SWEN30006 Project 2

Emaad Beig (1078145), Fu-Sheng Huang (1046432), Tiancheng Chen(987877)

Preface

The following is our account for the changes made to the Cribbage game program provided along with the specifications. The report contains a rationale for all the design choices made in order to align the system with the GRASP framework and all design patterns used.

Logger: Singleton Pattern

Since our first task when approaching the game of cribbage was to add scoring, we needed to first create a way to effectively keep track of all the events in the game so they could be accounted for in the scoring implementation. For this purpose, we created the class **`Logger`**. The Logger class uses the design principles of the Singleton design pattern, since the program requires only one instance of it that stays consistent during the entire runtime whilst also having global access. The Cribbage class is given access to the Singleton, which is called every time the game state is updated - so it can append all changes to the log file as a string containing the relevant information.

Even with the Logger class in place, the base code initially lacks any means of storing information as the game progresses. For instance, the discarding of cards. The discard function had to be altered such that when a card is discarded to the crib, its value is stored in a local variable before it is removed from the hand until it is logged away.

Scoring: Strategy Pattern

Having read the rules of cribbage, it is notable that there are multiple ways to win points in a game. Due to the variety of ways points can be scored, we decided to make use of the Strategy design pattern to implement scoring. By creating **`IScoringStrategy`**, we were left with a common interface to use for all the various methods to score. Each scoring possibility extends the **`IScoringStrategy`** Interface. This allows for all the methods to have high cohesion within their own classes and avoids any added confusion between different kinds of scoring. The interface itself allows the program to

assess the state of the game one segment of play at a time. As such the only variables within it are the Player, Segment, Score and the Logger to log the score calculated.

Scoring Context

When assessing a hand to find pairs to add to the players score, the segment in play has to be cloned before it is passed into the corresponding scoring strategy class. To ensure that no changes are made to the actual segment in the assessment process. In order to do this effectively, we created a **`ScoringContext`** class that contains all the relevant information required to calculate and log a score – including the type of scoring strategy being used, the segment of play, the concerned player and finally the score itself.

The following is a brief description of the individual scoring classes that can use the scoringContext and increment the scores and log events:

- **`ScoringStarter`:** This class is responsible for awarding the dealer two points in the event a Jack is pulled as the starting card.

Play Phase

- **`PlayCardSum`:** This class simply checks whether the total reached in this segment of play equals 31, 15 or neither and a go condition is reached, and points are awarded accordingly.
- **`PlaySames`:** This class checks for the number of cards with the same rank played in the segment. Starting with a check for 4 cards of the same rank, followed by checking for 3 and then 2 using the extractQuads, extractTrips and extractPairs functions of the Hand class from jcardgame package respectively. Points are awarded accordingly.
- **`PlayRuns`:** Similarly, this class is responsible for extracting any runs that occurred in this segment. The getSequences function helps find the longest possible sequences of cards that qualify as a run. Points are awarded accordingly.

Show Phase

- **`ShowJack`**: This class simply checks whether the jack is of the same suit as the starter. To get the starter of the segment, we use the segment.getLast function and we use the getSuitId to check if the card has the same suit as the starter.

- **`ShowRuns`**: This class checks for the number of runs from the dealer and non-dealer's hands plus the starter card.The method segment.extractSequences(n) helps extract the runs from the arraylist of hand.
- **`ShowPairs`**: This class is responsible for extracting the pairs, trips and quads that occurred in hands similar to the play phase. Also points are awarded accordingly.
- **`ShowFlush`**: For this class, we need to first check the number of flushes by the function getNumberOfCardsWithSuit and then extract the starter card to see if it is also the same suit as the cards in hands.
- **`ShowFifteen`**: In this class, we aim to find out the sum of any set of cards equal to fifteen and then plus the score. So we would first print the subsets whose sum is equal to the fifteen and create an array as per n(decimal number) with the function int[] x = new int[hand.getNumberOfCards()], then check the sum of the subset by using a for loop, at last we print the subset of cards for which the sum is 15 and pass it to the logger.

Phase dependant scoring: Composite Pattern

Since a certain segment of a round can obtain points in more ways than one (such as in the lay phase getting points due to both a total of 31 and having a pair of the same rank), the entire segment must be scored based on all relevant scoring strategies depending on the phase of the round. The methods to score can largely be divided into two sub-categories, that being the 2 phases of the round that the players can gain points in: the Play and Show phases (with the exception of when the starting card is a Jack).

As such, we decided to implement the composite design pattern to collectively check a segment and award points from all scoring strategies relevant to the phase of the game. So the **`ScoringSystem`** interface is extended by both composite classes such as **`CompositeScoring`** and atomic classes such as **`ScoringStarter`** alike. The CompositeScoring class now serves as the basis for both **`PlayCompositeScoring`** and **`ShowCompositeScoring`**. Refer to appendix [2]

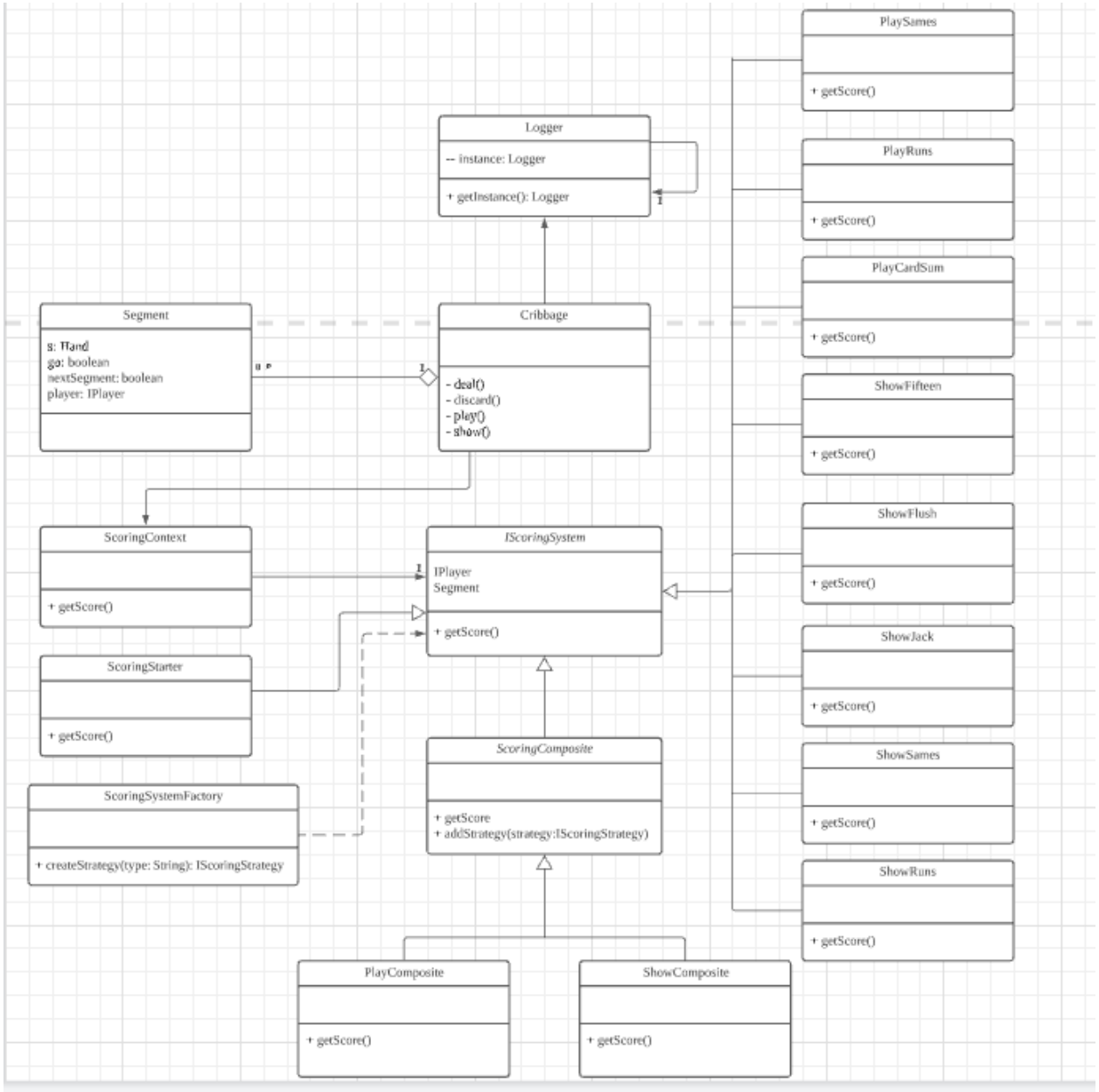
With pure fabrication, this method increases cohesion and reduces complexity in the Cribbage class. Since the addition or removal of a method of scoring from either of the phases can now be done modularly, allowing for easy modification when implementing a version of the game with altered rules.

Creating Scoring Classes: Factory Pattern

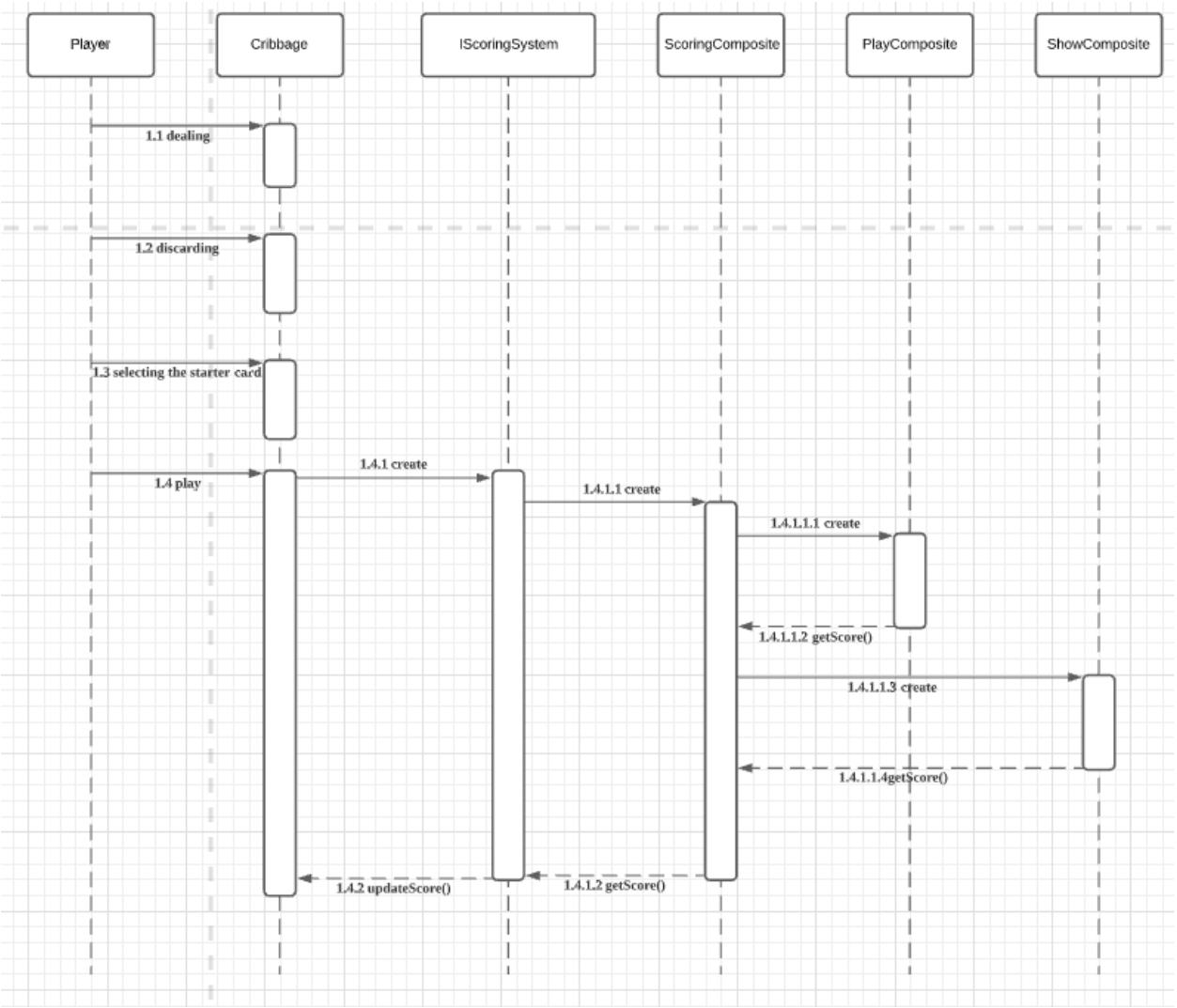
With both the scoring strategies and the composite classes containing them in place, we found the main Cribbage class to become exceedingly messy. We decided that in order to increase cohesion once again to delegate the task of the creation of these scoring objects to a **`ScoringSystemFactory`** class. This pure fabrication class contains a simple switch statement that can determine the phase of the game and accordingly create the corresponding scoring objects.

In order to integrate this class we had to make changes to the main Cribbage class such that it is called in the relevant phase of play, with the input of the word ‘play’ or ‘show’ accordingly to accurately navigate the switch statement. Since the Factory needs to be accessed from various different classes, it was necessary to also make it a singleton for global access.

Appendix



[1] A Domain model for our implementation for Cribbage Scoring



[2] A System Sequence Diagram outlining the structure of the Composite and Strategy Patterns implemented