SWEN30006-Project-2 Report

Written by *Workshop16Team02*

The report is going to introduce the additional features of the Cribbage card game and describe the changes our team have made to the system. Furthermore, the rationale for these changes would be discussed by using design patterns and principles and other reasonable options for the design would also be analysed. The following report would introduce in two main sections which are scoring and logging components. These new inserted classes are going to assist the program to meet the client's expectation.

Scoring

The way we implement the scoring feature is by applying strategy pattern together with composite pattern. After analysing the problem domain, we found out that there are two phases that requires scoring feature, one is the "Play" phase, the other is the "Show" phase. The scoring rules in these two phases seem to be similar but they are not exactly the same. One key difference is that the order of card would influence the results in the Play phase, yet the Show phase only concerns about the combination. Therefore, although some scoring rules exist in both phases, their implementing logic are quite different. To achieve numbers of different scoring logic, we suppose the strategy pattern is our best choice.

Strategy pattern is a very useful behavioural pattern when there are a variety of different algorithms to implement a particular feature of the system. In our case, the Cribbage scoring feature has many scoring strategies, such as Run, Pair and so on. And each of them needs to be implemented independently, since their logic are all unique. Therefore, the strategy pattern can perfectly map out the relationship between the scoring function and the concrete strategies which can also maintain protected variation while the game rules have to be changed. To implement this pattern, the abstract class called ScoringStrategy have been created, it has necessary attributes and methods that are required for the scoring process. Then, by applying polymorphism, the concrete scoring strategy classes, for instance *PlayRunStrategy* and *ShowPairStrategy*, are constructed.

After creating all strategy classes, another problem comes up. It is that multiple strategies are always required together during the game runtime. For example, when calculating the score of the Show phase, not only *ShowPairsStategy*, but a combination of all show scoring strategies is required.

This leads to the demand of composite pattern. Composite pattern provides a structure which enables the existence of multiple strategies in one scoring strategy object. By making an abstract class called *CompositeStrategy* which extends *ScoringStrategy*, we can then have a *CompositePlayStrategy* and *CompositeShowStrategy* which hold the scoring logic of Play phase and Show phase respectively. Also, to control which set of scoring strategy is going to be used, a *ScoringStrategyFactory* is introduced. It is a singleton class and takes charge of extracting required strategy during the game.

Beside the strategy pattern, using decorator pattern for scoring feature have been considered. Decorator pattern is always used to insert additional functionality. It does not change the base of the design but wrap it up as an updated design with new functions. In this case, decorator may be applied to check the requirement of each scoring types and record them. However, it would not have a composite method which can add or delete the scoring types freely. Generally, the decorator would only change the "skin" based on the original design. Yet, there is no basic class for the scoring system, hence collecting various strategy is a more rational solution in this case. Therefore, the strategy is a more reasonable way to implement this design.

In total, a scoring factory, an abstract scoring strategy class together with its several concrete child classes, and a set of composite strategy classes, are combined together to achieve the scoring feature (see more details in figure 1).

Logging

To construct a competent logging feature, two classes *Logger* and *LoggerHelper* are created for logging the required messages in a single round. Generally, *Logger* is only responsible for writing the log message into the log file *Cribbage.log*, and *LoggerHelper* is designed for setting the format of each log message which could be replaced by other formats easily.

Due to that *LoggerHelper* and *Logger* have their own responsibilities, the current design in the program obeys the principle of pure fabrication. *LoggerHelper* would be assigned the duty of deciding the formatting of the log message, and *Logger* would force the program to write the log messages into *cribbage.log*. Since the two implementations are asked for executing a specific and individual task, the program would maintain high cohesion.

Specifically, *LoggerHelper* as an intermediate class between ScoringStrategy and Logger could avoid directly coupled between them. Based on the one of the GRASP - indirection, it assists the design to maintain lower coupling. Additionally, if the program is going to have a new format for logging message or edit the current format, it could be easily done by modifying the *LoggerHelper*. On the other hand, if the client asks to log the messages into different files, i.e., ".txt" and ".csv", the demand would be satisfied by modifying *Logger*. To some extent, the collaboration between *LoggerHelper* and *Logger* would achieve the protection of variation.

Since the program would not need more than one *Logger* or *LoggerHelper* to log the texts, both *LoggerHelper* and *Logger* are followed the singleton pattern in the design. The program is expected to contain only one global static entry point of accessing *LoggerHelper* by calling *getInstance ()*. It would be more efficient while *Cribbage* or *ScoringStrategy* attempts to use it for logging message. The other important reason of making *Logger* be a singleton class is to make sure all the log messages would be written in a same log file. Otherwise, if *Logger* is designed to be a non-singleton class, the situation of creating multiple log files or repeatedly deleting and refreshing the log file may happen. Certainly, this severe error is unaccepted.

Beside the current structure, other implementations of logging system also have been considered. For instance, the log method was considered to be contained by each *ScoringStrategy* child class, hence the child class would decide how to log the massage individually. However, the issue is that *ScoringStrategy* would have low cohesion due to the multiple assigned tasks (*getScore ()* & *log ()*). Therefore, our team still decide to use the *Logger* and *LoggerHelper* design.

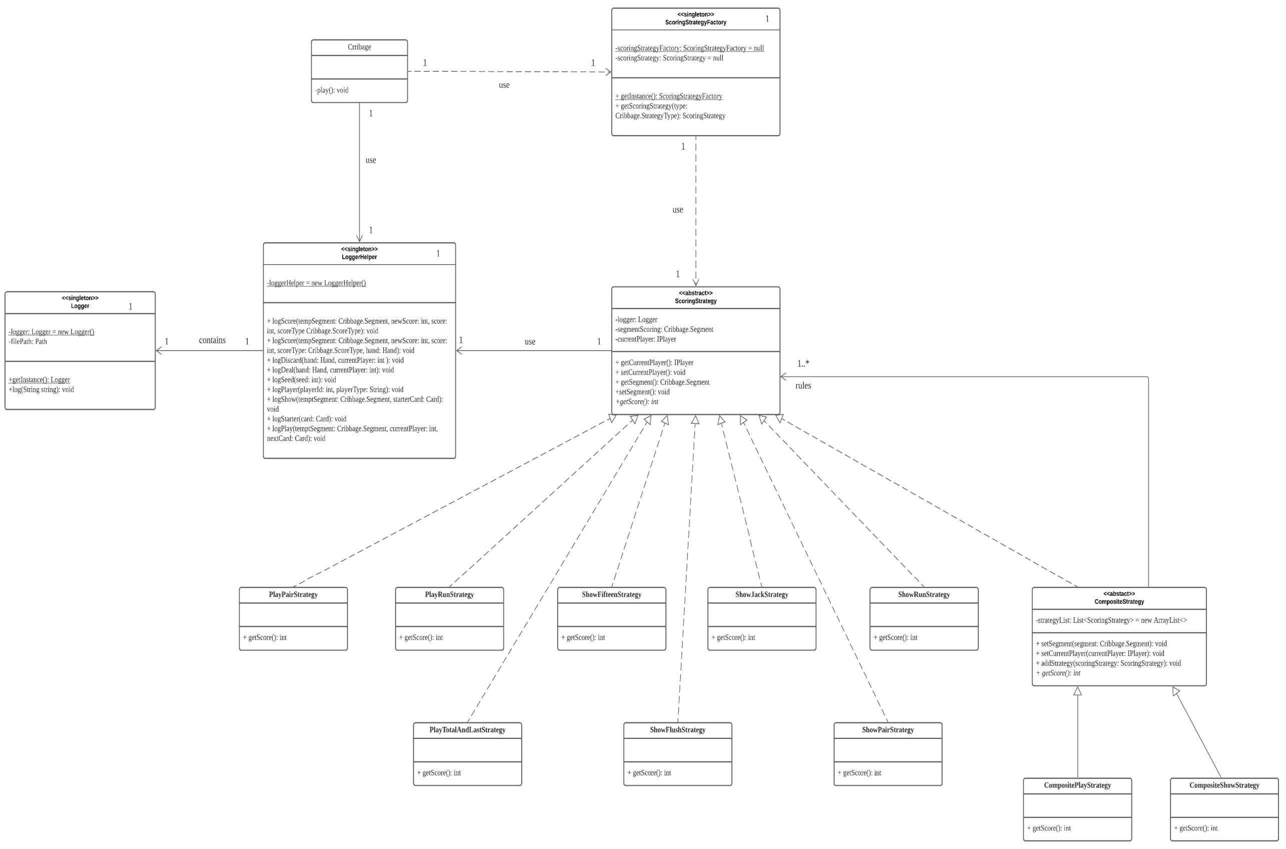
Crribage
-play(): void
<<singleton>>
ScoringStrategyFactory
-scoringStrategyFactory: ScoringStrategyFactory = null
-scoringStrategy: ScoringStrategy = null
+ getInstance(): ScoringStrategyFactory
+ getScoringStrategy(type: Cribbage.StrategyType): ScoringStrategy
<<singleton>>
LoggerHelper
-loggerHelper = new LoggerHelper()
+ logScore(tempSegment: Cribbage.Segment, newScore: int, score: int, scoreType Cribbage.ScoreType): void
+ logScore(tempSegment: Cribbage.Segment, newScore: int, score: int, scoreType: Cribbage.ScoreType, hand: Hand): void
+ logDiscard(hand: Hand, currentPlayer: int): void
+ logDeal(hand: Hand, currentPlayer: int): void
+ logSeed(seed: int): void
+ logPlayer(playerId: int, playerType: String): void
+ logShow(temptSegment: Cribbage.Segment, starterCard: Card): void
+ logStarter(card: Card): void
+ logPlay(temptSegment: Cribbage.Segment, currentPlayer: int, nextCard: Card): void
<<singleton>>
Logger
-logger: Logger = new Logger()
-filePath: Path
+getInstance(): Logger
+log(String string): void
<<abstract>>
ScoringStrategy
-logger: Logger
-segmentScoring: Cribbage.Segment
-currentPlayer: IPlayer
+ getCurrentPlayer(): IPlayer
+ setCurrentPlayer(): void
+ getSegment(): Cribbage.Segment
+setSegment(): void
+getScore(): int
use
contains
rules
1
1..*
PlayPairStrategy
PlayRunStrategy
ShowFifteenStrategy
ShowJackStrategy
ShowRunStrategy
PlayTotalAndLastStrategy
ShowFlushStrategy
ShowPairStrategy
+ getScore(): int
<<abstact>>
CompositeStrategy
-strategyList: List<ScoringStrategy> = new ArrayList<>
+ setSegment(segment: Cribbage.Segment): void
+ setCurrentPlayer(currentPlayer: IPlayer): void
+ addStrategy(scoringStrategy: ScoringStrategy): void
+ getScore(): int
CompositePlayStrategy
CompositeShowStrategy

Figure 1) UML Diagram of additional features including Scoring & Logging