SWEN30006 Software Modelling and Design

Assignment 2 Report

Paul Hutchins - 1160468
Jade Siang - 1170856
Kian Dsouza - 1142463

THE UNIVERSITY OF MELBOURNE

SWEN30006
The University Of Melbourne
21/10/2021

1 Introduction

We will discuss our design choices with respect to our extension of the Pasur package as supplied. The two main discussion points will relate to our flexible design decisions with regards to the GameLog and the addition of the scoring functionality, by support and allow for future maintenance and modifications of the game. Our implementation of the scoring and logging for the Pasur game largely revolved around creating a scoring subsystem with a Facade controller for the different scoring rules and Observer pattern for the logging.

Also included are 3 diagrams to document the modifications made to the existing system:

- Figure 1 - Design Class Diagram
- Figure 2 - Design Sequence Diagram: Computing Score of Player
- Figure 3 - Other Considered Design Class Diagram

2 Logging (GameLog)

2.1 Our Implementation

When implementing the logging to the trainer, we decided to use the Observer pattern for multiple reasons. Opting to use the Observer pattern allows logging to occur dynamically, which seemed appropriate since we needed to log after every move a player makes and it also integrates well with the dynamic scoring system.

Including the Observer pattern also allows High Cohesion within GameLog and Low Coupling between GameLog and the Configuration, Pasur, and Player classes from which it receives information. This is due to abstracting all logging information away from the game play classes, and containing it entirely inside the GameLog class. If the logging specifications were to be altered, our approach would mean that only GameLog would need to be refactored instead of needing to change every instance of it in the other classes.

We used Indirection to create the IObserver and IObservable interfaces to further abstract the logic of the Observer pattern from GameLog. The IObserver interface is the Observer and is implemented by the GameLog, whilst the IObservable interface is the Subject and implemented by the Configuration, Pasur, and Player classes. The GameLog then subscribes to the particular IObservable it wants to receive updates from. This accounts for potential future extensions of other Subjects. For example, if we wanted to record the analytics of the game, we can easily implement an analytics logging class, or if we wanted to restrict our logging to only contain the player log, we could create a separate player log and only subscribe the player observable. Through the implementation of these interfaces, Low Coupling is achieved as the subject classes (Configuration, Pasur, and Player) do not need to be aware of the methods and processes within GameLog in order to function and only need to call their notifyObserver() method which interacts through the Observer pattern to update GameLog of the new string thats needs to be written to file.

Another key aspect of our system design was choosing to make GameLog a Singleton pattern. The logic behind this implementation was to minimize having to pass each instance of the GameLog between classes and ensures we always have the same instance of the GameLog. A different approach we investigated was to

include instances of the GameLog within each class, however we decided against this as our use case only required a single log with all the preserved records and their order. By not requiring the instance of GameLog to be passed between classes and relying on the Singleton pattern, it reduces the Coupling between the 3 logging classes and provides an efficient way to ensure the correct instance of GameLog is obtained if new classes in future extensions of the program needed to be logged.

2.2 Other Considerations

Initially, we had planned on implementing IConfigurationObservable, IPasurObservable, IPlayerObservable interfaces (refer to Figure 3) which contained a static array to hold global observers, but we deemed this to be over complicated for our implementation. The thought process behind this at the start was to decrease Coupling within the system. However, we realized that this approach was overly complex and redundant since the same level of Coupling could be achieved by only implementing the IObserver and IObservable interfaces, in which case Cohesion would increase as well (refer to Figure 1). Passing GameLog amongst the subject classes would have increased Coupling and decreased Cohesion of code, hence was not an ideal implementation.

Another option we considered was creating a new class through Indirection and Pure Fabrication which would be responsible for outputting the log after being passed the relevant string. Although this would have worked fine, we were concerned about the extendability of the system and Low Cohesion of the code. Whereas, our current approach of having a notifyObserver() method in IObservable in conjunction with the update() method in GameLog, allows our system to be more extendable so that new observable classes can be added to the system easily in the future if required.

3 Scoring

3.1 Our Implementation

For scoring, we decided to create a stable interface around the scoring subsystem using a Facade. The player interacts with this scoring subsystem during their turn (refer to Figure 2), which provides Protected Variation for the scoring subsystem by introducing a layer of Indirection in addition to maintaining Low and tight Coupling between the player and the scoring subsystem.

Since there were different methods to calculate the score during the Pasur game, we decided to use the Strategy pattern with the creation of the IGetScore interface, which allows the game to choose the correct scoring strategy during runtime (refer to Figure 2). It provides a solution by using polymorphic objects to create an interchangeable family of algorithms. This pattern is used in combination with the Facade pattern in our design.

The IGetScore interface is implemented in a way that supports future alternations to the scoring. Our approach is easily extendable and caters for new scoring methods to be added, as well as changes to the current scoring system. Furthermore, in the scenario where a new scoring method requires the output of other scoring methods, our implementation allows for it all to be handled within the ScoreFacade class without modifying the player, further exhibiting the extendability and abstraction within the program.

When the ScoreFacade requires a score to be calculated, it calls the getCurrentScore() method and we pass in the segment object containing information about the current state of the game and action played. This results in the IGetScore strategy interface creating and returning an event representing the score and its details.

This is where the responsibility of Pasur ends in terms of creating events. The Player class never needs to know about how events are scored or created as the logic is abstracted away to the scoring system, and coordinated by the ScoreFacade. No excess logic is required other than selecting and applying the action of the correct scoring method. This maintains High Cohesion in the ScoreFacade class which is another benefit of using the Facade Strategy pattern.

3.2 Other Considerations

We considered implementing all the scoring methods within the Player class, however this adds unnecessary responsibility to the player as the Player class would need to know what its current hand is worth in terms of the scoring system. It also would have also led to issues in future extensions of the program as including extra functionality to the player just aids in Higher Cohesion and Lower Coupling, which is the exact opposite of what we want to achieve.

We also debated about creating separate classes for the different scoring methods and implementing them into the ScoreFacade. Whilst this could have reduced Coupling in the instances that we did not need to pass all the players' hand data to the scoring method, mainly the surs case. Using a strategy interface between the scoring method classes and the ScoreFacade allows us to easily extend the program and add new scoring methods. Moreover, in the event we require two different methods to calculate the score, we can easily extend our program to allow for this without having to copy over all the scoring methods and have replication of code. We can either implement multiple scoring methods in ScoreFacade and only return the relevant one, or we can implement two different ScoreFacades for different circumstances without having bloating of code.

4 Conclusion

Overall, our current implementation ensures that the scoring and logging systems are completely independent of each other. In our opinion, maintaining Low Coupling and Protected Variations between the systems is an important consideration and aspect of our design.

Final Design Model Diagram

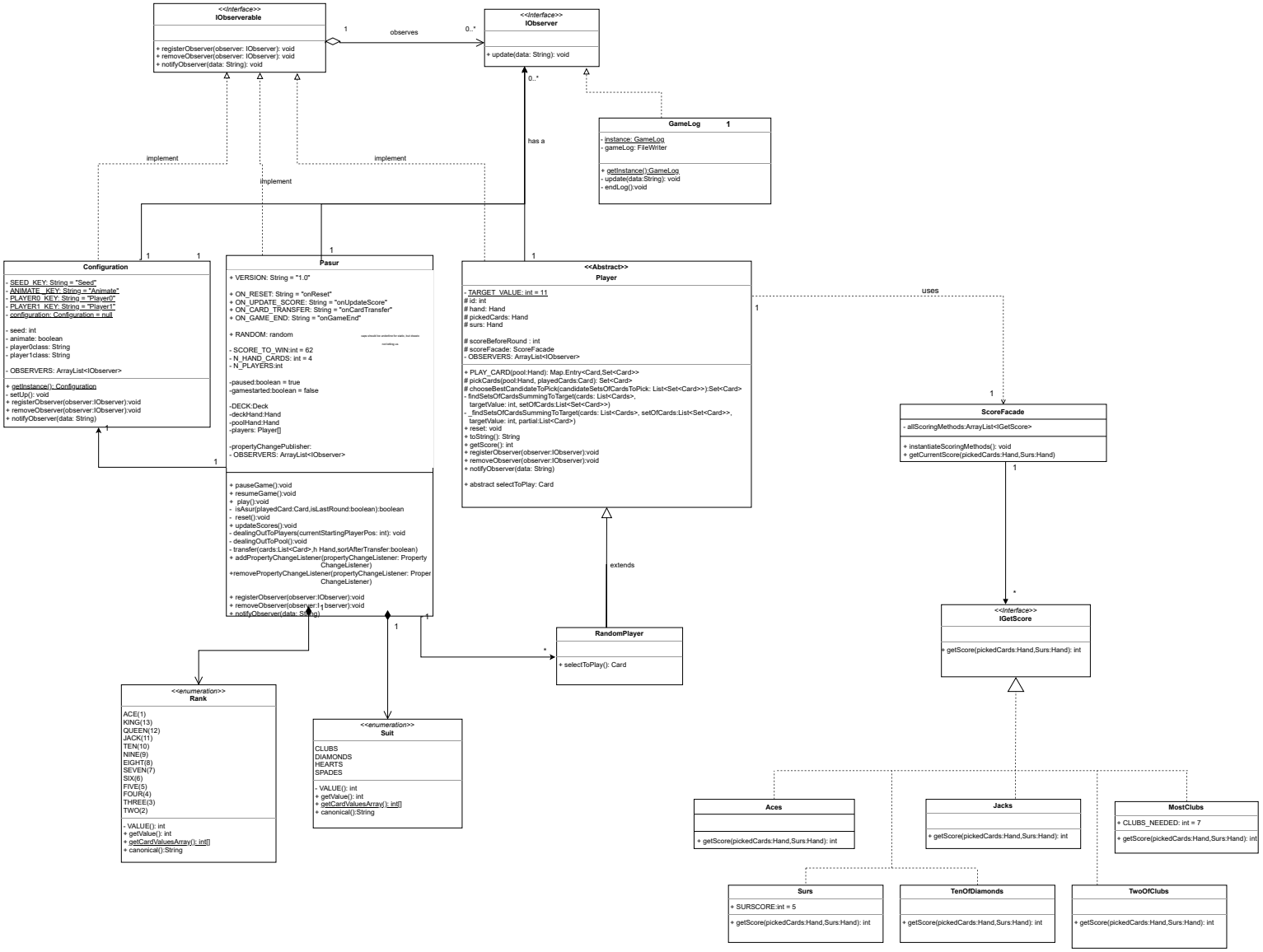

Figure 1: Final Design Class Diagram

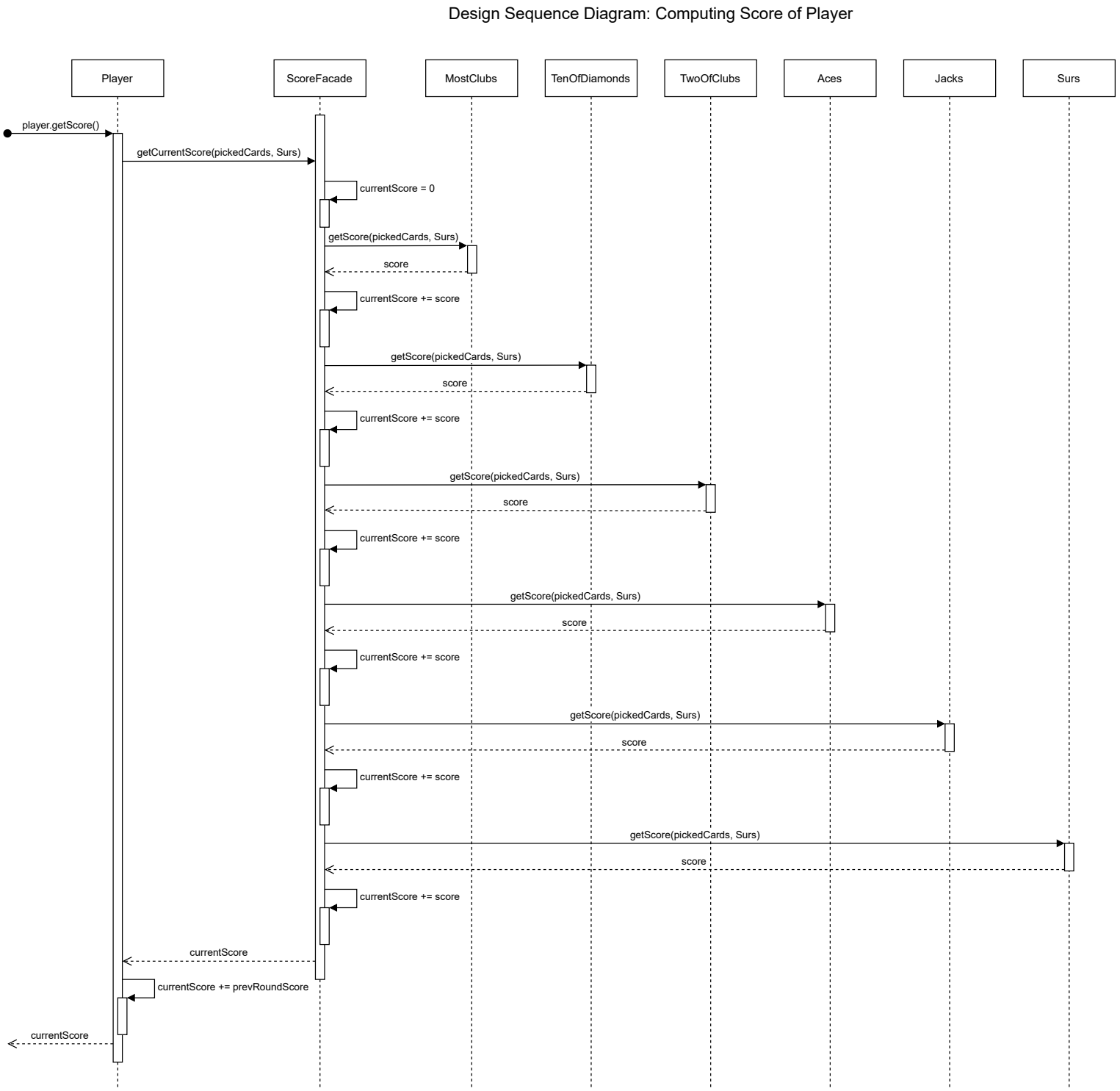

Figure 2: Design Sequence Diagram

Other Considered Design Class Diagram

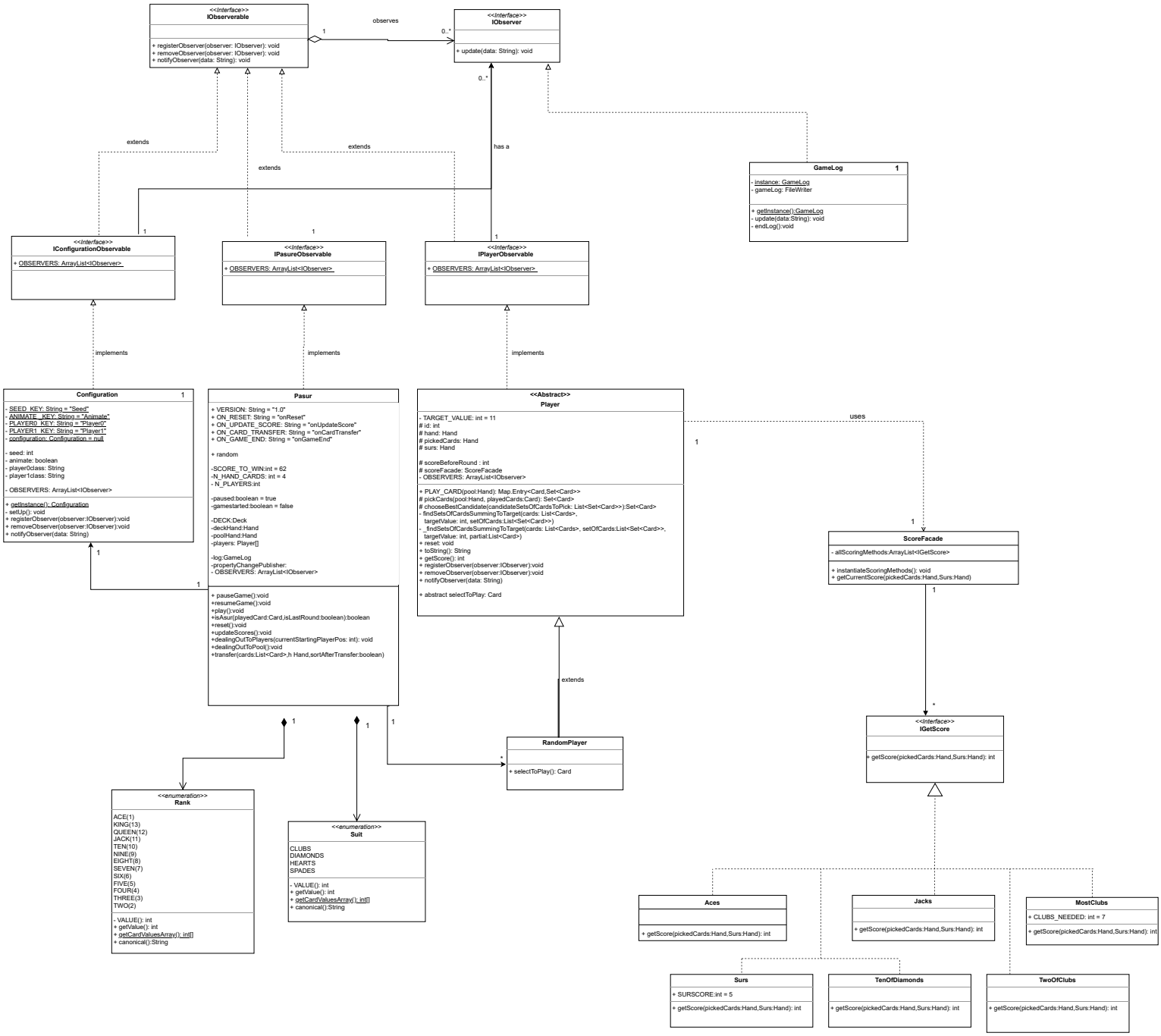

Figure 3: Other Considered Design Class Diagram