# Report for W16 Team4

In this project, we as a team added scoring and logging features to the trainer, with a flexible design which allows for future maintenance and modification by the NERD team.

## Changes made to the system regarding *Scoring*

To implement the scoring feature in the trainer. We as a team decided to implement a *singleton factory* with *composite strategy* pattern to add the feature to the system, while maintaining *low coupling* and *high cohesion* (see figure 1.1 below).

- **The differing Scoring Rules**
    - To design the varying, but related scoring algorithms, and the ability to *change* and *extend* these algorithms easily, we chose to implement a *strategy pattern*.
    - With the different types of scoring policies and strategies implementing a common interface that is **ScoreRule.** The **ScoreRule** possess a singular attribute that is score, as well as a single method that is getScore().
    - Implementing this design allows for an easily extendable system. In the future, when there is a need to implement additional scoring rules, we may have them implement **ScoreRule**, achieving *high cohesion* and *low coupling* as an aftermath.

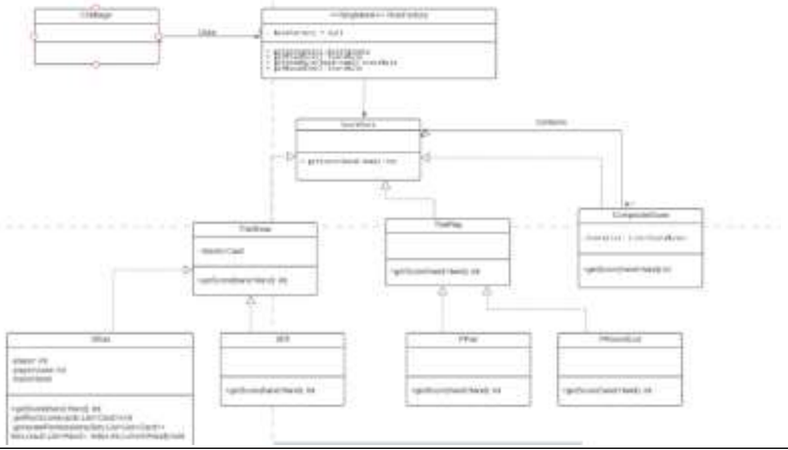
Figure 1.1 Design model for scoring system

- **The Play and The Show**
    - To distinguish the scoring strategies between the Play and the Show, we decided to utilize *polymorphism* to support the *Strategy pattern* in order to achieve *protected variation* to provide flexibility as well as a more structured design to the system (see figure 1.1 & figure 1.2). This design pattern allows for future potential rules or scoring strategies to be easily added and extend the design.
    - To resolve the problem that the parameters for calculating the play score and the show score are different, (the Show requires the player's hand as well as the starter card, the Play requires the total cards played) We implemented two abstract class **ThePlay** and **TheShow**, encapsulating the scoring strategies within the interface **ScoreRule** (see figure 1.2 *Additional strategies are neglected in the figure due to spacing reasons).
- **Composite Scoring Strategies**
    - As the scores calculated through playing the show and the play requires applying all the appropriate rules to it, we decided to implement a *Composite* scoring strategy which would determine the compound scoring strategy, hence achieving *low coupling* and *high cohesion*

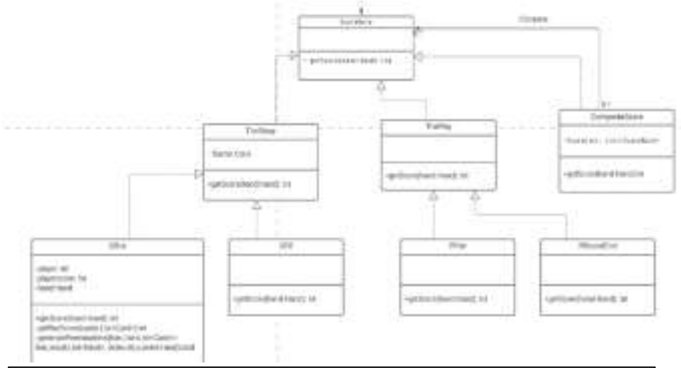
Figure 1.2 Using polymorphism

- Since the scoring rules for the Play and the Show is similar, as both would require the total addition of all the scoring rules achieved. We decided there is no need to implement multiple composite rules calculating strategies, instead we implemented a single composite class implementing this logic named **CompositeScore.**

- **Creation of the composite scoring strategies**
    - As the creation of this *strategy composite pattern* is complex. We implemented a **RuleFactory** class to solve this problem. By implementing a *Factory* class, we hide the complex score calculation logic and only make the access to the *composite* rule open. Hence achieving leaving the *extensibility open* and the *access to users closed*, improving the *cohesion* of this system.

## Alternative designs that we considered

### Alternative design 1

An alternative design that we considered regarding score calculation works with the realization that there are scoring strategies that are of the same nature. For example, the rule of pairs, both for the show and for the play the rule looks for existent cards with the same orders.

We developed a design that instead of having **ThePlay** and **TheShow**, we have an alternative *polymorphism* where the scoring rules act as a super class and the calculation of the rules as sub classes (see figure 2.1).

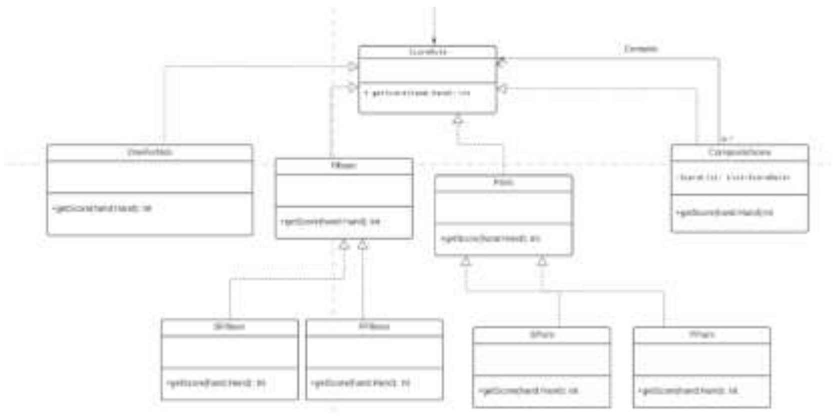
Figure 2.1 Alternative design 1

The reason we did not choose to implement this design is because of its lack of extensibility. As the number of rules grows, the code will have an increasingly difficult structure to maintain. It would also be difficult to manipulate the strategies within the *factory*.

### Alternative design2

Another alternative  design is like the previous design but instead of using polymorphism, we implemented multiple functions in the ScoreRule interface (see figure 2.2). Where getShowScore() would return the scores calculated in *the show* logic and getPlayScore() would return the scores calculated in *the play* logic. These methods will then be implemented by each of the rules.

The reason we chose not to implement this design is because of its low cohesion, poor extensibility, and the confusion that may arise. For example, for strategies such as One for the Nob are only for *the show* rules, it does not make much sense for it to have a method that is getting the play version of it.

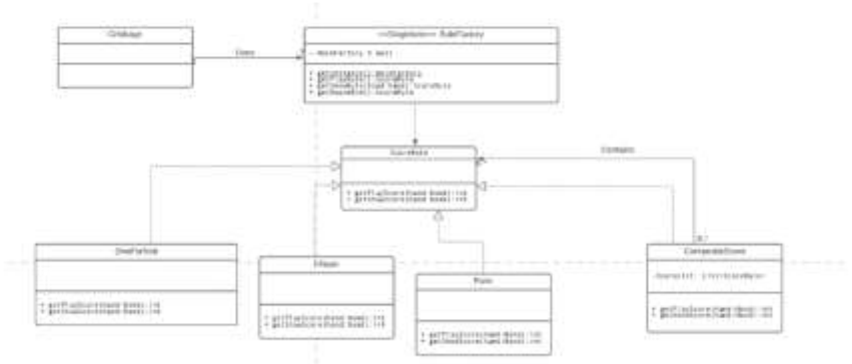
Figure 2.2 Alternative design 2

## Changes made to system regarding *logging*

The implement the Logging functionality, we decided to implement a **logger** that controls all the logging methods. We would then call the logger within **Cribbage** and call whichever methods we require to log the data.

To better implement the logging functionality in the framework given, we decided to refractor the **canonical** methods within the original framework and embed them as methods within the **logger** by the _information expert_ pattern as well as _the creator pattern_**.** We believe that assigning the **canonical** methods to **logger**, where they are being used closely, would achieve a _higher cohesion_ and _lower coupling_ than to leave them in the **cribbage** class, where they are not used at all.

## Alternatives

We have considered using a _Façade_ controller design pattern or using _an observer pattern_.

**Façade controller design pattern**

We have considered implementing a design where we would have a _façade controller_ that hides all the logging logic and only presents the necessary methods to the cribbage class. However, as we implement the logging methods, we find that our reasons of using a _façade controller_ would not be justified. As a _façade controller_ is designed for a system where it "requires a common, unified interface to a disparate set of implementations or interfaces—such as within a subsystem—is required.". However, within the logging system there isn't really a dissimilar set of implementations. Hence, we did not choose to implement a _facade controller_. (See figure 3.1)

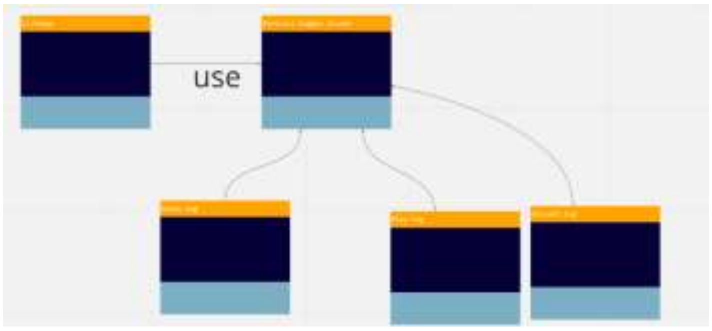

Figure 3.1 Alternative design using Façade controller

**Observer pattern**

We find the concept of an _observer pattern_ would fit nicely within a system where logging and scorings are interested in the state changes of the cards and would react in their own way (logging and scoring) when the **cribbage** system generates an event. However, with the framework provided and the system that we are currently working with, with **the jcardgame** and **jgamegrid** libraries, we do not see a way of implementing it, as the _observe_r would be observing the **cribbage** class, and updating the **cribbage** class, which seems counter intuitive.

_Potentials of an observer pattern_

The design for logging may be improved. Potentially having _a façade controller_ to unify the **jcardgame** as well as the **jgamegrid** libraries and de-couple the **cribbage** class. Which would then allow for an implementation of _observer pattern,_ where the observer would be observing the change in state of cards and log or score accordingly.

## Refactoring of the code

### GameDisplay

To provided framework has _low coupling and low cohesion_, having the play card, discard card, etc. operation logic and the display of the cards within the same class. To improve upon _the cohesion_ within this system. We refactored the code utilized _pure fabrication_ to create a new class named **GameDisplay** to support _high cohesion, low coupling, and reusability_ as well as based on the _information expert_. As we believe the current **Cribbage** class is bloated and consists of variables and methods that are _not cohesive_ when implemented within the class. Hence the responsibility is delegated to **GameDisplay**. This class contains the functionality of drawing the cards and storing the necessary information of the style of the presentation, such as fonts and text locations.