**11/17/2017**

**Head First Design Patterns**

**Project 3 Report**

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

**Table of Contents**

[Table of Figures 3](#_Toc498673079)

[Observer Pattern 6](#_Toc498673080)

[Strategy Pattern 23](#_Toc498673081)

[Decorator Pattern 35](#_Toc498673082)

[Factory Pattern 48](#_Toc498673083)

[References 50](#_Toc498673084)

# **Table of Figures**

[Figure 1. Subject-Observer Relationship 5](#_Toc498674833)

[Figure 2. Initial Player Class 6](#_Toc498674834)

[Figure 3. Initial Subject Class 7](#_Toc498674835)

[Figure 4. Initial MatchData Class 8](#_Toc498674836)

[Figure 5. Initial Observer Class 8](#_Toc498674837)

[Figure 6. Initial PlayerStats Class 9](#_Toc498674838)

[Figure 7. Initial TeamStats Class 10](#_Toc498674839)

[Figure 8. Initial Failed Tests 11](#_Toc498674840)

[Figure 9. Initial Failed Tests 12](#_Toc498674841)

[Figure 10. Initial Failed Tests 12](#_Toc498674842)

[Figure 11. Initial Passed Tests 13](#_Toc498674843)

[Figure 12. Initial Passed Tests 14](#_Toc498674844)

[Figure 13. Initial Passed Tests 15](#_Toc498674845)

[Figure 14. Revised Player Class 16](#_Toc498674846)

[Figure 15. Revised Subject Class 16](#_Toc498674847)

[Figure 16. Revised MatchData Class 17](#_Toc498674848)

[Figure 17. Revised MatchData Class 18](#_Toc498674849)

[Figure 18. Revised Observer Class 18](#_Toc498674850)

[Figure 19. Revised PlayerStats Class 19](#_Toc498674851)

[Figure 20. Revised TeamStats Class 20](#_Toc498674852)

[Figure 21. Passed Tests 21](#_Toc498674853)

[Figure 22: Strategy Pattern Outline 22](#_Toc498674854)

[Figure 23: Non-pattern Multiple Method Calculator 24](#_Toc498674855)

[Figure 24: Non-pattern Multiple Method Calculator 24](#_Toc498674856)

[Figure 25: Non-pattern Single Method Calculator 25](#_Toc498674857)

[Figure 26: Non-pattern Single Method Calculator 25](#_Toc498674858)

[Figure 27: Interface Class 26](#_Toc498674859)

[Figure 28: AverageByMean Concrete Class 27](#_Toc498674860)

[Figure 29: AverageByMedian Concrete Class 27](#_Toc498674861)

[Figure 30: AverageByMode Concrete Class 28](#_Toc498674862)

[Figure 31: AverageByGeometric Concrete Class 28](#_Toc498674863)

[Figure 32: AverageByHarmonic Concrete Class 29](#_Toc498674864)

[Figure 33: Calculator Class 29](#_Toc498674865)

[Figure 34: Test Suite List 30](#_Toc498674866)

[Figure 35: Results Close Enough Function 30](#_Toc498674867)

[Figure 36: Test AverageByMean 31](#_Toc498674868)

[Figure 37: Test AverageByMedian 31](#_Toc498674869)

[Figure 38: Test AverageByMode 31](#_Toc498674870)

[Figure 39: Test AverageByGeometric 32](#_Toc498674871)

[Figure 40: Test AverageByHarmonic 32](#_Toc498674872)

[Figure 41: Calculator Passed Test 33](#_Toc498674873)

[Figure 42: Overpopulated Subclass Map 34](#_Toc498674874)

[Figure 43: Decorator Implementation Outline 35](#_Toc498674875)

[Figure 44: Initial IFood Interface 36](#_Toc498674876)

[Figure 45: Cake and Pancake Classes 37](#_Toc498674877)

[Figure 46: Cake with Toppings Classes 38](#_Toc498674878)

[Figure 47: Cake with Toppings Classes (cont.) 38](#_Toc498674879)

[Figure 48: Pancakes with Toppings Classes 39](#_Toc498674880)

[Figure 49: Pancakes with Toppings Classes (cont.) 39](#_Toc498674881)

[Figure 50: Initialize Cake and Pancake Objects 40](#_Toc498674882)

[Figure 51: Cake and Pancake Objects Output 40](#_Toc498674883)

[Figure 52: Generate Cake and Pancakes with Toppings Objects 41](#_Toc498674884)

[Figure 53: Cake and Pancakes with Toppings Output 41](#_Toc498674885)

[Figure 54: Initial Decorator 42](#_Toc498674886)

[Figure 55: Concrete Decorator, Scent 42](#_Toc498674887)

[Figure 56: Concrete Decorator, Strawberry 43](#_Toc498674888)

[Figure 57: Concrete Decorator, Cream 43](#_Toc498674889)

[Figure 58: Generate Cake with Decorators Object 44](#_Toc498674890)

[Figure 59: Cake with Decorators Output 44](#_Toc498674891)

[Figure 60: Generate Pancakes with Decorators Object 45](#_Toc498674892)

[Figure 61: Pancakes with Decorators Output 45](#_Toc498674893)

[Figure 62. Factory Pattern Example 48](#_Toc498674894)

# **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.

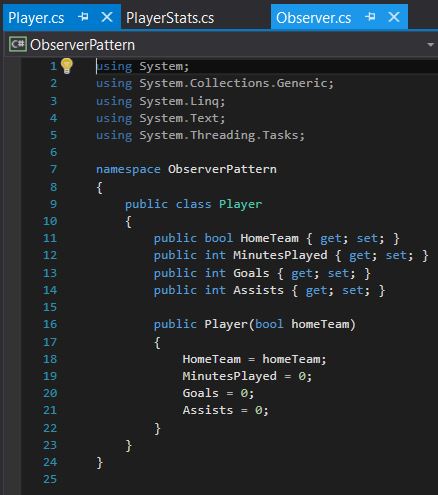


Figure 2. Initial Player Class

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

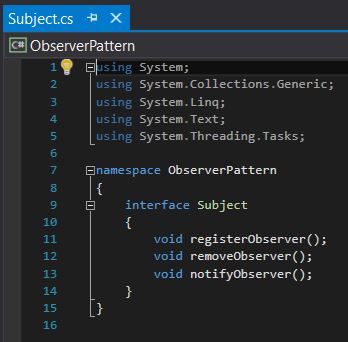


Figure 3. Initial Subject Class

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.

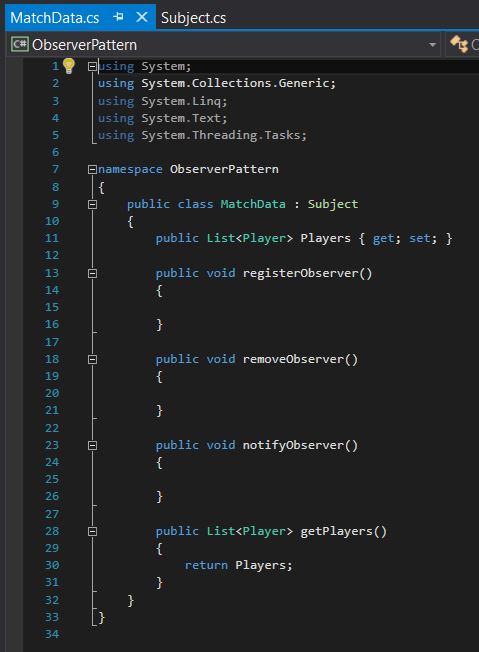


Figure 4. Initial MatchData Class

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

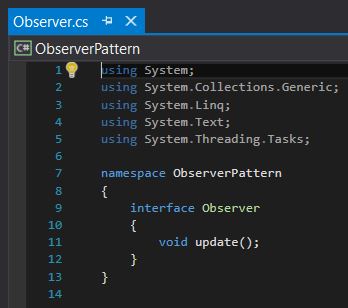


Figure 5. Initial Observer Class

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.

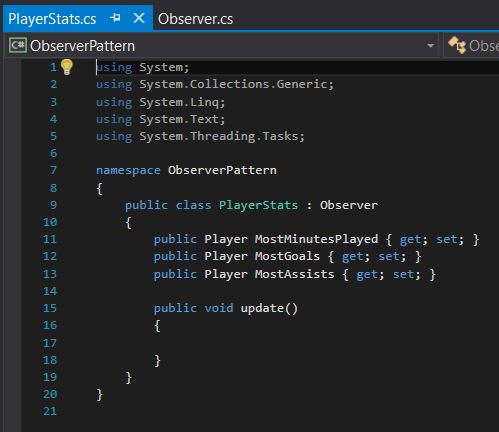


Figure 6. Initial PlayerStats Class

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.

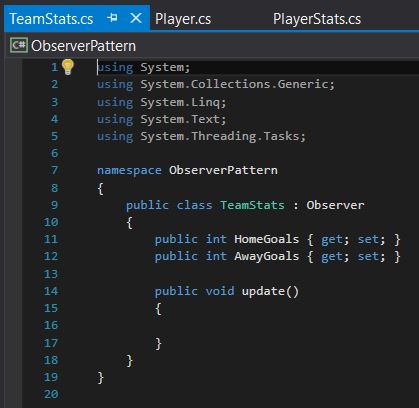


Figure 7. Initial TeamStats Class

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.

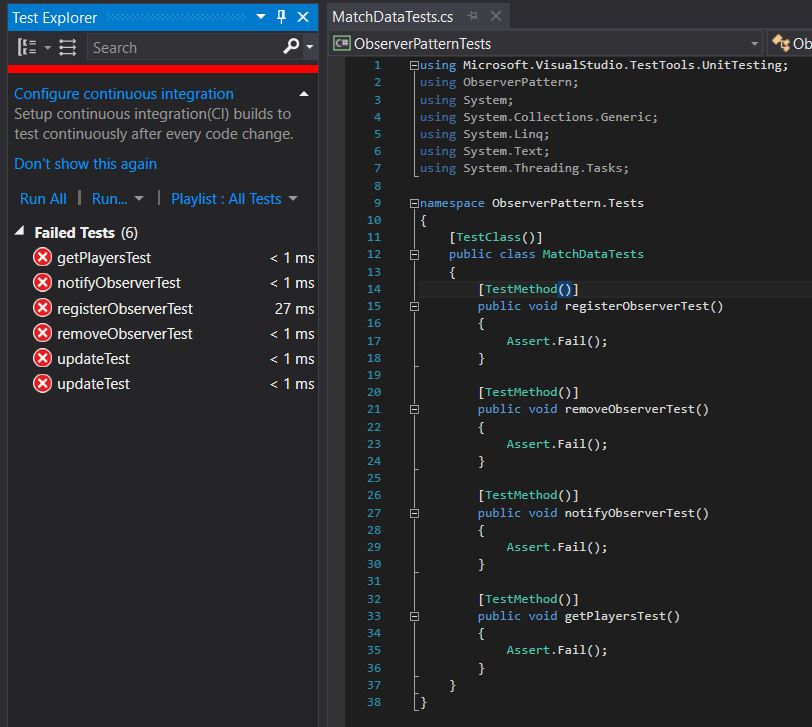


Figure 8. Initial Failed Tests

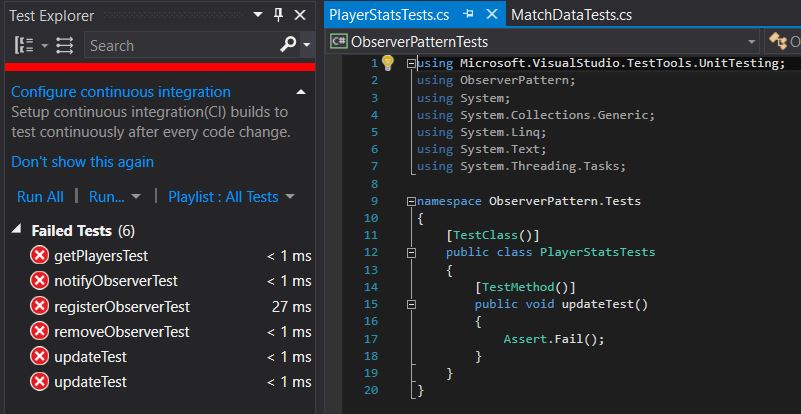


Figure 9. Initial Failed Tests



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.

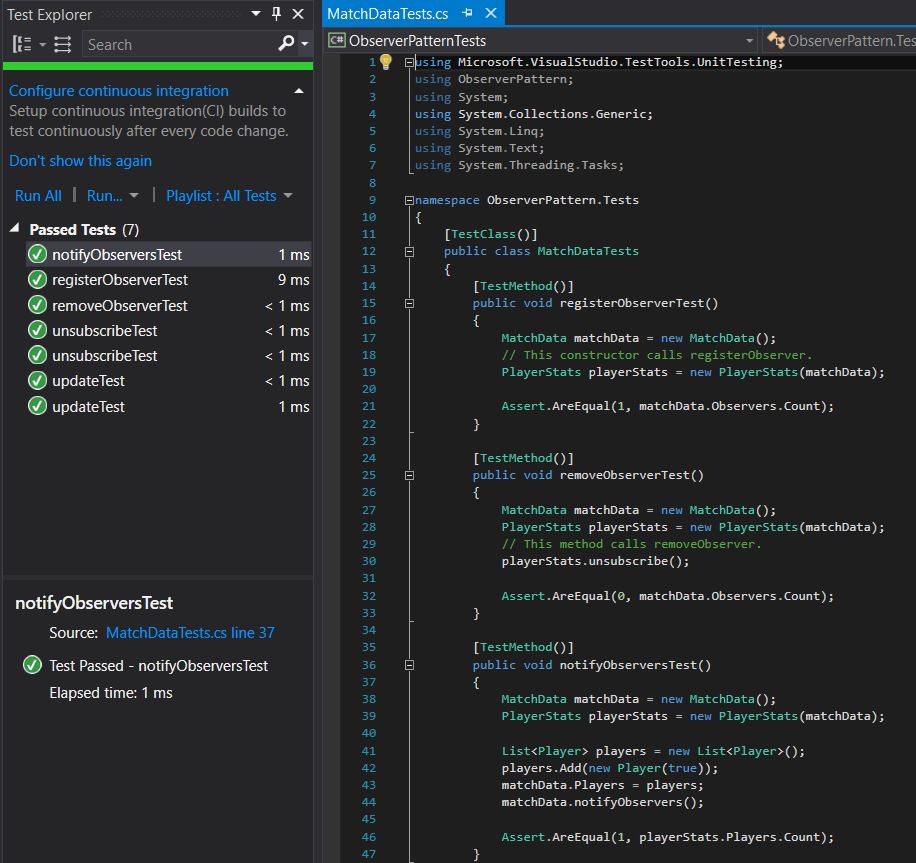


Figure 11. Initial Passed Tests

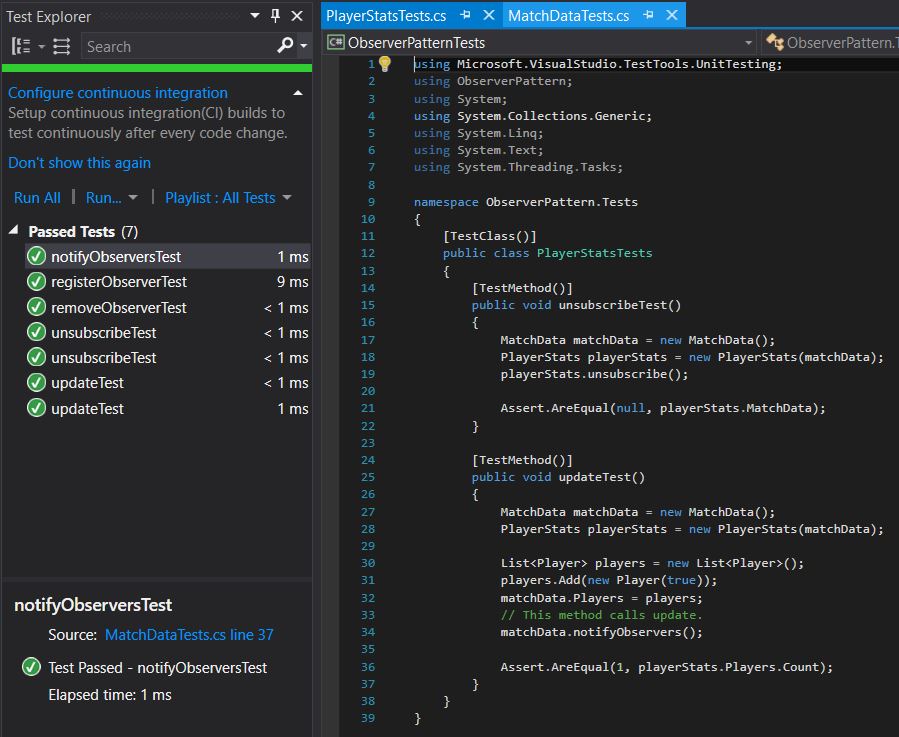


Figure 12. Initial Passed Tests

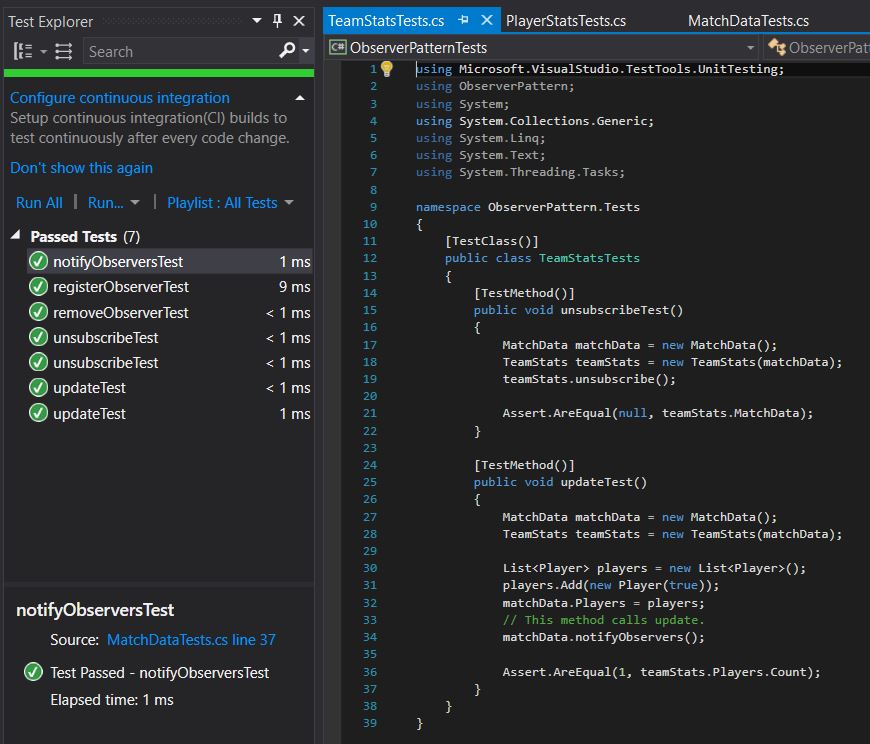


Figure 13. Initial Passed Tests

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.

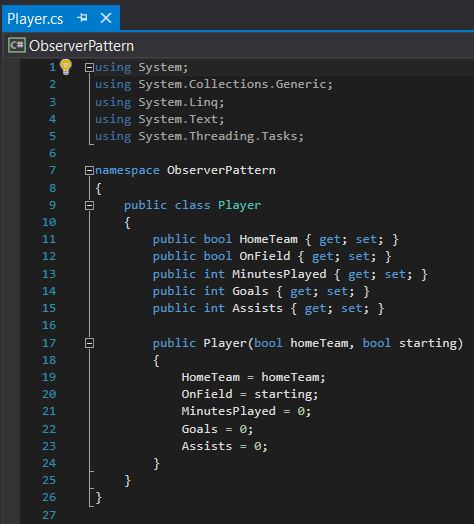


Figure 14. Revised Player Class

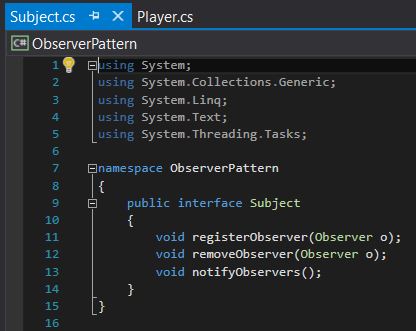


Figure 15. Revised Subject Class

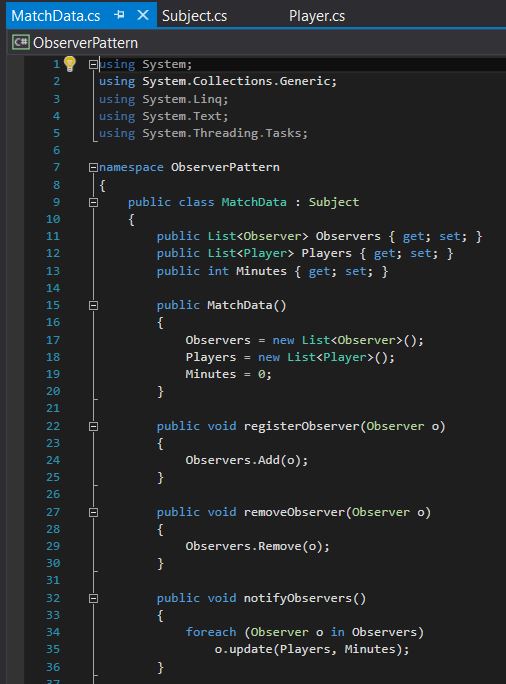


Figure 16. Revised MatchData Class



Figure 17. Revised MatchData Class

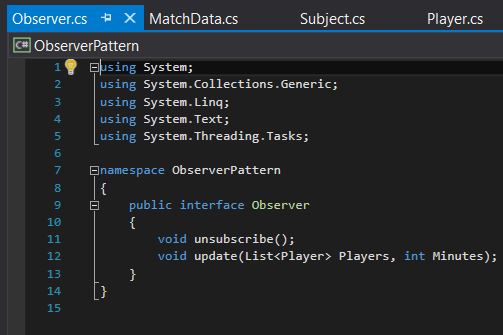


Figure 18. Revised Observer Class

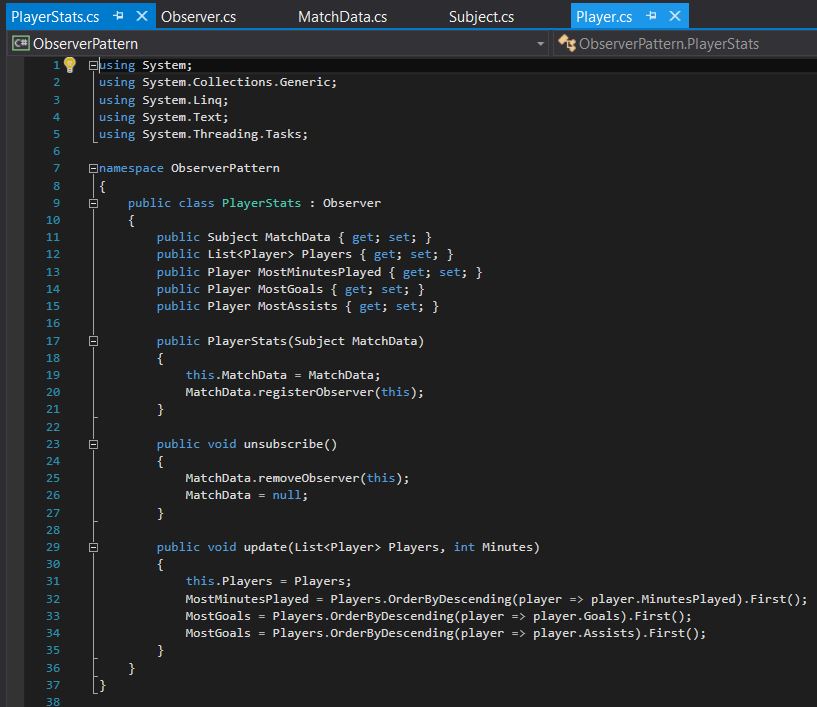


Figure 19. Revised PlayerStats Class

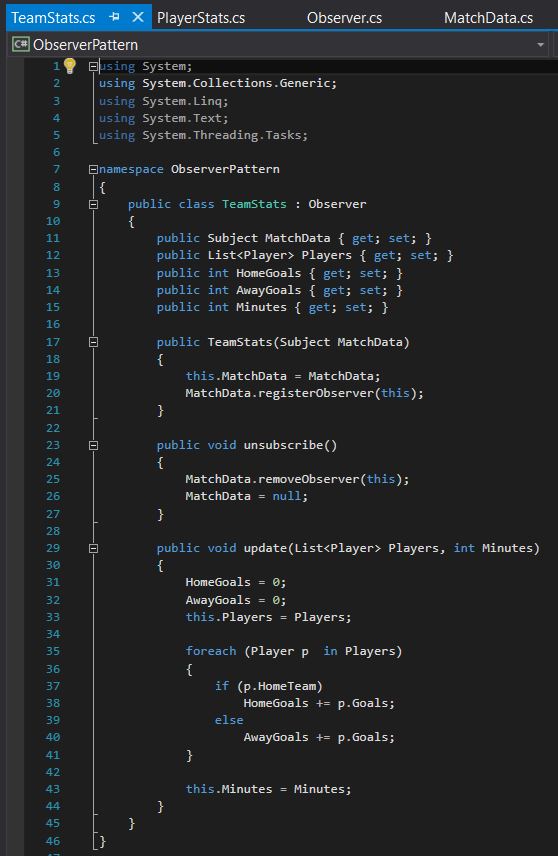


Figure 20. Revised TeamStats Class

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.

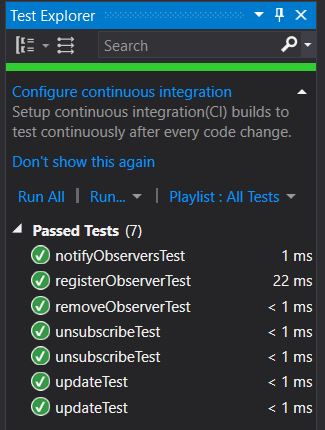


Figure 21. Passed Tests

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.

# **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.

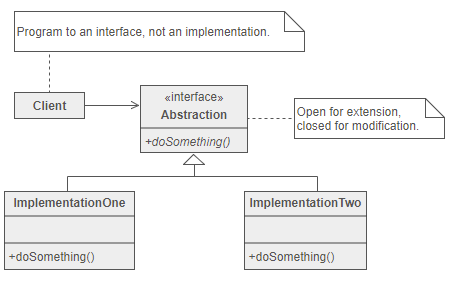


Figure 22: Strategy Pattern Outline

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.

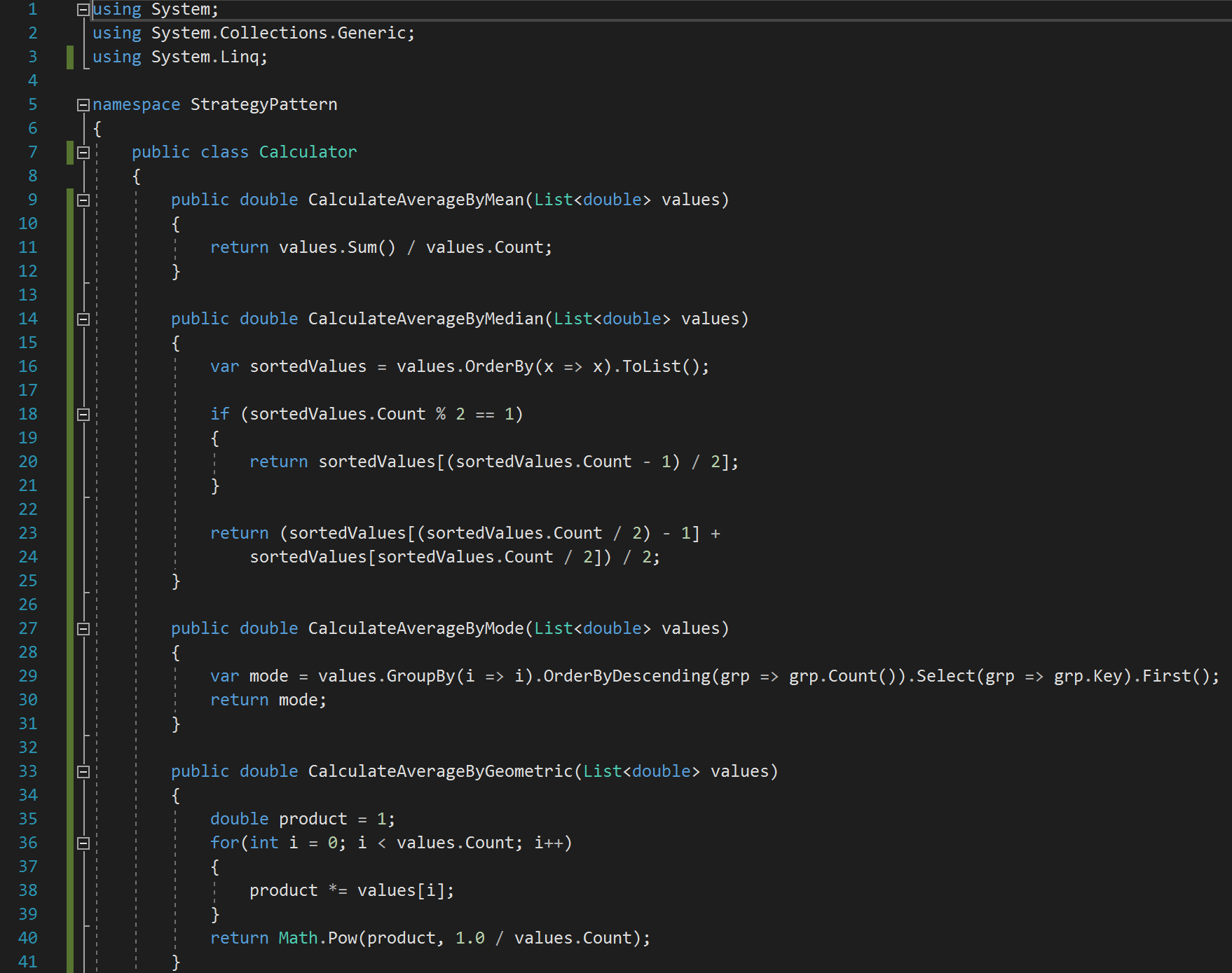


Figure 23: Non-pattern Multiple Method Calculator

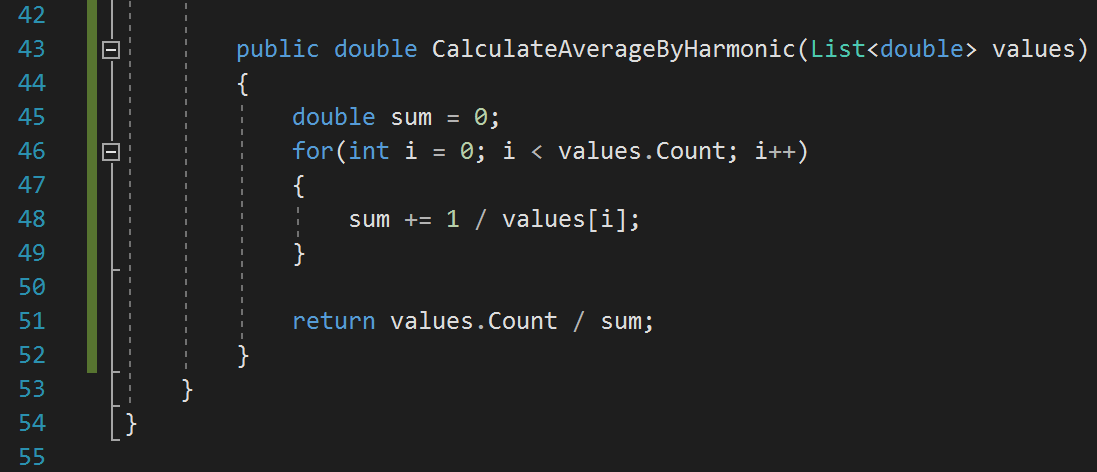


Figure 24: Non-pattern Multiple Method Calculator

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.

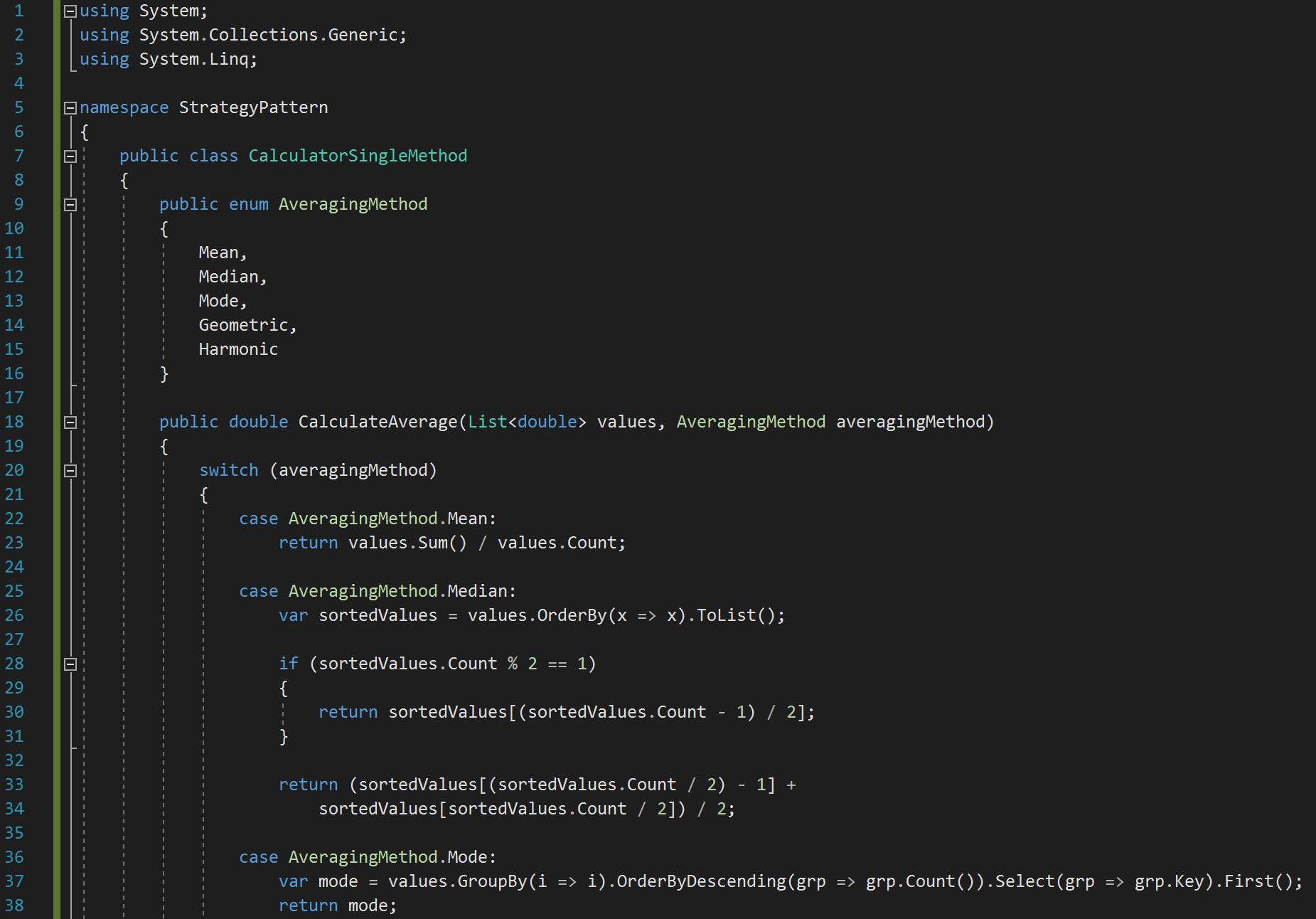


Figure 25: Non-pattern Single Method Calculator

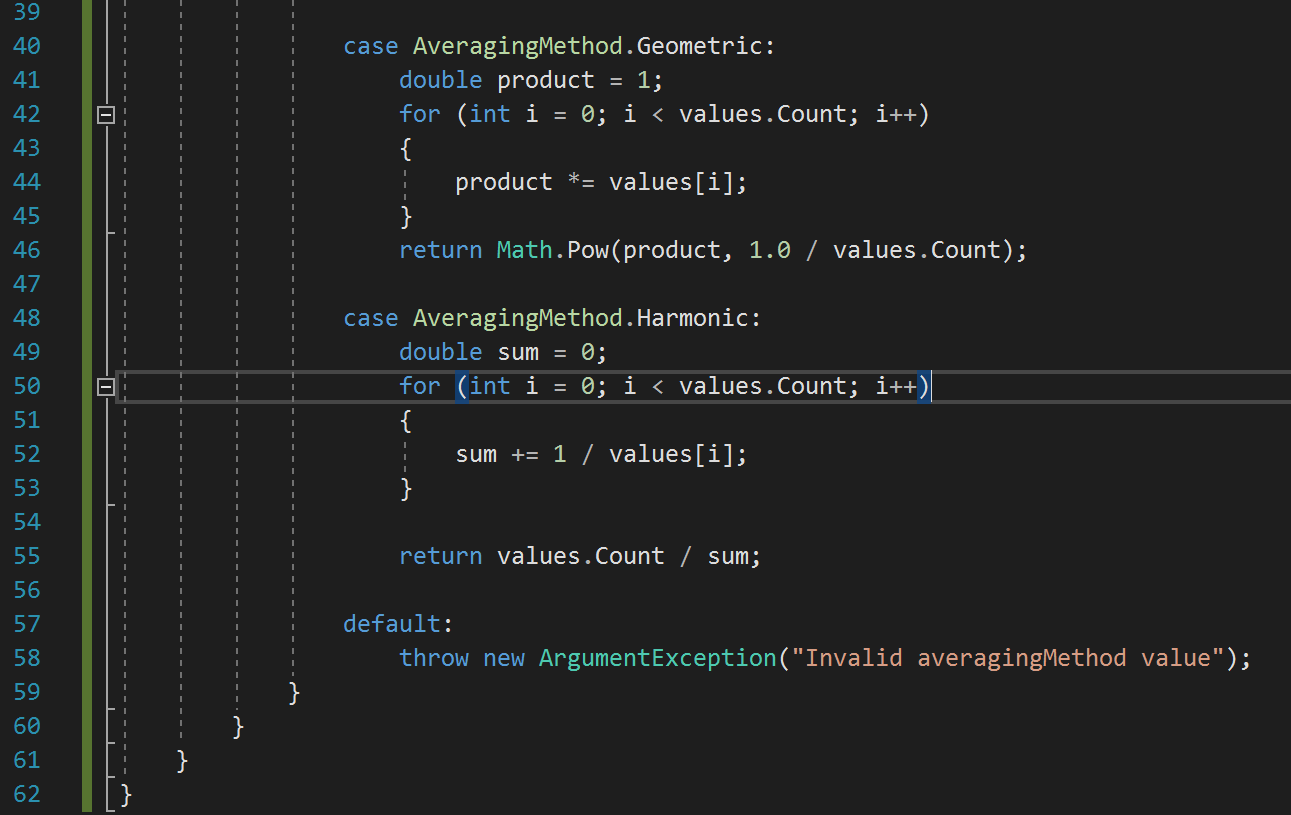


Figure 26: Non-pattern Single Method Calculator

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.

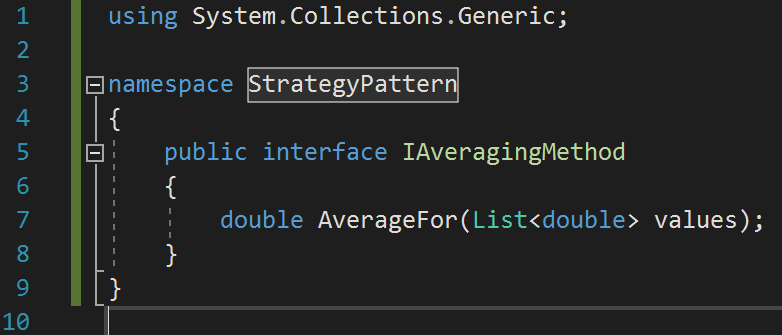


Figure 27: Interface Class

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

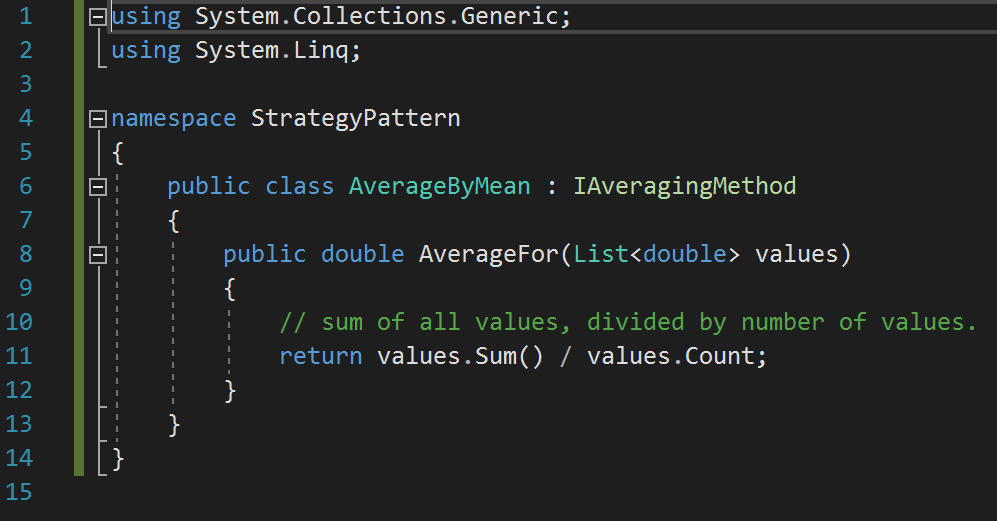


Figure 28: AverageByMean Concrete Class



Figure 29: AverageByMedian Concrete Class

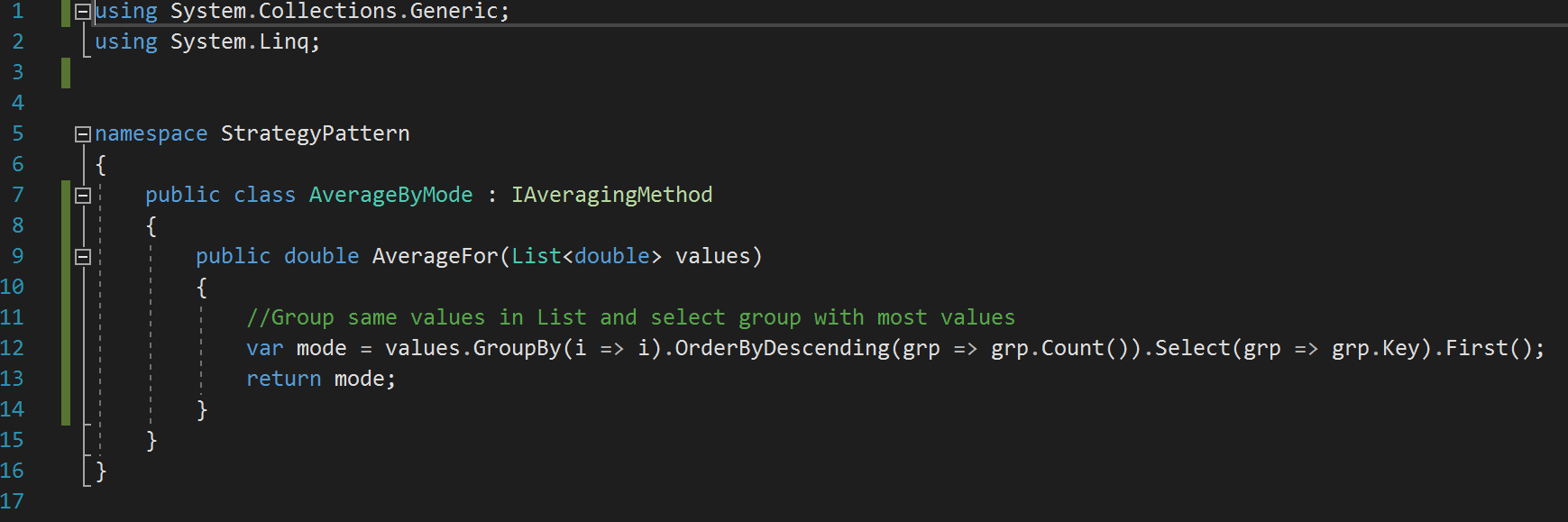


Figure 30: AverageByMode Concrete Class

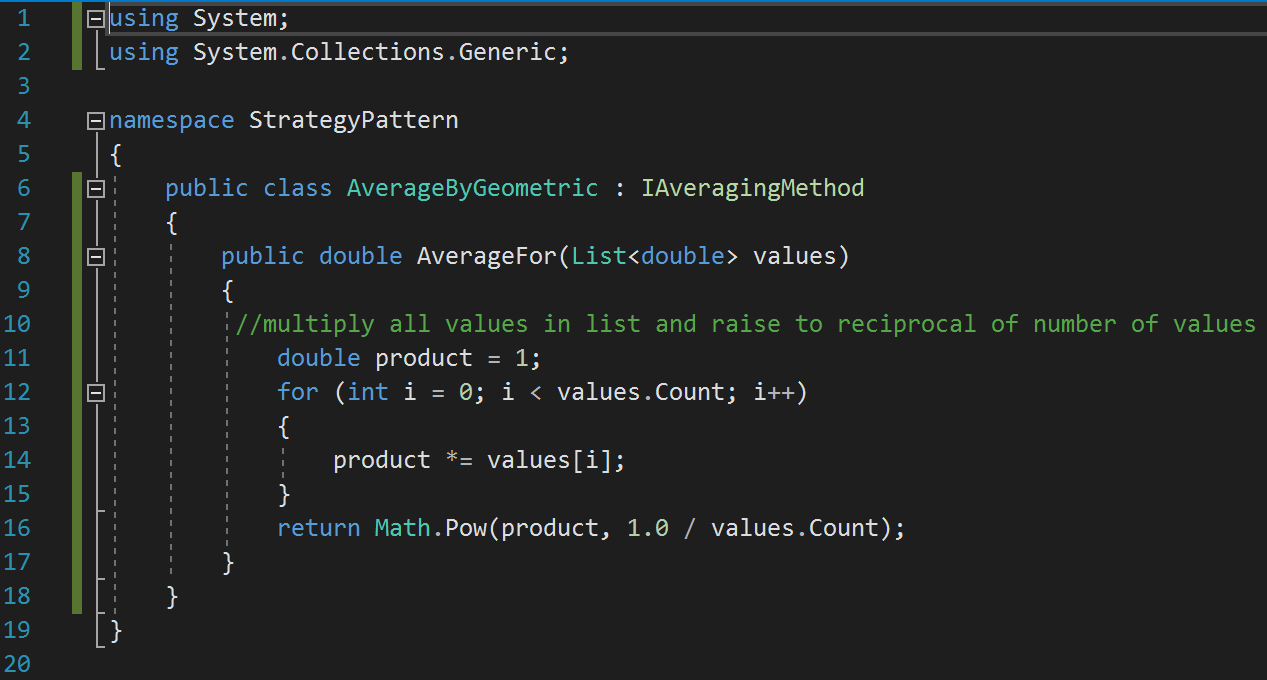


Figure 31: AverageByGeometric Concrete Class

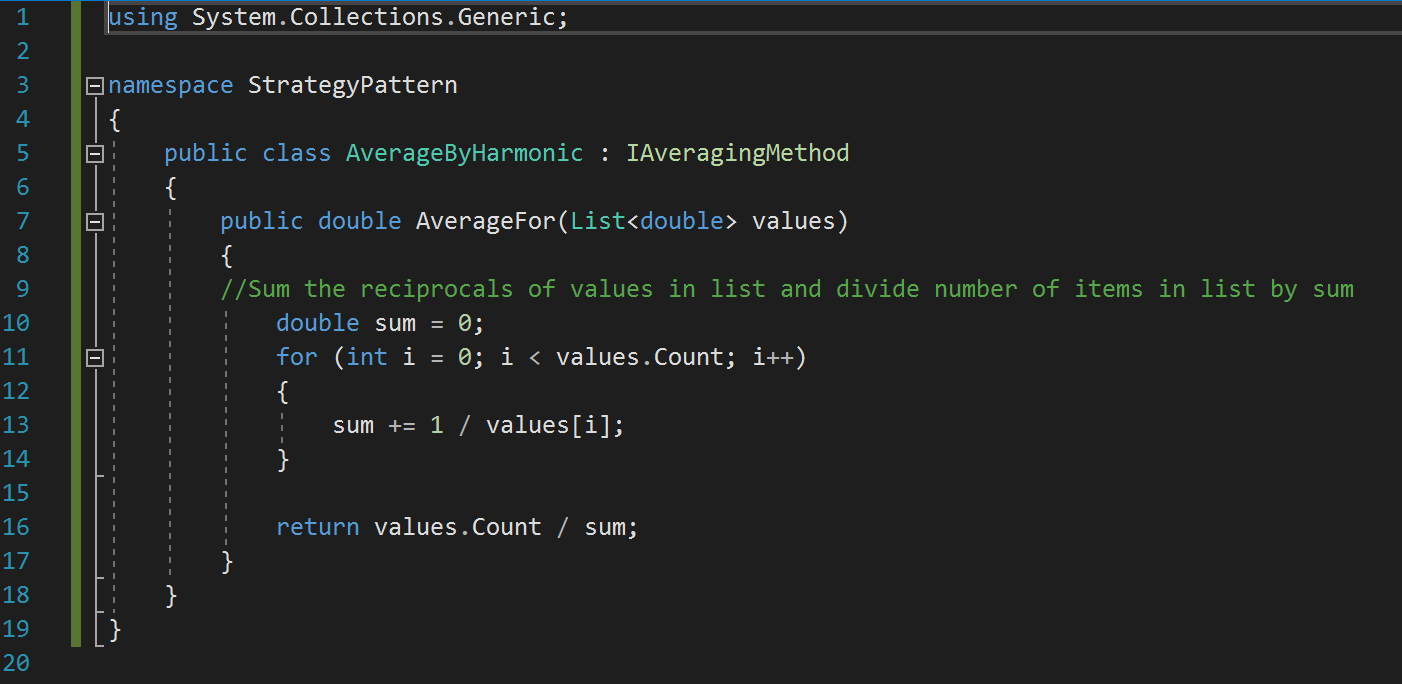


Figure 32: AverageByHarmonic Concrete Class

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.

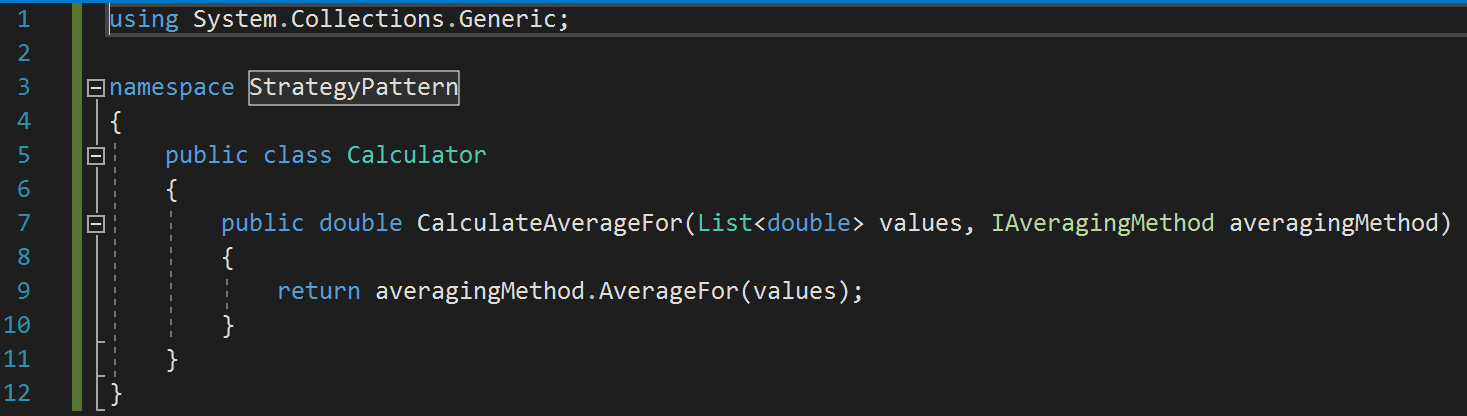


Figure 33: Calculator Class

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 concreate 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.

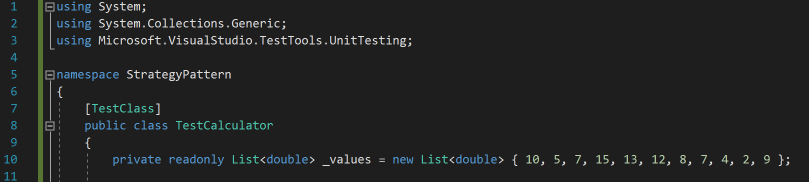


Figure 34: Test Suite List

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.

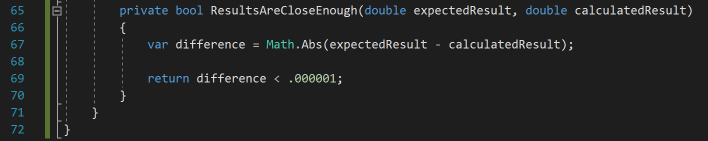


Figure 35: Results Close Enough Function

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.

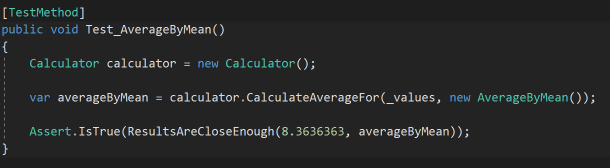


Figure 36: Test AverageByMean

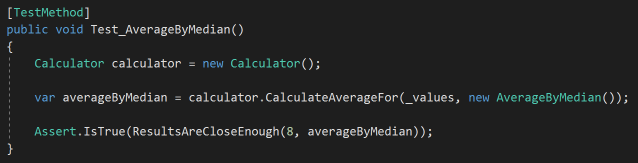


Figure 37: Test AverageByMedian

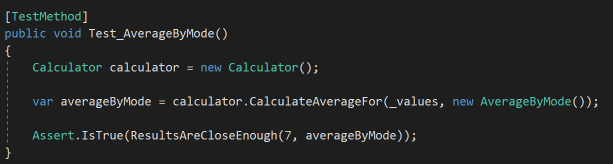


Figure 38: Test AverageByMode

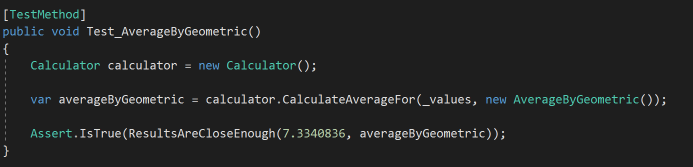


Figure 39: Test AverageByGeometric

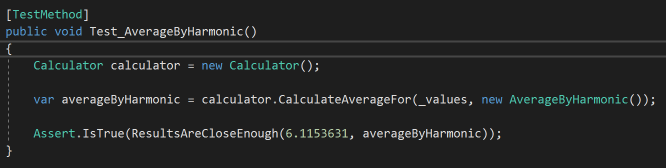


Figure 40: Test AverageByHarmonic

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

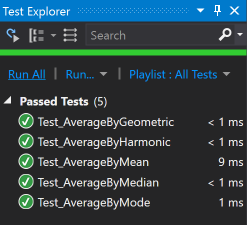


Figure 41: Calculator Passed Test

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.

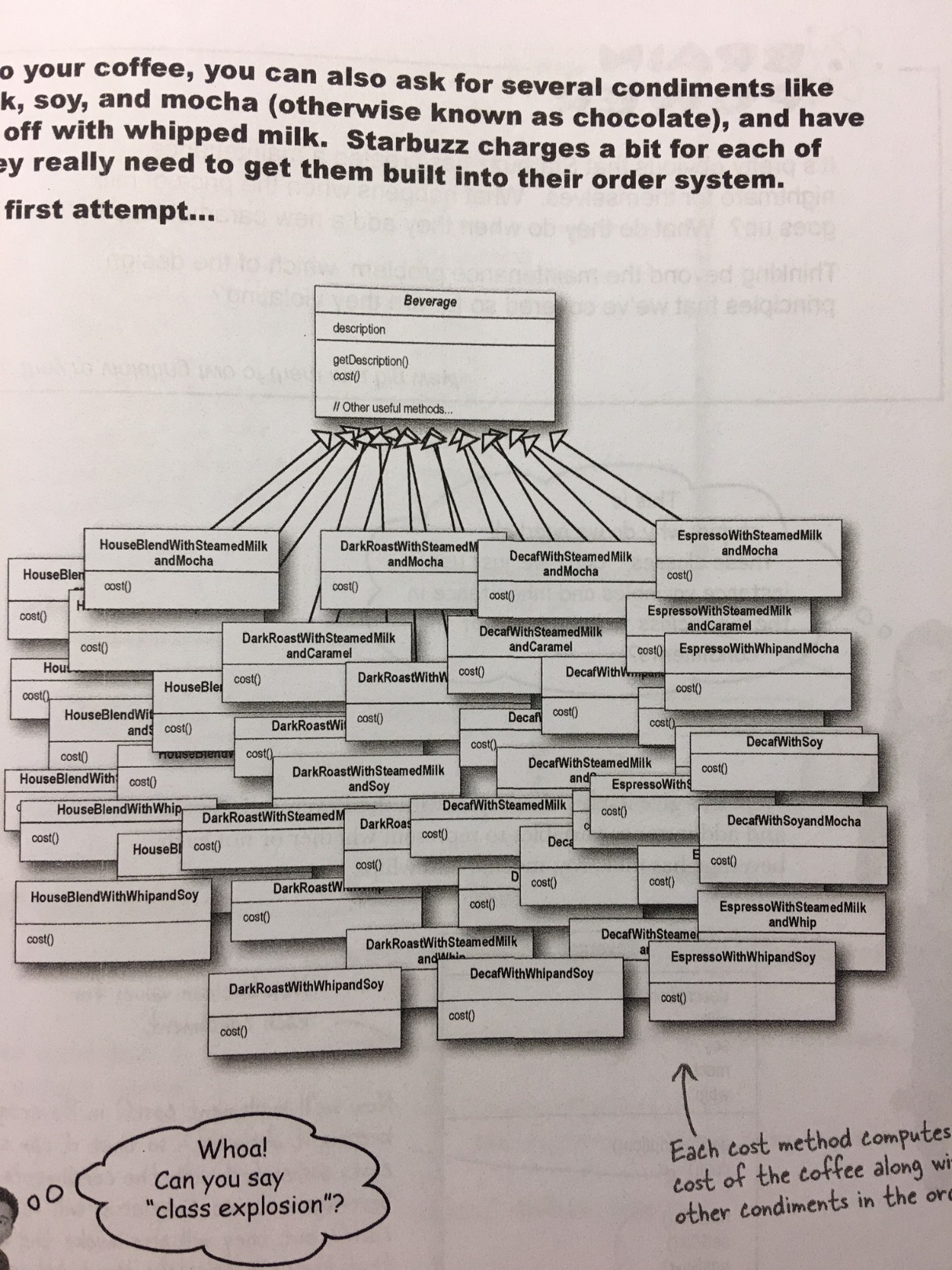


Figure 42: Overpopulated Subclass Map

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.

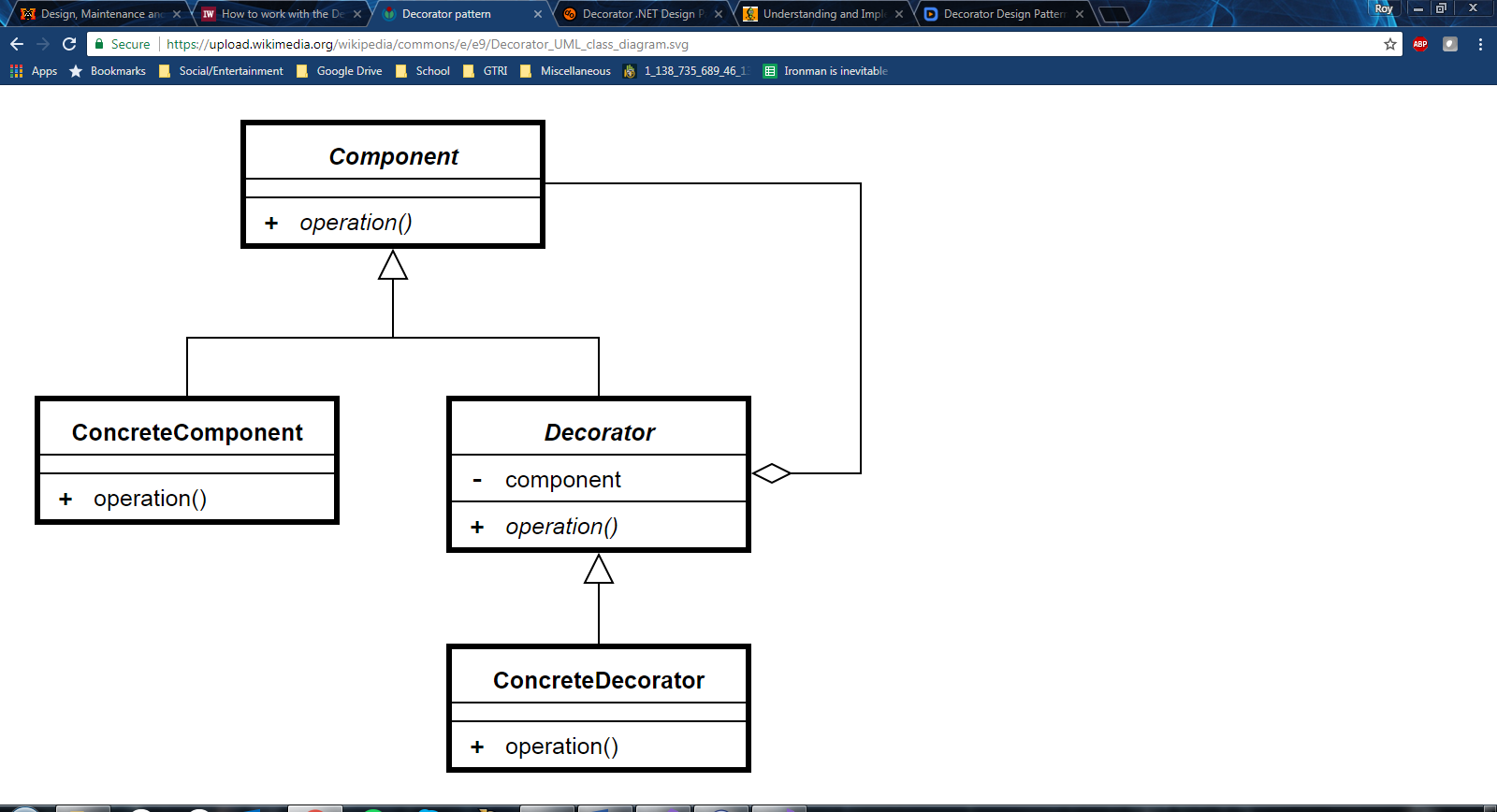


Figure 43: Decorator Implementation Outline

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.

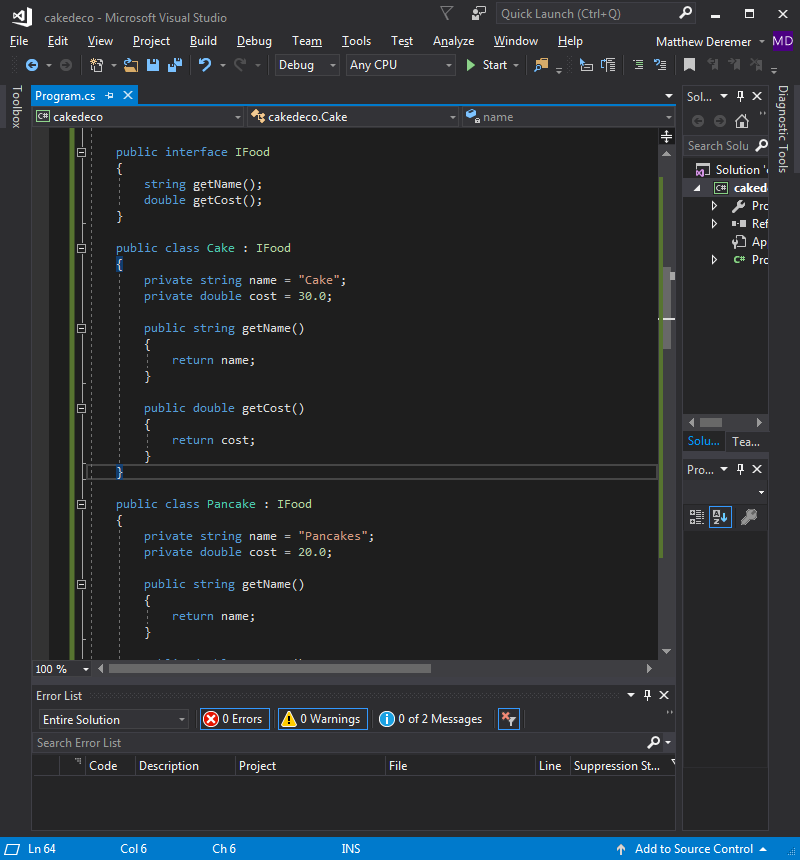


Figure 44: Initial IFood Interface

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

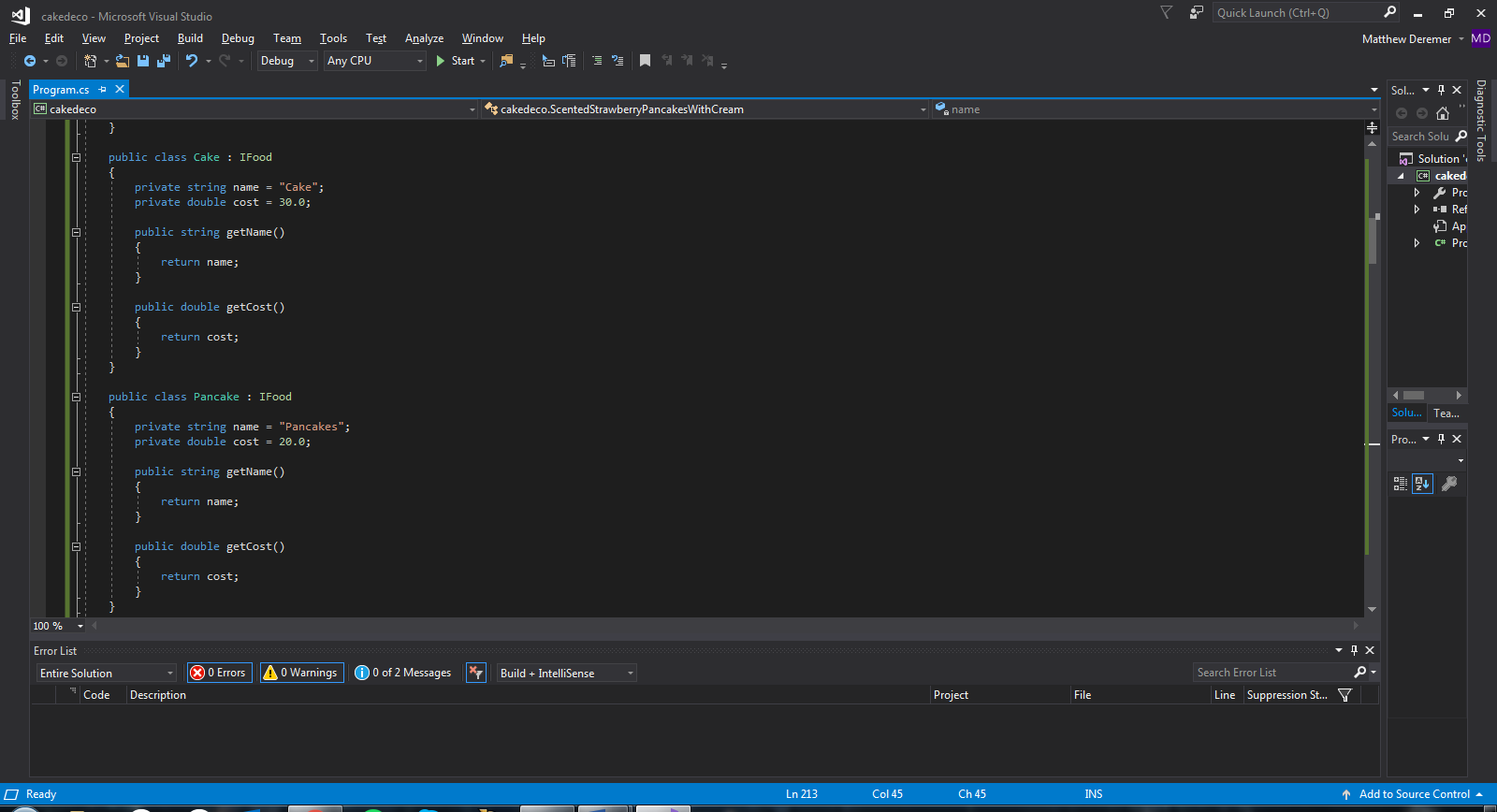


Figure 45: Cake and Pancake Classes

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.

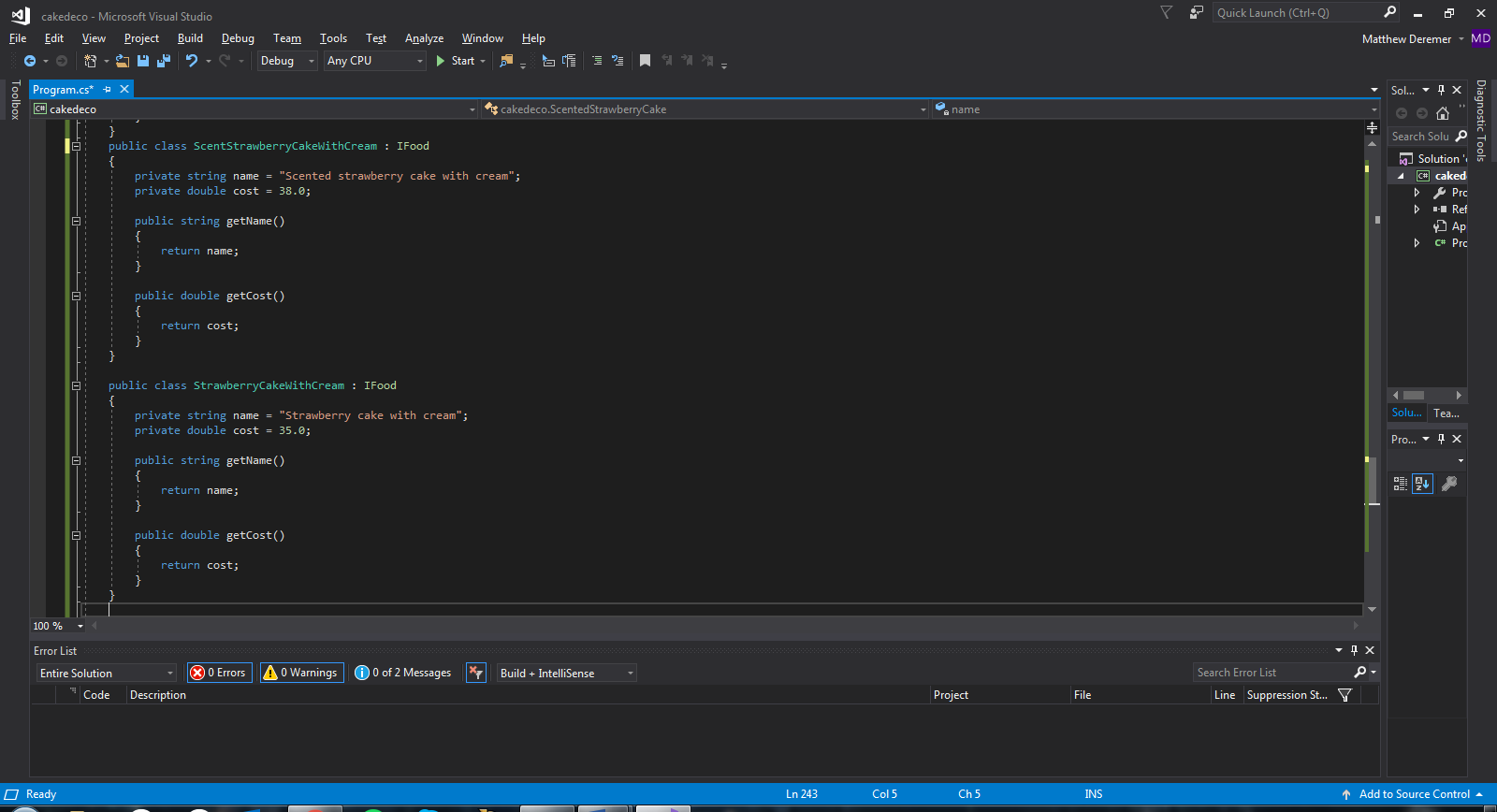


Figure 46: Cake with Toppings Classes

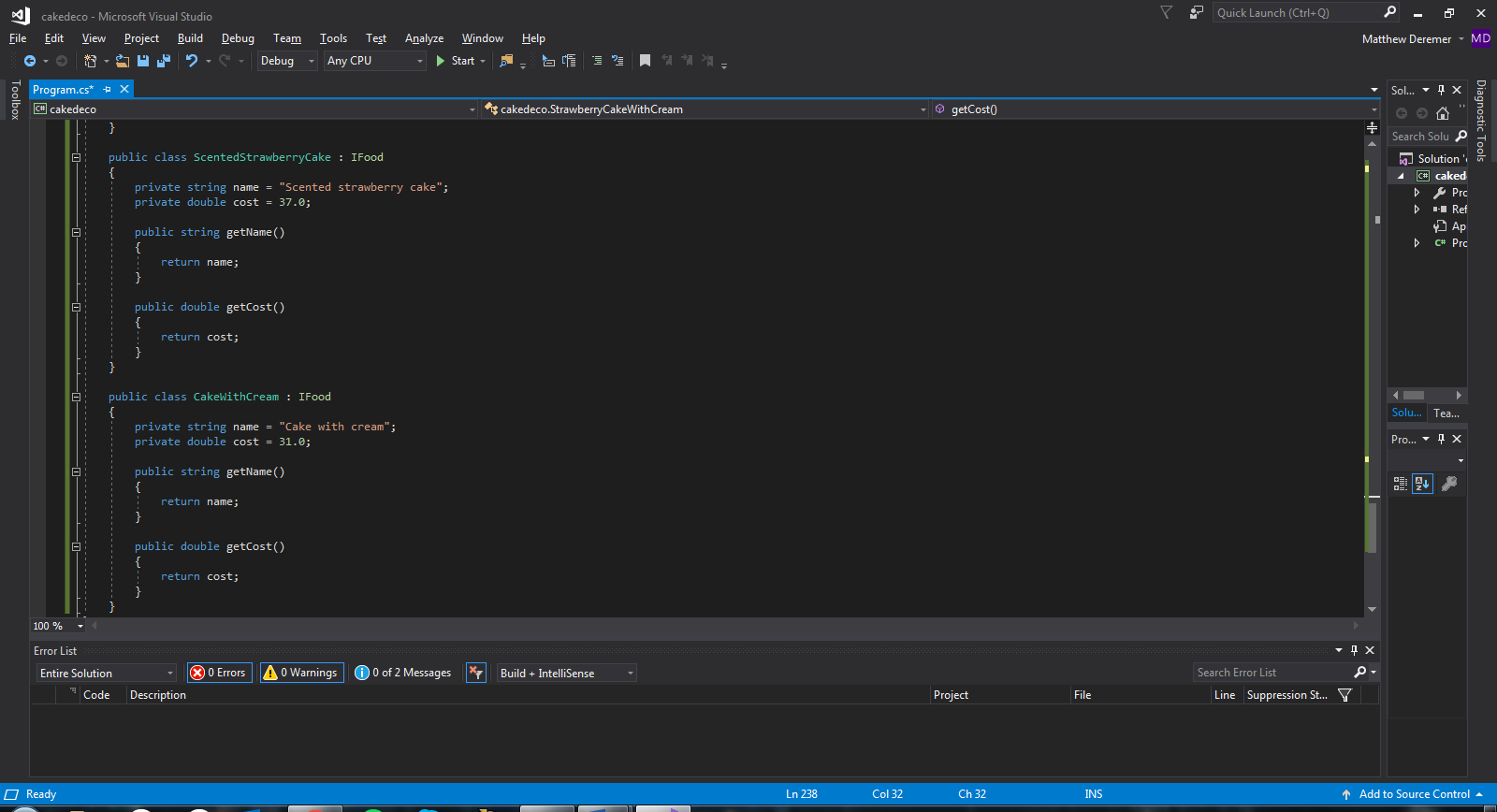


Figure 47: Cake with Toppings Classes (cont.)

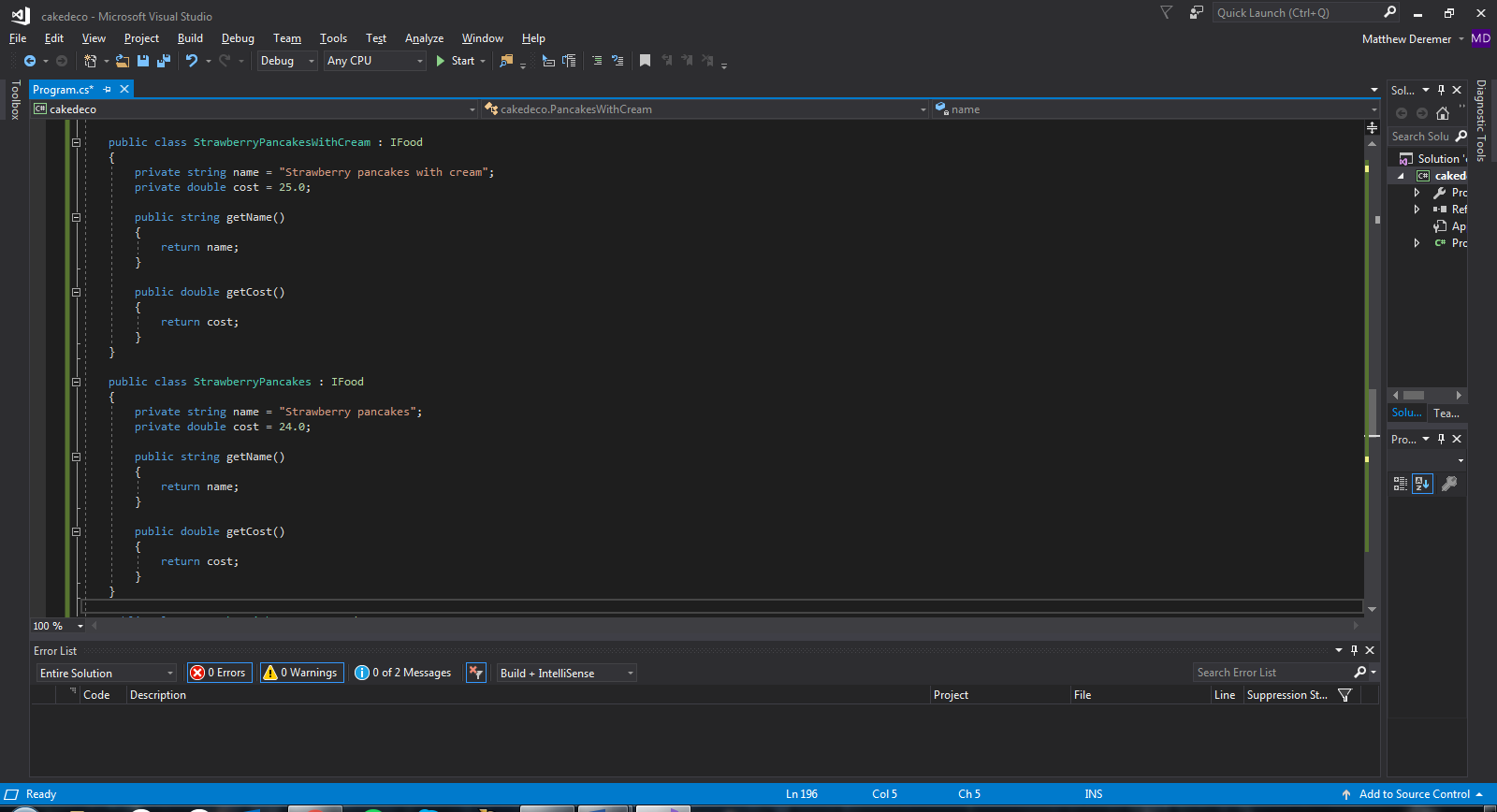


Figure 48: Pancakes with Toppings Classes

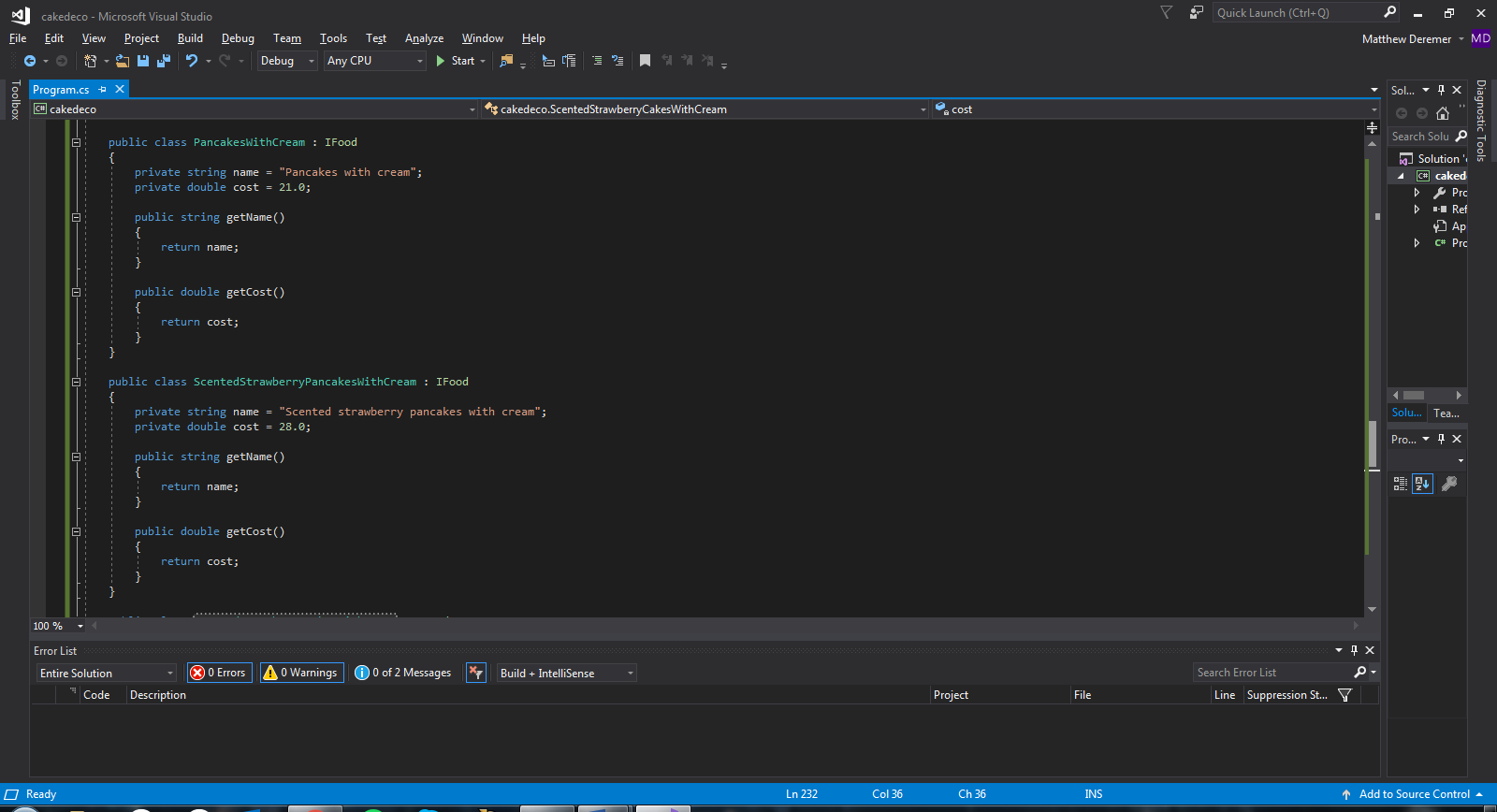


Figure 49: Pancakes with Toppings Classes (cont.)

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.

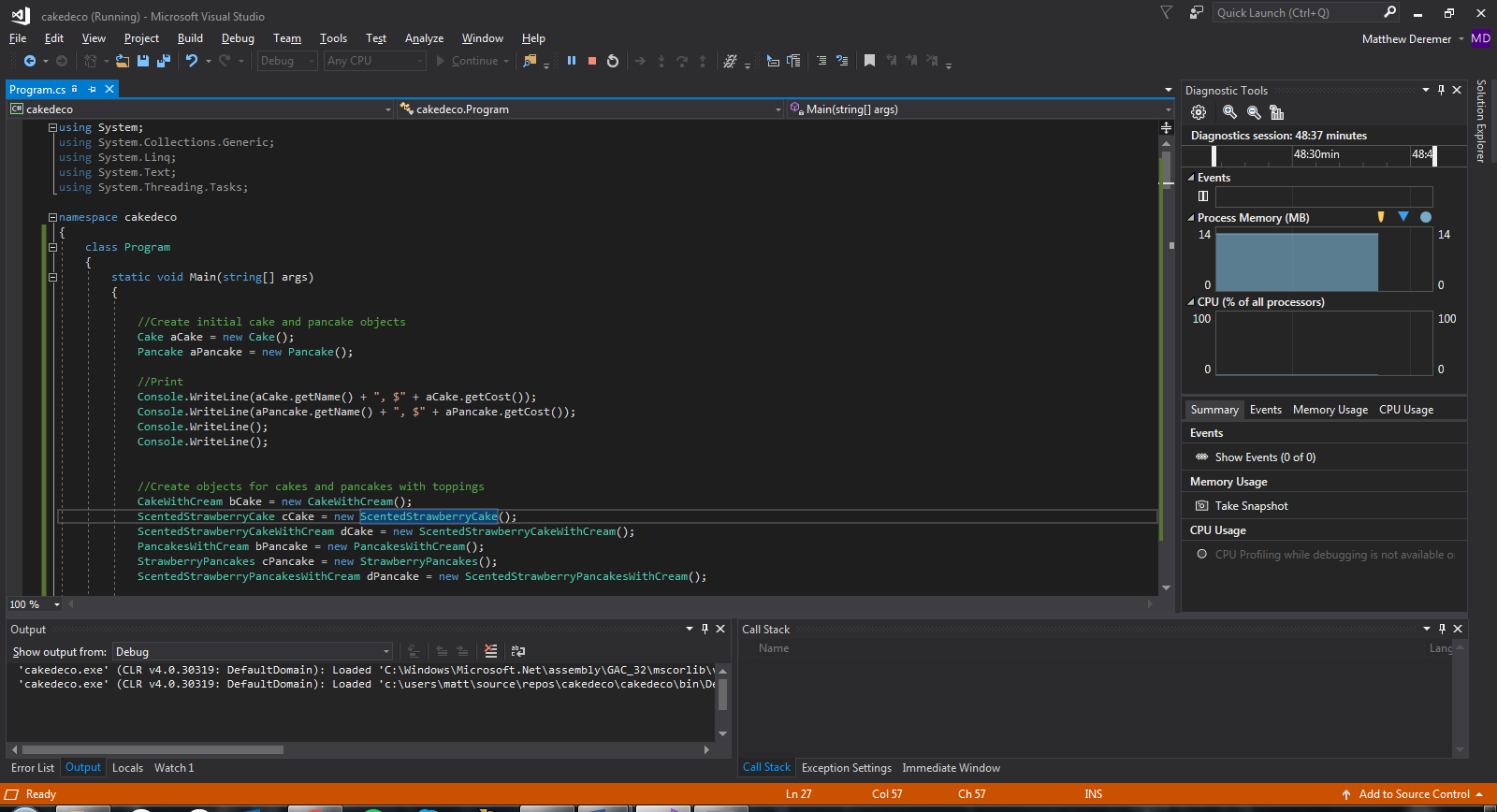


Figure 50: Initialize Cake and Pancake Objects

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

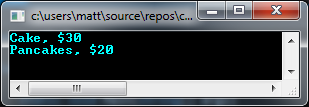


Figure 51: Cake and Pancake Objects Output

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

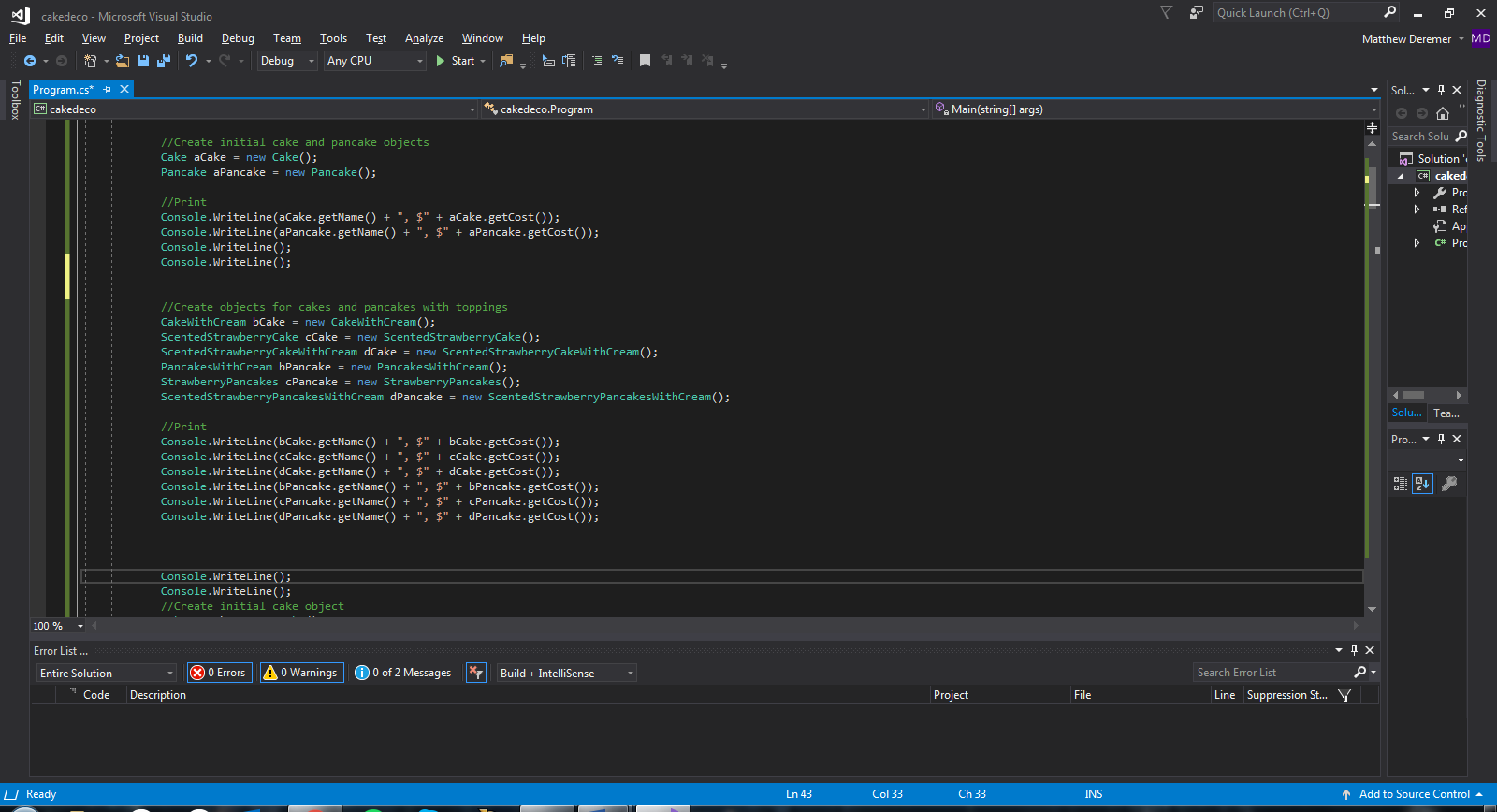


Figure 52: Generate Cake and Pancakes with Toppings Objects

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

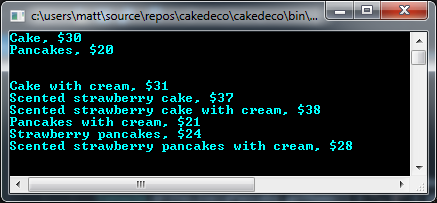


Figure 53: Cake and Pancakes with Toppings Output

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.

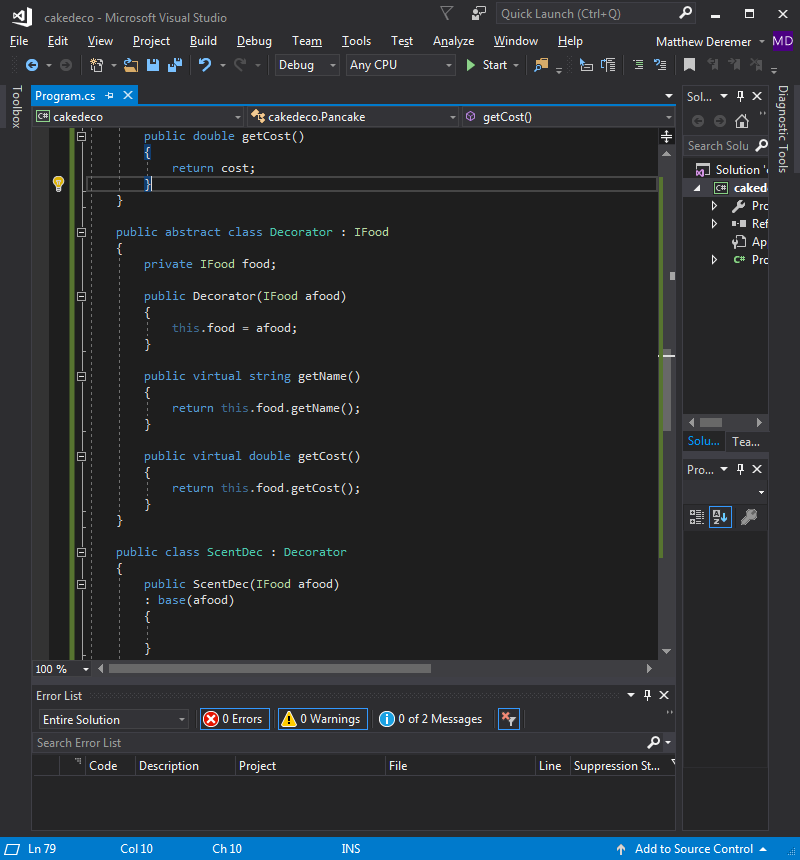


Figure 54: Initial Decorator

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.

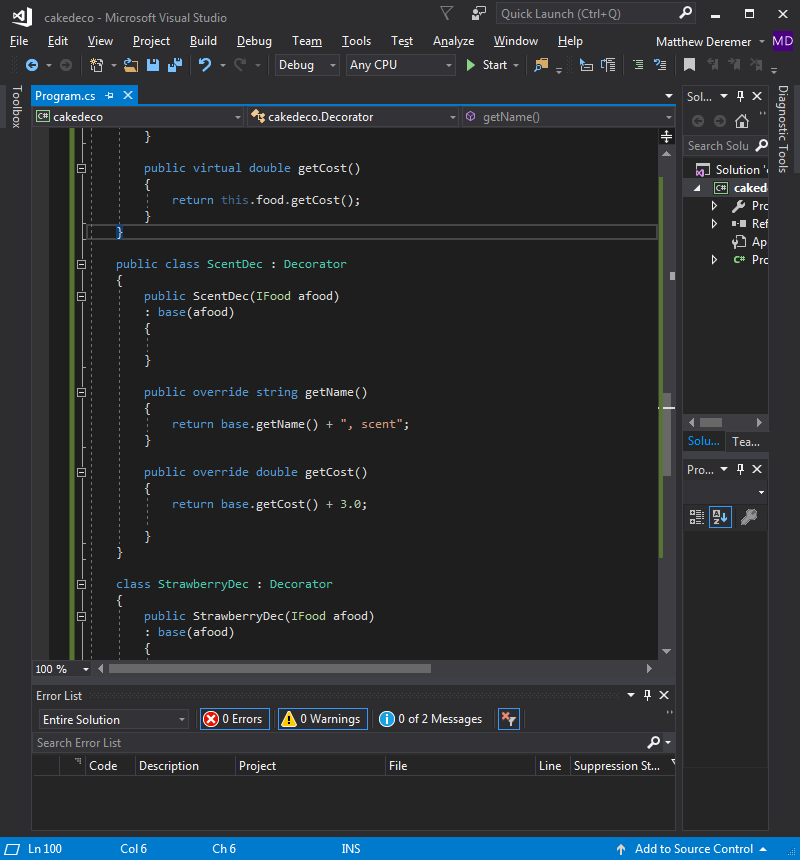


Figure 55: Concrete Decorator, Scent

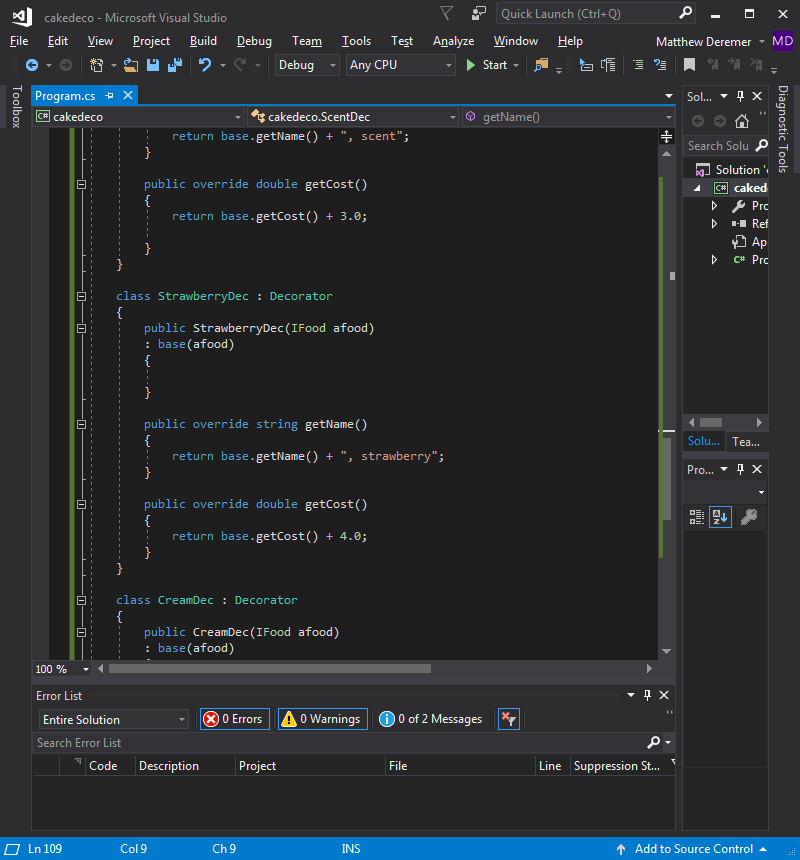


Figure 56: Concrete Decorator, Strawberry

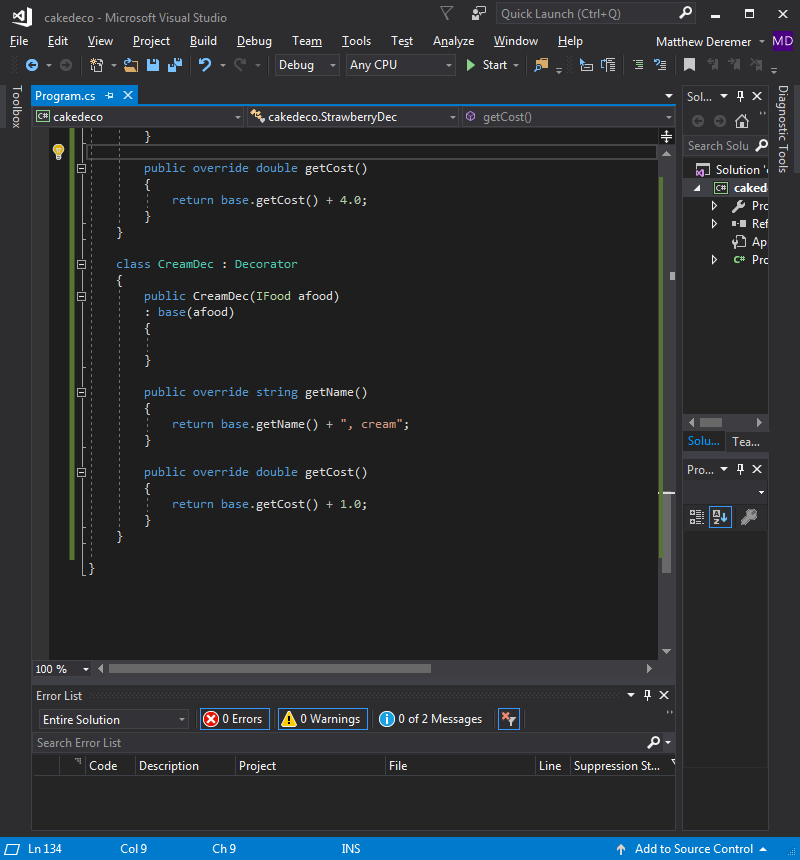


Figure 57: Concrete Decorator, Cream

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.

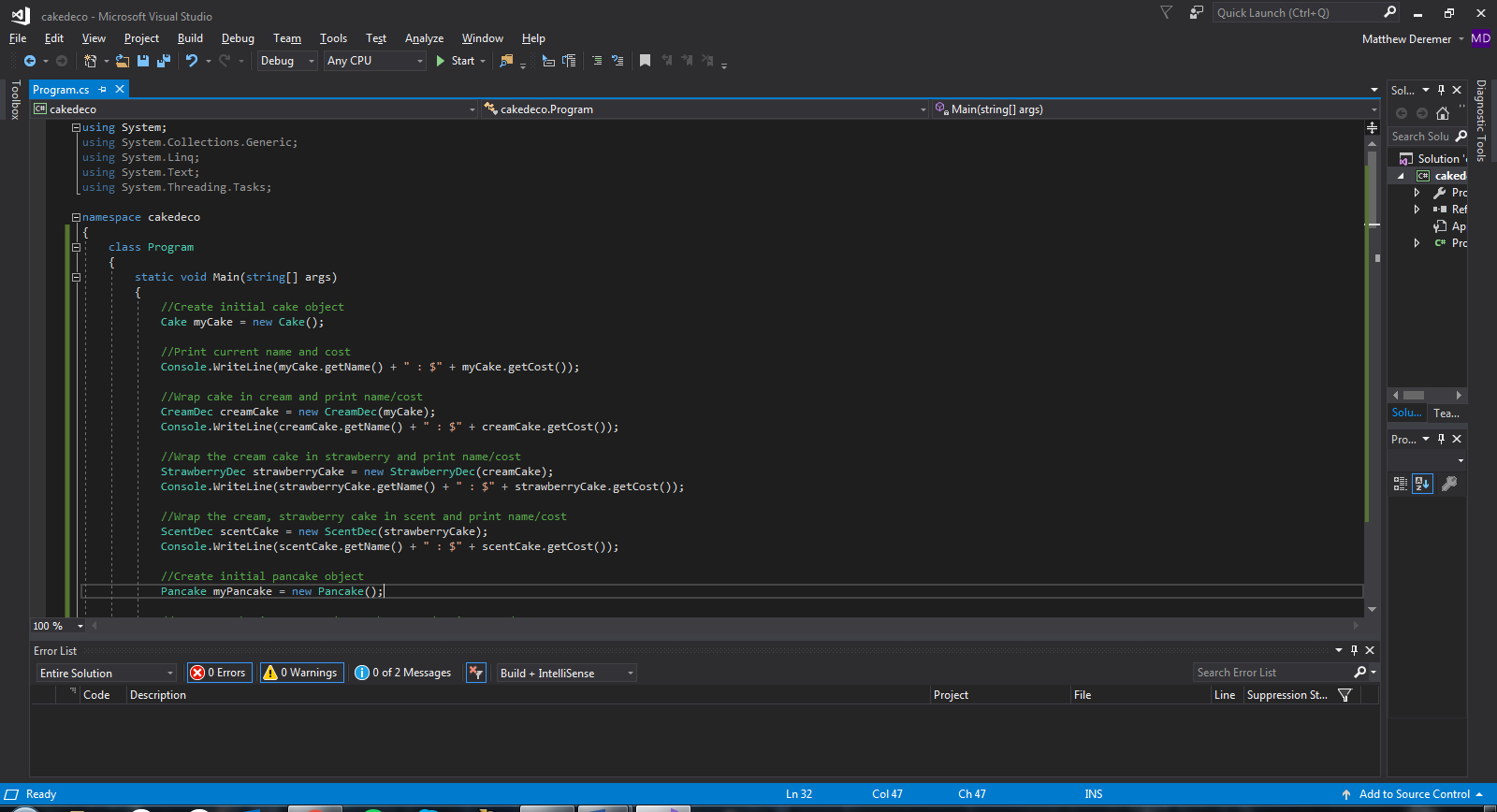


Figure 58: Generate Cake with Decorators Object

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

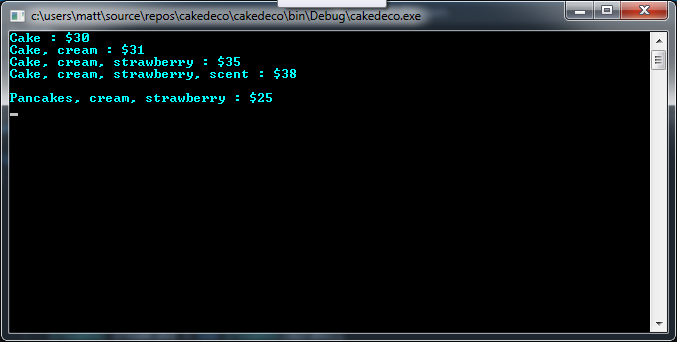


Figure 59: Cake with Decorators Output

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

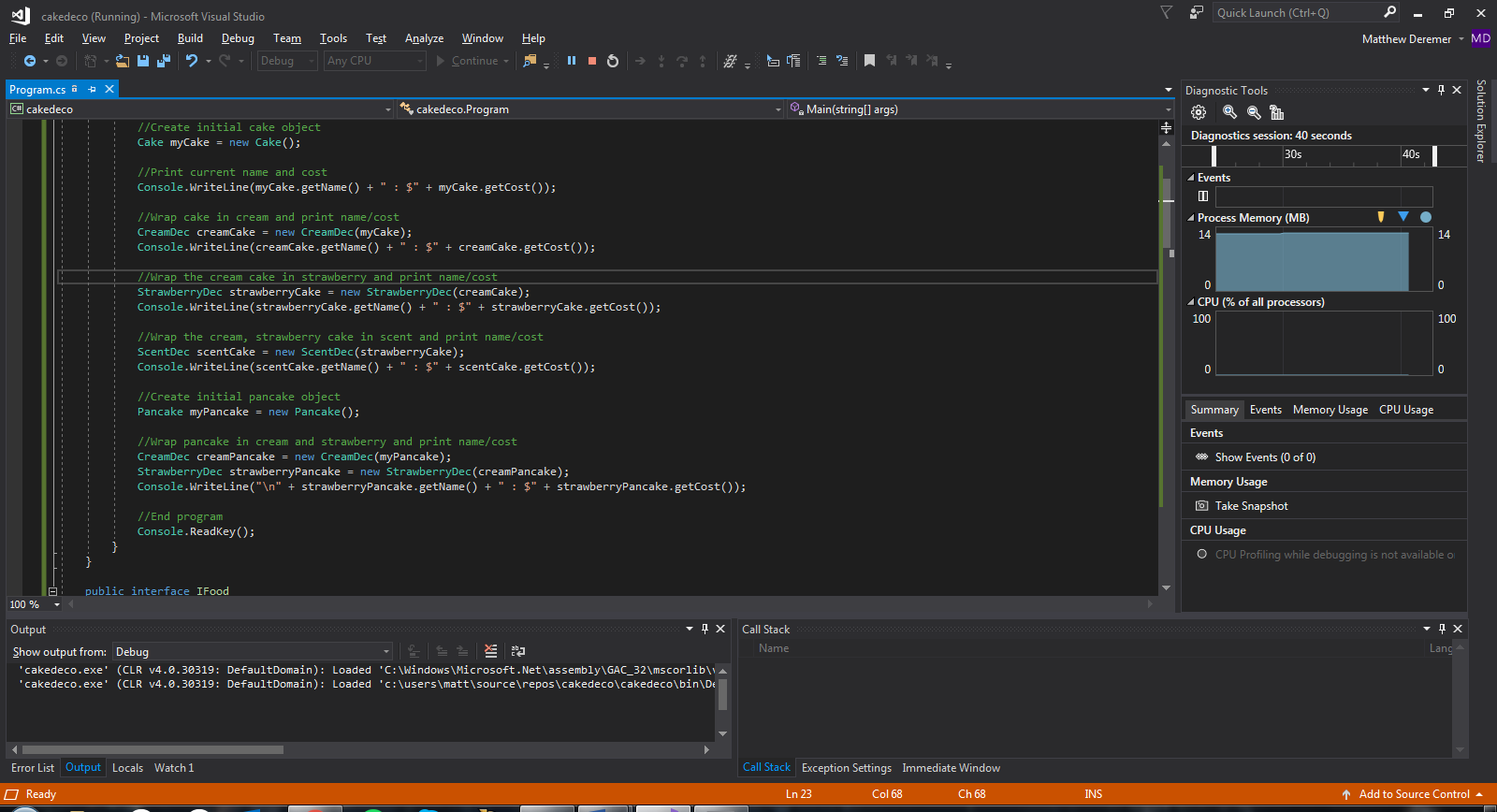


Figure 60: Generate Pancakes with Decorators Object

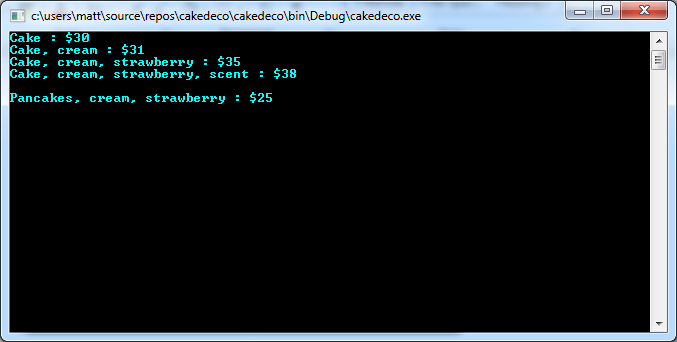


Figure 61: Pancakes with Decorators Output

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.

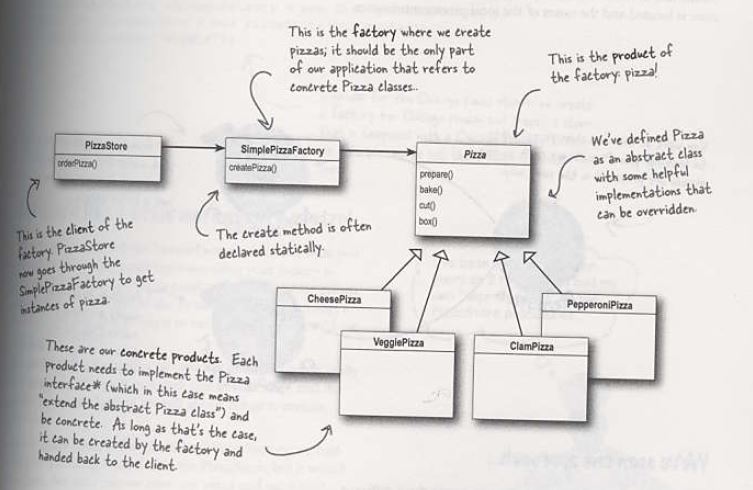


Figure 62: Factory Pattern Example

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.

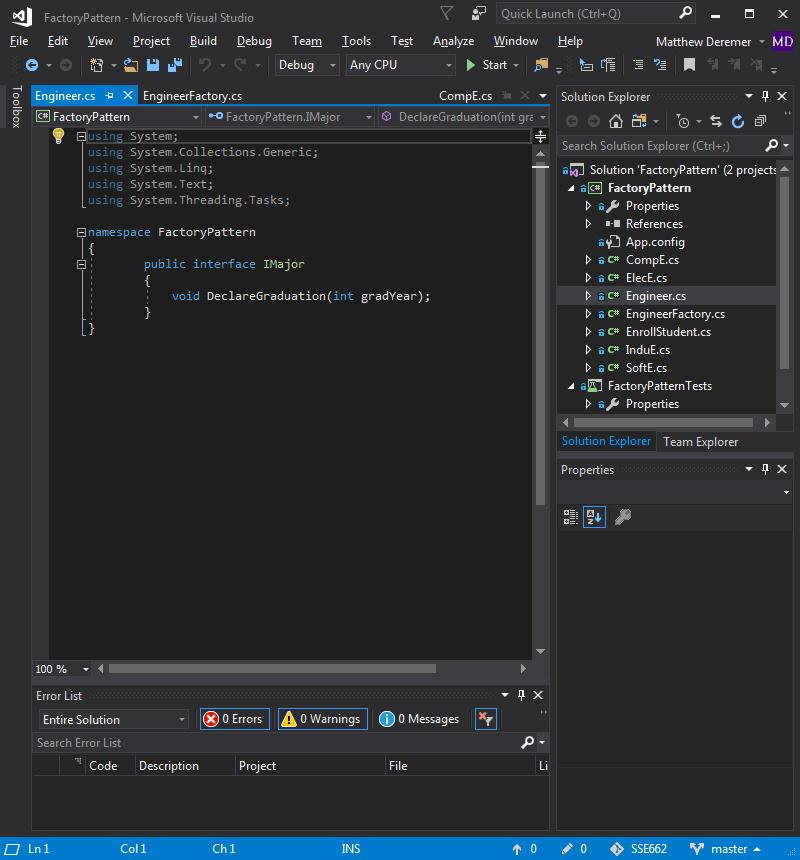


Figure : IMajor Interface

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.

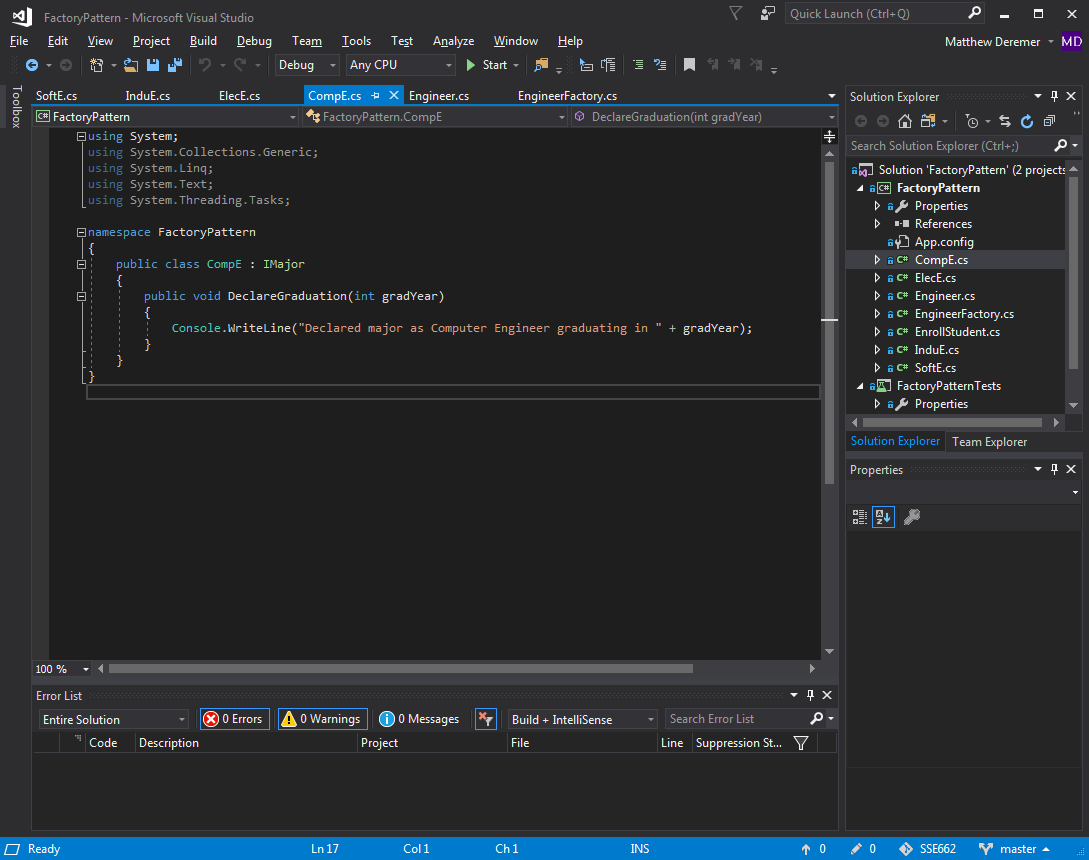


Figure : CompE Class

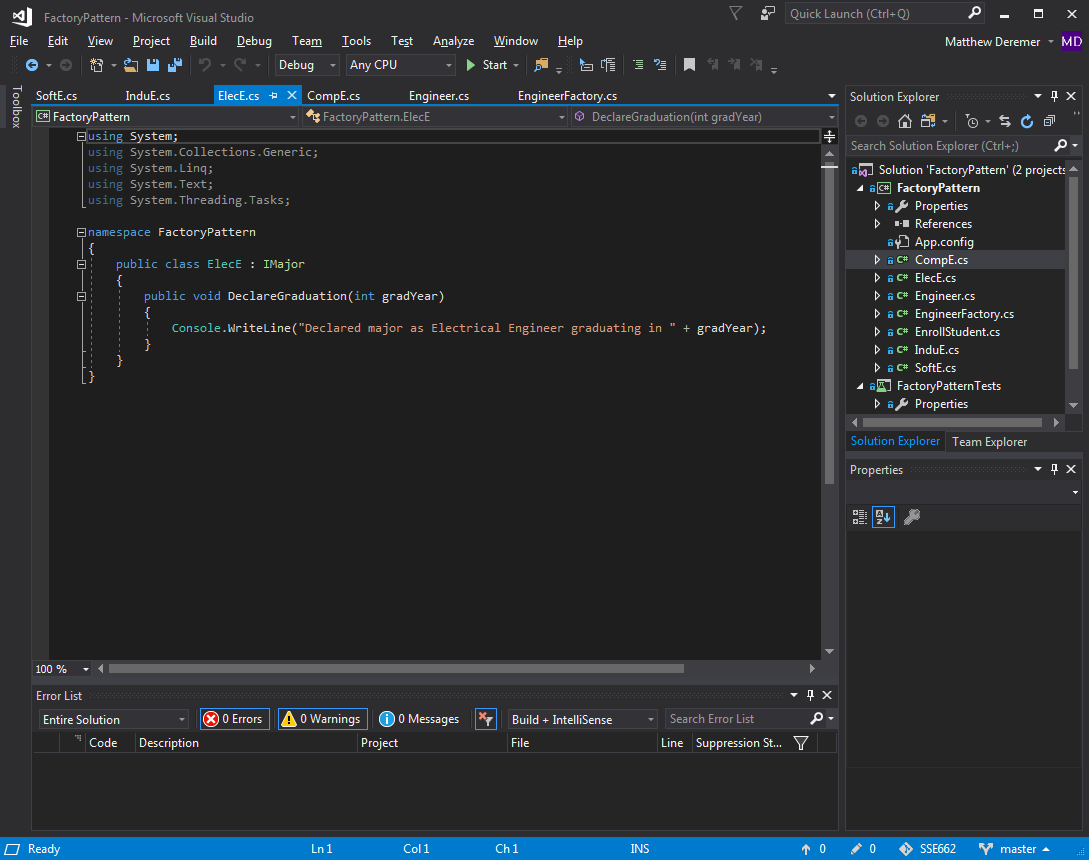


Figure : ElecE Class

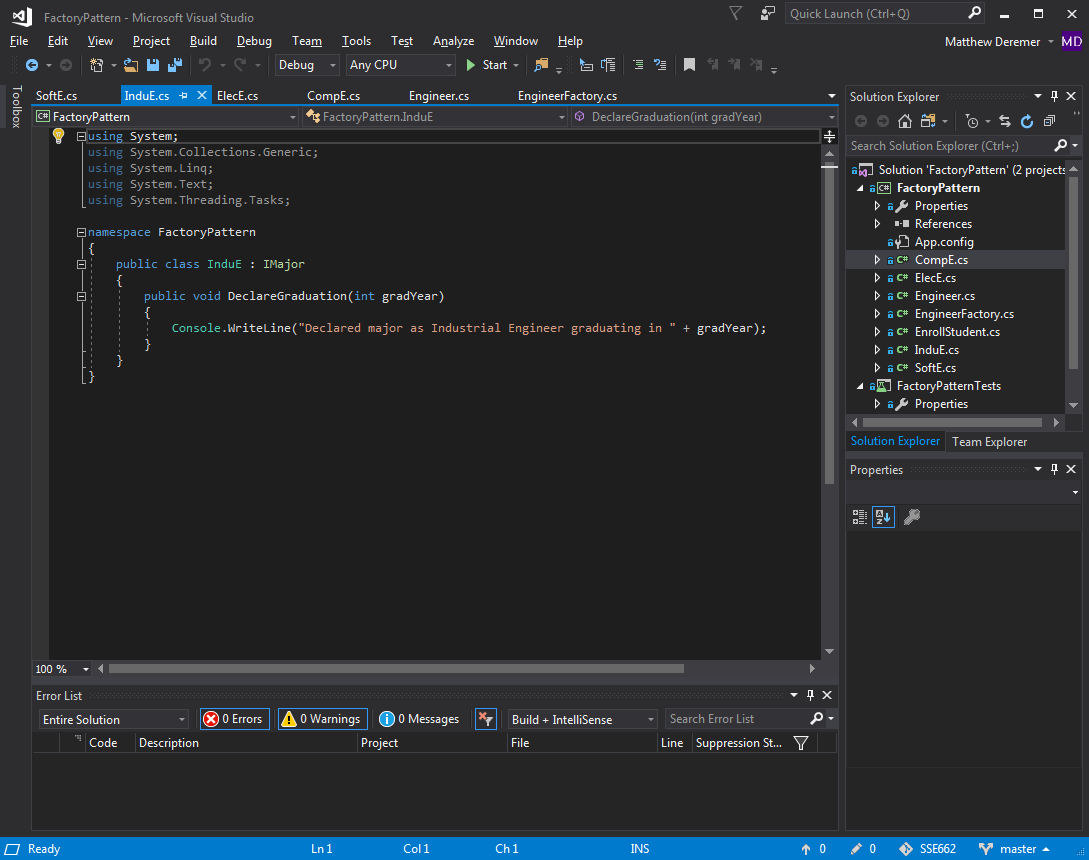


Figure : InduE Class

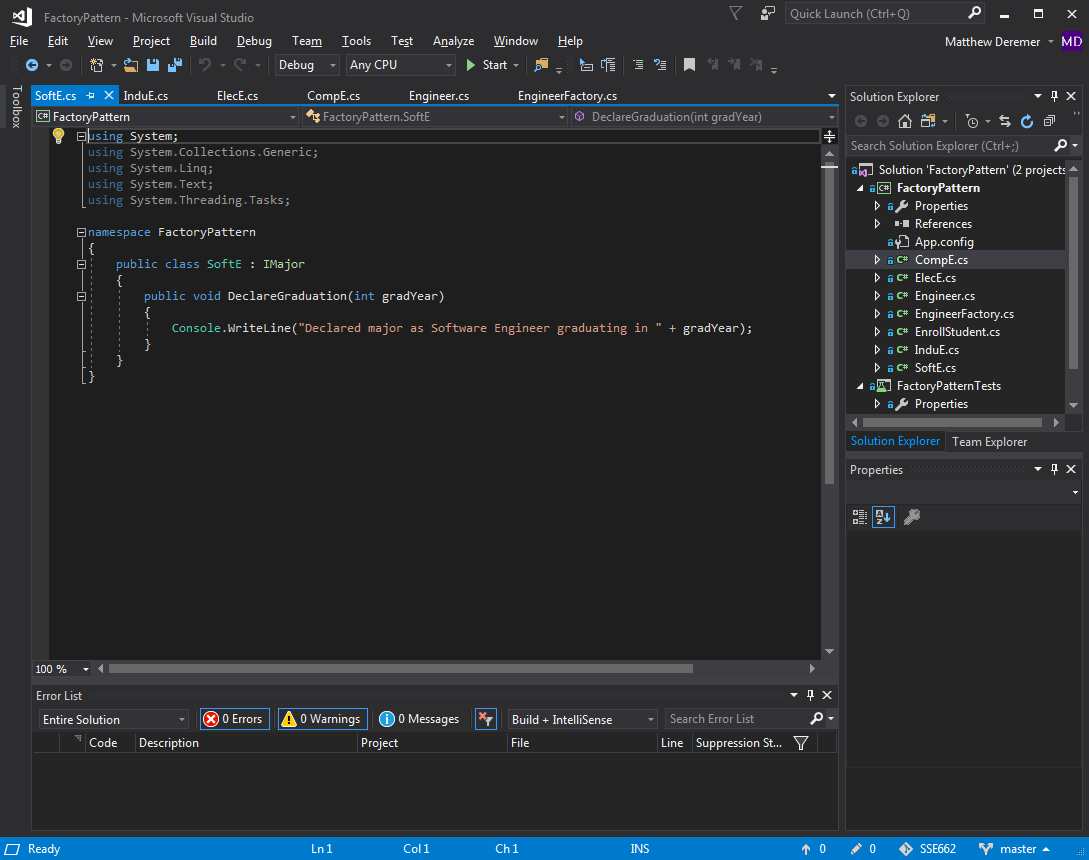


Figure : SoftE Class

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.

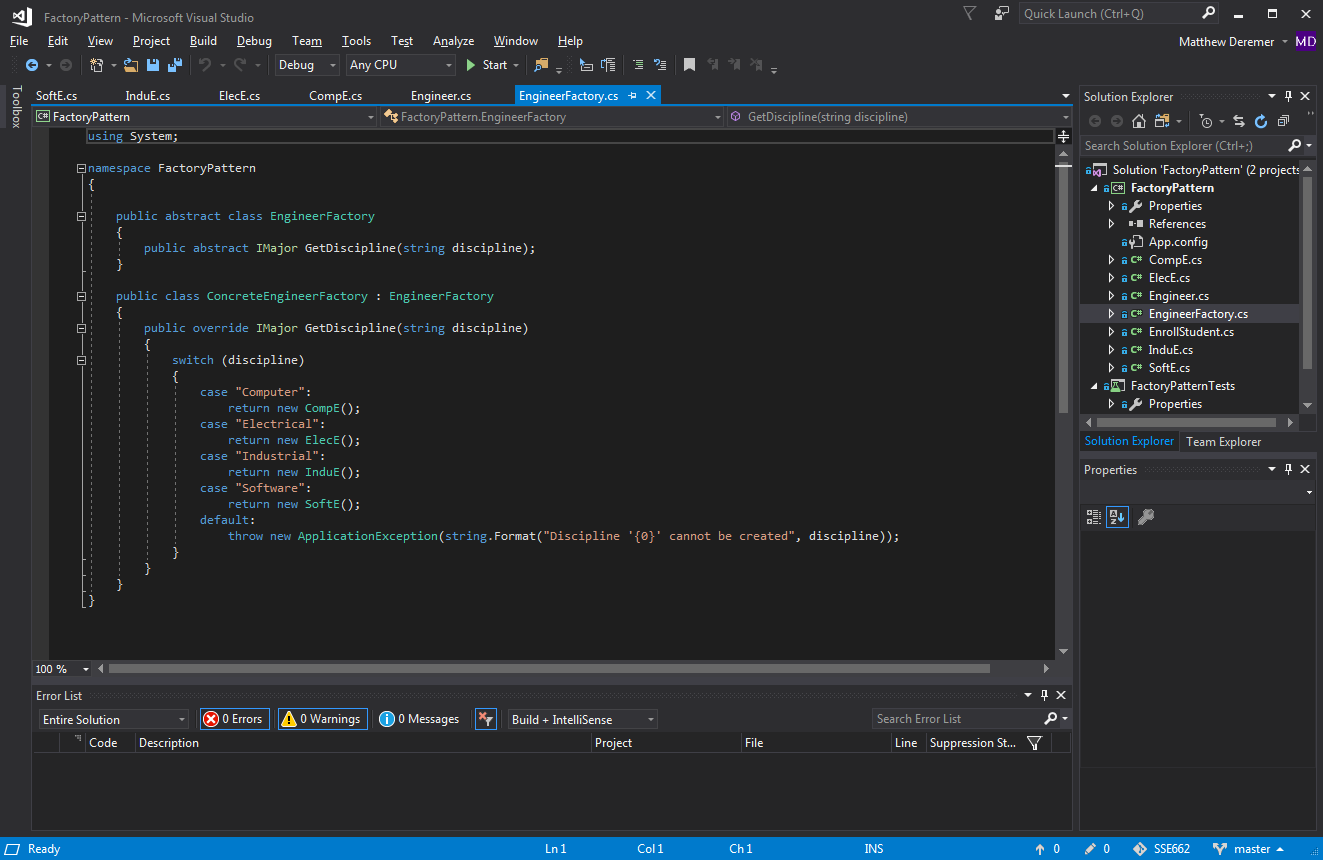


Figure : EngineerFactory

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

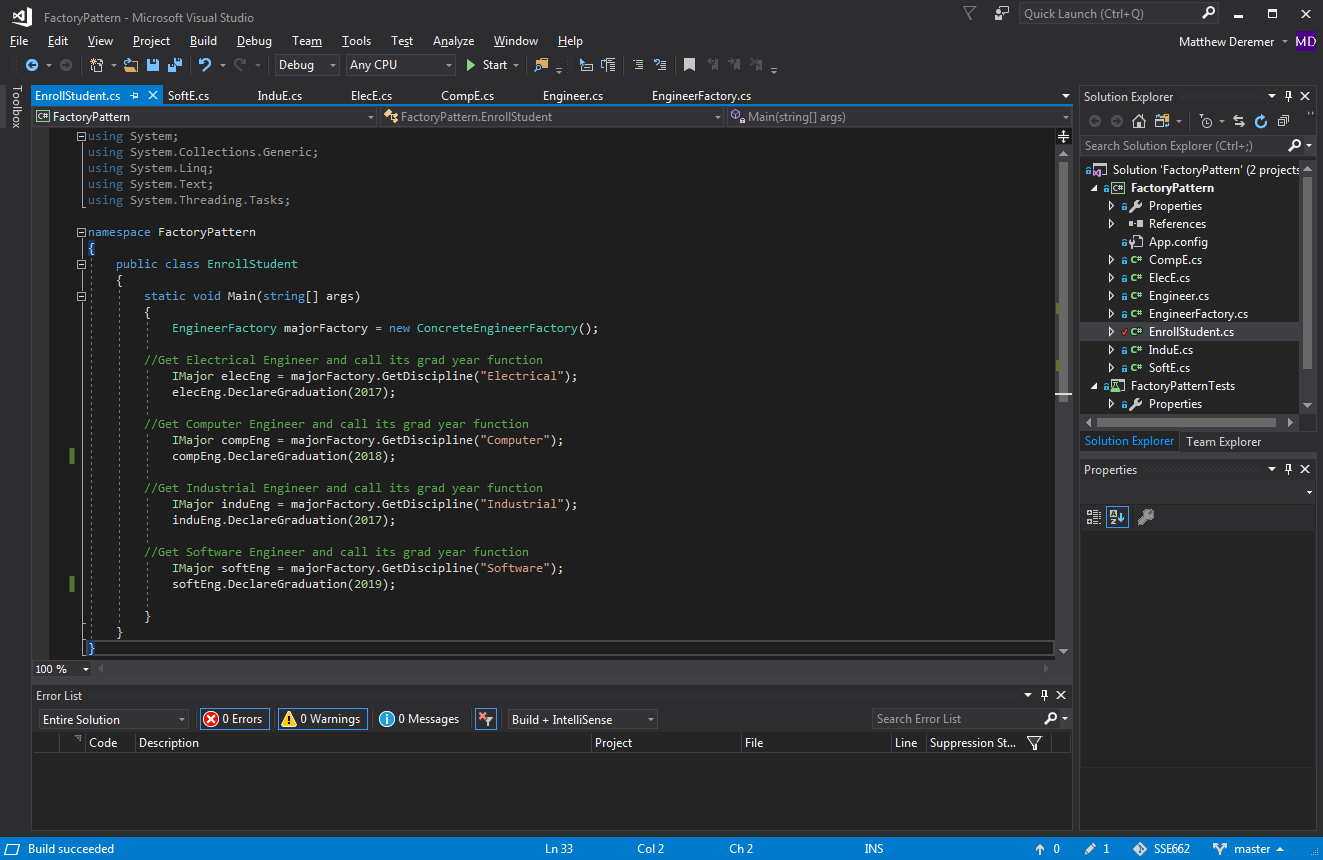


Figure : Client Code, Enroll Students

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.

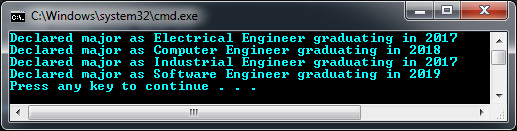


Figure : Enroll Students, Output

# **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>