CIS 350-001—Homework 4

Jonathan Levine
jonlevi@sas.upenn.edu

March 27, 2018

Question 1

The Presentation Layer in my code is comprised of the abstract parent UserInterface class and the CommandLineUserInterface implementation of that class.

The Controller Layer has the DataController class which runs the bulk of the data parsing, and the Bracket class which is responsible for using the data in the Data Layer for tournament simulation, and the Team and Game class, which are just data stores, but support more complicated updating operations such as resetting and being either subjects/observers in the Observer pattern. Since the bracket class uses all of the different types of GameSimulator classes, all of those classes(GameSimulator and its implementations CoinFlipSimulator, PennAlwaysWinsSimulator, FavoriteWinsSimulator, and EloSimulator) are also part of the Controller Layer, as well as the Observer and Subject classes which help with the simulation of the tournament

The Data Layer is more complex, and contains both the abstract EloFileReader as well as the two different implementations: EloCSVReader and EloJSONReader. It also contains the concrete factor EloFileFactory in order for the controller to access the appropriate EloFileReader. The Data Layer also contains the abstract GamesFileReader and its implementation in GamesCSVReader, but since there is only one implementation of GamesFileReader, no factory was made. In future renditions with alternate forms of GamesFiles it would be easy to change it to a similar concrete factory pattern like the Elo Files.

Question 2

My abstract class EloFileReader is implemented in 2 ways, one for JSON files and one for CSV files. The data controller uses the concrete factory EloFileFactory and its method getReader in order to get the proper reader instantiated and set up to extract data. As mentioned, it is a concrete factory.

Question 3

The observer and subject parents classes were used as given. The Team class and the Bracket class, extended the observer class. In Team.java, Lines 62-67 represent the update function, which takes in a Subject which is always a Game, and updates the Team's win/loss record based on the winner of that game. In Bracket.java, Lines 53-68, the update function takes in a Subject which is always a Game, and updates the bracket to represent that one game is over and that the winning team will either be the champion or advance to the next game, which should then be simulated. In Bracket Lines 70-85, the game simulation attaches the bracket and the two teams to each Game object, and then when the Game is finished, detaches the bracket and two teams to that game object. Each game object extends Subject, and gets the bracket and team objects from the attach function (and detach function). Inside of Game.java, Lines 38-41 when a game is finished playing (i.e. a winner is set), the game object notifies all of its currently attached observers that the game is over, and passes them a handle to find out who won the game and who lost the game, which they will use as described above.

Question 4

The GameSimulator interface provided me with the ability to implement different types of game simulations without having to change any of the code in Bracket.java. Bracket.java takes in a GameSimulator in its constructor (Line 14), and uses it to simulate the games in the bracket (Line 77). Bracket does not have access nor does it need to know how the games are being simulated, it just gets passed in a GameSimulator and simulates it with it. CoinFlipSimulator, PennAlwaysWinsSimulator, FavoriteWinsSimulator, and EloSimulator are all implementations of simulators, which are be passed into Bracket constructors based on how the user chooses to simulate the tournament. The User interface finds the users choice, and passes a number to the DataController, which uses that number to decide which GameSimulator to instantiate and

to pass to the Bracket object prior to simulation (DataController Lines 51-64). This strategy pattern allows for the same code in Bracket to be shared among all simulation types, and allows to add or take away any number of simulation styles without having to change any of the code in Bracket.java. I was able to add the option of PennAlwaysWinsSimulator by only changing a few lines of code in the DataController, and not having to change the Bracket implementation at all.

Question 5

Compare this program to previous work you've coded before we covered design and design patterns (such as HW1). Do you feel that this code is more maintainable? Why or why not? Be completely honest here, you can say you felt it was a waste of time. But also consider your time spent not just developing, but testing and debugging as well.

This code was a lot more stable and easy to adapt than the code in HW1. Once the framework was set up to have an abtract file reader, it was easy to implement it using a CSV and using a JSON, without having to change the way the other layers interact with those classes. Similarly, it was easy to create new simulation techniques given that the GameSimulator interface was implemented, since the other classes didn't need to know how the simulation occurred. This code was also a lot easier to test – the functionally independent parts made it easy to localize bugs. For example, once I knew that the CSV reader was working completely, and then when using the JSON reader the code broke, it was obvious that the bug was local to EloJSONReader, since I was guaranteed that the only changed code occurred there. Similarly, the logging was all handled by one Singleton and was a lot less work to have all of it localized to one class, and not have to deal with opening and closing a file in every single class like I did in HW1.