Software Modelling and Design

Assignment 2

Project 2: Pasur Trainer

Antony Tragas: 1080479

Rajneesh Gokool: 1101512

Anjaney Chirag Mahajan: 1119668

SWEN30006

University of Melbourne

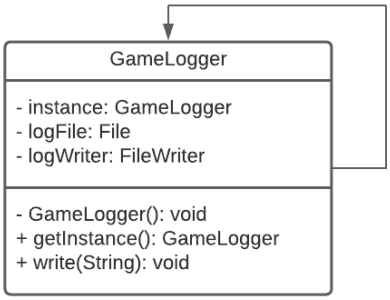
Due: 22 October 2021

Introduction

The purpose of this report is to describe the changes and extensions implemented for *NERD* game's Pasur game program. For the project we were required to implement the game scoring and logging feature for the project as well as extend the program to be easily maintainable and upgradeable for the *NERD* games. Below are the additions we have implemented.

Game Logger

The purpose of the game logger is to help players review the impact of their choices when playing the game. This is done by writing events that happen during the game to the file "*pasur.log*" in the project directory. To accomplish this we created a Singleton class called GameLogger which is a single point of access for the program to log events that happen in the game. Below is a UML class diagram of the singleton:



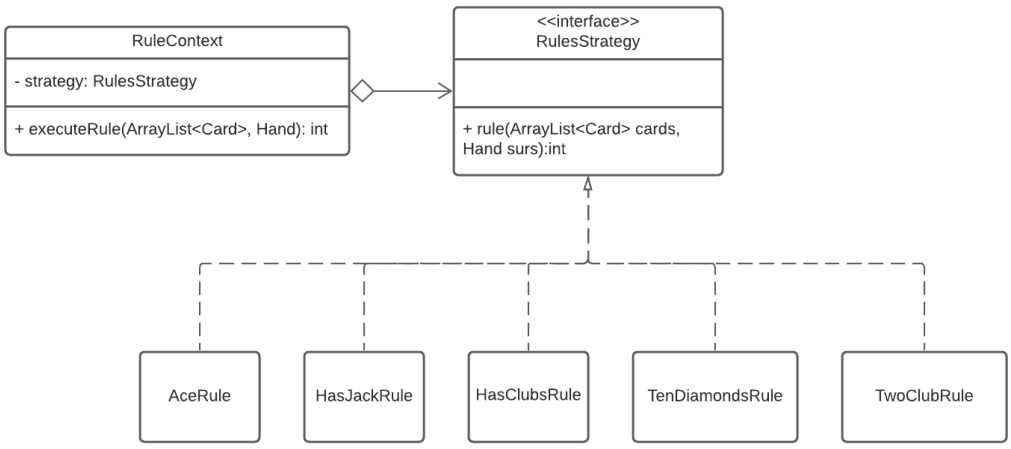
We chose to use a Singleton class as the FileWriter object for writing to files, does not need to be reinitialized in the different objects that need to write to the "*pasur.log*" file. This allows objects to store a reference to the GameLogger object by calling **GameLogger.getInstance()** and write to the

Rules Strategy

The next feature we had to implement for the project was the scoring system that adhered to a set of requirement listed in the project requirements which are listed below:
Player who has 7 or more clubs: **7 points** (there are only 13 clubs so they must have the most if they have 7 or more)
Player who has the 10 of diamonds: **3 points**
Player who has the 2 of clubs: **2 points**
Each Ace: **1 point**

Each Jack: **1 point**
Each Sur: **5 points.**

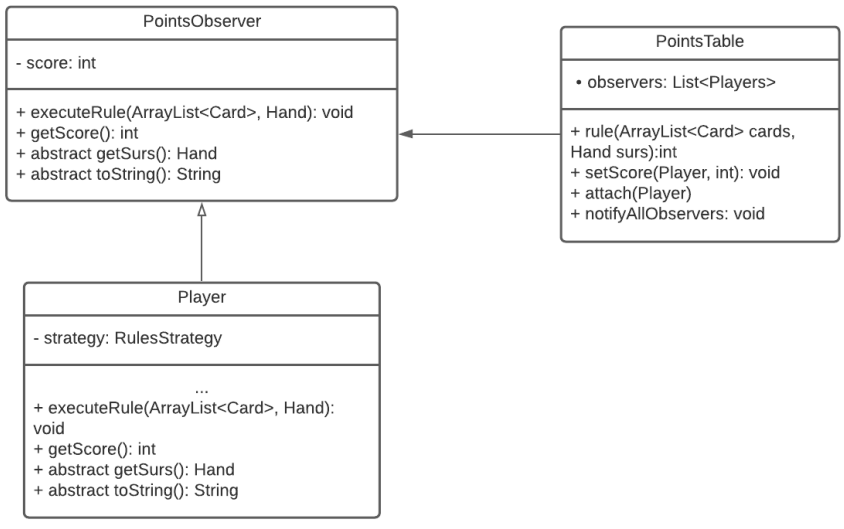
For this we decided to use a strategy pattern that could be designed to support different types of rules by using the list of cards a player has and their surs the object will calculate the number of points earned by the player. Below is the UML diagram which describes our implementation for the strategy pattern.



For this pattern we created an interface called RulesStrategy which provides a template to create new Rules for the game. We call use a Context class which provides a reference to one of the rules and communicates with this object only via the strategy interface.

Scoring Observer

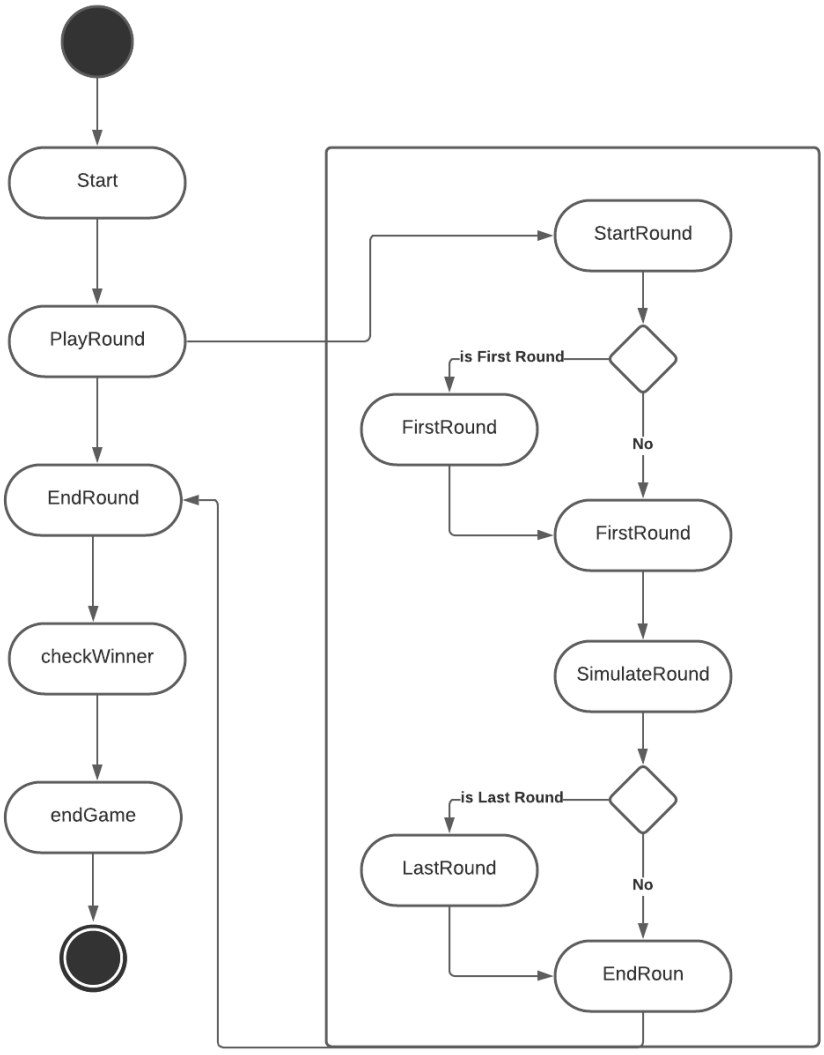
Inspired by the other Strategy Pattern for the rules system, our team decided to implement a points observer that updates the GUI interface everytime a player plays a card. This was accomplished by creating a modified Observer pattern that updates the interface whenever the player plays a card.



Updated State Machine

In order to make the system a lot more maintainable for the team at *NERD* games we took the liberty to modify the *play()* method in the *Pasur* class to use a state machine to reduce the complexity of the game play mechanics rather than use a linear system that can keep growing in complexity.

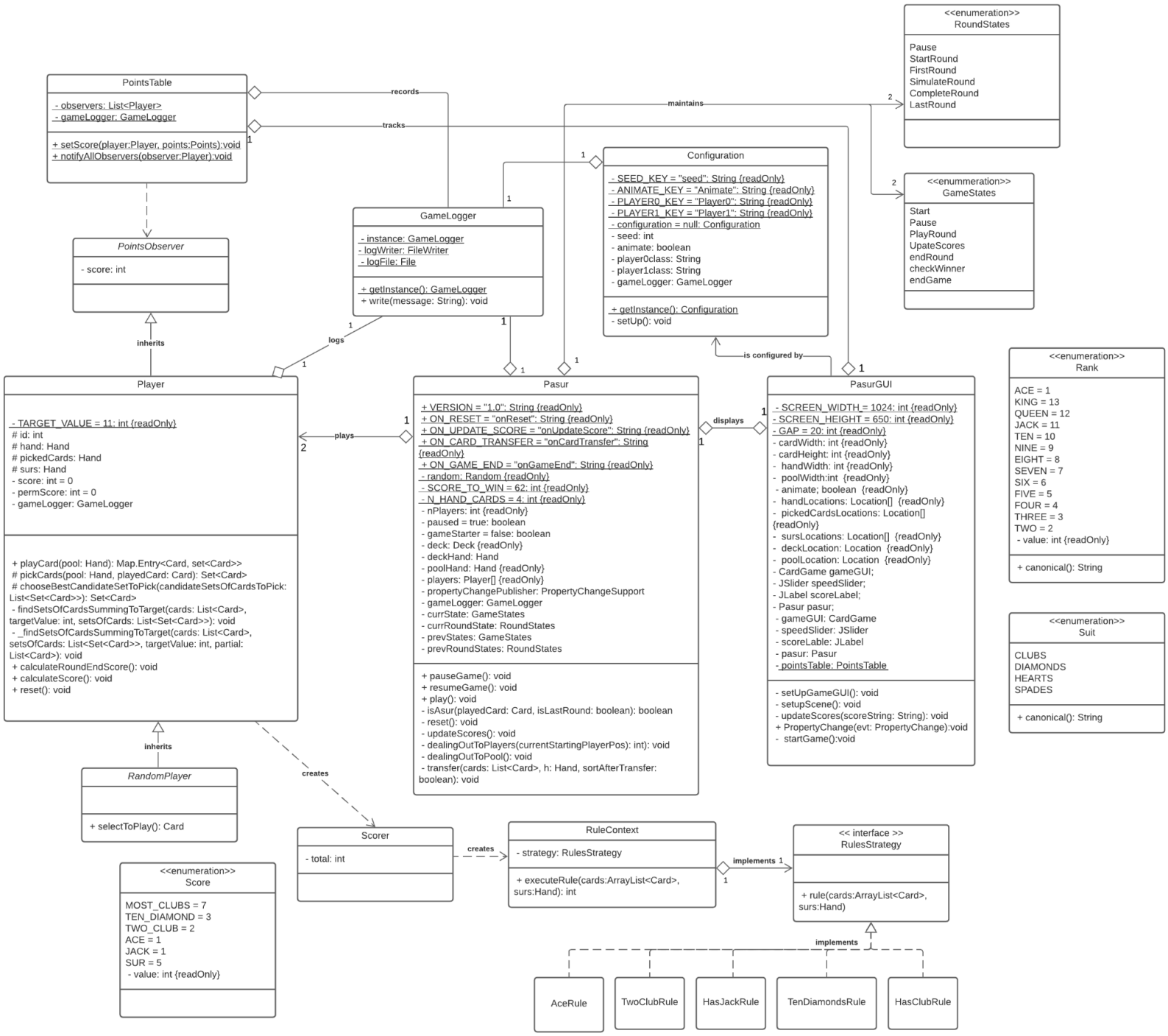
Below is a design-level state model which describes the logic flow of events that occurs in the program.



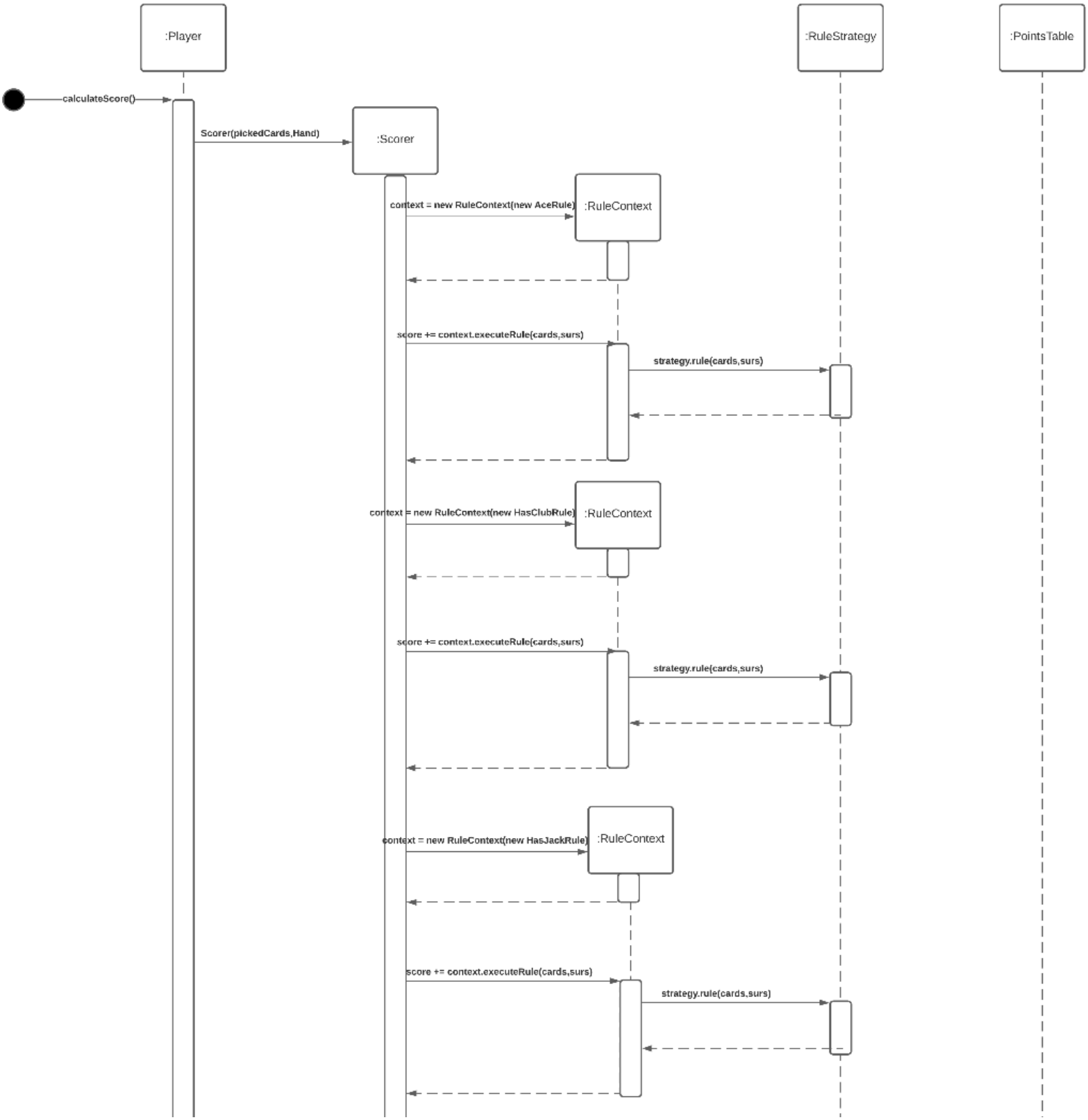
Other Considerations

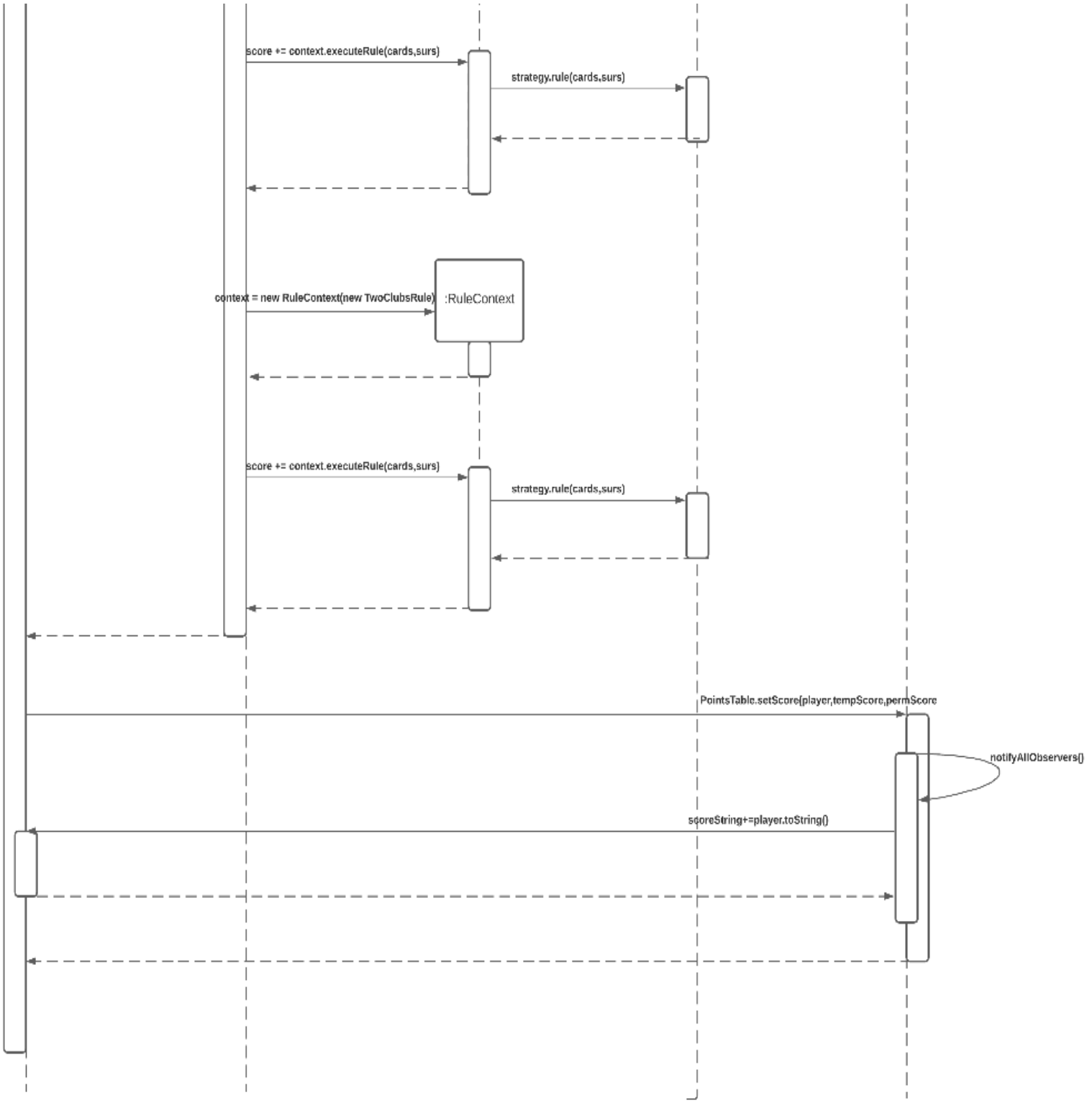
When the scoring system was first implemented we introduced a method scorer() which was used to update the points of each user. All the scoring rules were implemented in that single method with the number of points allocated based on each hardcoded in ‘if’ statements for each rule. We realised that this would not be easily extendable if NERD games wanted to add new rules or change the amount of points allocated for each rule. We then decided to use a new class, Scorer, to handle the calculating of points, an observer to tell the player when to create a Scorer to calculate the points, a strategy pattern to handle the rules and a Score enum to hold constants of points awarded for each achievement. This means that if a new rule needed to be added, one would simply need to create a new Rule class, add a new constant to the Score enum and update the constructor in Scorer.

Design Class Diagram



Design Sequence Diagram
(for calculating score)




score += context.executeRule(cards,surs)
strategy.rule(cards,surs)
context = new RuleContext(new TwoClubsRule)
:RuleContext
score += context.executeRule(cards,surs)
strategy.rule(cards,surs)
PointsTable.setScore(player,tempScore,permScore
notifyAllObservers()
scoreString+=player.toString()