Pei Lun Hung
CIS 350 001
March 27th, 2018
Homework 4 – Write-up

1. Presentation Layer: Main.java, CommandLineUserInterface.java
   Controller Layer: ControllerTier.java, Bracket.java, Observer.java, Subject.java, SimulationStrategy.java, CoinFlipStrategy.java, EloStrategy.java, FavoriteWinsStrategy.java
   Data Layer: DataTier.java, GameList.java, TeamEloStore.java, TeamEloReaderFactory.java, TeamEloReader.java, TeamEloCSVReader.java, TeamEloJSONReader.java

2. I use a concrete factory. In **DataTier.java**, within the method *makeTeamEloStore()*, we determine the file type of the Elo file given. This method then makes a new **TeamEloStore.java** instance; and, within TeamEloStore.java's *getTeamEloStore(String filetype)* method, we use **TeamEloReaderFactory.java** to generate a new **TeamEloReader**, which is either a **TeamEloCSVReader** or a **TeamEloJSONReader**.

3. I use the Observer pattern to simulate tournaments (see *simulateTournament(SimulationStrategy s)* in ControllerTier.java). More specifically, I used the Observer pattern in the following classes and methods:

• Bracket.java [Observer]

- [line 20] attachBracketToAllGames() – makes it so that the Bracket is watching/cognizant of the results of all Games (who wins?) so that teams can be appropriately set for future bracket games.
- [line 29] attachTeamsToGames() – makes it so that all Teams are notified when any Game's winner is set so that, if appropriate, that Team can update its win/loss record
- [line 95] update(Subject s) – when the Subject notifies the Bracket, the Bracket appropriately sets a future game's match up
- [line 74] updateGame(Game g) – gets called from update(Subject s)
- [line 53] simulate() – calls attachBracketToAllGames() and attachTeamsToGames() and then simulates the tournament, game by game

• Game.java [Subject]

- [line 38] setWinner(SimulationStrategy s) – determines the winner based on some SimulationStrategy. Once the winner is set, this method calls notifyAllObservers()

• Team.java [Observer]

- [line 30] update(Game g) – if the winner/loser of the Game is this Team, then the number of wins/losses is updated appropriately
- [line 38] update(Subject s) – calls update(Game g)

4. I used the Strategy pattern to determine winners of games (i.e. whether to use coin flips, Elo randomization, or favoritism to determine game winners in the tournament).

• ControllerTier.java

- [line 48] simulateTournament(SimulationStrategy s) – tells Bracket how to determine winners [line 49]
- [line 26] simulateGame(String teamA, String teamB) – uses the EloStrategy to simulate a single game [lines 31, 36]

• Bracket.java

    - [lines 8, 11, 57] simulate() – in this method, we go through all the Games and set their winners [lines 56 – 58; specifically, see line 58: “g.setWinner(strategy)”], where each game determines its winner according to this strategy

- Game.java
    - [line 38] setWinner(SimulationStrategy s) – sets winner of game according to the given strategy [line 39]

- SimulationStrategy.java: interface; the three classes below implement this interface, which requires implementing classes to implement determineWinner().
    - CoinFlipStrategy.java
        - [line 6] determineWinner(String teamA, String teamB, double eloA, double eloB) – ignores passed in elos and randomly determines winner
    - EloStrategy.java
        - [line 7] determineWinner(…) – winner is randomly determined based on Elos
    - FavoriteWinsStrategy.java
        - [line 6] determineWinner(…) – team with higher Elo wins; otherwise, winner is randomly determined

The user determines how they want to simulate brackets, and I wanted a way to determine this at runtime. The underlying behavior is the same—we want to determine the winner of games in a tournament—and the only thing that’s different is *how* (what strategy is used) that’s done. Each of the different winner-determining-methods is just a different way/methodology to do the same thing, which is why I decided to use the Strategy pattern here, as this is similar to the example in lecture where we used different payment strategies (different ways to do the same thing; i.e., paying). This is why I decided to create a SimulationStrategy interface and three different implementations. The ControllerTier, Bracket, and Game then use different strategies as determined by the user to simulate option 2.

5. Yes, this code is more maintainable. It’s easier to <u>analyze</u> because of the organization of the code into three tiers. That is, because of the separation of functions (presentation, control, and data store), it’s easier to understand different aspects of the program (the user interface vs. the code logic vs. the parsing and storage). It’s also easier to make modifications (<u>changeability</u> is improved): as an example, I can easily incorporate reading a new data format to store Elo scores by adding another type of TeamEloReader for the TeamEloReaderFactory to generate (i.e. I could easily add in a .tsv parser, for example; the code is easy to change because of its modularity). The software is also more <u>stable</u>, as changes don’t propagate as far: I can modify EloStrategy.java, for example, and not worry about affecting other strategies at all. Finally, the software is more testable by virtue of the modular separation of tiers: it’s much easier to test the logic of the software because once the UI is done, I don’t need to worry about it, and if something is wrong, I can pinpoint it to the ControllerTier if DataTier is also functioning correctly.