combinations of cards in hand with discarded cards, and seeing how often each card sums to thirteen with a discarded card.
    - The private card that contributes most frequently to potential sums of thirteen is discarded.
    - *Justification: discarded cards that lead to a sum of thirteen the most with current private cards indicates that the chances of appearing in the game again are very low.*