

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

11/17/2017

Head First Design Patterns

Project 3 Report

Several thin, curved lines in dark blue and light grey extending from the bottom of the vertical bar towards the center of the page.

*Braeden Brettin,
Matthew Deremer,
and Luke Pace*

Table of Contents

Table of Figures	3
Observer Pattern.....	8
Strategy Pattern	25
Decorator Pattern	37
Factory Pattern	50
References	59

Table of Figures

Figure 1. Subject-Observer Relationship.....	8
Figure 2. Initial Player Class.....	9
Figure 3. Initial Subject Class	10
Figure 4. Initial MatchData Class	11
Figure 5. Initial Observer Class	11
Figure 6. Initial PlayerStats Class.....	12
Figure 7. Initial TeamStats Class	13
Figure 8. Initial Failed Tests	14
Figure 9. Initial Failed Tests	15
Figure 10. Initial Failed Tests	15
Figure 11. Initial Passed Tests	16
Figure 12. Initial Passed Tests	17
Figure 13. Initial Passed Tests	18
Figure 14. Revised Player Class	19
Figure 15. Revised Subject Class.....	19
Figure 16. Revised MatchData Class.....	20
Figure 17. Revised MatchData Class.....	21

Figure 18. Revised Observer Class	21
Figure 19. Revised PlayerStats Class.....	22
Figure 20. Revised TeamStats Class.....	23
Figure 21. Passed Tests.....	24
Figure 22: Strategy Pattern Outline	25
Figure 23: Non-pattern Multiple Method Calculator.....	27
Figure 24: Non-pattern Multiple Method Calculator.....	27
Figure 25: Non-pattern Single Method Calculator	28
Figure 26: Non-pattern Single Method Calculator	28
Figure 27: Interface Class	29
Figure 28: AverageByMean Concrete Class.....	30
Figure 29: AverageByMedian Concrete Class	30
Figure 30: AverageByMode Concrete Class	31
Figure 31: AverageByGeometric Concrete Class.....	31
Figure 32: AverageByHarmonic Concrete Class.....	32
Figure 33: Calculator Class.....	32
Figure 34: Test Suite List.....	33
Figure 35: Results Close Enough Function	33

Figure 36: Test AverageByMean.....	34
Figure 37: Test AverageByMedian.....	34
Figure 38: Test AverageByMode.....	34
Figure 39: Test AverageByGeometric	35
Figure 40: Test AverageByHarmonic	35
Figure 41: Calculator Passed Test.....	36
Figure 42: Overpopulated Subclass Map	37
Figure 43: Decorator Implementation Outline.....	38
Figure 44: Initial IFood Interface.....	39
Figure 45: Cake and Pancake Classes.....	40
Figure 46: Cake with Toppings Classes.....	41
Figure 47: Cake with Toppings Classes (cont.)	41
Figure 48: Pancakes with Toppings Classes	42
Figure 49: Pancakes with Toppings Classes (cont.)	42
Figure 50: Initialize Cake and Pancake Objects	43
Figure 51: Cake and Pancake Objects Output	43
Figure 52: Generate Cake and Pancakes with Toppings Objects	44
Figure 53: Cake and Pancakes with Toppings Output.....	44

Figure 54: Initial Decorator	45
Figure 55: Concrete Decorator, Scent.....	45
Figure 56: Concrete Decorator, Strawberry.....	46
Figure 57: Concrete Decorator, Cream.....	46
Figure 58: Generate Cake with Decorators Object.....	47
Figure 59: Cake with Decorators Output.....	47
Figure 60: Generate Pancakes with Decorators Object	48
Figure 61: Pancakes with Decorators Output	48
Figure 62: Factory Pattern Example	51
Figure 63: IMajor Interface.....	52
Figure 64: CompE Class	53
Figure 65: ElecE Class.....	53
Figure 66: InduE Class.....	54
Figure 67: SoftE Class	54
Figure 68: EngineerFactory	55
Figure 69: Client Code, Enroll Students	56
Figure 70: Enroll Students, Output	57
Figure 71: Factory Pattern Test.....	57

Figure 72: Passed Factory Pattern Test.....	58
---	----

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.

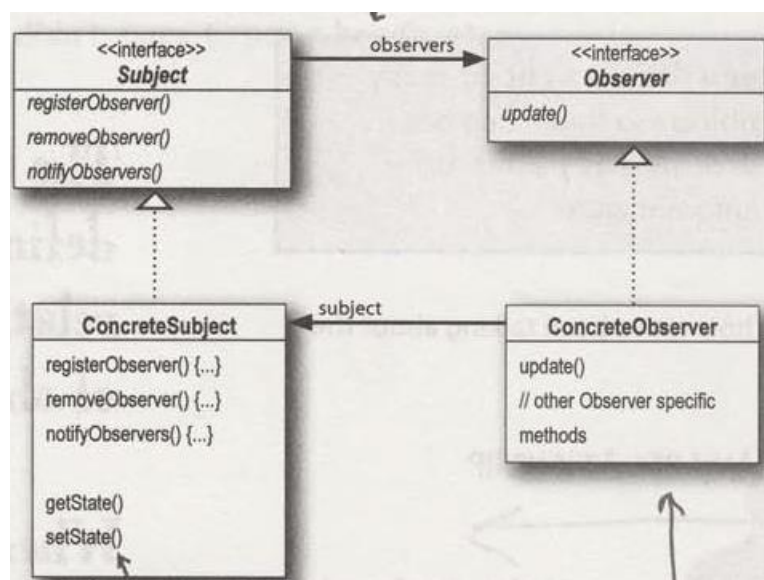
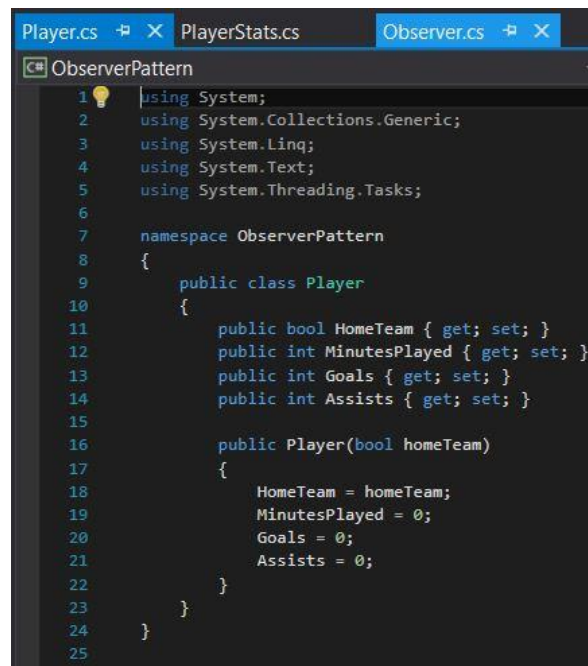


Figure 1. Subject-Observer Relationship

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.

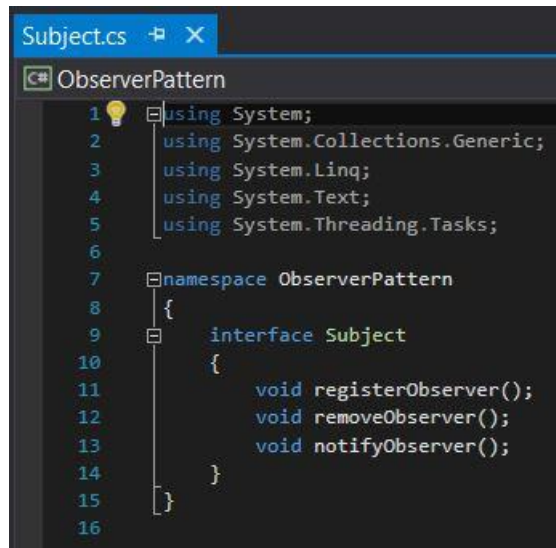
To begin, we created a skeleton class for our Subject and Observer interfaces, and we set up a test suite that outlines the various methods and variables that we believe this test will utilize. We will first test the methods contained in the MatchData class, which implements the Subject interface. We 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 image shows a screenshot of a code editor with three tabs at the top: 'Player.cs', 'PlayerStats.cs', and 'Observer.cs'. The 'Observer.cs' tab is active, showing a file named 'ObserverPattern.cs'. The code is in C# and defines a 'Player' class within the 'ObserverPattern' namespace. The class has four public properties: 'HomeTeam' (bool), 'MinutesPlayed' (int), 'Goals' (int), and 'Assists' (int), each with get and set methods. A constructor 'Player(bool homeTeam)' initializes these properties: 'HomeTeam' to the passed value, and 'MinutesPlayed', 'Goals', and 'Assists' to 0. The code is numbered from 1 to 25 on the left margin.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class Player
10    {
11        public bool HomeTeam { get; set; }
12        public int MinutesPlayed { get; set; }
13        public int Goals { get; set; }
14        public int Assists { get; set; }
15
16        public Player(bool homeTeam)
17        {
18            HomeTeam = homeTeam;
19            MinutesPlayed = 0;
20            Goals = 0;
21            Assists = 0;
22        }
23    }
24 }
25
```

Figure 2. Initial Player Class

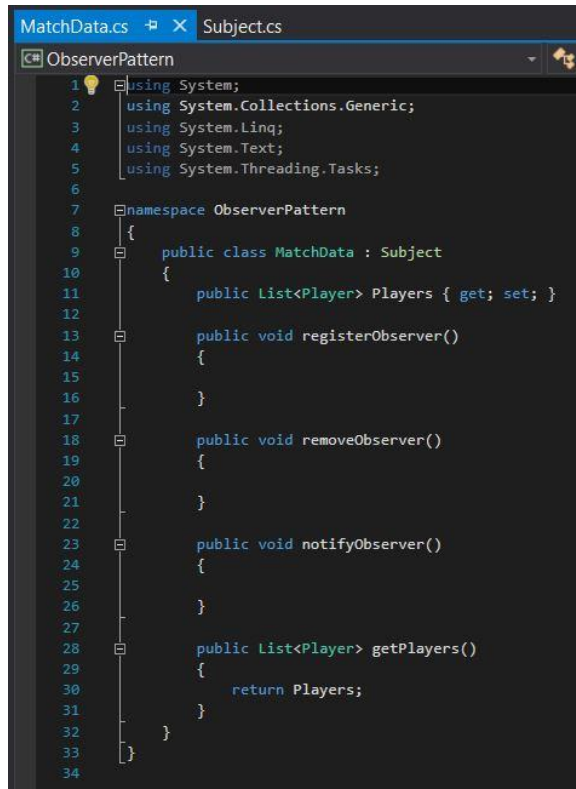
The initial Subject interface was as shown in Figure 3, below.

The image is a screenshot of a code editor window titled 'Subject.cs'. The editor shows the 'ObserverPattern' namespace containing an interface named 'Subject'. The interface has three methods: 'registerObserver()', 'removeObserver()', and 'notifyObserver()'. The code is written in C# and includes several 'using' statements at the top. The line numbers 1 through 16 are visible on the left side of the editor.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     interface Subject
10    {
11        void registerObserver();
12        void removeObserver();
13        void notifyObserver();
14    }
15 }
16
```

Figure 3. Initial Subject Class

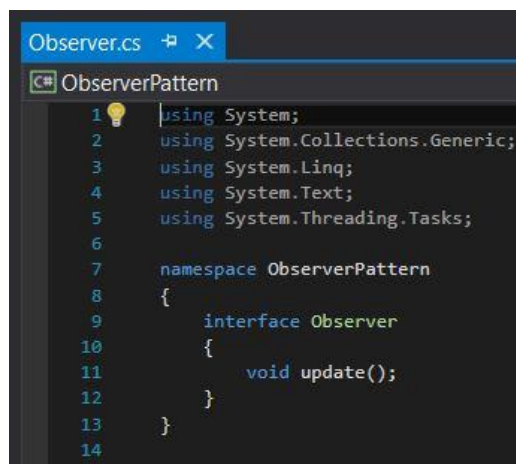
We then created a class that will implement this Subject interface, the MatchData class, as shown in Figure 4, below. This MatchData class contains a list of Player objects, methods for registering, removing, and notifying observers, and a method for returning the list of Player objects.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class MatchData : Subject
10     {
11         public List<Player> Players { get; set; }
12
13         public void registerObserver()
14         {
15
16         }
17
18         public void removeObserver()
19         {
20
21         }
22
23         public void notifyObserver()
24         {
25
26         }
27
28         public List<Player> getPlayers()
29         {
30             return Players;
31         }
32     }
33 }
34
```

Figure 4. Initial MatchData Class

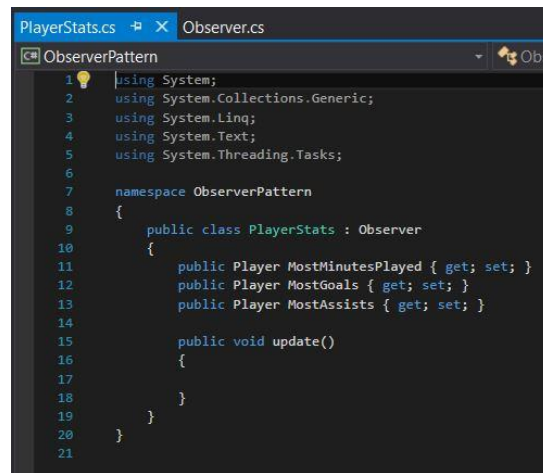
The initial Observer interface was as shown in Figure 5, below.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     interface Observer
10     {
11         void update();
12     }
13 }
14
```

Figure 5. Initial Observer Class

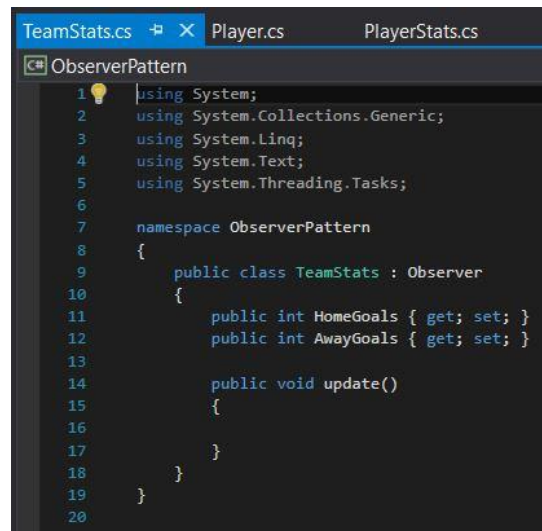
We next created a PlayerStats class that will implement this Observer interface, as shown in Figure 6, below. This class will provide a statistical analysis of Player data, such as the Player with the most minutes played, the top scorer, and the Player with the most assists.

The image shows a code editor with two tabs: 'PlayerStats.cs' and 'Observer.cs'. The 'PlayerStats.cs' tab is active, displaying the following C# code:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class PlayerStats : Observer
10     {
11         public Player MostMinutesPlayed { get; set; }
12         public Player MostGoals { get; set; }
13         public Player MostAssists { get; set; }
14
15         public void update()
16         {
17         }
18     }
19 }
20
21
```

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.

A screenshot of a code editor with three tabs: TeamStats.cs, Player.cs, and PlayerStats.cs. The TeamStats.cs tab is active, showing the following C# code:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class TeamStats : Observer
10     {
11         public int HomeGoals { get; set; }
12         public int AwayGoals { get; set; }
13
14         public void update()
15         {
16
17         }
18     }
19 }
20
```

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.

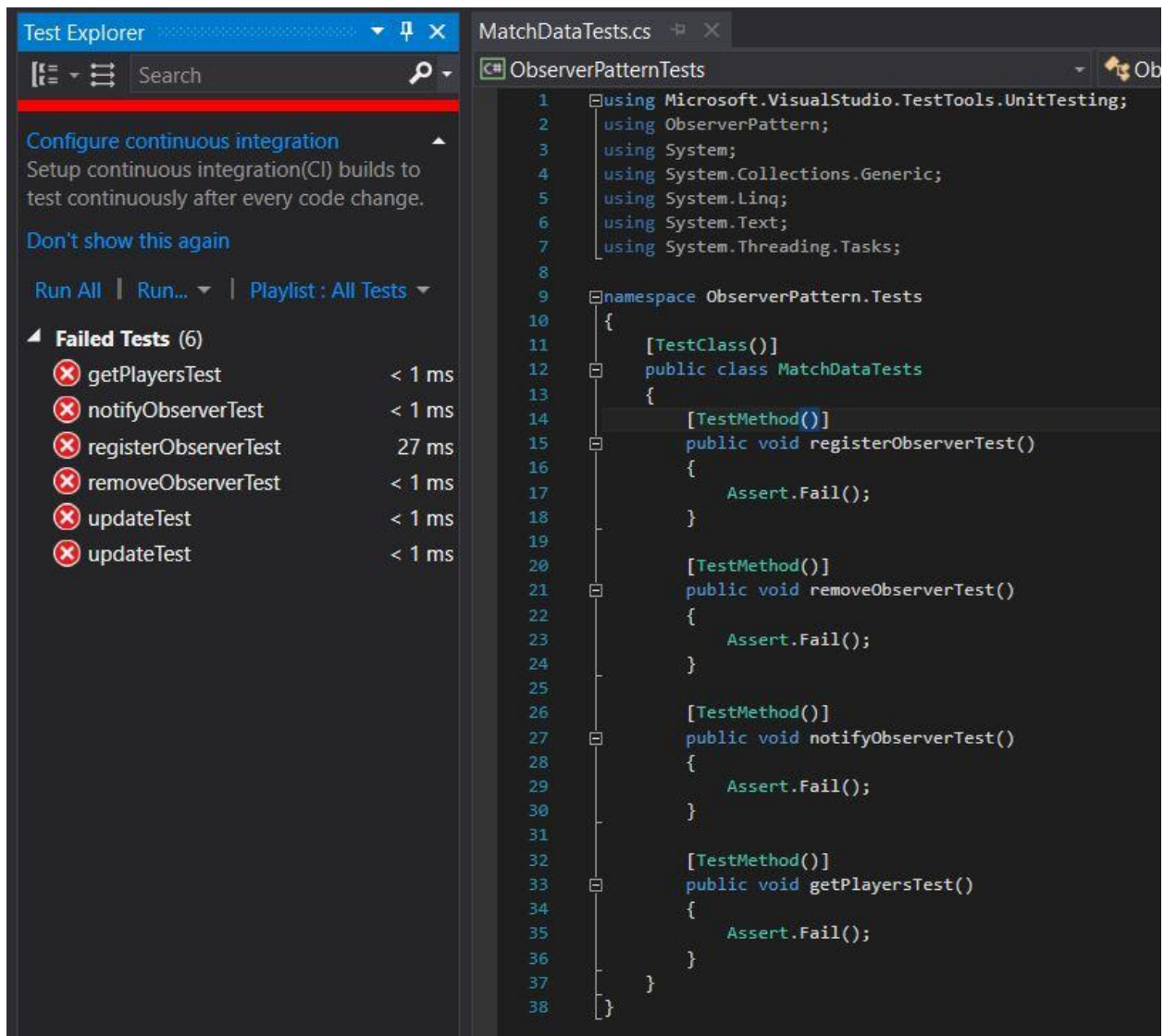


Figure 8. Initial Failed Tests

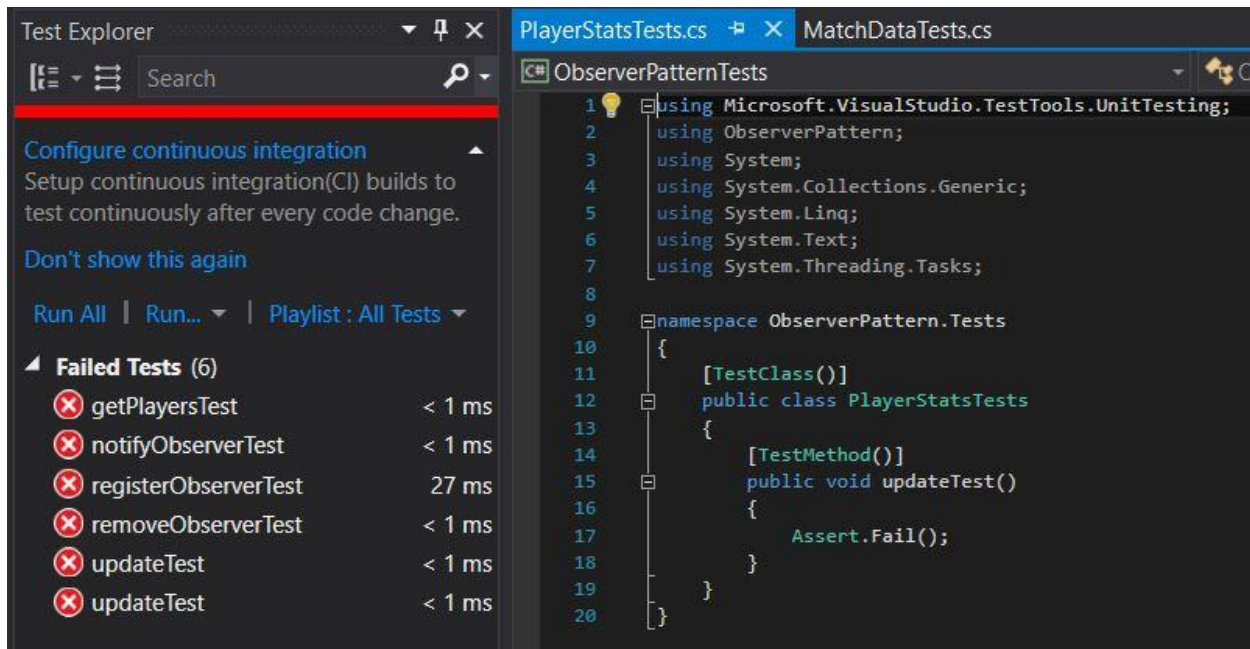


Figure 9. Initial Failed Tests

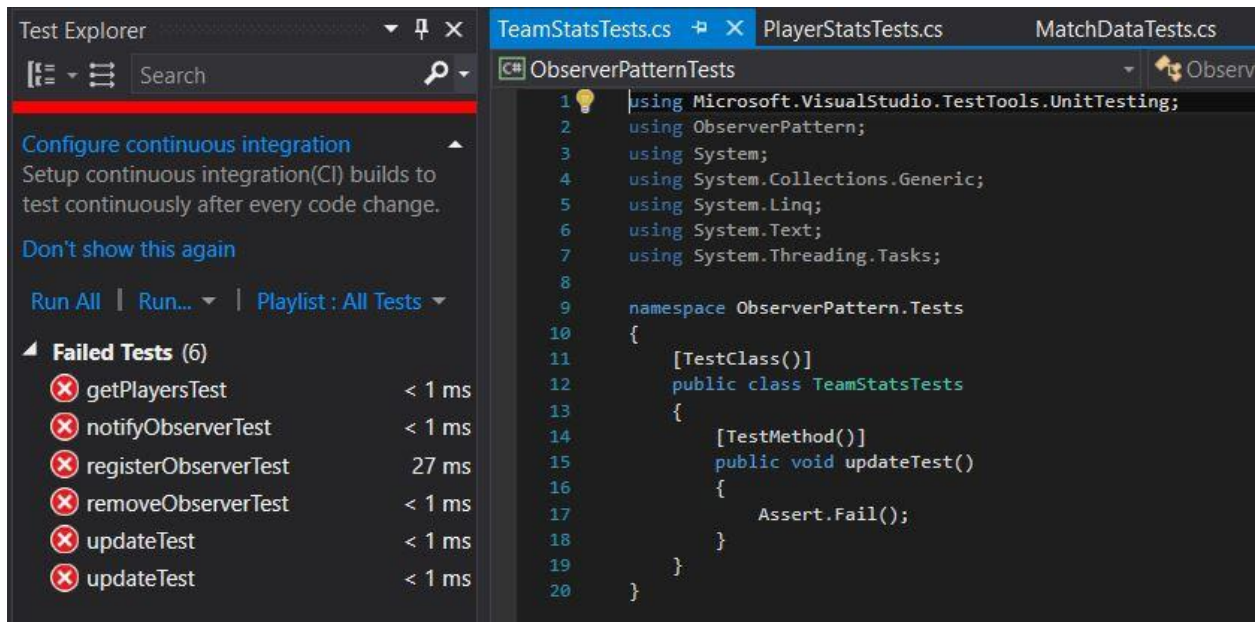


Figure 10. Initial Failed Tests

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.

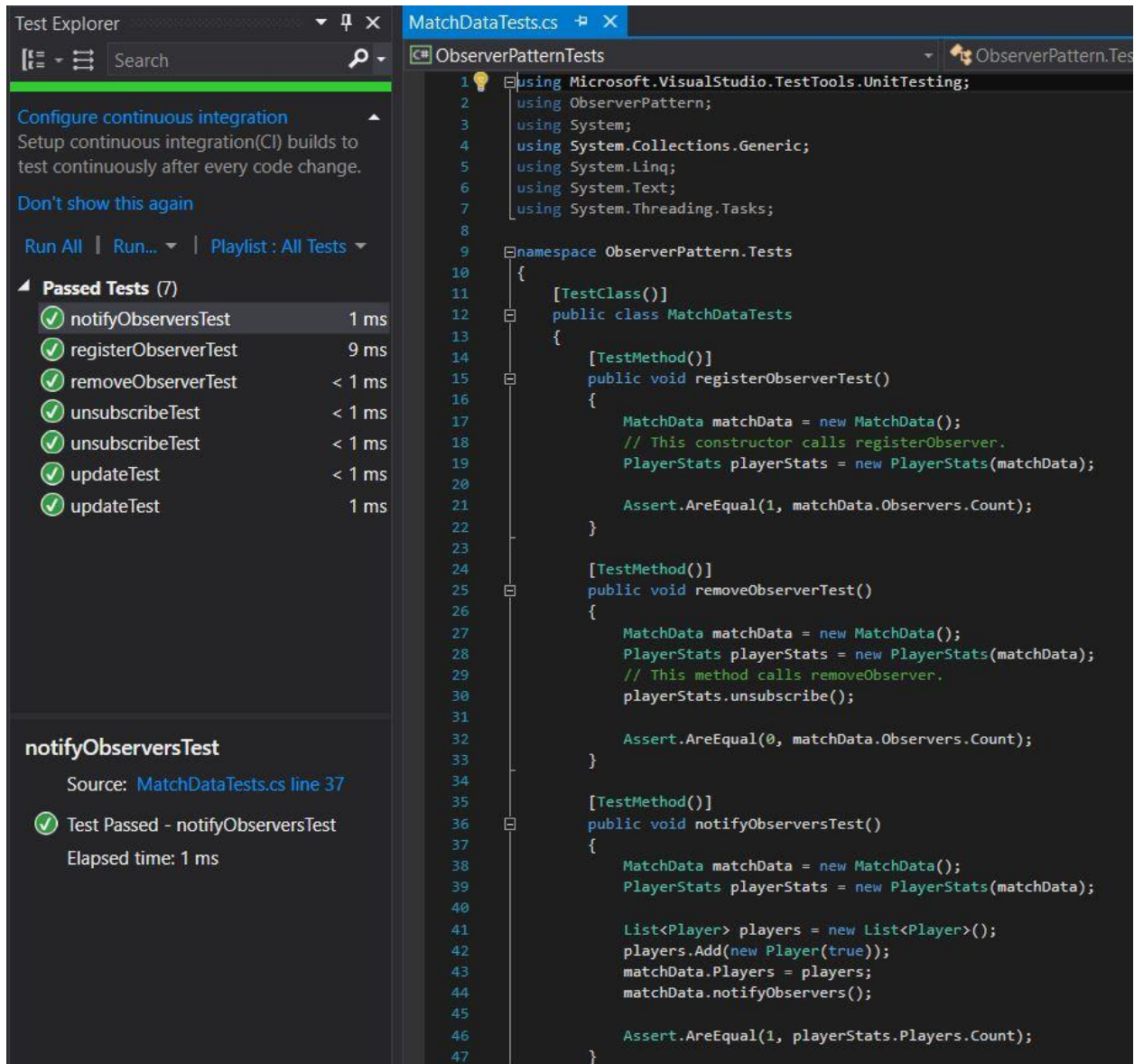


Figure 11. Initial Passed Tests

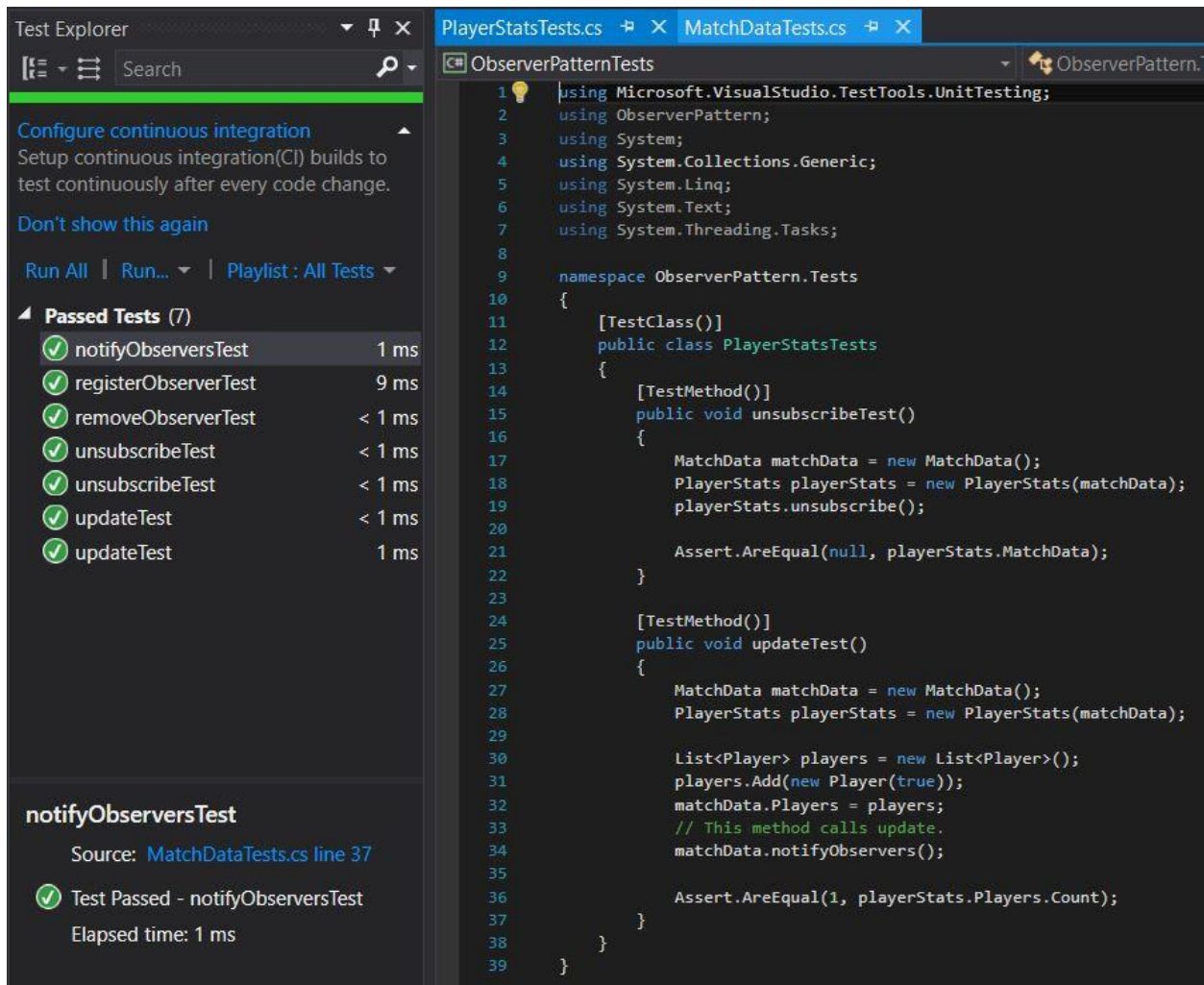


Figure 12. Initial Passed Tests

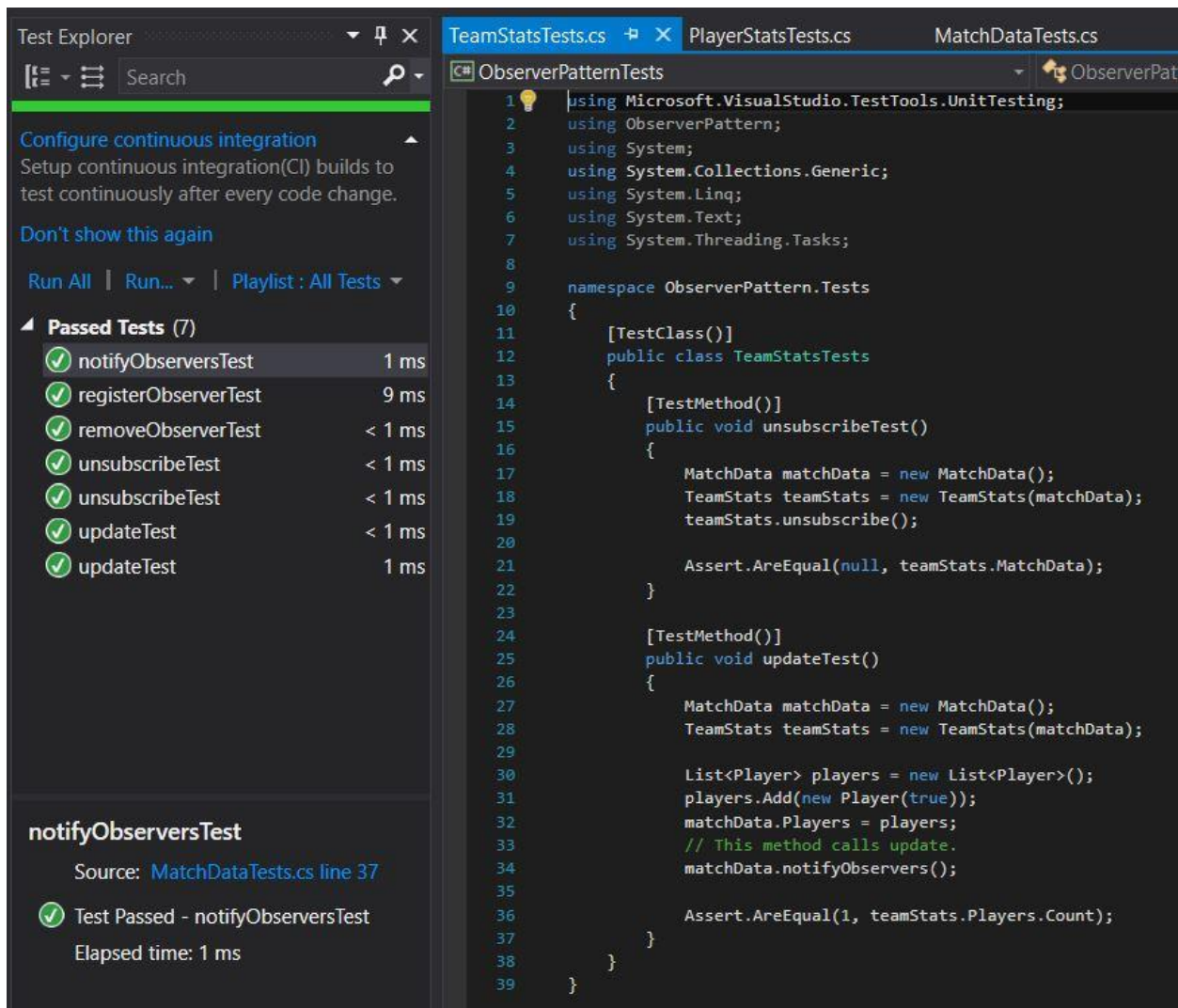


Figure 13. Initial Passed Tests

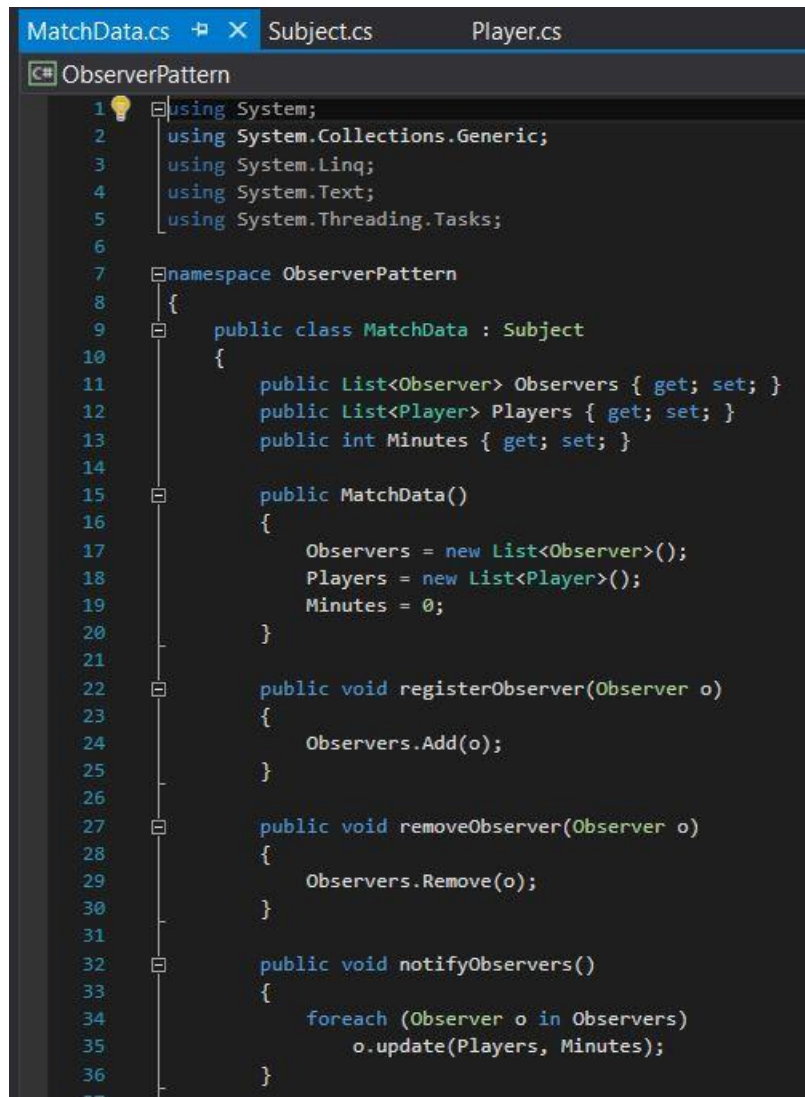
With these tests now running successfully, we no longer must amend this test suite and can simply re-run it every time we refactor the source code. We now need to add methods for updating the minutes played by each player and recording when a goal is scored and/or an assist is recorded. The classes were amended to include all information needed for our Observers, as shown in Figures 14 through 20 below.

```
Player.cs X
ObserverPattern
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class Player
10    {
11        public bool HomeTeam { get; set; }
12        public bool OnField { get; set; }
13        public int MinutesPlayed { get; set; }
14        public int Goals { get; set; }
15        public int Assists { get; set; }
16
17        public Player(bool homeTeam, bool starting)
18        {
19            HomeTeam = homeTeam;
20            OnField = starting;
21            MinutesPlayed = 0;
22            Goals = 0;
23            Assists = 0;
24        }
25    }
26 }
27
```

Figure 14. Revised Player Class

```
Subject.cs X Player.cs
ObserverPattern
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public interface Subject
10    {
11        void registerObserver(Observer o);
12        void removeObserver(Observer o);
13        void notifyObservers();
14    }
15 }
16
```

Figure 15. Revised Subject Class



```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ObserverPattern
8  {
9      public class MatchData : Subject
10     {
11         public List<Observer> Observers { get; set; }
12         public List<Player> Players { get; set; }
13         public int Minutes { get; set; }
14
15         public MatchData()
16         {
17             Observers = new List<Observer>();
18             Players = new List<Player>();
19             Minutes = 0;
20         }
21
22         public void registerObserver(Observer o)
23         {
24             Observers.Add(o);
25         }
26
27         public void removeObserver(Observer o)
28         {
29             Observers.Remove(o);
30         }
31
32         public void notifyObservers()
33         {
34             foreach (Observer o in Observers)
35                 o.update(Players, Minutes);
36         }
37     }
```

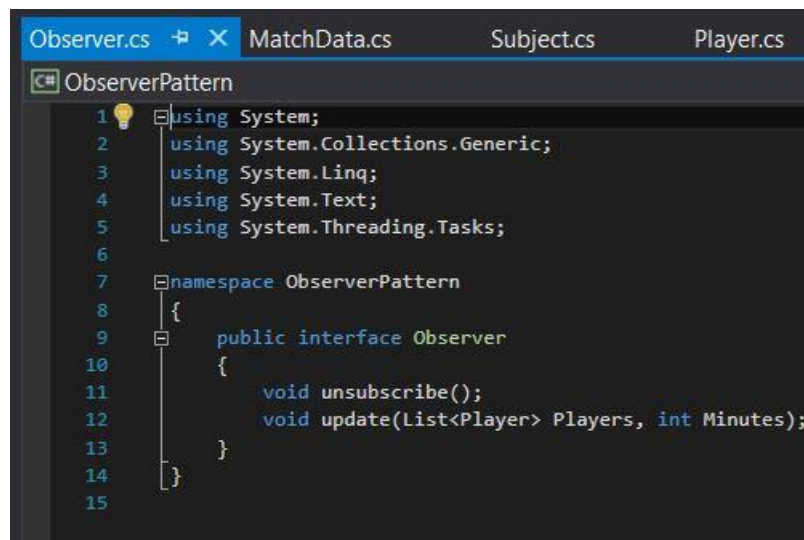
Figure 16. Revised MatchData Class

```

38 // Increment the total number of minutes the match has been played for.
39 // If the Player is currently on the field, increment the number of minutes he has played.
40 public void incrementMinutes()
41 {
42     Minutes++;
43
44     foreach (Player p in Players)
45     {
46         if (p.OnField)
47             p.MinutesPlayed++;
48     }
49
50     notifyObservers();
51 }
52
53 // g is the Player who scored the goal.
54 // a is the Player who recorded the assist (if a is not passed in, no player recorded an assist).
55 public void goalScored(Player g, Player a = null)
56 {
57     int scorer = Players.IndexOf(g);
58     Players[scorer].Goals++;
59
60     if (a != null)
61     {
62         int assister = Players.IndexOf(a);
63         Players[assister].Assists++;
64     }
65
66     notifyObservers();
67 }
68 }
69 }

```

Figure 17. Revised MatchData Class

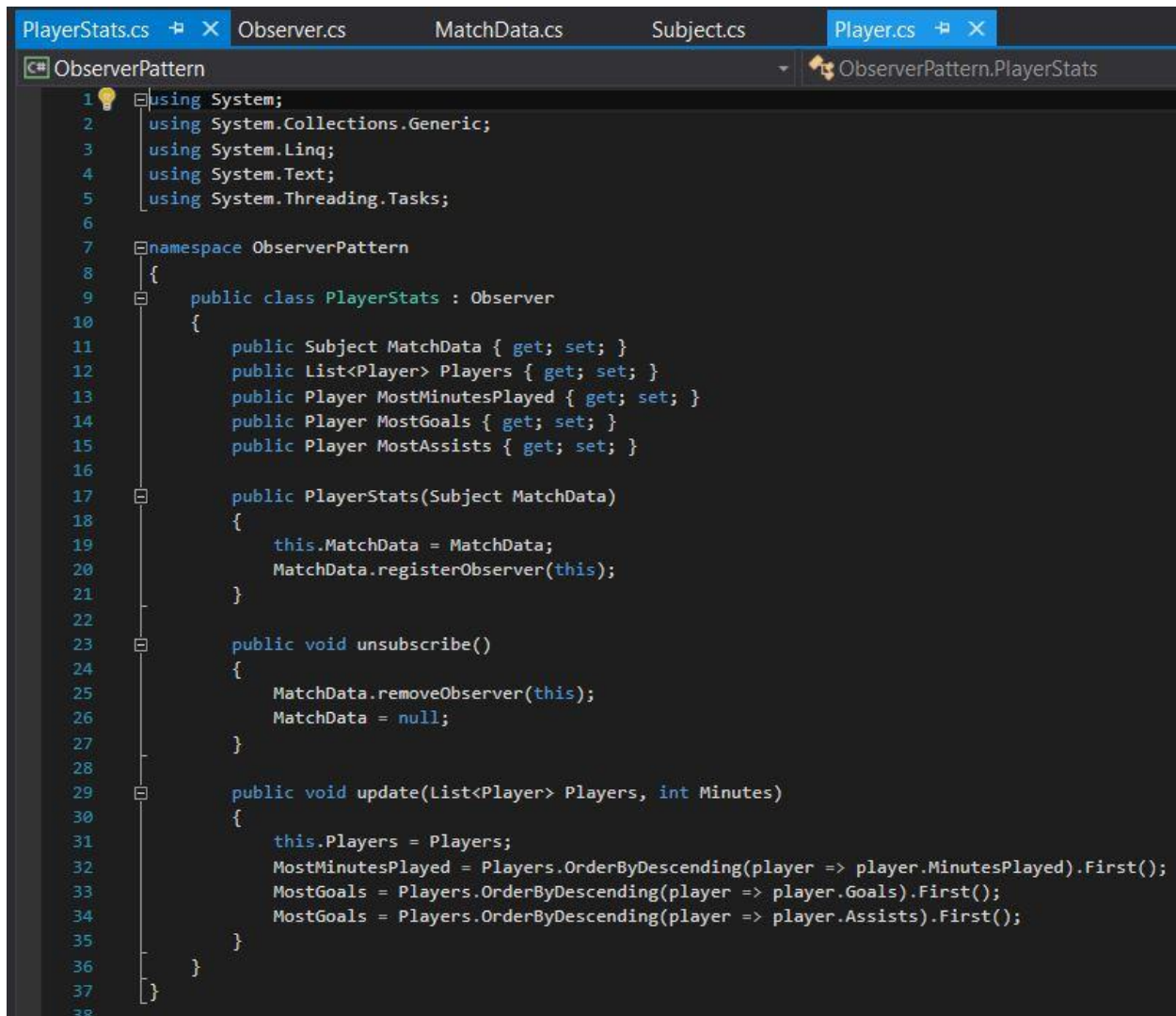


```

Observer.cs MatchData.cs Subject.cs Player.cs
ObserverPattern
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public interface Observer
10     {
11         void unsubscribe();
12         void update(List<Player> Players, int Minutes);
13     }
14 }
15

```

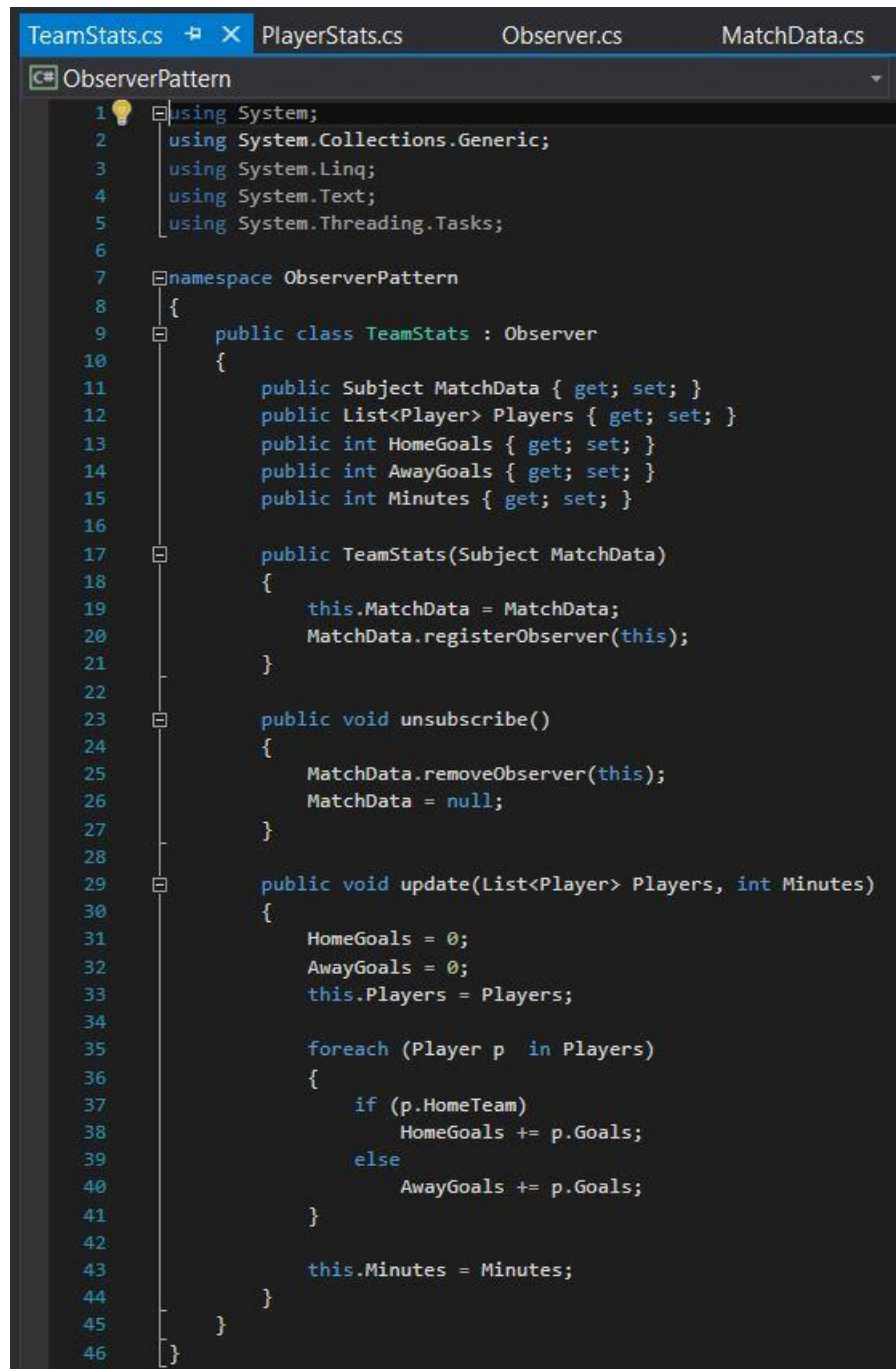
Figure 18. Revised Observer Class



The image shows a Visual Studio code editor with five tabs: PlayerStats.cs, Observer.cs, MatchData.cs, Subject.cs, and Player.cs. The PlayerStats.cs tab is active, showing the following C# code:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class PlayerStats : Observer
10     {
11         public Subject MatchData { get; set; }
12         public List<Player> Players { get; set; }
13         public Player MostMinutesPlayed { get; set; }
14         public Player MostGoals { get; set; }
15         public Player MostAssists { get; set; }
16
17         public PlayerStats(Subject MatchData)
18         {
19             this.MatchData = MatchData;
20             MatchData.registerObserver(this);
21         }
22
23         public void unsubscribe()
24         {
25             MatchData.removeObserver(this);
26             MatchData = null;
27         }
28
29         public void update(List<Player> Players, int Minutes)
30         {
31             this.Players = Players;
32             MostMinutesPlayed = Players.OrderByDescending(player => player.MinutesPlayed).First();
33             MostGoals = Players.OrderByDescending(player => player.Goals).First();
34             MostAssists = Players.OrderByDescending(player => player.Assists).First();
35         }
36     }
37 }
38
```

Figure 19. Revised PlayerStats Class

The image shows a Visual Studio code editor with four tabs at the top: TeamStats.cs, PlayerStats.cs, Observer.cs, and MatchData.cs. The 'ObserverPattern' namespace is selected in the Explorer on the left. The main editor area displays the code for the ObserverPattern namespace, which contains a TeamStats class. The code is as follows:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ObserverPattern
8 {
9     public class TeamStats : Observer
10     {
11         public Subject MatchData { get; set; }
12         public List<Player> Players { get; set; }
13         public int HomeGoals { get; set; }
14         public int AwayGoals { get; set; }
15         public int Minutes { get; set; }
16
17         public TeamStats(Subject MatchData)
18         {
19             this.MatchData = MatchData;
20             MatchData.registerObserver(this);
21         }
22
23         public void unsubscribe()
24         {
25             MatchData.removeObserver(this);
26             MatchData = null;
27         }
28
29         public void update(List<Player> Players, int Minutes)
30         {
31             HomeGoals = 0;
32             AwayGoals = 0;
33             this.Players = Players;
34
35             foreach (Player p in Players)
36             {
37                 if (p.HomeTeam)
38                     HomeGoals += p.Goals;
39                 else
40                     AwayGoals += p.Goals;
41             }
42
43             this.Minutes = Minutes;
44         }
45     }
46 }
```

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.

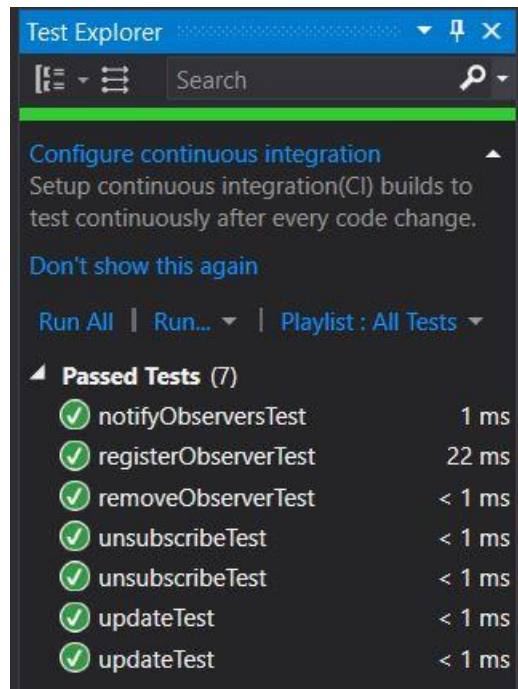


Figure 21. Passed Tests

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.

Strategy Pattern

The Strategy Pattern is one of the most fundamental patterns used by developers. The intent of the pattern is to “define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from the clients that use it.”

Without the use of the Strategy Pattern, algorithms would be implemented directly within the class that uses it. This makes the class inflexible because the algorithm written in the class is used at compile-time even though it might not be correct for the type of data. Also, since the algorithm would be written directly in the class, it cannot be modified in the future without changing the class and limits the reusability of the class. By using the Strategy Pattern, we can encapsulate the interface in a base class and bury the implementation of different algorithms in derived classes. This is also an example of the open-closed principle which means the interface is open to extension but closed from modification. Figure 22, below, shows the outline of the pattern and how the separation of algorithms from the base class allows for changes to be made to the derived classes without affect the client.

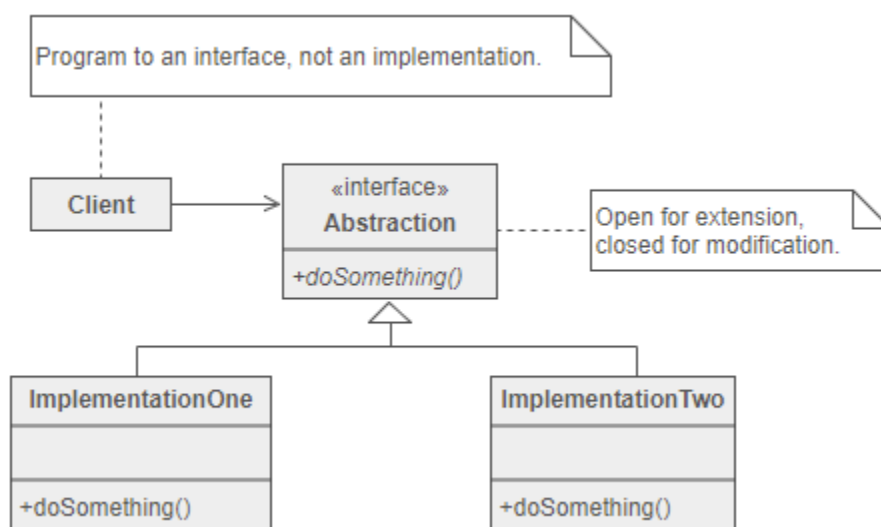


Figure 22: Strategy Pattern Outline

As with most designs, there are advantages and disadvantages. There are many advantages of the Strategy Pattern including the ability to alter behavior without affecting the client, easily add new algorithms, and avoid the use of series of “switch” and “if-else” statements. There are also drawbacks for the pattern which include the application must be aware of all possible strategies to select the correct one, base classes must expose interface for all required behaviors, and since most applications configure the client with the base class, two objects are created instead of one.

To demonstrate the use of the Strategy Pattern, we have created a Calculator class that needs to calculate the average of a list of numbers using either the mean, median, mode, geometric, or harmonic technique. To show the effectiveness of the pattern, we started by writing multiple methods in a single class which is an alternative to the Strategy Pattern. The non-patterned version can be seen below in Figures 23 and 24.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace StrategyPattern
6  {
7      public class Calculator
8      {
9          public double CalculateAverageByMean(List<double> values)
10         {
11             return values.Sum() / values.Count;
12         }
13
14         public double CalculateAverageByMedian(List<double> values)
15         {
16             var sortedValues = values.OrderBy(x => x).ToList();
17
18             if (sortedValues.Count % 2 == 1)
19             {
20                 return sortedValues[(sortedValues.Count - 1) / 2];
21             }
22
23             return (sortedValues[(sortedValues.Count / 2) - 1] +
24                 sortedValues[sortedValues.Count / 2]) / 2;
25         }
26
27         public double CalculateAverageByMode(List<double> values)
28         {
29             var mode = values.GroupBy(i => i).OrderByDescending(grp => grp.Count()).Select(grp => grp.Key).First();
30             return mode;
31         }
32
33         public double CalculateAverageByGeometric(List<double> values)
34         {
35             double product = 1;
36             for(int i = 0; i < values.Count; i++)
37             {
38                 product *= values[i];
39             }
40             return Math.Pow(product, 1.0 / values.Count);
41         }

```

Figure 23: Non-pattern Multiple Method Calculator

```

42
43         public double CalculateAverageByHarmonic(List<double> values)
44         {
45             double sum = 0;
46             for(int i = 0; i < values.Count; i++)
47             {
48                 sum += 1 / values[i];
49             }
50
51             return values.Count / sum;
52         }
53     }
54 }
55

```

Figure 24: Non-pattern Multiple Method Calculator

Another alternative to using the Strategy Pattern is to have all the algorithms contained in a single method with switch statements as shown in Figures 25 and 26 below.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace StrategyPattern
6  {
7      public class CalculatorSingleMethod
8      {
9          public enum AveragingMethod
10         {
11             Mean,
12             Median,
13             Mode,
14             Geometric,
15             Harmonic
16         }
17
18         public double CalculateAverage(List<double> values, AveragingMethod averagingMethod)
19         {
20             switch (averagingMethod)
21             {
22                 case AveragingMethod.Mean:
23                     return values.Sum() / values.Count;
24
25                 case AveragingMethod.Median:
26                     var sortedValues = values.OrderBy(x => x).ToList();
27
28                     if (sortedValues.Count % 2 == 1)
29                     {
30                         return sortedValues[(sortedValues.Count - 1) / 2];
31                     }
32
33                     return (sortedValues[(sortedValues.Count / 2) - 1] +
34                             sortedValues[sortedValues.Count / 2]) / 2;
35
36                 case AveragingMethod.Mode:
37                     var mode = values.GroupBy(i => i).OrderByDescending(grp => grp.Count()).Select(grp => grp.Key).First();
38                     return mode;
39             }
40         }
41     }
42 }
```

Figure 25: Non-pattern Single Method Calculator

```
39
40
41         case AveragingMethod.Geometric:
42             double product = 1;
43             for (int i = 0; i < values.Count; i++)
44             {
45                 product *= values[i];
46             }
47             return Math.Pow(product, 1.0 / values.Count);
48
49         case AveragingMethod.Harmonic:
50             double sum = 0;
51             for (int i = 0; i < values.Count; i++)
52             {
53                 sum += 1 / values[i];
54             }
55             return values.Count / sum;
56
57         default:
58             throw new ArgumentException("Invalid averagingMethod value");
59     }
60 }
61
62 }
```

Figure 26: Non-pattern Single Method Calculator

While the code shown above will work, it is not ideal. The code is long making it hard to read and debug and if any changes want to be made, the class must be rewritten. Now we will implement the class using the Strategy Pattern. The project does become more complex when implementing the project due to the use of multiple files but will be more useful in the future when changes need to be made. We start the implementation of the pattern by creating the interface as shown in Figure 27, below. The interface declares what must exist in each concrete strategy class and the concrete strategy classes implement the interface.

```
1      using System.Collections.Generic;
2
3      namespace StrategyPattern
4      {
5          public interface IVeragingMethod
6          {
7              double AverageFor(List<double> values);
8          }
9      }
10
```

Figure 27: Interface Class

After writing the interface, each concrete class was written. Each concrete class must include an “AverageFor” function as declared in the interface. The concrete classes are shown below in Figures 28 through 32 in the following order: mean, median, mode, geometric, and harmonic.

```

1  using System.Collections.Generic;
2  using System.Linq;
3
4  namespace StrategyPattern
5  {
6      public class AverageByMean : IAveragingMethod
7      {
8          public double AverageFor(List<double> values)
9          {
10             // sum of all values, divided by number of values.
11             return values.Sum() / values.Count;
12         }
13     }
14 }
15

```

Figure 28: AverageByMean Concrete Class

```

1  using System.Collections.Generic;
2  using System.Linq;
3
4  namespace StrategyPattern
5  {
6      public class AverageByMedian : IAveragingMethod
7      {
8          public double AverageFor(List<double> values)
9          {
10             // Median average is the middle value of the values in the list.
11             var sortedValues = values.OrderBy(x => x).ToList();
12
13             // Determine if number of values is even or odd
14             if (sortedValues.Count % 2 == 1)
15             {
16                 // Number of values is odd.
17                 return sortedValues[(sortedValues.Count - 1) / 2];
18             }
19
20             // Number of values is even.
21             return (sortedValues[(sortedValues.Count / 2) - 1] +
22                 sortedValues[sortedValues.Count / 2]) / 2;
23         }
24     }
25 }
26

```

Figure 29: AverageByMedian Concrete Class

```

1  using System.Collections.Generic;
2  using System.Linq;
3
4
5  namespace StrategyPattern
6  {
7      public class AverageByMode : IAveragingMethod
8      {
9          public double AverageFor(List<double> values)
10         {
11             //Group same values in List and select group with most values
12             var mode = values.GroupBy(i => i).OrderByDescending(grp => grp.Count()).Select(grp => grp.Key).First();
13             return mode;
14         }
15     }
16 }
17

```

Figure 30: AverageByMode Concrete Class

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace StrategyPattern
5  {
6      public class AverageByGeometric : IAveragingMethod
7      {
8          public double AverageFor(List<double> values)
9          {
10             //multiply all values in list and raise to reciprocal of number of values
11             double product = 1;
12             for (int i = 0; i < values.Count; i++)
13             {
14                 product *= values[i];
15             }
16             return Math.Pow(product, 1.0 / values.Count);
17         }
18     }
19 }
20

```

Figure 31: AverageByGeometric Concrete Class

```

1  using System.Collections.Generic;
2
3  namespace StrategyPattern
4  {
5      public class AverageByHarmonic : IAveragingMethod
6      {
7          public double AverageFor(List<double> values)
8          {
9              //Sum the reciprocals of values in list and divide number of items in list by sum
10             double sum = 0;
11             for (int i = 0; i < values.Count; i++)
12             {
13                 sum += 1 / values[i];
14             }
15
16             return values.Count / sum;
17         }
18     }
19 }
20

```

Figure 32: AverageByHarmonic Concrete Class

After all the concrete classes had been written, we wrote the client class to use the interface class. As shown in Figure 33, below, the function receives a list of numbers and an object of the type of our interface class. This allows us to pass an object that implements the correct concrete class.

```

1  using System.Collections.Generic;
2
3  namespace StrategyPattern
4  {
5      public class Calculator
6      {
7          public double CalculateAverageFor(List<double> values, IAveragingMethod averagingMethod)
8          {
9              return averagingMethod.AverageFor(values);
10             }
11         }
12     }

```

Figure 33: Calculator Class

The final step in implementing the Strategy Pattern was to test the design. To do this, we create Unit test that would create a Calculator object, call the CalculateAverageFor function for each of the concrete classes, and compare the returned value to expected value. Figure 34, below, shows the setup of the test suite where the list of values that are used in each test is created.

```
1  using System;
2  using System.Collections.Generic;
3  using Microsoft.VisualStudio.TestTools.UnitTesting;
4
5  namespace StrategyPattern
6  {
7      [TestClass]
8      public class TestCalculator
9      {
10         private readonly List<double> _values = new List<double> { 10, 5, 7, 15, 13, 12, 8, 7, 4, 2, 9 };
11     }
```

Figure 34: Test Suite List

Before we could write any test, we need to write a short function to return a bool if the expected value and actual value are within .000001. This function is required because we are using floating point numbers and values may not be an exact match. The function can be seen in Figure 35, below.

```
65  private bool ResultsAreCloseEnough(double expectedResult, double calculatedResult)
66  {
67      var difference = Math.Abs(expectedResult - calculatedResult);
68
69      return difference < .000001;
70  }
71
72 }
```

Figure 35: Results Close Enough Function

We then wrote the test for each concrete class which can be seen in Figures 36 through 40 below. Each test creates a Calculator object, calls the CalculateAverageFor function passing the desired concrete class, and checks if the returned value is close enough to the expected value.

```
[TestMethod]
public void Test_AverageByMean()
{
    Calculator calculator = new Calculator();

    var averageByMean = calculator.CalculateAverageFor(_values, new AverageByMean());

    Assert.IsTrue(ResultsAreCloseEnough(8.3636363, averageByMean));
}
```

Figure 36: Test AverageByMean

```
[TestMethod]
public void Test_AverageByMedian()
{
    Calculator calculator = new Calculator();

    var averageByMedian = calculator.CalculateAverageFor(_values, new AverageByMedian());

    Assert.IsTrue(ResultsAreCloseEnough(8, averageByMedian));
}
```

Figure 37: Test AverageByMedian

```
[TestMethod]
public void Test_AverageByMode()
{
    Calculator calculator = new Calculator();

    var averageByMode = calculator.CalculateAverageFor(_values, new AverageByMode());

    Assert.IsTrue(ResultsAreCloseEnough(7, averageByMode));
}
```

Figure 38: Test AverageByMode

```

[TestMethod]
public void Test_AverageByGeometric()
{
    Calculator calculator = new Calculator();

    var averageByGeometric = calculator.CalculateAverageFor(_values, new AverageByGeometric());

    Assert.IsTrue(ResultsAreCloseEnough(7.3340836, averageByGeometric));
}

```

Figure 39: Test AverageByGeometric

```

[TestMethod]
public void Test_AverageByHarmonic()
{
    Calculator calculator = new Calculator();

    var averageByHarmonic = calculator.CalculateAverageFor(_values, new AverageByHarmonic());

    Assert.IsTrue(ResultsAreCloseEnough(6.1153631, averageByHarmonic));
}

```

Figure 40: Test AverageByHarmonic

As shown in Figure 41, below, all test passed proving the Strategy Patten is working correctly.

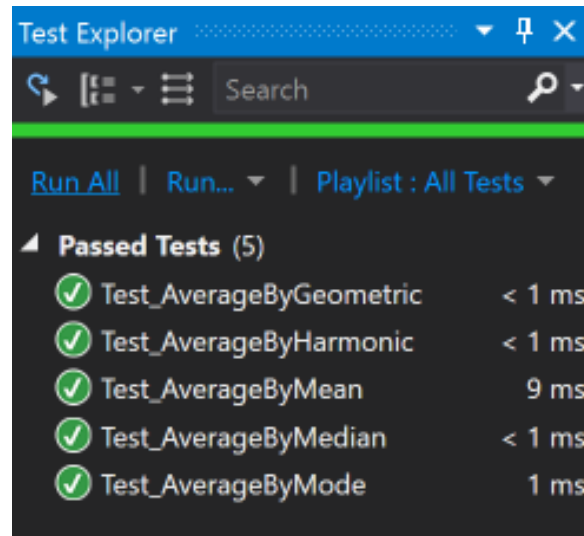


Figure 41: Calculator Passed Test

As the section of the project has demonstrated, the Strategy Pattern is a useful way to organized projects with more than one way to accomplish a task. The separation of each algorithm in its own class allows for changes to be made to existing algorithms and new algorithms to be add or subtracted without affecting the functionality of the interface or client. This is a better option to the alternative of having all algorithms in one class but different functions or having all algorithms in one class and one function. These alternative methods are okay for smaller projects but in a larger project, having hundreds of lines of code in a single class can be difficult to read and debug.

Decorator Pattern

There are several occasions in applications when developers need to create an object with some basic functionality in such a way that some extra functionality can be added to this object dynamically. Assuming a developer writes code for a basic object that can be customized in multiple different ways, the developer may end up having an extraordinary number of subclasses. One such example was shared in the book: at a coffee shop, customers can add several different condiments to their order of coffee. Due to the multitude of possible combinations, the developer may end up with an excessively large number of subclasses, as shown in Figure 42.

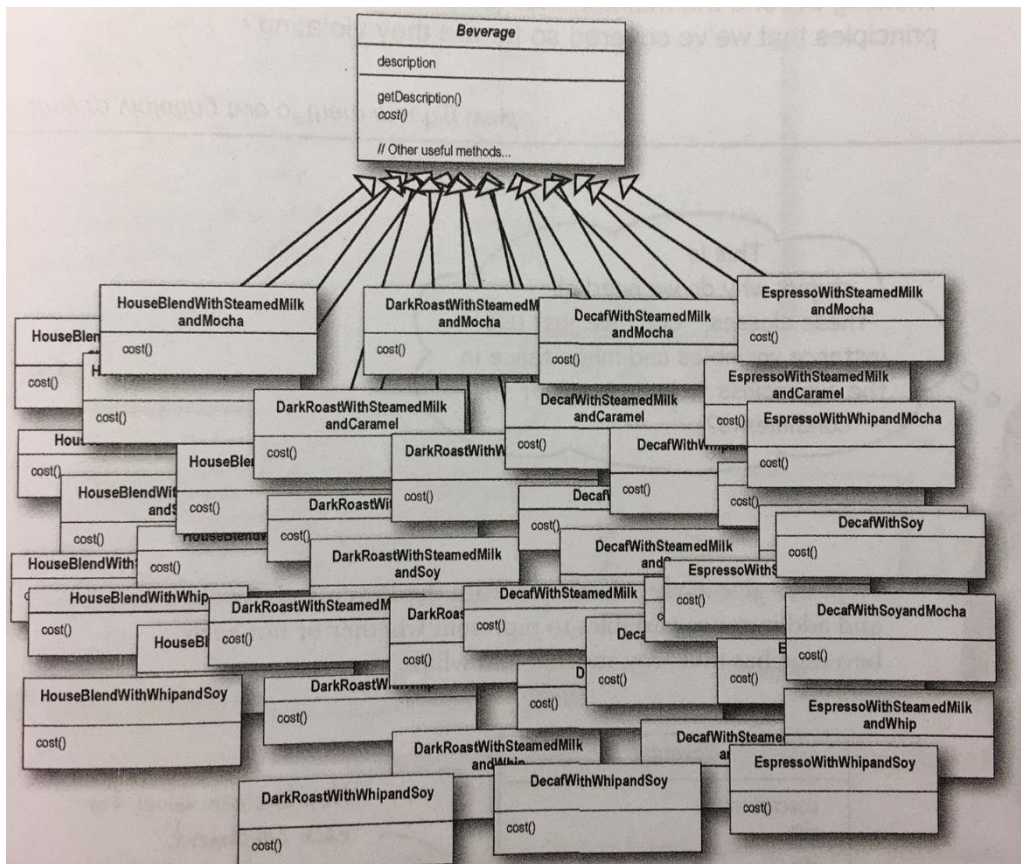


Figure 42: Overpopulated Subclass Map

This massive map of subclasses is highly undesirable. So what can be done to optimize a setup like the one shown above? Look no further than the Decorator pattern. The Decorator pattern is used to add new functionality to an existing object without changing its structure. In this way, the Decorator pattern provides an alternative way to inheritance for modifying the behavior of an object. This approach follows the Open-Closed Principle: classes should be open for extension, but closed for modification. Figure 43 shows a general outline of how a decorator is implemented.

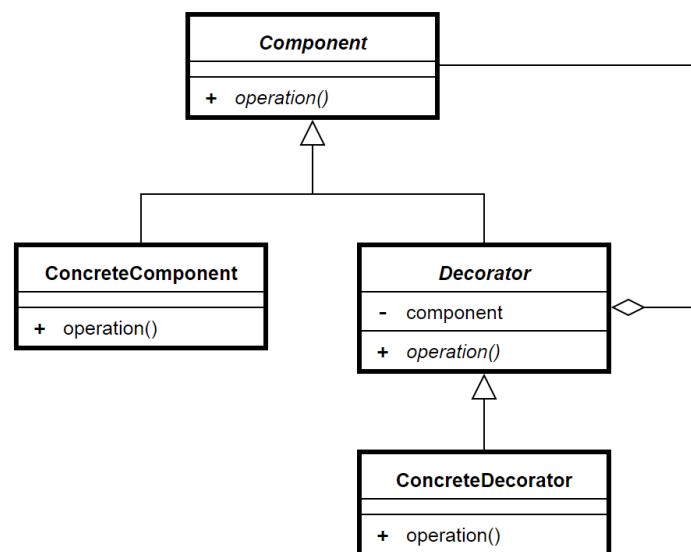
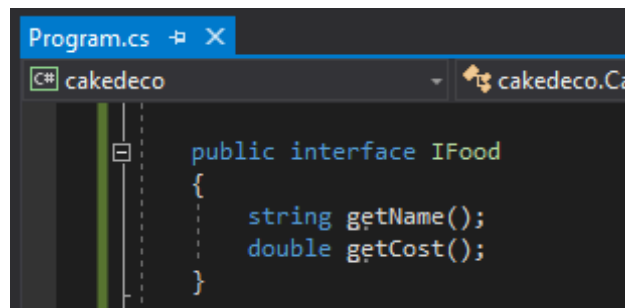


Figure 43: Decorator Implementation Outline

The component is an interface containing members that will be implanted by **ConcreteComponent** and **Decorator**. **ConcreteComponent** is a class which implements the **Component** interface. **Decorator** is an abstract class which implements the **Component** interface and contains the reference to a **Component** instance. This class also acts as a base class for all

decorators for components. ConcreteDecorator is a class which inherits from Decorator class and provides a decorator for components.

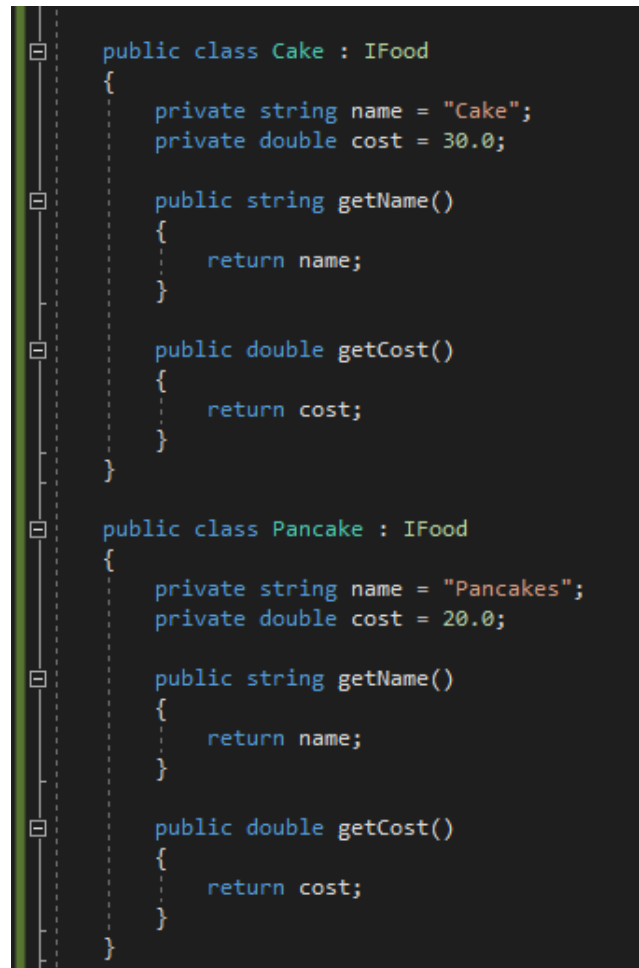
The adjustment from subclassing to applying the Decorator pattern causes a shift from compiler time to run-time, all while maintaining the integrity of the original code structure. This should be obvious given the multitude of possible combinations an object can have. The following examples will show this in more detail. Let's start by creating a public interface for food, shown in Figure 44.

A screenshot of a C# code editor window titled 'Program.cs'. The editor shows a public interface named 'IFood' with two methods: 'getName()' returning a 'string' and 'getCost()' returning a 'double'. The code is written in a dark-themed editor with syntax highlighting. The file explorer on the left shows a project named 'cakedeco' with a file 'cakedeco.Ca' selected.

```
public interface IFood
{
    string getName();
    double getCost();
}
```

Figure 44: Initial IFood Interface

Let's now create two classes for IFood: Cake and Pancake, shown in Figure 45.

A screenshot of a code editor with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for file operations (like save, open, close) and a line-numbered gutter. The code is written in C++ and defines two classes, Cake and Pancake, both inheriting from an IFood interface. The Cake class has a private string name "Cake" and a private double cost 30.0. The Pancake class has a private string name "Pancakes" and a private double cost 20.0. Both classes implement the getName() and getCost() methods. The code is as follows:

```
public class Cake : IFood
{
    private string name = "Cake";
    private double cost = 30.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

public class Pancake : IFood
{
    private string name = "Pancakes";
    private double cost = 20.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}
```

Figure 45: Cake and Pancake Classes

Sometimes, plain cakes and pancakes aren't enough to satisfy the appetite, so we throw in a lot of toppings. Following are several examples of possible combinations of cakes and pancakes, given the toppings of scent, strawberries, and cream. See Figures 46-49.


```

public class ScentStrawberryCakeWithCream : IFood
{
    private string name = "Scented strawberry cake with cream";
    private double cost = 38.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

public class StrawberryCakeWithCream : IFood
{
    private string name = "Strawberry cake with cream";
    private double cost = 35.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

```

Figure 46: Cake with Toppings Classes

```

public class ScentedStrawberryCake : IFood
{
    private string name = "Scented strawberry cake";
    private double cost = 37.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

public class CakeWithCream : IFood
{
    private string name = "Cake with cream";
    private double cost = 31.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

```

Figure 47: Cake with Toppings Classes (cont.)

```

public class StrawberryPancakesWithCream : IFood
{
    private string name = "Strawberry pancakes with cream";
    private double cost = 25.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

public class StrawberryPancakes : IFood
{
    private string name = "Strawberry pancakes";
    private double cost = 24.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

```

Figure 48: Pancakes with Toppings Classes

```

public class PancakesWithCream : IFood
{
    private string name = "Pancakes with cream";
    private double cost = 21.0;

    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

public class ScentedStrawberryPancakesWithCream : IFood
{
    private string name = "Scented strawberry pancakes with cream";
    private double cost = 28.0;

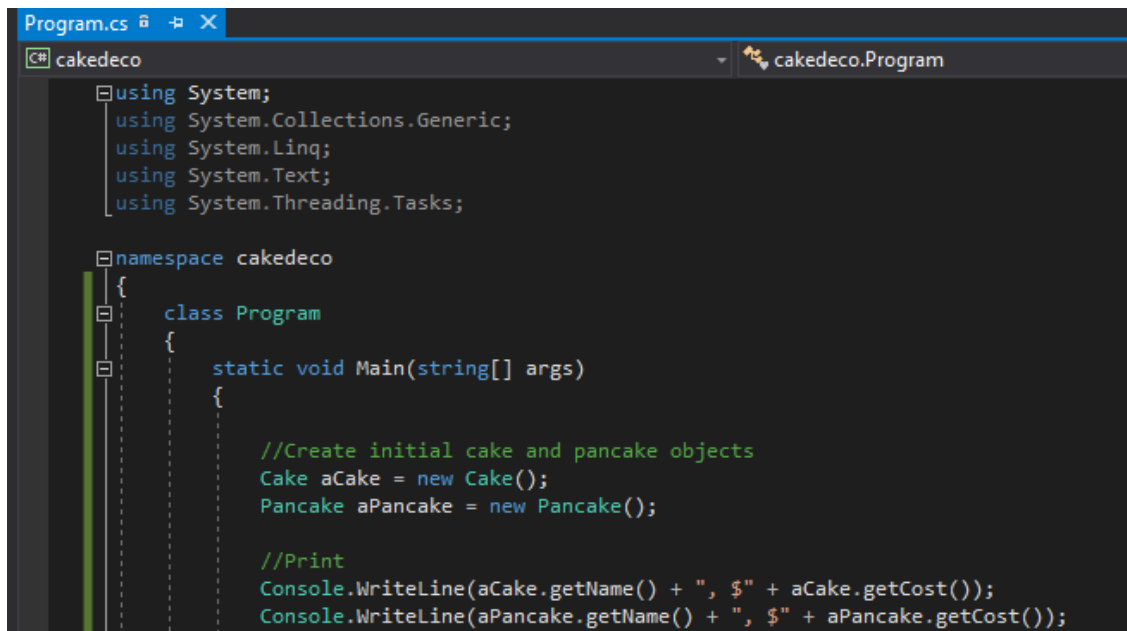
    public string getName()
    {
        return name;
    }

    public double getCost()
    {
        return cost;
    }
}

```

Figure 49: Pancakes with Toppings Classes (cont.)

As you can see, there are several different combinations of toppings for these food items, so several classes have to be generated and compiled with each run. However, this simplifies the writing of the main method. You can see basic cake and pancake objects being generated in Figure 50.



```
Program.cs
cakedeco
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace cakedeco
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create initial cake and pancake objects
            Cake aCake = new Cake();
            Pancake aPancake = new Pancake();

            //Print
            Console.WriteLine(aCake.getName() + ", $" + aCake.getCost());
            Console.WriteLine(aPancake.getName() + ", $" + aPancake.getCost());
        }
    }
}
```

Figure 50: Initialize Cake and Pancake Objects

Figure 51 shows the simple output and description of these two objects.

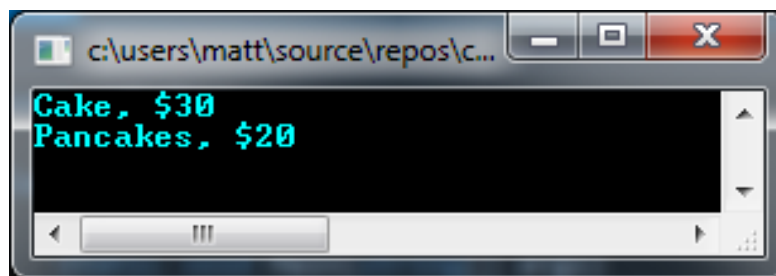


Figure 51: Cake and Pancake Objects Output

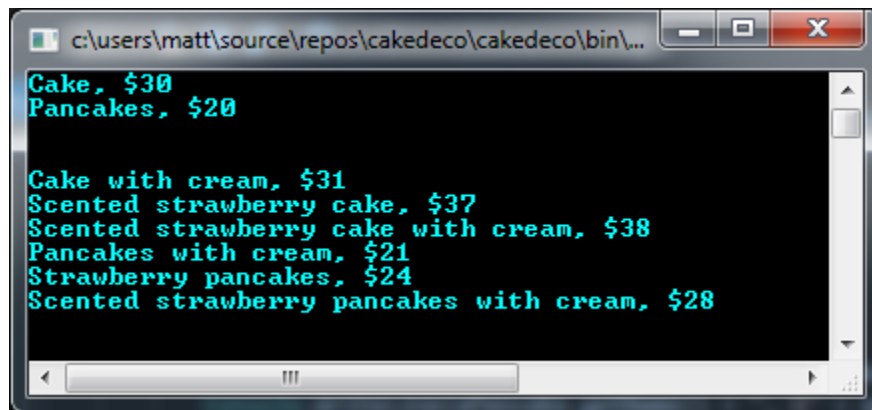
Figure 52 shows several of the combinations of toppings on cakes and pancakes being generated in the main method.

```
//Create objects for cakes and pancakes with toppings
CakeWithCream bCake = new CakeWithCream();
ScentedStrawberryCake cCake = new ScentedStrawberryCake();
ScentedStrawberryCakeWithCream dCake = new ScentedStrawberryCakeWithCream();
PancakesWithCream bPancake = new PancakesWithCream();
StrawberryPancakes cPancake = new StrawberryPancakes();
ScentedStrawberryPancakesWithCream dPancake = new ScentedStrawberryPancakesWithCream();

//Print
Console.WriteLine(bCake.getName() + ", $" + bCake.getCost());
Console.WriteLine(cCake.getName() + ", $" + cCake.getCost());
Console.WriteLine(dCake.getName() + ", $" + dCake.getCost());
Console.WriteLine(bPancake.getName() + ", $" + bPancake.getCost());
Console.WriteLine(cPancake.getName() + ", $" + cPancake.getCost());
Console.WriteLine(dPancake.getName() + ", $" + dPancake.getCost());
```

Figure 52: Generate Cake and Pancakes with Toppings Objects

Figure 53 shows the updated output with the descriptions and prices of the cakes and pancakes.



```
c:\users\matt\source\repos\cakedeco\cakedeco\bin\...
Cake, $30
Pancakes, $20

Cake with cream, $31
Scented strawberry cake, $37
Scented strawberry cake with cream, $38
Pancakes with cream, $21
Strawberry pancakes, $24
Scented strawberry pancakes with cream, $28
```

Figure 53: Cake and Pancakes with Toppings Output

It's time to use the Decorator Pattern. Let's start this coding process over and go back to when we only had the original Cake and Pancake classes; these will be our concrete components. This time, instead of generating several additional classes with toppings, let's create a Decorator class. This class is shown in Figure 54.

```

public abstract class Decorator : IFood
{
    private IFood food;

    public Decorator(IFood afood)
    {
        this.food = afood;
    }

    public virtual string getName()
    {
        return this.food.getName();
    }

    public virtual double getCost()
    {
        return this.food.getCost();
    }
}

```

Figure 54: Initial Decorator

The Decorator references its super to obtain the original name and cost of the object. Let's now create three concrete decorators, one for each topping – scent, strawberries, and cream. These decorators are shown in Figures 55-57.

```

public class ScentDec : Decorator
{
    public ScentDec(IFood afood)
    : base(afood)
    {
    }

    public override string getName()
    {
        return base.getName() + ", scent";
    }

    public override double getCost()
    {
        return base.getCost() + 3.0;
    }
}

```

Figure 55: Concrete Decorator, Scent

```

class StrawberryDec : Decorator
{
    public StrawberryDec(IFood afood)
    : base(afood)
    {
    }

    public override string getName()
    {
        return base.getName() + ", strawberry";
    }

    public override double getCost()
    {
        return base.getCost() + 4.0;
    }
}

```

Figure 56: Concrete Decorator, Strawberry

```

class CreamDec : Decorator
{
    public CreamDec(IFood afood)
    : base(afood)
    {
    }

    public override string getName()
    {
        return base.getName() + ", cream";
    }

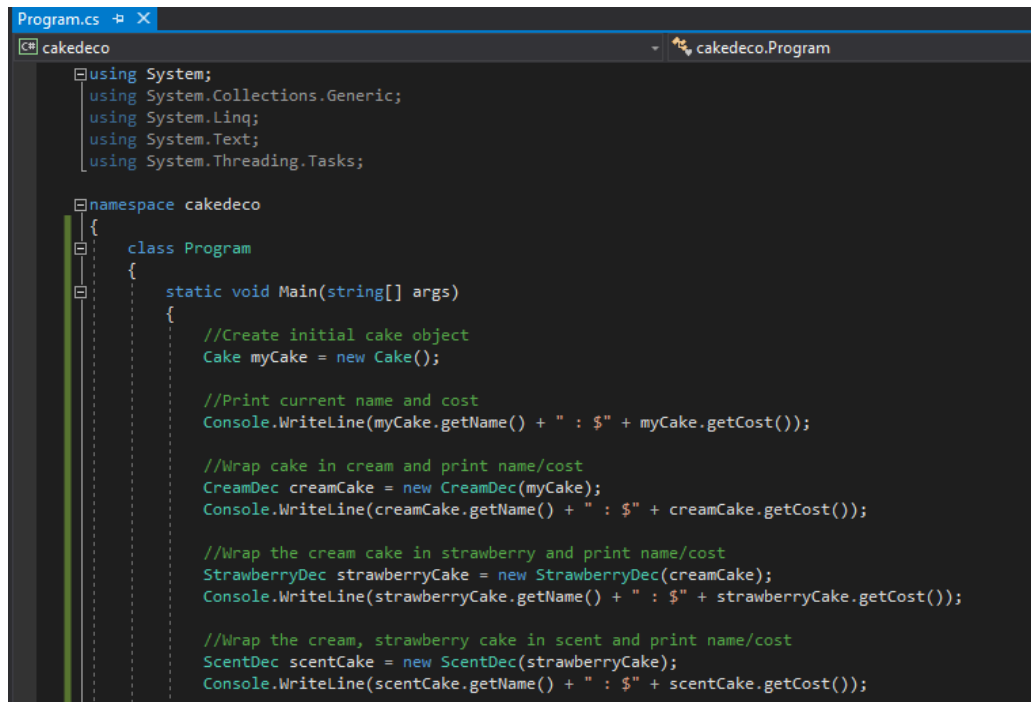
    public override double getCost()
    {
        return base.getCost() + 1.0;
    }
}

```

Figure 57: Concrete Decorator, Cream

As you can see, each decorator class references the super class to retrieve the original name and cost. The decorator classes then take these values and add their own unique values to them. For example, StrawberryDec adds “, strawberry” to the name of the object and adds 4.0 to the cost. Writing code in this manner allows the developer to only write one class per topping, rather than one class for each possible combination of toppings. The developer isn’t limited to using just one

topping; the developer can first wrap the object in a topping, and then wrap the new object in another topping. This can be clearly seen in Figure 58. See how a new Cake object is first generated, then wrapped in CreamDec, StrawberryDec, and finally ScentDec.



```
Program.cs
cakedeco
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace cakedeco
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create initial cake object
            Cake myCake = new Cake();

            //Print current name and cost
            Console.WriteLine(myCake.getName() + " : $" + myCake.getCost());

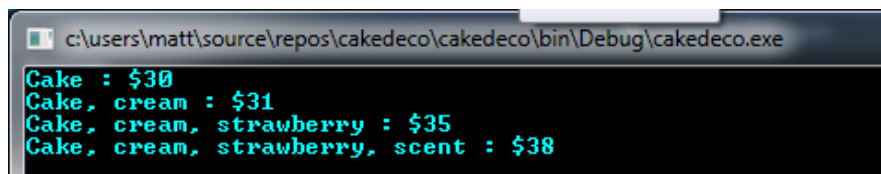
            //Wrap cake in cream and print name/cost
            CreamDec creamCake = new CreamDec(myCake);
            Console.WriteLine(creamCake.getName() + " : $" + creamCake.getCost());

            //Wrap the cream cake in strawberry and print name/cost
            StrawberryDec strawberryCake = new StrawberryDec(creamCake);
            Console.WriteLine(strawberryCake.getName() + " : $" + strawberryCake.getCost());

            //Wrap the cream, strawberry cake in scent and print name/cost
            ScentDec scentCake = new ScentDec(strawberryCake);
            Console.WriteLine(scentCake.getName() + " : $" + scentCake.getCost());
        }
    }
}
```

Figure 58: Generate Cake with Decorators Object

Figure 59 shows the step-by-step output of each layer of topping being added, printing the updated description and cost.



```
c:\users\matt\source\repos\cakedeco\cakedeco\bin\Debug\cakedeco.exe
Cake : $30
Cake, cream : $31
Cake, cream, strawberry : $35
Cake, cream, strawberry, scent : $38
```

Figure 59: Cake with Decorators Output

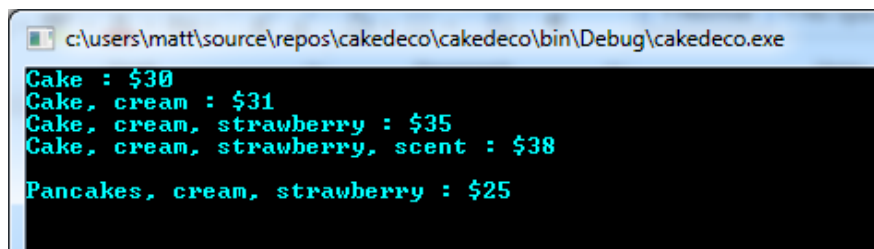
The same process can be shown in Figures 60 and 61 for generating a Pancake object and adding strawberries and cream.

```
//Create initial pancake object
Pancake myPancake = new Pancake();

//Wrap pancake in cream and strawberry and print name/cost
CreamDec creamPancake = new CreamDec(myPancake);
StrawberryDec strawberryPancake = new StrawberryDec(creamPancake);
Console.WriteLine("\n" + strawberryPancake.getName() + " : $" + strawberryPancake.getCost());

//End program
Console.ReadKey();
}
```

Figure 60: Generate Pancakes with Decorators Object



```
c:\users\matt\source\repos\cakedeco\cakedeco\bin\Debug\cakedeco.exe
Cake : $30
Cake, cream : $31
Cake, cream, strawberry : $35
Cake, cream, strawberry, scent : $38
Pancakes, cream, strawberry : $25
```

Figure 61: Pancakes with Decorators Output

Writing code in this manner allows the developer to only write one class per topping, rather than one class for each possible combination of toppings. Based on the number of combinations an object has, the Decorator pattern has the potential to greatly reduce compile time. Overall, the Decorator Pattern is a useful method for creating flexible designs and staying true to the Open-Closed Principle.

However, designs using this pattern often result in a large number of small classes that can be overwhelming to a developer trying to use the Decorator pattern. This also results in a design that's not as straightforward for others to understand. Objects generated using these classes also

have the potential of needing to be wrapped in several decorators, which also complicates the code.

Factory Pattern

In all our patterns up to now, we have consistently used the “new” operator to instantiate a new object. Unfortunately, using this operator can lead to coupling problems. As seen from our previous patterns, we would like to refactor code to implement interfaces rather than concrete classes. However, the “new” operator creates an instance of a concrete class. This can lead to the programmer having to create a long list of if-else statements to create instances of different concrete classes. Even worse, the decision of which class “to instantiate is made at runtime depending on some set of conditions” (Freeman 110). How do we avoid this coupling, though? Simple. We utilize the factory pattern.

The factory pattern creates a separate class that handles the construction of our object depending on the specified type. It also allows the programmer to add or remove specific types of objects by modifying only this factory class, rather than the method holding the constructor itself.

Developers can even go so far as to add methods that allow the user to add constructible subclasses at runtime, although our example will not delve this deeply into the factory pattern.

For the purposes of this project, we will model a student major declaration system for Engineering students. Depending on the string passed to the function in our factory class, the student will be instantiated as a specific Engineering major (Computer, Electrical, etc.). An example of this pattern outline is shown in Figure 62, below, using a PizzaFactory class to construct different types of pizzas.

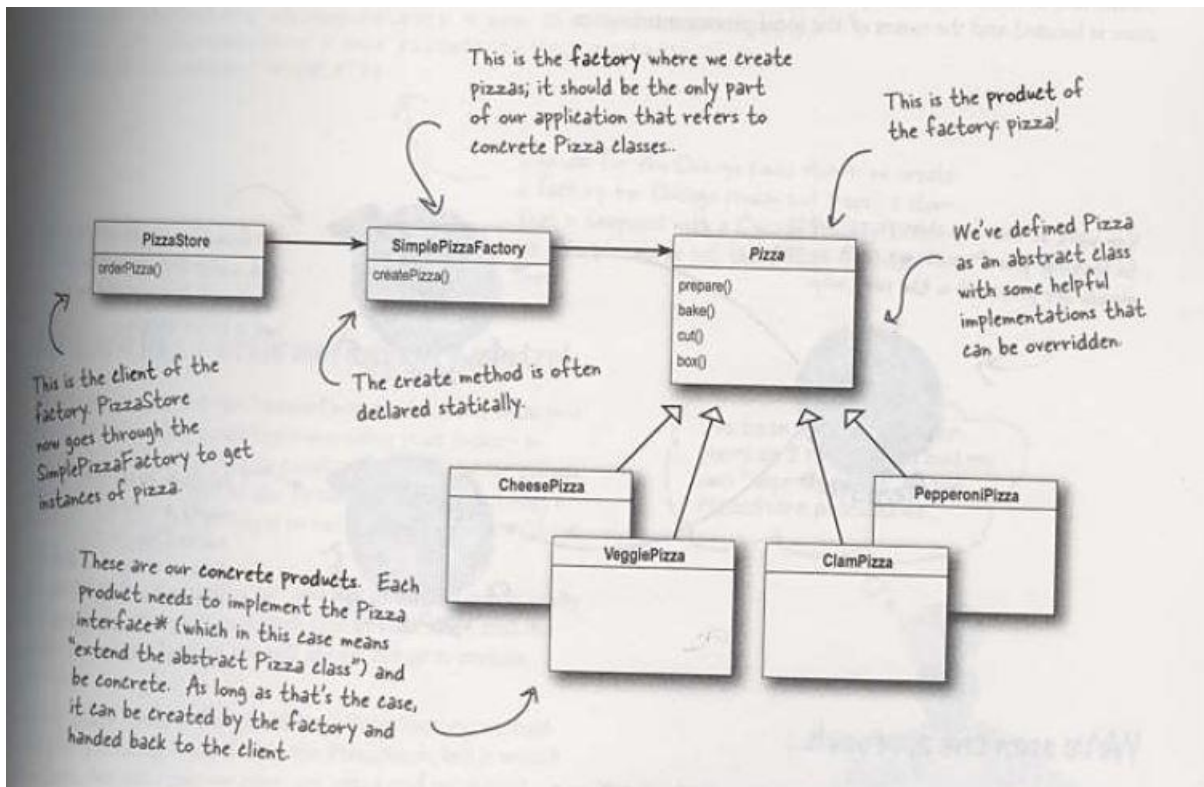
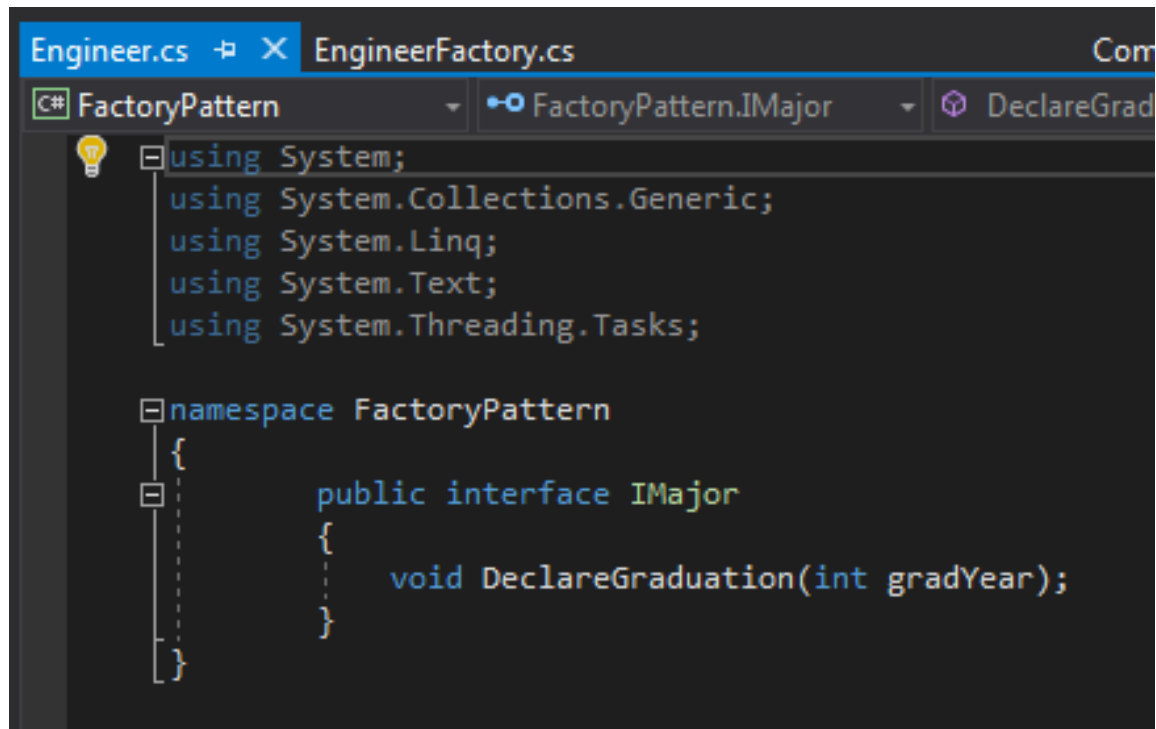


Figure 62: Factory Pattern Example

One advantage of this factory class is that it can be utilized by any number of clients. It can then be used in conjunction with other classes and other factory classes to provide a wealth of information. All changes then need to be made to only one class. The factory pattern is extremely useful when implementing polymorphism into a design, as it handles the instantiation of subclasses with no extra input required from the user. Once again, we are decoupling the implementation of our object from its use.

See Figure 63 for our implementation of the IMajor interface in our Factory Pattern example.



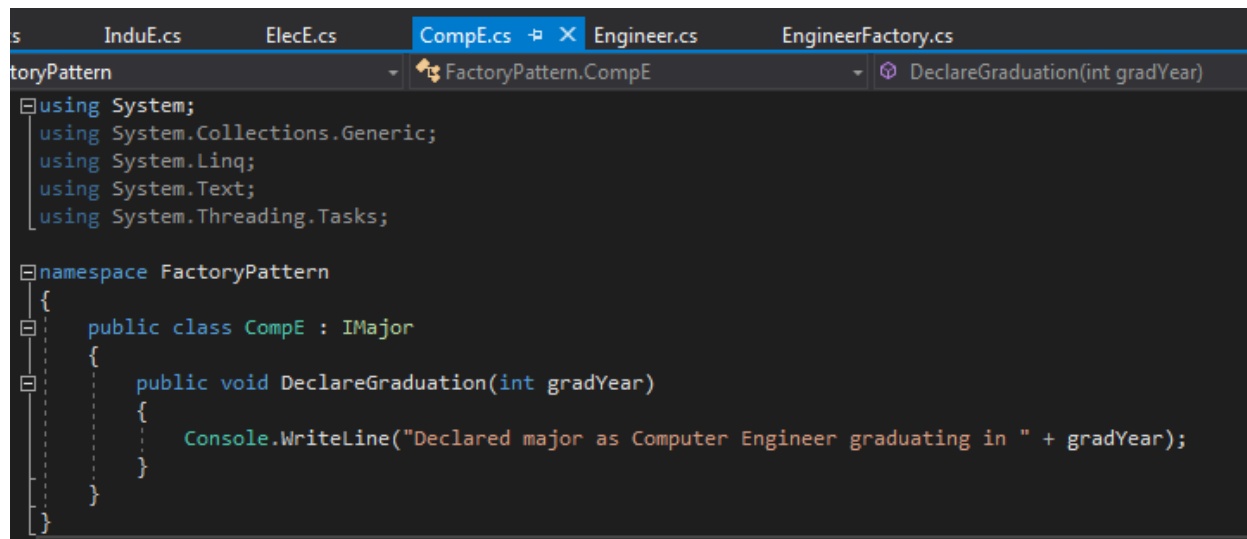
The screenshot shows a code editor with two tabs: 'Engineer.cs' and 'EngineerFactory.cs'. The 'EngineerFactory.cs' tab is active, showing the following C# code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryPattern
{
    public interface IMajor
    {
        void DeclareGraduation(int gradYear);
    }
}
```

Figure 63: IMajor Interface

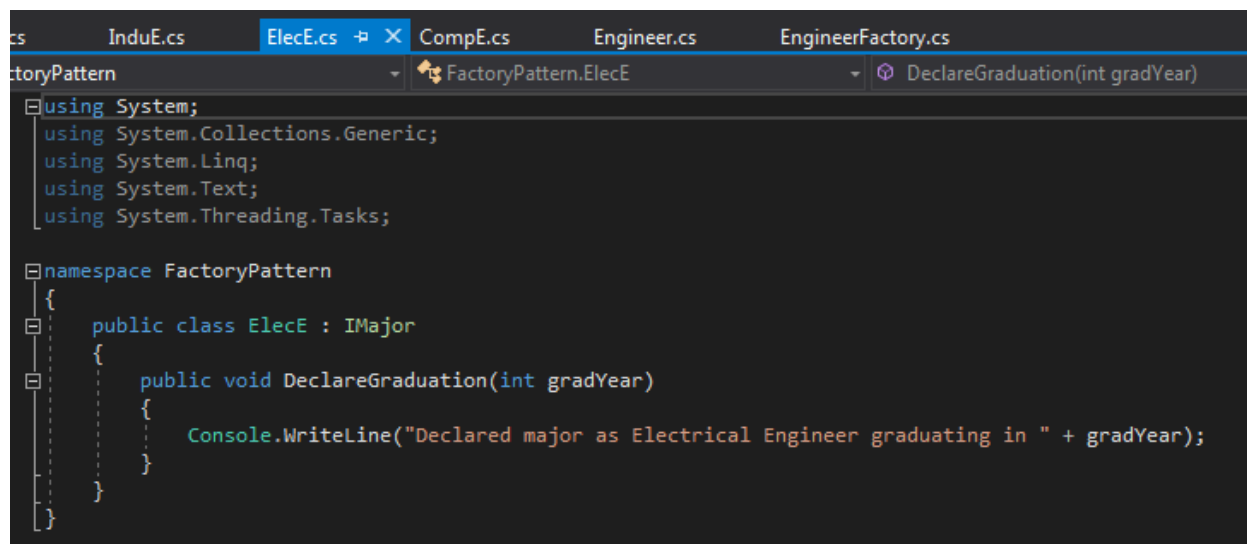
We decided to give our interface one method, `DeclareGraduation()`, which the developer can use to specify a graduation year for each student that falls under the `IMajor` interface. We then created four classes that implement the `IMajor` interface: `CompE`, `ElecE`, `InduE`, and `SoftE`. These represent four different specializations of engineering. See Figures 64-67 for these classes and their unique output from the `DeclareGraduation()` method.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryPattern
{
    public class CompE : IMajor
    {
        public void DeclareGraduation(int gradYear)
        {
            Console.WriteLine("Declared major as Computer Engineer graduating in " + gradYear);
        }
    }
}
```

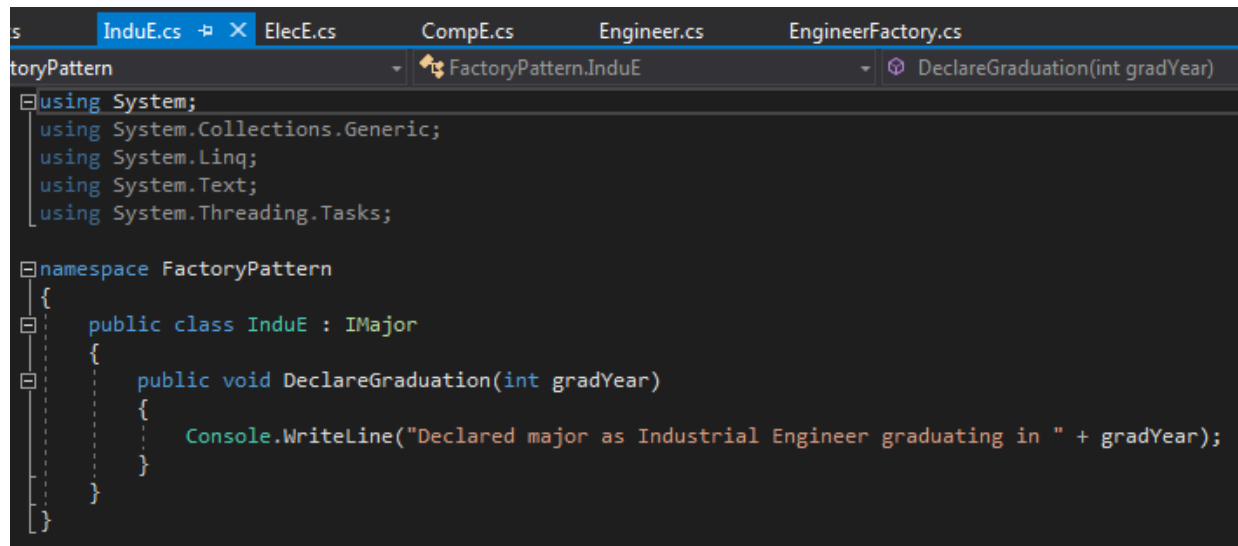
Figure 64: CompE Class



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryPattern
{
    public class ElecE : IMajor
    {
        public void DeclareGraduation(int gradYear)
        {
            Console.WriteLine("Declared major as Electrical Engineer graduating in " + gradYear);
        }
    }
}
```

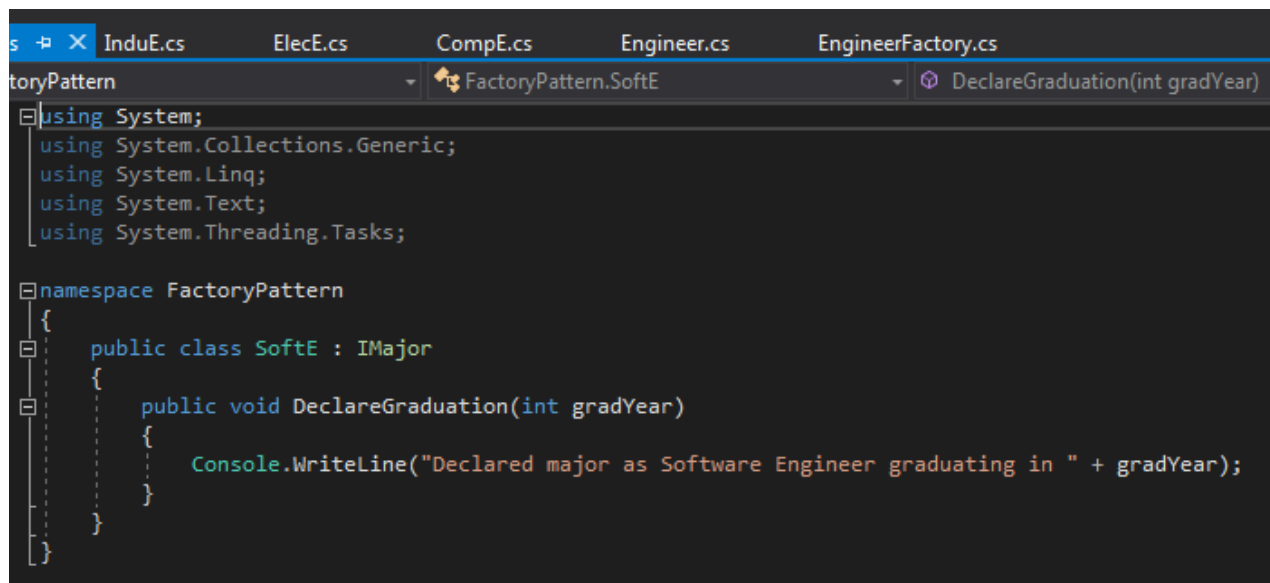
Figure 65: ElecE Class



```
s  InduE.cs  ElecE.cs  CompE.cs  Engineer.cs  EngineerFactory.cs
FactoryPattern  FactoryPattern.InduE  DeclareGraduation(int gradYear)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryPattern
{
    public class InduE : IMajor
    {
        public void DeclareGraduation(int gradYear)
        {
            Console.WriteLine("Declared major as Industrial Engineer graduating in " + gradYear);
        }
    }
}
```

Figure 66: InduE Class

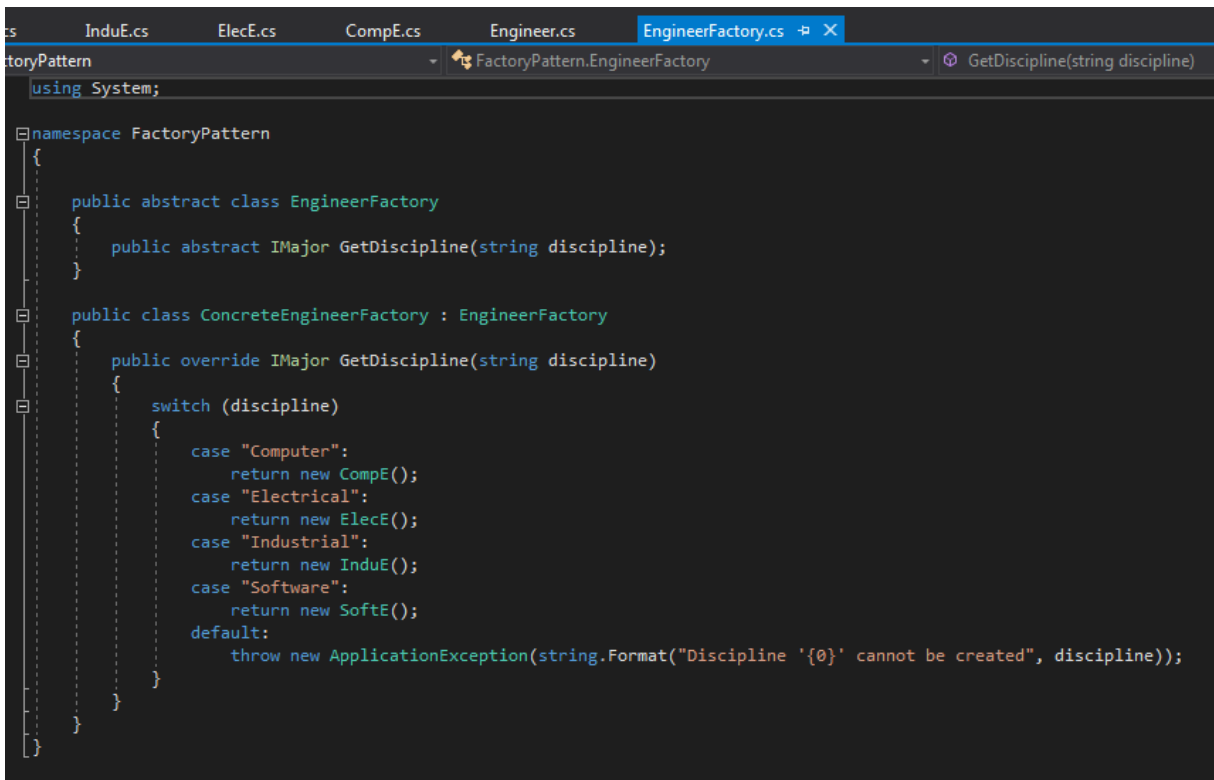


```
s  InduE.cs  ElecE.cs  CompE.cs  Engineer.cs  EngineerFactory.cs
FactoryPattern  FactoryPattern.SoftE  DeclareGraduation(int gradYear)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryPattern
{
    public class SoftE : IMajor
    {
        public void DeclareGraduation(int gradYear)
        {
            Console.WriteLine("Declared major as Software Engineer graduating in " + gradYear);
        }
    }
}
```

Figure 67: SoftE Class

The next step was to create the factory. We wrote both a public abstract class named `EngineerFactory`, and a public class named `ConcreteEngineerFactory` that implemented our abstract class. A `GetDiscipline` method was written that would generate a specific type of object given the input. Figure 68 shows the switch statement used to decide what class would be used to generate a new object.



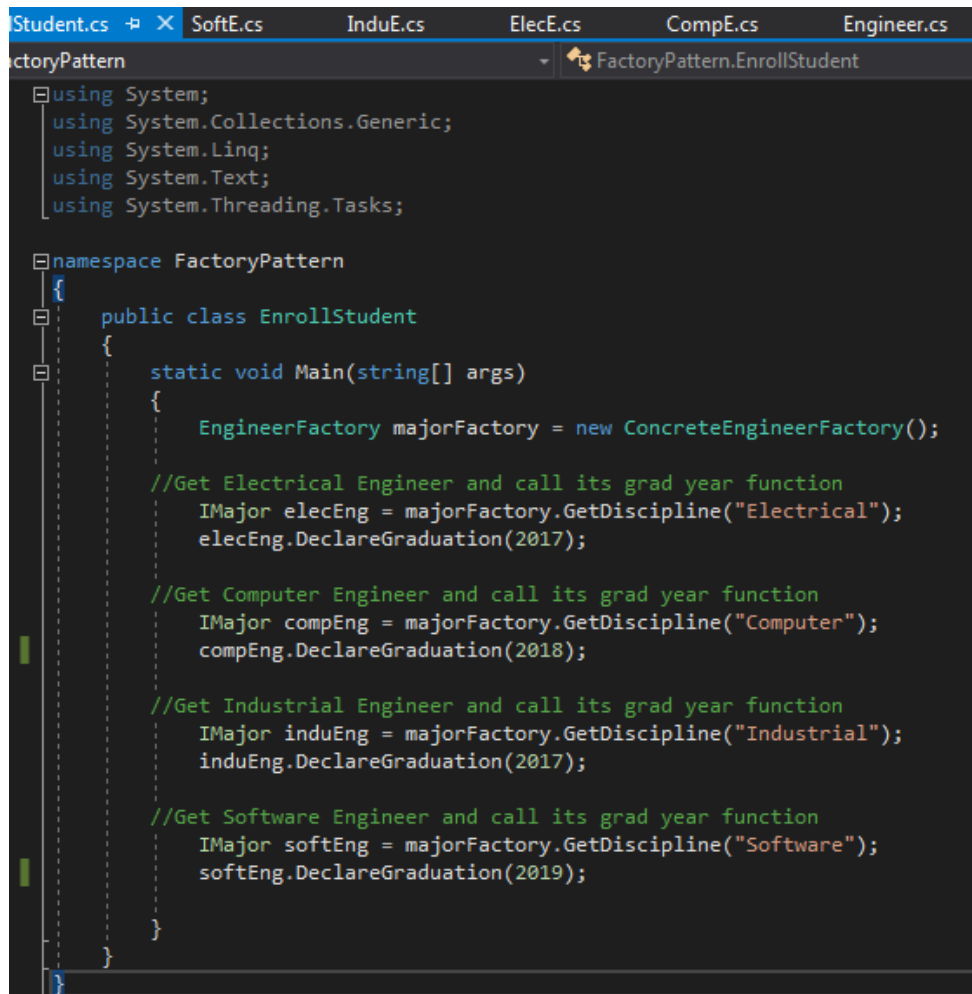
```
cs    InduE.cs    ElecE.cs    CompE.cs    Engineer.cs    EngineerFactory.cs  X
FactoryPattern
    FactoryPattern.EngineerFactory
    GetDiscipline(string discipline)
using System;

namespace FactoryPattern
{
    public abstract class EngineerFactory
    {
        public abstract IMajor GetDiscipline(string discipline);
    }

    public class ConcreteEngineerFactory : EngineerFactory
    {
        public override IMajor GetDiscipline(string discipline)
        {
            switch (discipline)
            {
                case "Computer":
                    return new CompE();
                case "Electrical":
                    return new ElecE();
                case "Industrial":
                    return new InduE();
                case "Software":
                    return new SoftE();
                default:
                    throw new ApplicationException(string.Format("Discipline '{0}' cannot be created", discipline));
            }
        }
    }
}
```

Figure 68: `EngineerFactory`

Out client code demonstrates how to use the factory pattern in a practical way. See Figure 69.



```
Student.cs  X  SoftE.cs  InduE.cs  ElecE.cs  CompE.cs  Engineer.cs
FactoryPattern
FactoryPattern.EnrollStudent
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryPattern
{
    public class EnrollStudent
    {
        static void Main(string[] args)
        {
            EngineerFactory majorFactory = new ConcreteEngineerFactory();

            //Get Electrical Engineer and call its grad year function
            IMajor elecEng = majorFactory.GetDiscipline("Electrical");
            elecEng.DeclareGraduation(2017);

            //Get Computer Engineer and call its grad year function
            IMajor compEng = majorFactory.GetDiscipline("Computer");
            compEng.DeclareGraduation(2018);

            //Get Industrial Engineer and call its grad year function
            IMajor induEng = majorFactory.GetDiscipline("Industrial");
            induEng.DeclareGraduation(2017);

            //Get Software Engineer and call its grad year function
            IMajor softEng = majorFactory.GetDiscipline("Software");
            softEng.DeclareGraduation(2019);
        }
    }
}
```

Figure 69: Client Code, Enroll Students

This client code generates a simple output based on the specific type of class object generated and the graduation year input. This is shown in Figure 70.

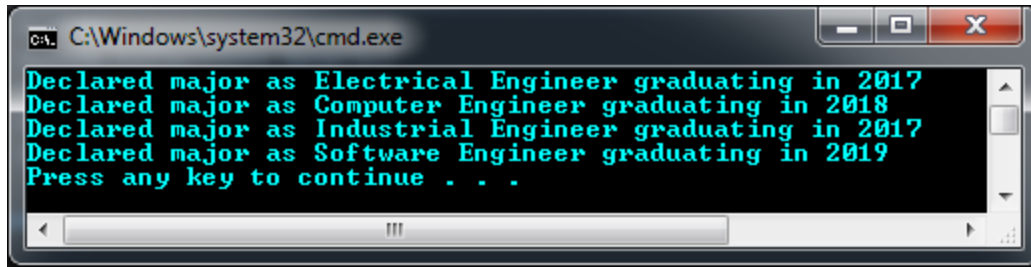


Figure 70: Enroll Students, Output

We then wrote the test for the client code which can be seen in Figure 71, below. The test creates an EngineerFactory object which is then used to create object of each engineering specialization. To determine if the concrete classes are being utilized correctly, we then assert that the engineer object is of the correct type.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace FactoryPattern.Tests
{
    [TestClass()]
    public class ConcreteEngineerFactoryTests
    {
        [TestMethod()]
        public void GetDisciplineTest()
        {
            EngineerFactory majorFactory = new ConcreteEngineerFactory();
            IMajor engineer;

            // Get Electrical Engineer
            engineer = majorFactory.GetDiscipline("Electrical");
            Assert.IsInstanceOfType(engineer, typeof(ElecE));
            engineer.DeclareGraduation(2017);

            // Get Computer Engineer
            engineer = majorFactory.GetDiscipline("Computer");
            Assert.IsInstanceOfType(engineer, typeof(CompE));
            engineer.DeclareGraduation(2017);

            // Get Industrial Engineer
            engineer = majorFactory.GetDiscipline("Industrial");
            Assert.IsInstanceOfType(engineer, typeof(InduE));
            engineer.DeclareGraduation(2017);

            // Get Software Engineer
            engineer = majorFactory.GetDiscipline("Software");
            Assert.IsInstanceOfType(engineer, typeof(SoftE));
            engineer.DeclareGraduation(2017);
        }
    }
}
```

Figure 71: Factory Pattern Test

After running all test, Figure 72 shows that all passed.

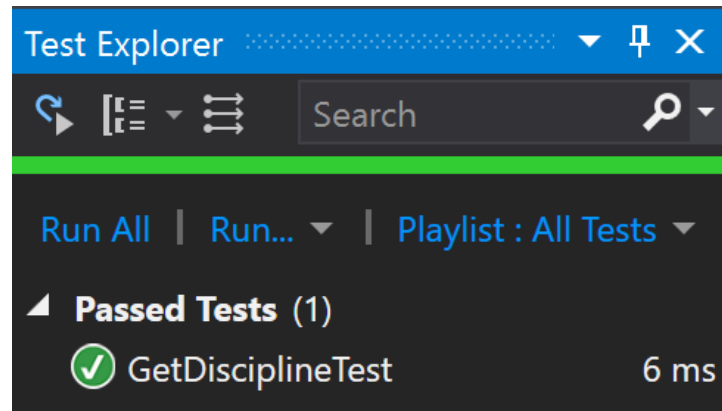


Figure 72: Passed Factory Pattern Test

As the above example has demonstrated, we were able to eliminate the use of the new operator using a factory. This has reduced coupling issues and removes the need for list of if-else statements to create instances of classes at runtime. The use of the factory pattern to create a class that is used to construct objects can be helpful for developers in projects where many similar objects need to be created.

References

- Chauhan. (2013). Decorator design Pattern – C#. Retrieved November 16, 2017, from <http://www.dotnettricks.com/learn/designpatterns/decorator-design-pattern-dotnet>
- Freeman, E., & Freeman, E. (2005). Head First Design Patterns. O'Reilly.
- Naidu, Damodhar. (2014). Decorator Pattern in C#. Retrieved November 16, 2017, from <http://www.c-sharpcorner.com/UploadFile/damubetha/decorator-pattern-in-csharp/>
- Singh, Rahul Rajat. (2012). Understanding and Implementing Decorator Pattern in C#. Retrieved November 14, 2017, from <https://www.codeproject.com/Articles/479635/UnderstandingplusandplusImplementingplusDecoratorp>
- Wagner, B. (n.d.). Interfaces (C# Programming Guide). Retrieved November 9, 2017, from <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/>
- Design Patterns and Refactoring. (n.d.). Retrieved November 16, 2017, from https://sourcemaking.com/design_patterns/strategy
- 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](https://msdn.microsoft.com/en-us/library/ms182524(v=vs.90).aspx)