**11/12/2017**

**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 20](#_Toc497743448)

# **Table of Figures**

[Figure 1. Subject-Observer Relationship 4](#_Toc498254019)

[Figure 2. Initial Player Class 5](#_Toc498254020)

[Figure 3. Initial Subject Class 6](#_Toc498254021)

[Figure 4. Initial MatchData Class 7](#_Toc498254022)

[Figure 5. Initial Observer Class 7](#_Toc498254023)

[Figure 6. Initial PlayerStats Class 8](#_Toc498254024)

[Figure 7. Initial TeamStats Class 8](#_Toc498254025)

[Figure 8. Initial Failed Tests 9](#_Toc498254026)

[Figure 9. Initial Failed Tests 10](#_Toc498254027)

[Figure 10. Initial Failed Tests 10](#_Toc498254028)

[Figure 11. Initial Passed Tests 11](#_Toc498254029)

[Figure 12. Initial Passed Tests 12](#_Toc498254030)

[Figure 13. Initial Passed Tests 13](#_Toc498254031)

[Figure 14. Revised Player Class 14](#_Toc498254032)

[Figure 15. Revised Subject Class 14](#_Toc498254033)

[Figure 16. Revised MatchData Class 15](#_Toc498254034)

[Figure 17. Revised MatchData Class 16](#_Toc498254035)

[Figure 18. Revised Observer Class 16](#_Toc498254036)

[Figure 19. Revised PlayerStats Class 17](#_Toc498254037)

[Figure 20. Revised TeamStats Class 18](#_Toc498254038)

[Figure 21. Passed Tests 19](#_Toc498254039)

# **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#. Our Observer pattern will have the same functionality as Java’s built -in Observer pattern, allowing observers to be registered and removed from the subject and allowing the subject to notify the observers. 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.

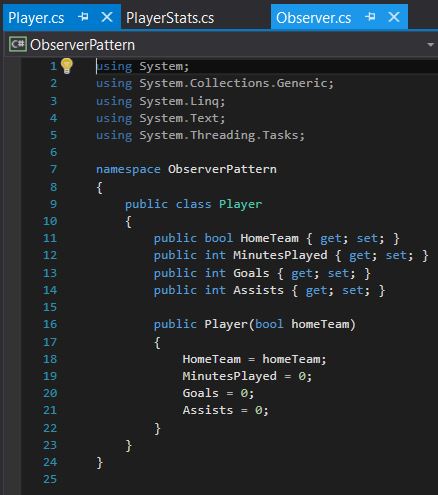


Figure 2. Initial Player Class

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

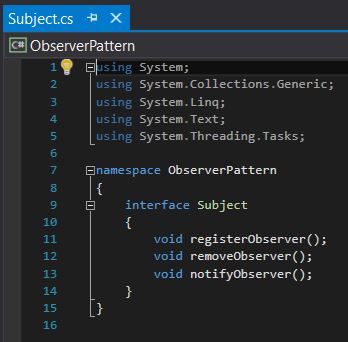


Figure 3. Initial Subject Class

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.

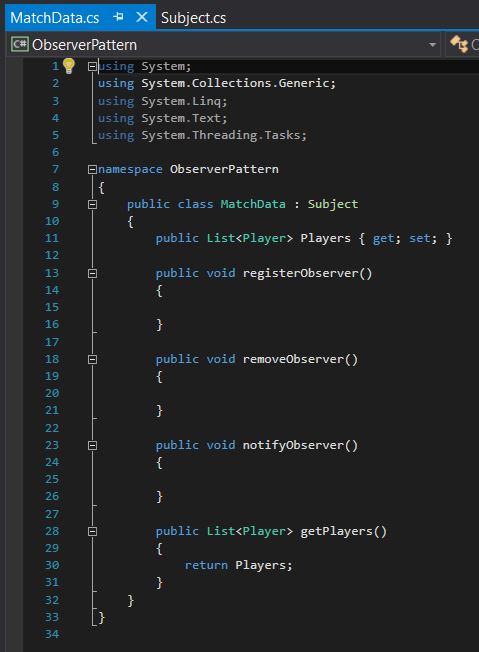


Figure 4. Initial MatchData Class

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

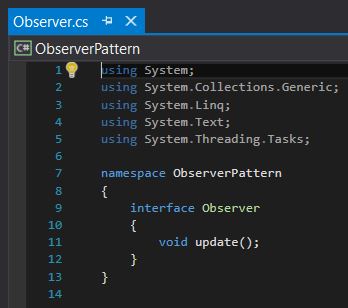


Figure 5. Initial Observer Class

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.

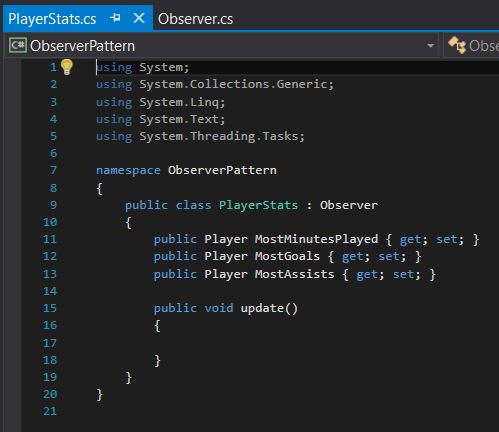


Figure 6. Initial PlayerStats Class

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.

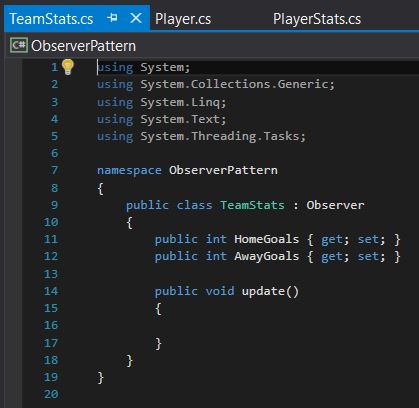


Figure 7. Initial TeamStats Class

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.

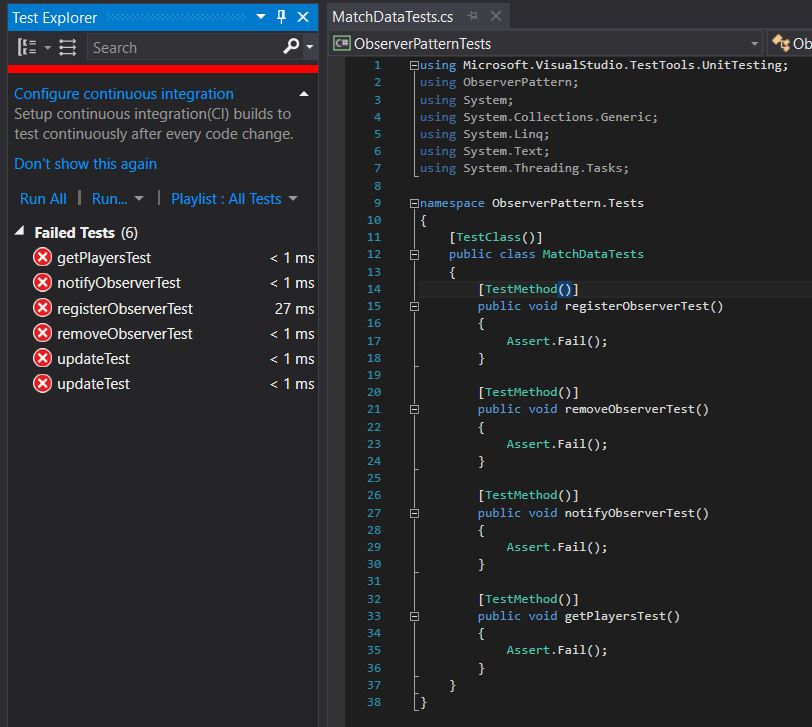


Figure 8. Initial Failed Tests

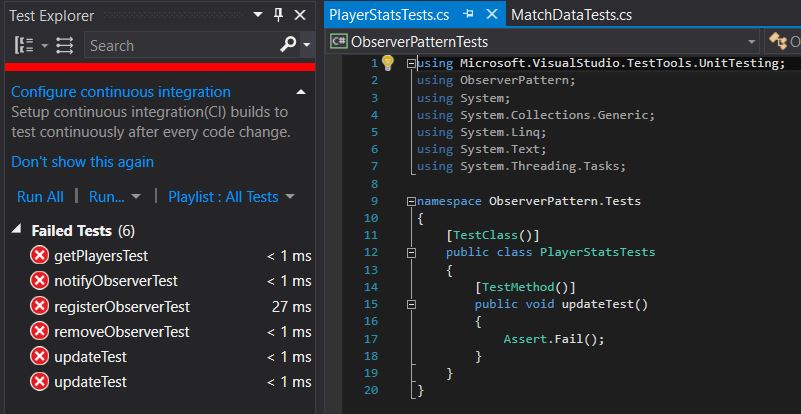


Figure 9. Initial Failed Tests

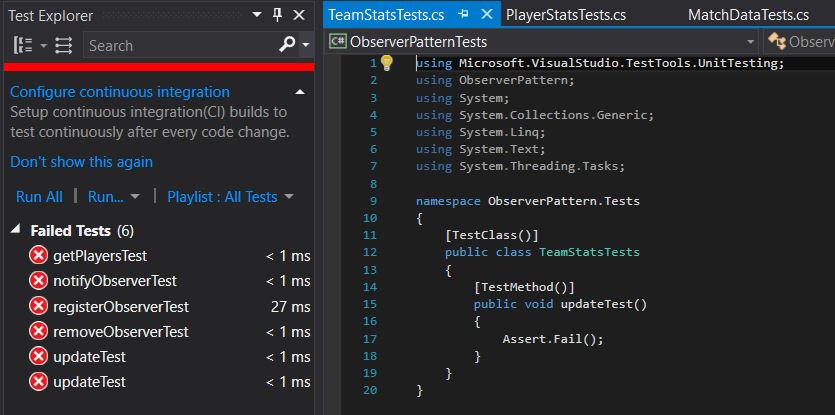


Figure 10. Initial Failed Tests

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.

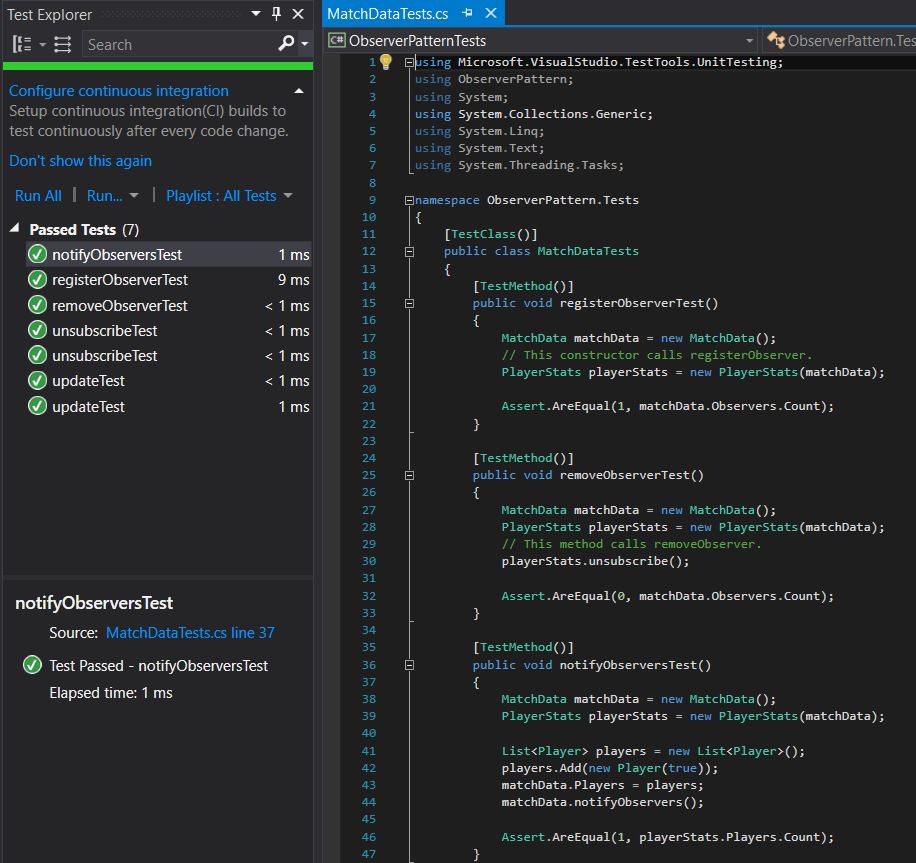


Figure 11. Initial Passed Tests

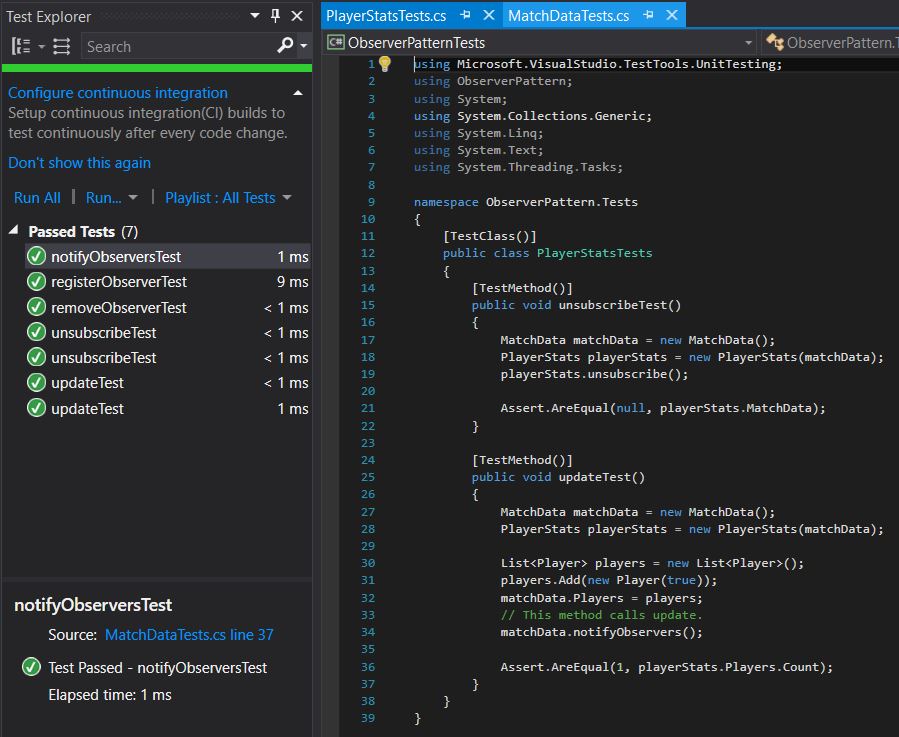


Figure 12. Initial Passed Tests

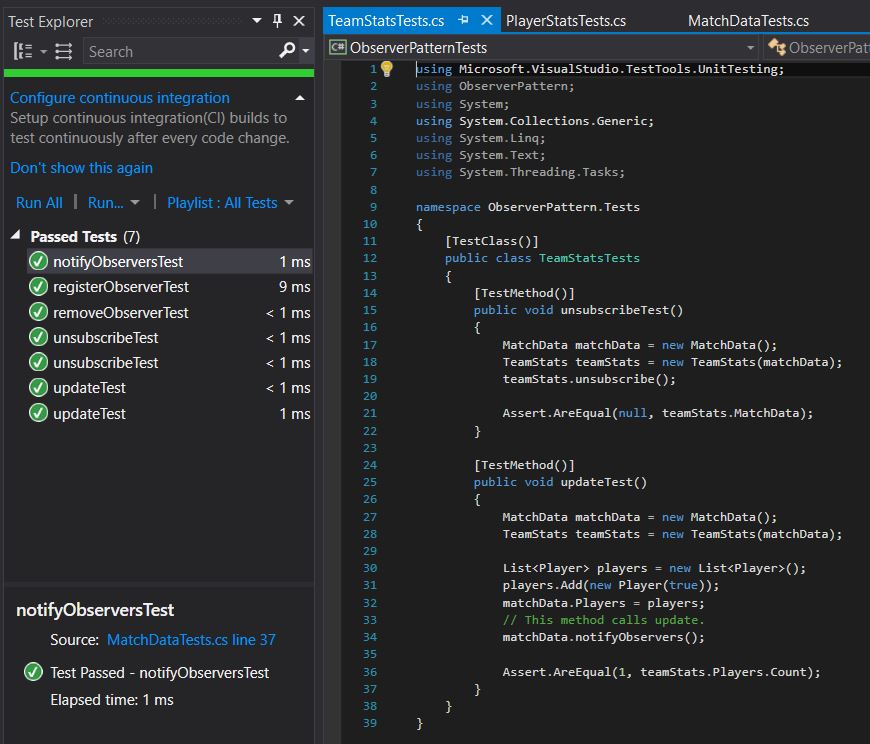


Figure 13. Initial Passed Tests

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 14 through 20 below.

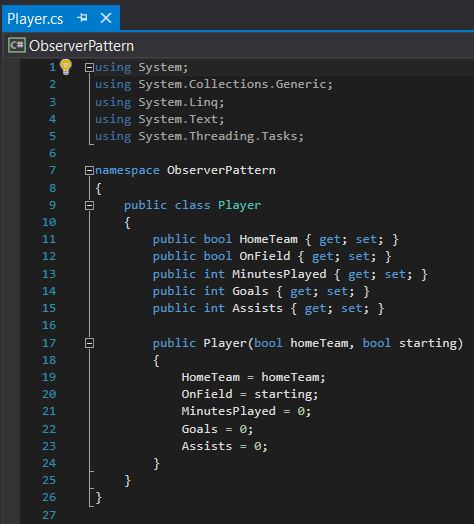


Figure 14. Revised Player Class

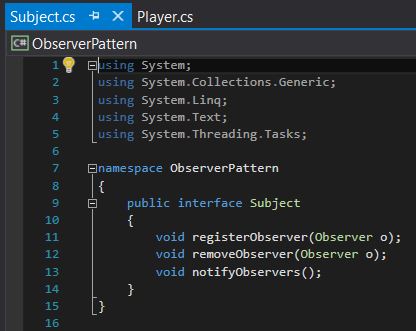


Figure 15. Revised Subject Class

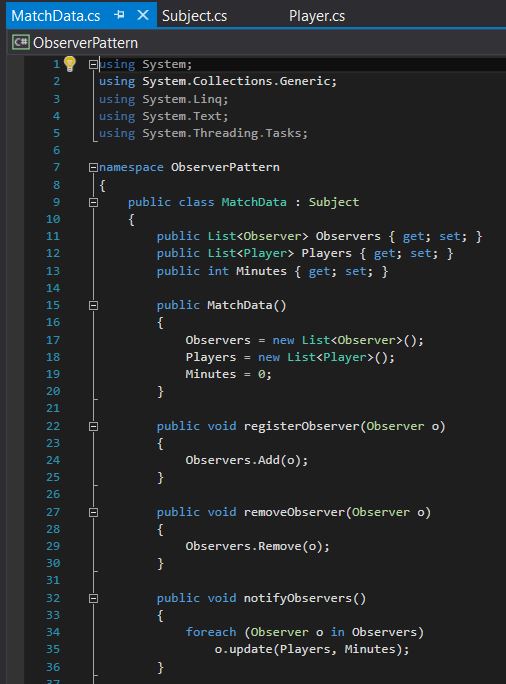


Figure 16. Revised MatchData Class

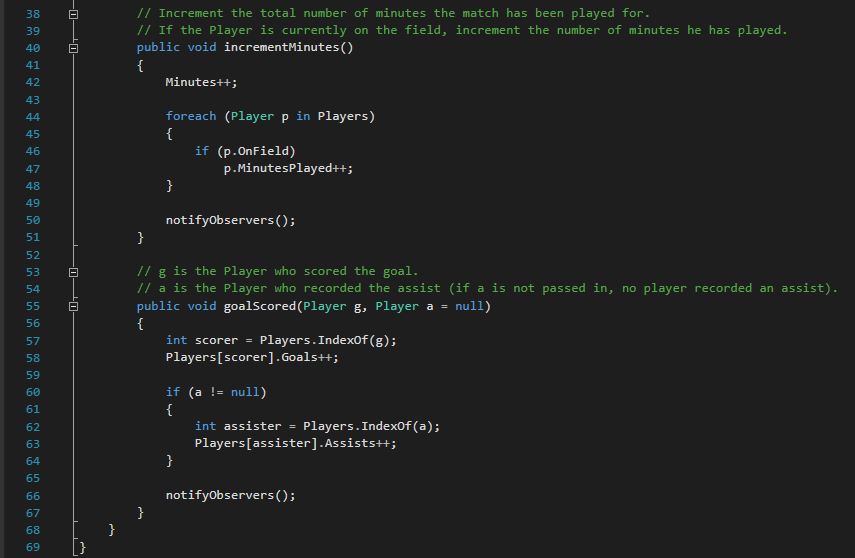


Figure 17. Revised MatchData Class

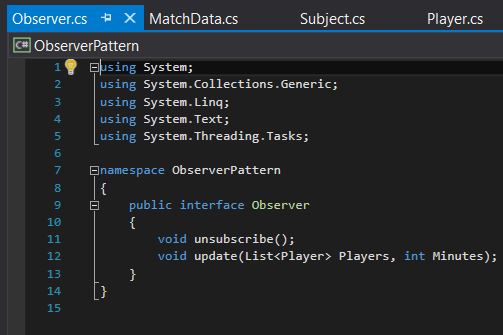


Figure 18. Revised Observer Class

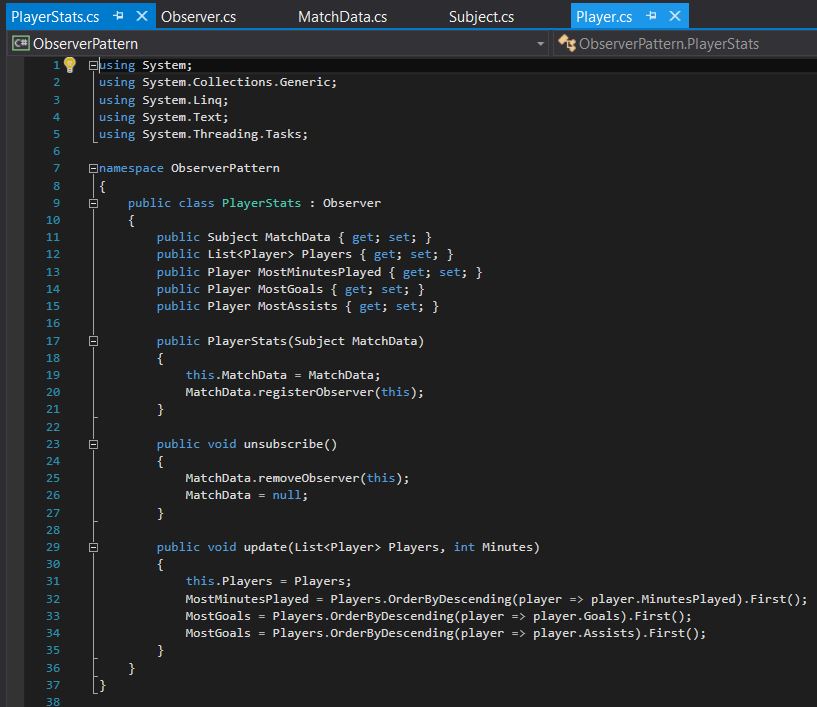


Figure 19. Revised PlayerStats Class

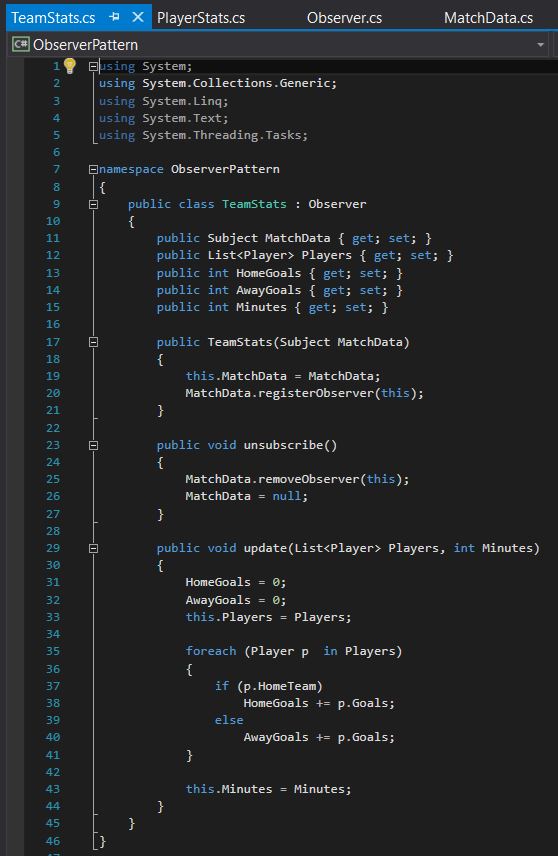


Figure 20. Revised TeamStats Class

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 Figure 21, below.

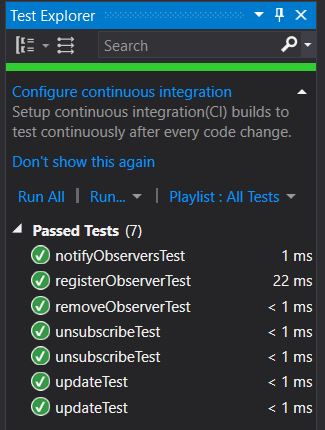


Figure 21. Passed Tests

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.

Unfortunately, our Observer pattern requires the subject to notify the Observers with a specific set of information, and the Observer cannot specify which information they would like to receive. In a more in-depth project, we would give the Observer the ability to request specific information from the Subject, reducing the amount of overhead required.

# **References**

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

Wagner, B. (n.d.). Interfaces (C# Programming Guide). Retrieved November 9, 2017, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/

Get Max Value in a List of Points. (n.d.). Retrieved November 5, 2017, from https://stackoverflow.com/questions/26593348/get-max-value-in-a-list-of-points

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