SWEN30006 Software Modelling and Design
Project 2: Cribbage Trainer
Team number: W04 Team 02 [Mon 01:00PM]
Team member: Yu-Wen Michael Zhang (1089117), Linyan Zhu (1074009), Xinyue Zhang (984983)

Part 1: Domain Model

In this section, a partial domain model is drawn to help us fully understand what functionalities we need to implement to fulfill the requirements (calculate the scores for each player).

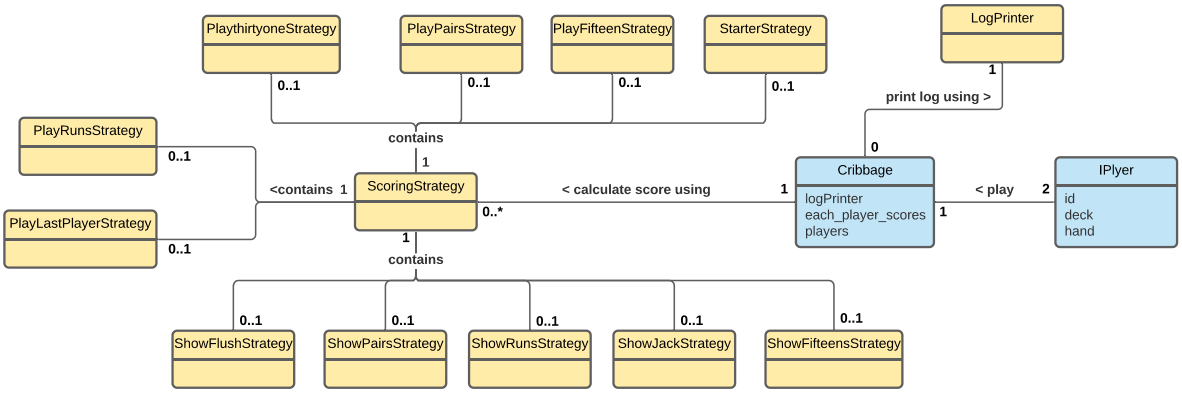

Figure 1: Domain Model Diagram of our design. The classes shaded in yellow are created by us and the blue one is the classes already exist in Cribbage Trainer.

Part 2: Design pattern and partial design model

A complete partial design (excludes Class Utility) model diagram will be in Part 4 figure 8.

2.1. Strategy Pattern

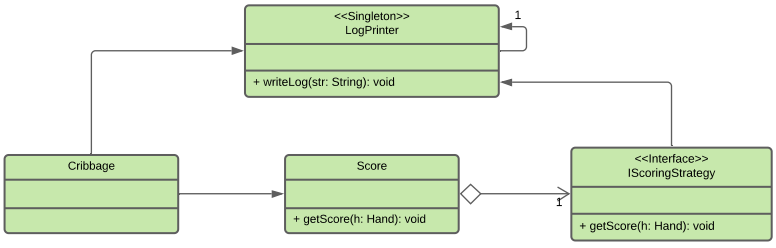

Figure 2: partial design model of Strategy Pattern

Since there are lots of ways a player can score a point, we decide to implement Strategy Pattern in our program to achieve that. For every scoring rule we create a strategy class that implements the interface IScoringStrategy. In this way it is very convenient to combine rules together to calculate a player's score, as well as provides great extendibility for any future modifications (**Protected Variation**).

Where Score class is used as a Context class in the strategy pattern. Singleton LogPrinter in the IScoringStrategy allows every concrete strategy to implement it and grants each strategy to have the ability to access the log file.

If any new scoring strategy is required, we can simply add a new concrete class that implements IScoringStrategy. However, in the original implementation, the Cribbage class (client class) needs to know every concrete scoring class and instantiate it. In this project, we need to go through some of the play strategies while a player places a card and add up the score to the corresponding player. As we are applying same action to multiple strategies and we don't want Cribbage class to concerned about how every strategy is used, we delegate the responsibility to other classes by applying composite pattern and factory pattern to make each class focus on their own job. By doing this, **cohesion** between classes is increased.

2.2. Factory Pattern

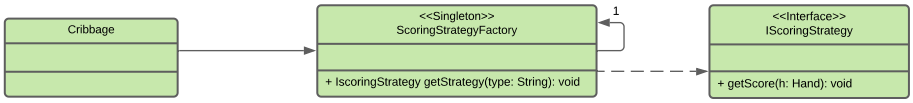

Figure 3: Partial Design Model for Factory pattern

One instance of the Singleton Pattern is the ScoringStrategyFactory. This class is created to separate the responsibility of creating IScoringStrategy (mentioned in the previous section) from other classes. It also hides the complex creation logic from the main program and provides **Protected Variation**. As there is no need to create multiple Factory instances, the Singleton Pattern is used to provide a single access point through global visibility.

2.3. Composite Pattern

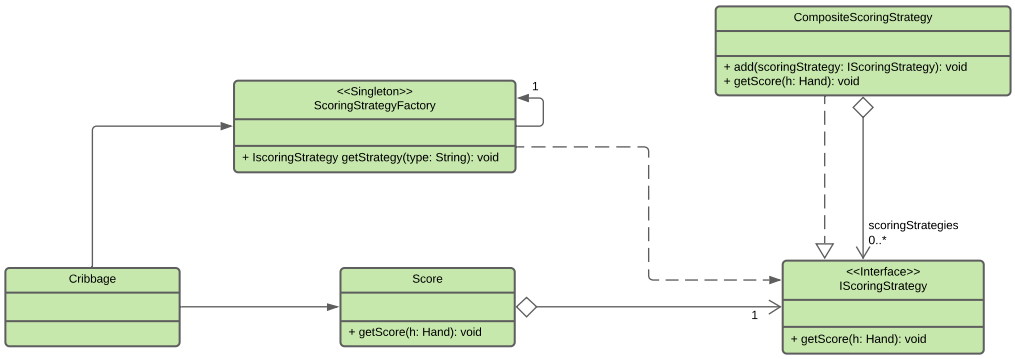

Figure 4: Partial Design Model for Composite Pattern

Two main parts of this game for a player to score points is the Play Stage and the Show Stage, where each stage contains several ways to score some points. Thus, a Composite Pattern can be helpful in this situation.

The CompositeScoringStrategy class implements the IScoringStrategy interface and contains a list that stores all the strategies follow the specified rules. We can simply add any applicable IScoringStrategy to the CompositeScoringStrategy's list and let it loop through all possible rules to score some points (**Protected Variation**).

Furthermore, after the ScoringStrategyFactory instantiates all the strategies we require, it then passes these strategies to the context class Score from Cribbage to execute the getScore method which defines in CompositeScoringStrategy. More advantages of creating class Score can be found in section 2.4.

2.4. Context Object Pattern

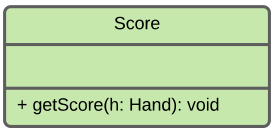

Figure 5: UML Diagram for Score Class

We create a new object called Score as a context object so that we can simply call the getScore method in Score to get the score of current strategy regardless of what type of strategy it is. In this way, **High Cohesion** is achieved. Also, reusability and maintainability of our application can be improved.

2.5. More on Singleton Pattern

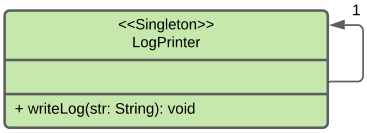

Figure 6: UML Diagram for LogPrinter Class

As no classes in the original version should be responsible for printing logs into a file, by **Information Expert**, we assign the responsibility to a new class called LogPrinter. This class is used to print logs only. Furthermore, we decide to consider LogPrinter as Singleton because it only needs to be instantiated once in Cribbage. After that, we can use the same instance to print logs.

2.6. Helper Class

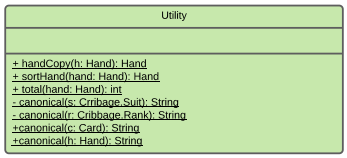

Figure 7: UML Diagram for Utility Class

Some utility methods, such as creating a copy of exiting Hand or sorting a Hand, are used in several classes. Hence, we decide to separate those methods from the Cribbage class to achieve **High Cohesion** and **Low Coupling**. A helper class Utility is created to handle

these common operations. As this class only provides utility methods for other classes, there is no need to create instances of this class. Thus, 1. we mark this class as final to prevent others extending from this class; 2. all methods within this class are declared static and 3. it has a private constructor to prevent others create instances.

Part 3: Discussion on reasonable options

We have considered making the LogPrinter not be Singleton but just a simple class. However, we prefer making it as the former one because we can save memory by not creating a new LogPrinter at each request. If LogPrinter is not Singleton, we need to use Java keyword: "new" to create memory for this LogPrinter and use this created new instance to print logs. It is memory-consuming if the application is large. Hence, considering LogPrinter as Singleton is better.

Furthermore, we also try to make CompositeScoringStrategy as an abstract or an interface class and create two more composite strategies called CompositePlayScoringStrategy and CompositeShowScoringStrategy. However, since we are applying same actions on both play strategies and show strategies, one concrete CompositeScoringStrategy class is enough for both play and show scoring strategies.

Part 4: Diagram

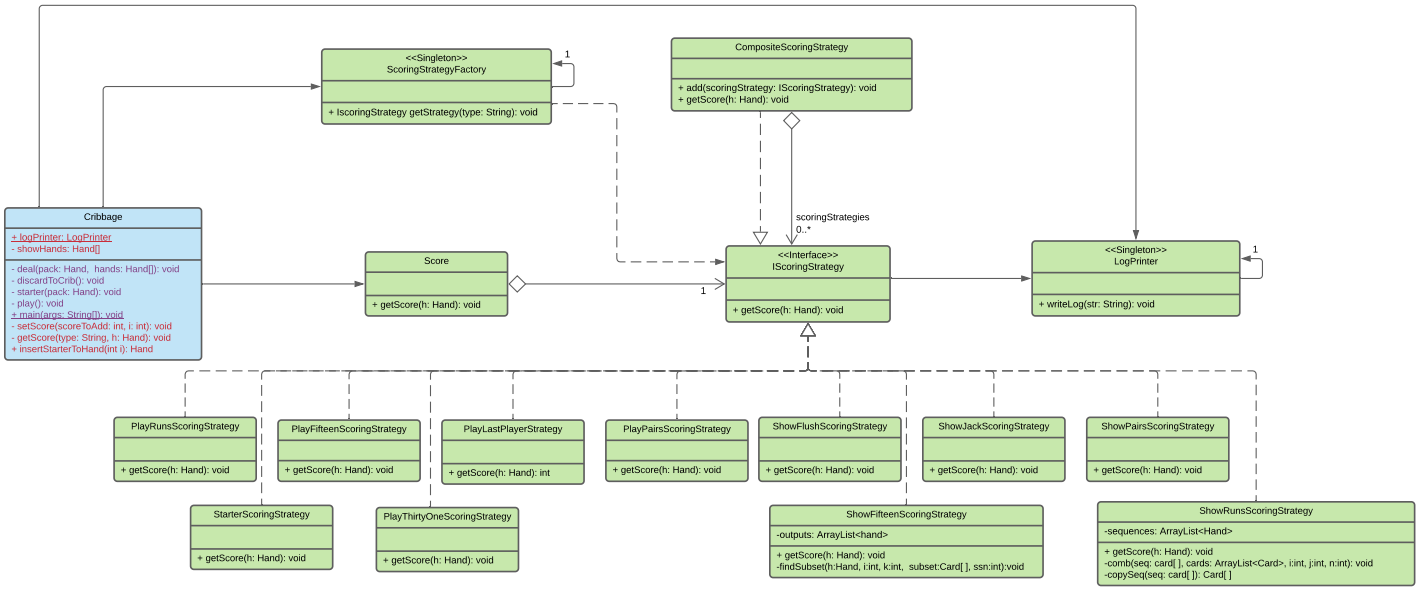

Figure 8: complete Design Model, where the class shaded in blue is the class exist in the original Cribbage Trainer and the green classes are created by us.

In Figure 8, As there are too many methods and attributes in Cribbage, we only highlight the one we modify: the red one is created by our group and the purple one already exists in the Cribbage Trainer package, but we do some modifications on it.