Cribbage Report

Pin Wang 1056745
Shiqi Zhang 978600
Deepthy Das kalathil 1104890

W01 Team 06

1 Scoring

There is a package called Play with 11 classes, each class represents a rule of Cribbage during playing. Under the Show package, there are 10 classes representing those rules during the show. Also, each class implements the interface called IScoringStrategy, having its own algorithm to calculate the score and can be regarded as a leaf of the composite. Although The structure of Show is similar with Play, the implementation and logic are different.

1.1 Play VS Show

Since during the play, the card will be placed orderly, if one player forms a run3, the other may form a run4 (so does pair), therefore, we can calculate run7,6,5,4,3 or pair4,3,2 separately. However, when it comes to Show, the five cards will be placed together to calculate points. To avoid calculating repeatedly, for example, if cards meet run5, we only add points of run5 rather than add points of run4,3 repeatedly. Also, if cards contains pair and run, there may be multiple run combinations. So we cannot consider pair and run separately in Show.

1.1.1 Fifteen

In Play, if value of total cards is 15, the player placing the latest card can get 2 points. However, in Show, we combine the 5 cards in different sets of two or more cards and see how many combinations have value of 15.

1.1.2 Run

In Play, we add points to the corresponding player if he completes run of cards, however, in Show, since there may be run and pair together, like 2H, 2S, 3H, 3S, 4D, which means there are 4 run3 combinations. Firstly, we count the number of each rank, then check whether any run exists, if there is any rank with multiple suits, combine them with others to form different sets of run and count the run number.

1.1.3 Pair

In Play, points can be added to the player completing pair of cards, while in Show, since the five cards will be displayed together, we count the number of each rank and only calculate pairs with highest priority. (pair4 >pair3 >pair2 >none)

1.2 Design Patterns and Principles

1.2.1 Strategy

Since we have a bunch of play rules, each rule has its own algorithm and calculation method. Strategy pattern can help encapsulate each algorithm into separate strategies so that we can switch them according to specific situations and the changes will not affect the program performance. Strategy pattern separates the responsibility of using the algorithm and the implementation of the algorithm, and delegates to different objects to manage these algorithms.

1.2.2 Composite

We have mentioned that each rule has been encapsulated into a single strategy, which means there are numerous strategies need to be managed. Also, for example, from run7 to run3, we want to have the execution sequence in order. And for run, pair and total, we need to combine them. Therefore, the composite pattern is used to manage these strategies. IScoringStrategy.class is the component, which is to declare public interfaces for leaf and composite objects and to implement their default behaviors(i.e. define general method–getScore and add). CompositeScoringStrategy.class is the composite as an abstract class, which is extended by two different classes: SameCategoryScoringStrategy.class and DifferentCategoryStrategy.class. For SameCategoryScoringStrategy.class, it defines exclusive strategy combination method for those main types of strategies, during the play, total, run and pair are main types of strategies, while during the show, main types are flush, jack, run, pair and fifteen. We combine the same type strategies and add them to separate same scoring strategy lists, while the same composite strategy only focuses on and picks up the highest priority strategy among the same type, which means in play, run7>run6>run5>run4>run3; pair4>pair3>pair2; 31>15 and in show, run5>run4>run3; pair4>pair3>pair2; flush5 >flush4. Then we use another strategy list with different composite strategy to sum those same strategy lists mentioned above and calculate the final scores. In this composition, we can guarantee a reasonable sequence of score calculation.

1.2.3 Singleton Factory

As for obtaining different strategies, under consideration that avoiding creating many strategy objects each time, we choose Singleton Factory to initialize corresponding strategies. During the creating strategy process, the factory is also combining strategies as mentioned in 1.2.2. Finally, the factory will return the game strategy according to the stage of the game(i.e. play or show). Additionally, by using singleton pattern, it can ensure that there is only one instance to combining strategies, reducing the memory overhead and avoiding consuming multiple occupation of resources as well as performing effectively.

2 Logging

2.1 Implementation

2.1.1 CribbageGame

From Cribbage.class, we extracted several functions, which are updateScore, deal, discardToCrib, starter, go, playCard and showHand to generate an interface called CribbageGame. The aim is that, CribbageGame is the component in Decorator pattern, which will be implemented by Cribbage and CribbageGameDecorator.

2.1.2 CribbageGameDecorator

This is an abstract decorator class, the base decorator class for managing concrete decorators. It defines an interface consistent with the CribbageGame interface and holds an CribbageGame instance. Through this class, additional methods are dynamically added to Cribbage, which is the concrete component.

2.1.3 LoggingCribbageGameDecorator

This class extends CribbageGameDecorator.class, implementing parts of methods mentioned in CribbageGame – deal, discardToCrib, starter, playCard and showHand. The function of this decorator is printing the information of current statements which are not related with scores.

2.1.4 ScoringCribbageGameDecorator

This is the other concrete decorator class, which implements playCard, showHandsCrib and go methods in CribbageGame. It can update the latest score and output the score to the log file.

2.1.5 Others

We have changed the places calling decorator methods. In main method, the logging decorator is created and the instance calls deal, discardToCrib and starter methods to show the information of initial phase of each playing round.

2.2 Design Pattern – Decorator

Because logging is to help players study the impact of their choices on the play and score, it can be regarded as an additional function. So decorator pattern is utilized for implementing logging. It allows to add new features to an existing object without changing its structure. It creates a decorative class to wrap the original class and provide additional functions while maintaining class integrity.

3 Other Design Choices

3.1 Two Decorators VS One Decorator

Considering that we have to display game statements, firstly we thought implemented only one decorator, which can update scores and output logs. It combines two functions and has low coupling with the original Cribbage.class. However, it is difficult for us to implement this hybrid decorator class and there is high coupling between score and logging. Instead of single decorator, we came out another method–two decorators. For logging decorator, it only outputs the information with the current statement unrelated with scores, such as play, show, discard, deal and starter. For score decorator, not only does it update scores, but also outputs the log about scores. In that case, we decoupling the relationship between logging and score, also do not influence Cribbage.class.

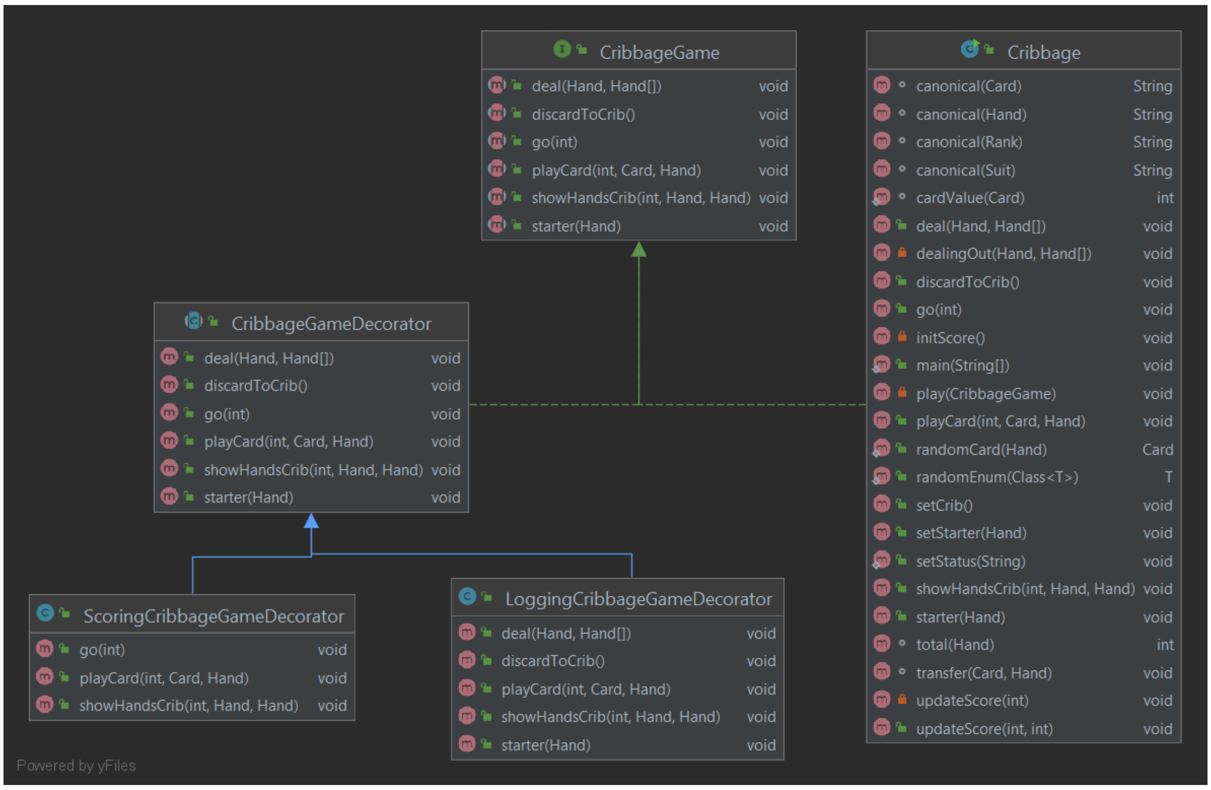
3.2 Adapter VS Decorator

Under the condition that we have implemented score and logging functions and hope to combine them with Cribbage.class, also these two functions are quite different from Cribbage.class, we can also choose adapter pattern to connect them. It transforms one interface into another one, and to achieve reusing by changing the interface.
Compared with decorator pattern, the adapter pattern establishes connection between log, score and

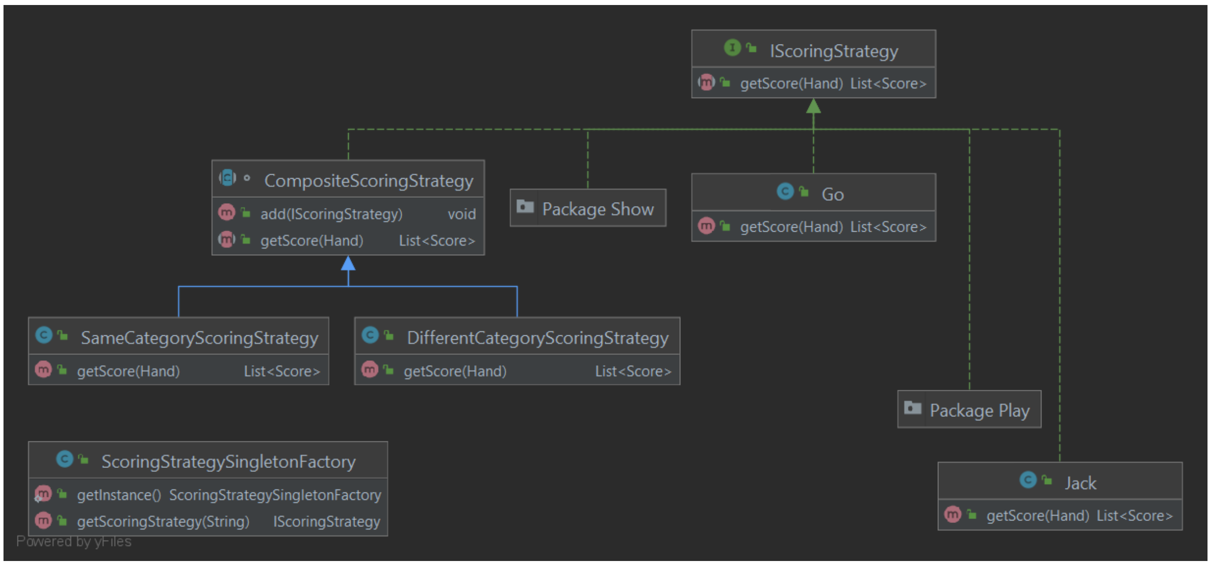
cribbage. We need to add a series of interfaces and classes, leading increasing overall complexity. Meanwhile, because it is more difficult to remove or modify, we need to consider whether it can be revoked or changed from the whole system for future modification. However, decorator is scalable and easy to be modified for future maintenance. Since each strategy has its own points, if the points would be changed, we can modify the point variable in corresponding strategy class. If the rules of point calculation would be changed, we can modify the combination of those small strategies conveniently.

4 UML Diagrams

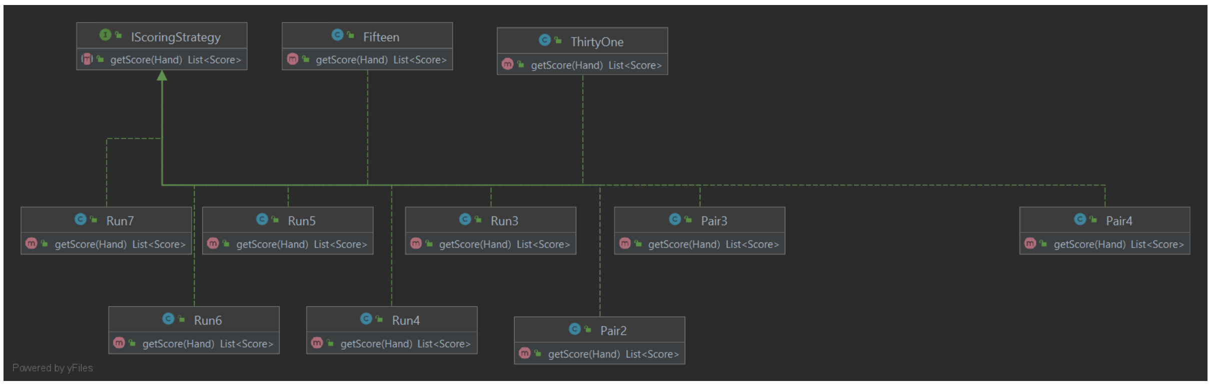
4.1 Cribbage Decorator



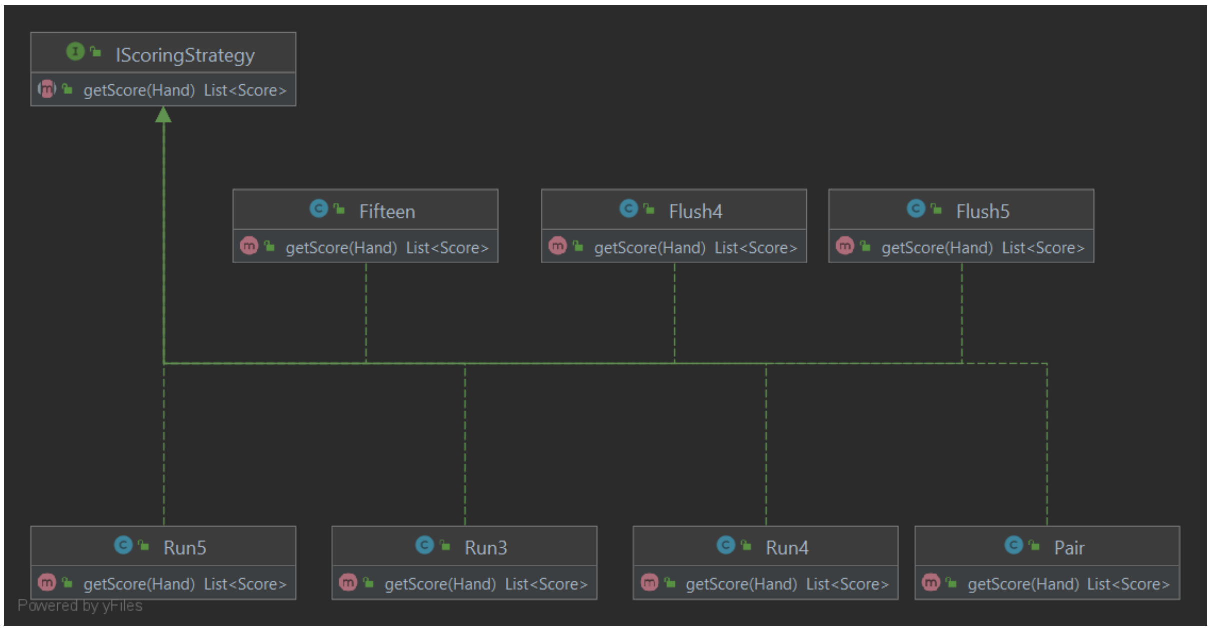
4.2 Score Strategy Composite


IScoringStrategy
getScore(Hand) List<Score>
CompositeScoringStrategy
add(IScoringStrategy) void
getScore(Hand) List<Score>
Package Show
Go
getScore(Hand) List<Score>
SameCategoryScoringStrategy
getScore(Hand) List<Score>
DifferentCategoryScoringStrategy
getScore(Hand) List<Score>
Package Play
ScoringStrategySingletonFactory
getInstance() ScoringStrategySingletonFactory
getScoringStrategy(String) IScoringStrategy
Jack
getScore(Hand) List<Score>
Powered by yFiles

4.3 Play Strategy


IScoringStrategy
getScore(Hand) List<Score>
Fifteen
getScore(Hand) List<Score>
ThirtyOne
getScore(Hand) List<Score>
Run7
getScore(Hand) List<Score>
Run5
getScore(Hand) List<Score>
Run3
getScore(Hand) List<Score>
Pair3
getScore(Hand) List<Score>
Pair4
getScore(Hand) List<Score>
Run6
getScore(Hand) List<Score>
Run4
getScore(Hand) List<Score>
Pair2
getScore(Hand) List<Score>
Powered by yFiles

4.4 Show Strategy


IScoringStrategy
getScore(Hand) List<Score>
Fifteen
getScore(Hand) List<Score>
Flush4
getScore(Hand) List<Score>
Flush5
getScore(Hand) List<Score>
Run5
getScore(Hand) List<Score>
Run3
getScore(Hand) List<Score>
Run4
getScore(Hand) List<Score>
Pair
getScore(Hand) List<Score>
Powered by yFiles