**11/10/2017**

**Head First Design Patterns**

**Project 3 Report**

***Braeden Brettin, Matthew Deremer, and Luke Pace***

**Table of Contents**

[Table of Figures 3](#_Toc497743446)

[Observer Pattern 4](#_Toc497743447)

[References 9](#_Toc497743448)

# **Table of Figures**

[Figure 1. Subject-Observer Relationship 4](#_Toc497745222)

# **Observer Pattern**

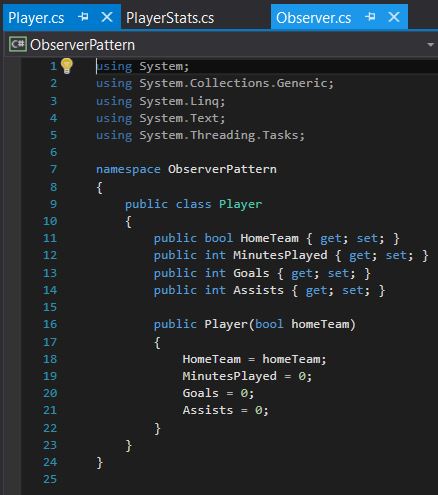
The Observer pattern is one of the most widely used patterns in all of software development. Using it, observers can request information from subjects at any time, thereby promoting the idea of loosely-coupled objects. Like a newspaper subscription service, observers can decide if they want to remain subscribed to the subject, the newspaper in this example. Every week, the subject updates the observer with a brand-new newspaper. The Observer pattern is built into the JDK; however, we would like to create our own Observer pattern in our chosen language of C#. In the following project, we will create our own Observer class and interface and Subject class and interface. We will then incorporate them together in a one-to-many relationship to implement the Observer pattern. A general outline of the relationship between the Subject and Observer is shown in Figure 1, below.



Figure 1. Subject-Observer Relationship

In our example, the Subject class will contain data for each player in a soccer match. This data will include number of minutes played, goals, and assists. Every minute of match time, an Observer class will request certain information from the Subject class. The two Observer classes in this project, a statistical analysis for all players and an overview of the current stats for each team, will then be displayed.

To begin, we created a skeleton class for out Subject and Observer interfaces, and we set up a test suite that outlines the various methods and variables that we believe this test will utilize. We will first test the methods contained in the MatchData class, which implements the Subject interface. We first created a Player object, as shown in Figure 2, below, which contains the number of minutes played, goals, and assists for a player.



The initial Subject interface was as shown in Figure 3, below.

FIGURE 2

We then created a class that will implement this Subject interface, the MatchData class, as shown in Figure 4, below. This MatchData class contains a list of Player objects, methods for registering, removing, and notifying observers, and a method for returning the list of Player objects.

FIGURE 3

The initial Observer interface was as shown in Figure 5, below.

FIGURE 4

We next created a PlayerStats class that will implement this Observer interface, as shown in Figure 6, below. This class will provide a statistical analysis of Player data, such as the Player with the most minutes played, the top scorer, and the Player with the most assists.

FIGURE 5

Finally, we created a TeamStats class that will implement the Observer interface, as shown in Figure 7, below. This class will simply display the score of the match.

FIGURE 6

With the skeleton structure of these initial classes set up, we created a test suite for each class that implements an interface. These initial tests were set up and run so that they would purposefully fail, as shown in Figures 8 through 10, below.

FIGURES 7-9

The next step in test driven development is to add code to our classes being tested so these tests will now pass successfully. After revising the classes being tested, we re-ran the test suite, producing the successful output shown in Figures 11 through 13, below.

FIGURES 11-13

With these tests now running successfully, we no longer must amend this test suite and can simply re-run it every time we refactor the source code. We now need to add methods for updating the minutes played by each player and recording when a goal is scored and/or an assist is recorded. The classes were amended to include all information needed for our Observers, as shown in Figures XXX below.

FIGURES XXX

To confirm that none of our refactoring affected the functionality of the code, we ran our test suite again, producing the successful output shown in Figures XXX, below.

FIGURES XXX

As seen from this project, the Observer pattern is a widely-used, useful pattern for exchanging data between subjects and observers. It is exceptionally handy in the case of performing a statistical analysis on a sporting event, as shown in this project. Through loosely coupling the subjects and observers, developers can quickly and efficiently pass a wealth of information between multiple classes.

1. For each design pattern, explain in detail what your alternative design would be, and what problem with that design the use of this pattern solves (if it does).
2. Explain in detail what is good about your Design Pattern (DP)-based design, and what is problematic about your DP-based design.

# **References**

Freeman, E., & Freeman, E. (2005). Head First Design Patterns. O'Reilly.

Wagner, B. (n.d.). Interfaces (C# Programming Guide). Retrieved November 09, 2017, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/

How to: Create and Run a Unit Test. (n.d.). Retrieved November 2, 2017, from https://msdn.microsoft.com/en-us/library/ms182524(v=vs.90).aspx

https://stackoverflow.com/questions/26593348/get-max-value-in-a-list-of-points