11/29/2017

Head First Design Patterns

Project 4 Report

Braeden Brettin,
Matthew Deremer,
and Luke Pace

Table of Contents

Table of Figures

Adapter Pattern

The object-oriented notion of an adapter is not too different from that of a real-life adapter. Think back to any trip you may have made to a foreign country. In most other countries, a normal AC plug will not connect to wall outlets. Why is this? It could be due to a difference in required voltage or a difference in socket design. How did you fix the problem? You used an adapter. This adapter adapted your design (the American AC plug) to a client (the wall outlet) without changing either of these components. In much the same way, object-oriented adapters provide functionality to connect an existing system to a client without changing the code of either of these components. Instead, new code is written in the adapter to adapt the two components. This adaptation can be visualized as a jigsaw puzzle, as shown in Figure 1, below.

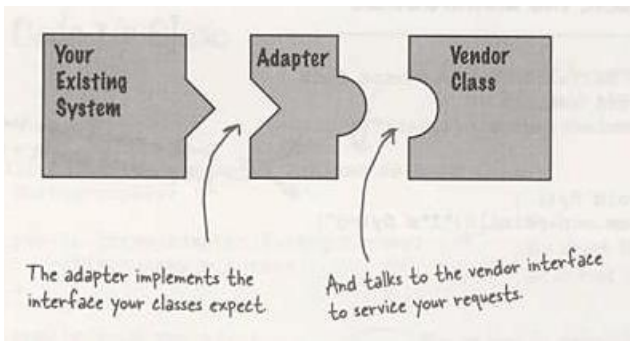

Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.

Figure 1. Adapter Design

The adapter pattern "converts the interface of a class into another interface the clients expect" [1]. This pattern enables classes to work in tandem that otherwise would not be able to because of incompatible interfaces. This pattern also preserves the decoupling of the adapter and the client. Neither class has any knowledge of the inner workings of the other class, an ideal condition in object-oriented design. The client sees only the Target interface, and all requests get delegated to the Adaptee, as shown in Figure 2, below.

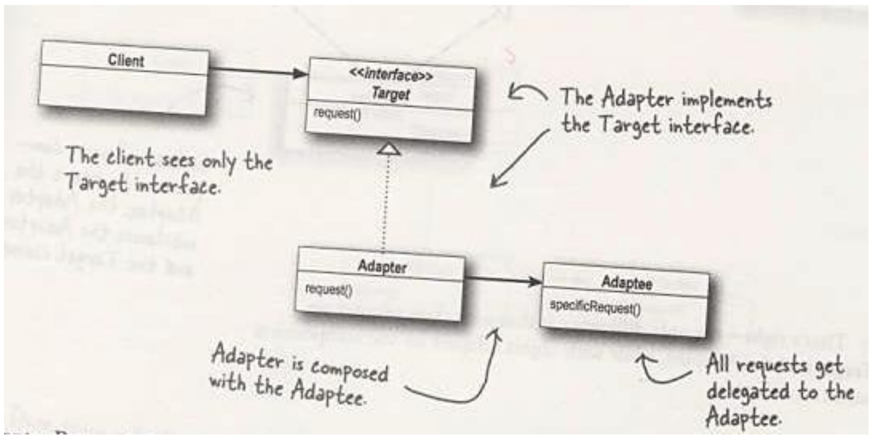
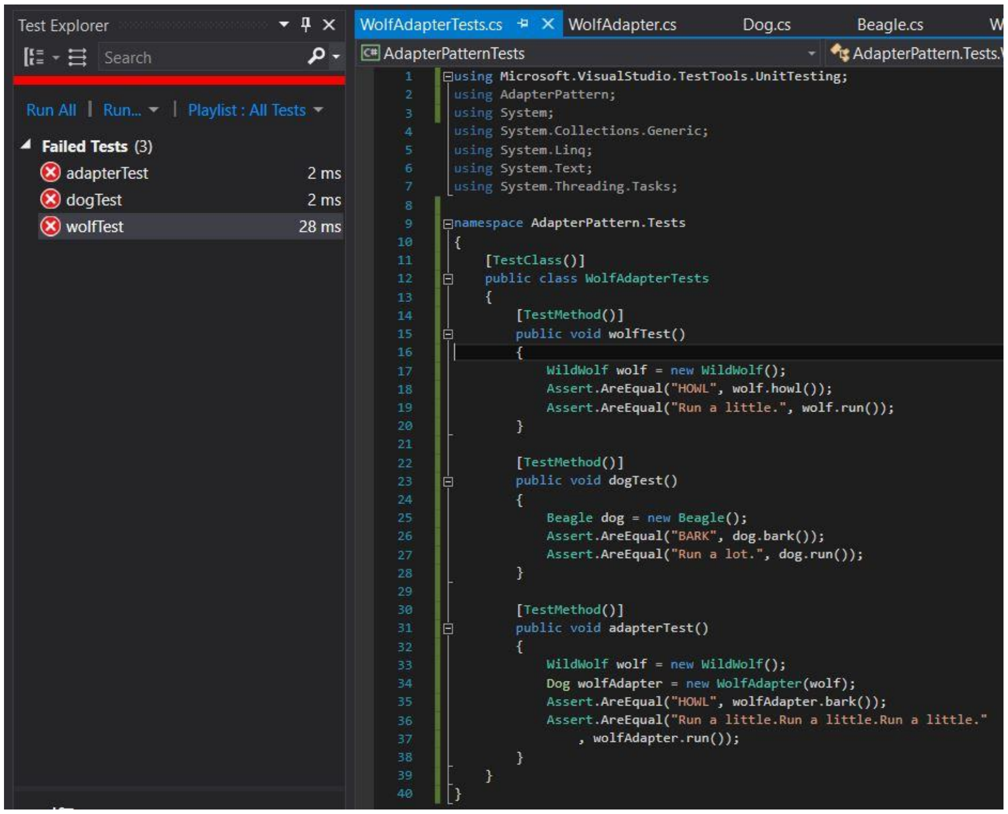

Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

Figure 2. Adapter and Target Design

For the purposes of this project, I will create a Wolf interface that can be adapted to a Dog interface. Considering the close genetic relationship between these two species, these two animals share similar attributes and behaviors. As such, these two interfaces will share similar functions to replicate the real-world behaviors of these two species. The Dog interface contains functions for barking and running, bark() and run(), respectively. The Wolf interface contains functions for howling and running, howl() and run(), respectively. The bark() and howl() functions will differ slightly in the sound that the animal makes. The run() functions in the two classes will differ slightly in the amount of time that the animal runs. We have also created two concrete classes, each implementing either the Dog or Wolf interface.

A test suite was created that outlines the specific functionality we hope to achieve in this project. This test suite was purposefully failed, as shown in Figure 3, below.

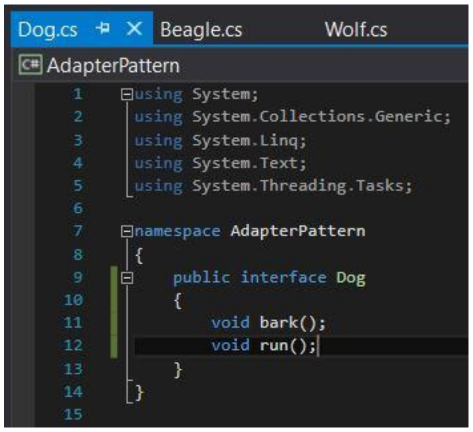


```
Failed Tests (3)
  adapterTest    2 ms
  dogTest        2 ms
  wolfTest      28 ms
```

```csharp
using Microsoft.VisualStudio.TestTools.UnitTesting;
using AdapterPattern;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AdapterPattern.Tests
{
    [TestClass()]
    public class WolfAdapterTests
    {
        [TestMethod()]
        public void wolfTest()
        {
            WildWolf wolf = new WildWolf();
            Assert.AreEqual("HOWL", wolf.howl());
            Assert.AreEqual("Run a little.", wolf.run());
        }

        [TestMethod()]
        public void dogTest()
        {
            Beagle dog = new Beagle();
            Assert.AreEqual("BARK", dog.bark());
            Assert.AreEqual("Run a lot.", dog.run());
        }

        [TestMethod()]
        public void adapterTest()
        {
            WildWolf wolf = new WildWolf();
            Dog wolfAdapter = new WolfAdapter(wolf);
            Assert.AreEqual("HOWL", wolfAdapter.bark());
            Assert.AreEqual("Run a little.Run a little.Run a little."
                , wolfAdapter.run());
        }
    }
}
```

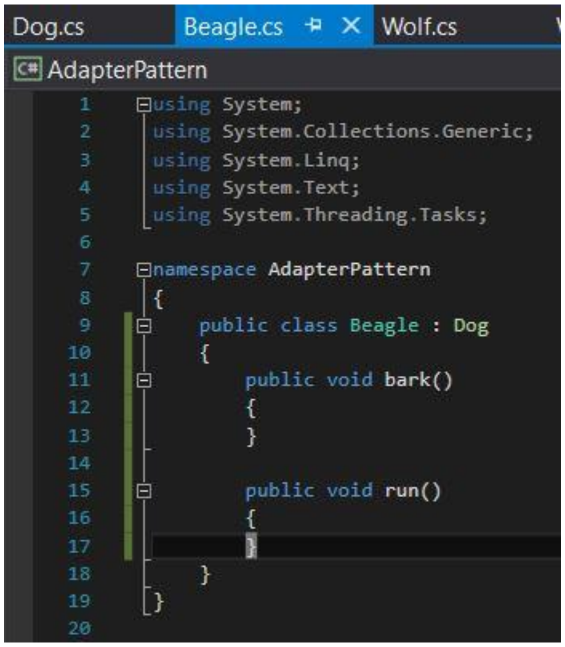
Figure 3. Initial Failed Test

The code in the interfaces and concrete classes at the time of this failed test was as shown in Figures 4 through 7, below.



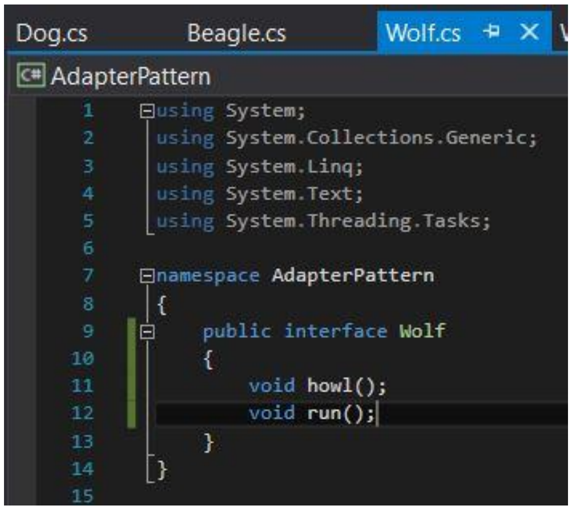
```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AdapterPattern
{
    public interface Dog
    {
        void bark();
        void run();
    }
}
```

Figure 4. Initial Dog Class



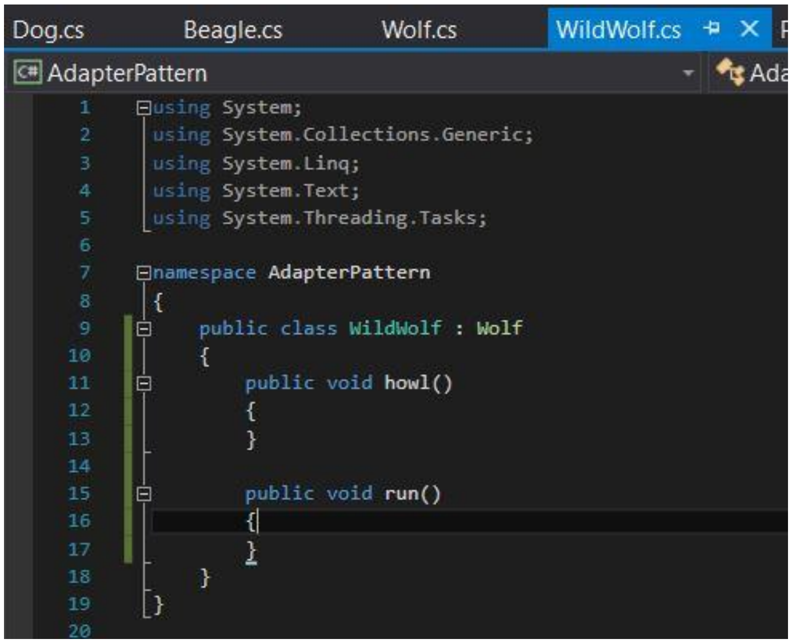
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AdapterPattern
{
    public class Beagle : Dog
    {
        public void bark()
        {
        }

        public void run()
        {
        }
    }
}
```

Figure 5. Initial Beagle Class



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

namespace AdapterPattern
{
    public interface Wolf
    {
        void howl();
        void run();
    }
}
```

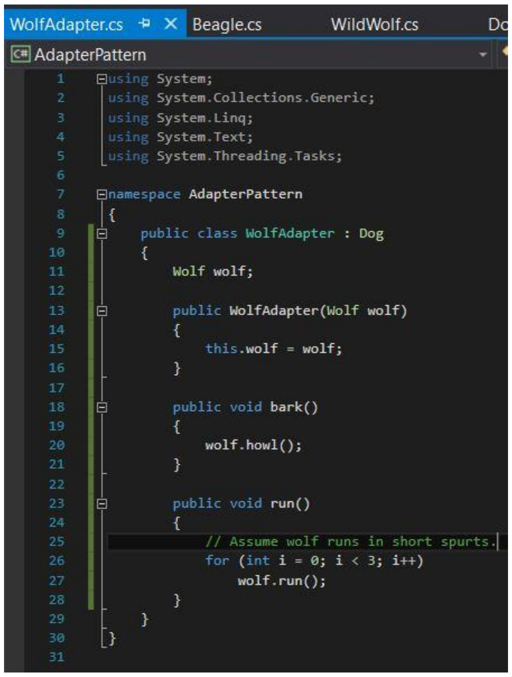
Figure 6. Initial Wolf Class



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

namespace AdapterPattern
{
    public class WildWolf : Wolf
    {
        public void howl()
        {
        }

        public void run()
        {
        }
    }
}
```

Figure 7. Initial WildWolf Class

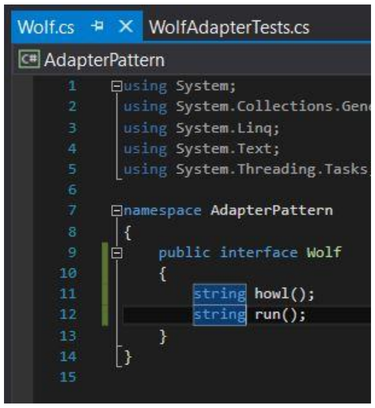
Now that these initial classes have been created and the test suite created and outlined, we need to create an adapter for the Wolf interface. How do we go about doing so? We can examine the differences between dogs and wolves to create this adapter. Wolves hunt in packs, so they do not have to run for long periods of time like dogs must. The pack mentality and organization of wolves allows them to conserve energy when hunting. As such, they merely need to run in short spurts. In To to adapt a wolf to a dog, we need to call the wolf's run() function multiple times to replicate the dog's run() function, as shown in Figure 8, below.



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

namespace AdapterPattern
{
    public class WolfAdapter : Dog
    {
        Wolf wolf;

        public WolfAdapter(Wolf wolf)
        {
            this.wolf = wolf;
        }

        public void bark()
        {
            wolf.howl();
        }

        public void run()
        {
            // Assume wolf runs in short spurts.
            for (int i = 0; i < 3; i++)
                wolf.run();
        }
    }
}
```

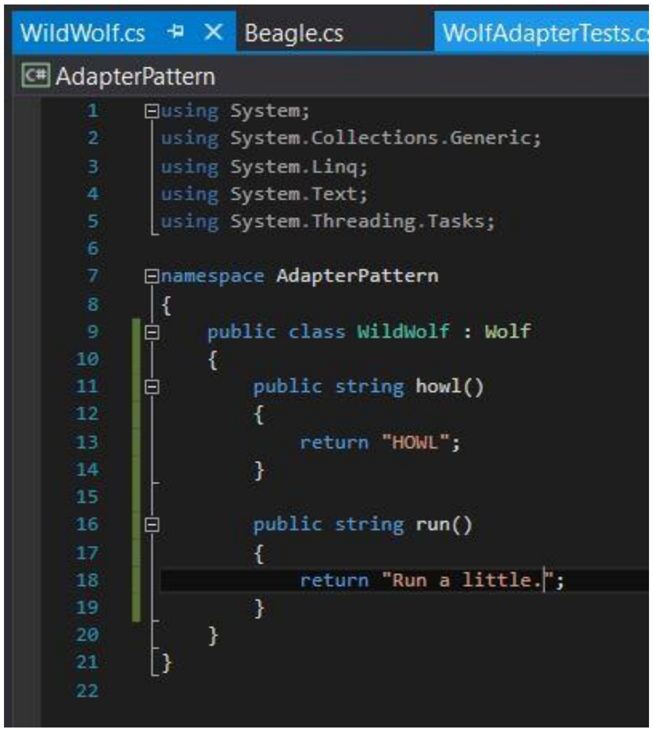
Figure 8. WolfAdapter Class

This adapter now provides the needed functionality to adapt a wolf to a dog and replicate the dog class's behavior. We now merely need to update the Wolf, WildWolf, Dog, and Beagle classes to return the correct output when their respective functions are called, as shown in Figures 9 through 12, below.



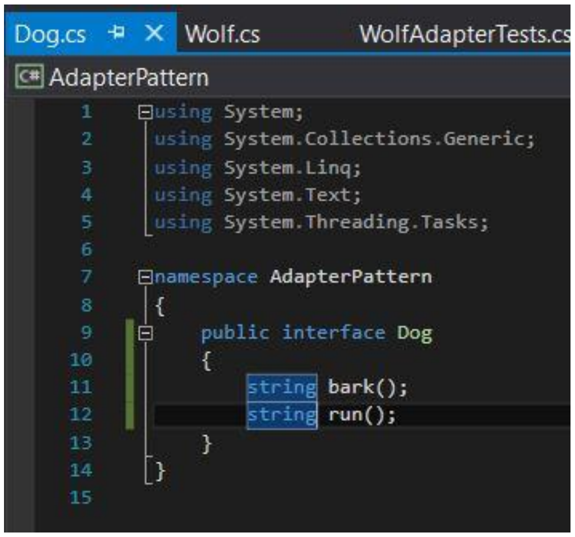
```
using System;
using System.Collections.Gen
using System.Linq;
using System.Text;
using System.Threading.Tasks

namespace AdapterPattern
{
    public interface Wolf
    {
        string howl();
        string run();
    }
}
```

Figure 9. Refactored Wolf Class



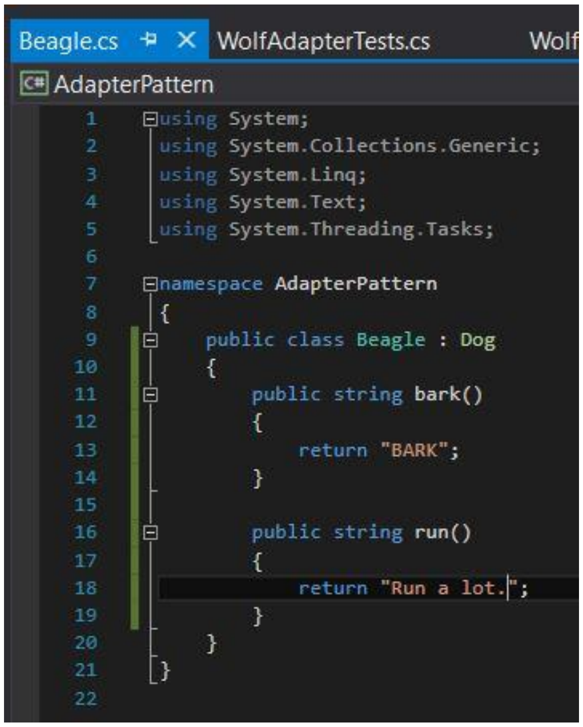
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AdapterPattern
{
    public class WildWolf : Wolf
    {
        public string howl()
        {
            return "HOWL";
        }

        public string run()
        {
            return "Run a little.";
        }
    }
}
```

Figure 10. Refactored WildWolf Class



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

namespace AdapterPattern
{
    public interface Dog
    {
        string bark();
        string run();
    }
}
```

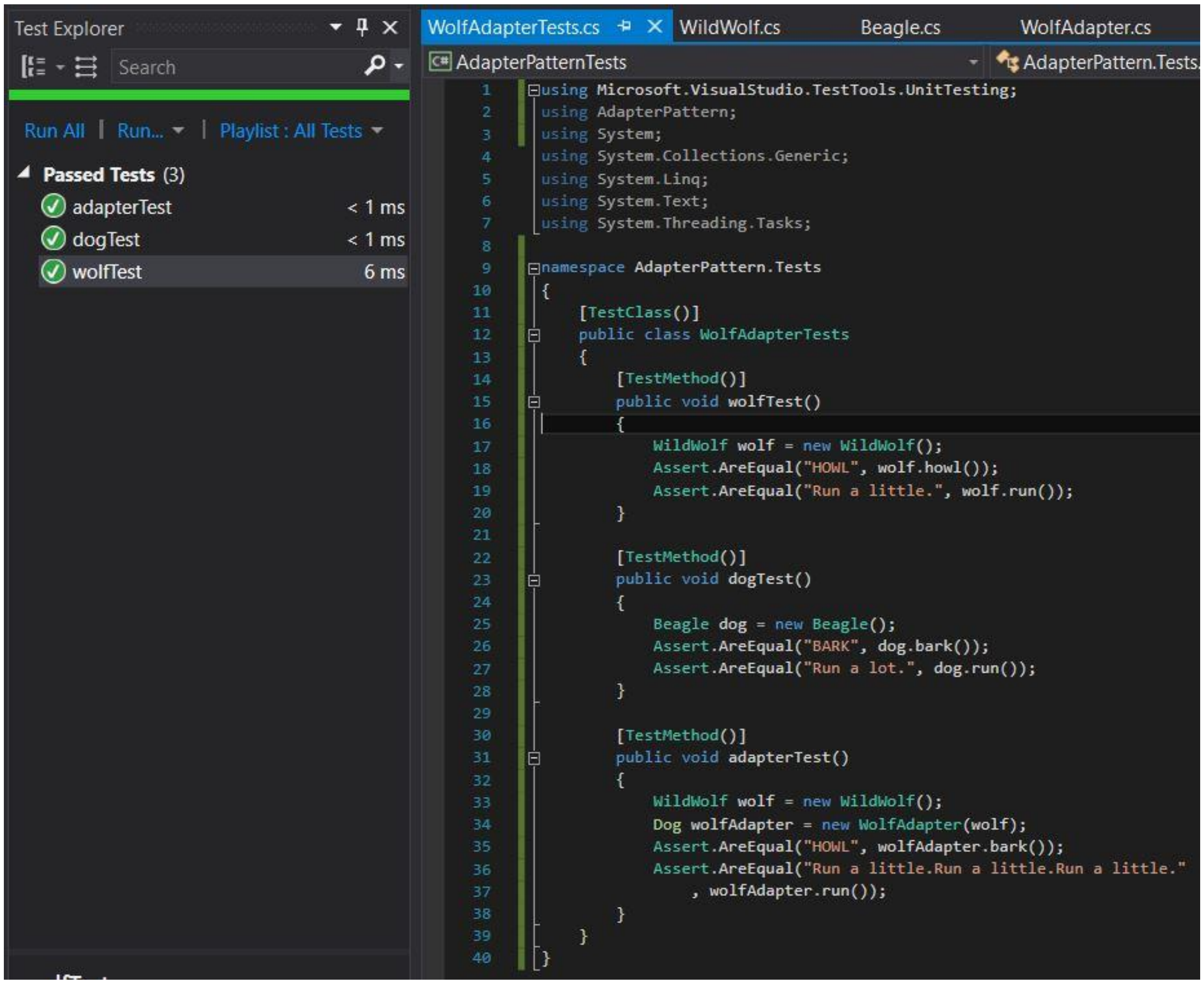
Figure 11. Refactored Dog Class



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

namespace AdapterPattern
{
    public class Beagle : Dog
    {
        public string bark()
        {
            return "BARK";
        }

        public string run()
        {
            return "Run a lot.";
        }
    }
}
```

Figure 12. Refactored Beagle Class

With these classes refactored, we need to ensure that these changes result in successful tests, preserving the functionality of the project. The test suite was run again, producing the successful output shown in Figure 13, below.



```
Passed Tests (3)
  adapterTest    < 1 ms
  dogTest        < 1 ms
  wolfTest         6 ms
```

```csharp
using Microsoft.VisualStudio.TestTools.UnitTesting;
using AdapterPattern;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AdapterPattern.Tests
{
    [TestClass()]
    public class WolfAdapterTests
    {
        [TestMethod()]
        public void wolfTest()
        {
            WildWolf wolf = new WildWolf();
            Assert.AreEqual("HOWL", wolf.howl());
            Assert.AreEqual("Run a little.", wolf.run());
        }

        [TestMethod()]
        public void dogTest()
        {
            Beagle dog = new Beagle();
            Assert.AreEqual("BARK", dog.bark());
            Assert.AreEqual("Run a lot.", dog.run());
        }

        [TestMethod()]
        public void adapterTest()
        {
            WildWolf wolf = new WildWolf();
            Dog wolfAdapter = new WolfAdapter(wolf);
            Assert.AreEqual("HOWL", wolfAdapter.bark());
            Assert.AreEqual("Run a little.Run a little.Run a little."
                , wolfAdapter.run());
        }
    }
}
```

Figure 13. Successful Test

As can be seen from this project, object-oriented adapters can be used to adapt one interface to another without needing to change the code of either interface. The adapter pattern is extremely useful for adapting closely-related classes and has a plethora of real-world applications. When running low on objects of one class, consider using the adapter pattern to adapt this class to another class.

Composite Pattern

The idea behind the use of the Composite Pattern can be seen all around us. For example, think about the last time you went to the doctor. You probably saw a nurse first who took some general information, then the doctor would perform an exam and the nurse may come back to finish the visit. Then you go to the lab to have some different readings taken and wait for the results. So now we have created a detailed hierarchy with lots of different objects that documents the visit. Then comes the important part, the bill. Billing does not care about the details of the visit, they just want to know what was done that is billable. To obtain this information with the current setup would be difficult because it is embedded in the records which is clogged up with different objects and is not always consistent with the hierarchy because each visit does not go through the same process. This issue of too many objects and an inconsistent graph of data can be solved by implementing the Composite Pattern. We could define a base class with a billing property and each encounter would appear as a container with the base class inside. Billing can now simply enumerate everything inside an encounter and not worry about if it is a node or leaf. Figure 14 shows a visualization of the organization of the data using the Composite Pattern where the composites are nodes (i.e., nurse exam, lab visit) and the leaves are information such as height and weight.

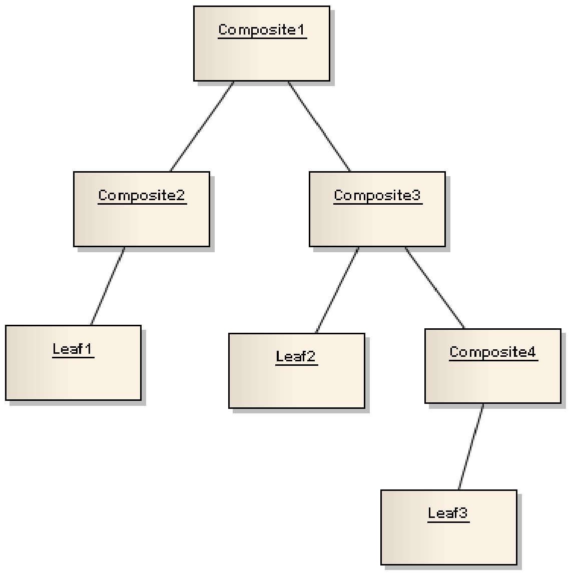


Figure 14: Visualization of Composite Pattern

The Composite Pattern "allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly" [1]. What our text means by part-whole is the tree is composed of parts but can be treated as a whole. This can be very useful because it allows us to write simple code to apply an operation to the entire structure. Figure 15 shows the class diagram of the Composite Pattern.

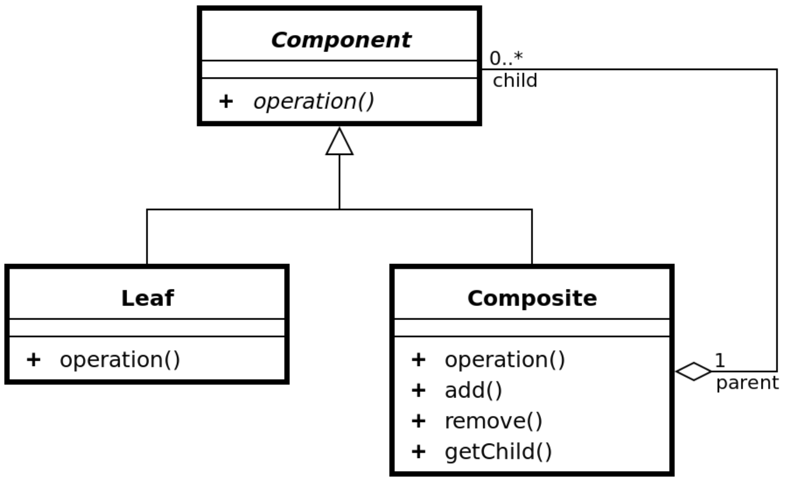


Figure 15: Composite Pattern Class Diagram

As the above figure shows, there are three components needed for the Composite Pattern. The first is the component which declares an interface for objects in the composition and implements behavior common to all objects. It also must implement an interface for adding and removing its own children. The second component is a leaf which implements the behavior for a leaf. The final component is a composite which defines behavior for nodes and implements the adding/removing interface from the component.

To model the Composite Pattern, we have implemented a Freestyle Coke machine which has a hierarchy of drinks starting with brand and working down to flavor. Figure 16, below, shows a model of the hierarchy.

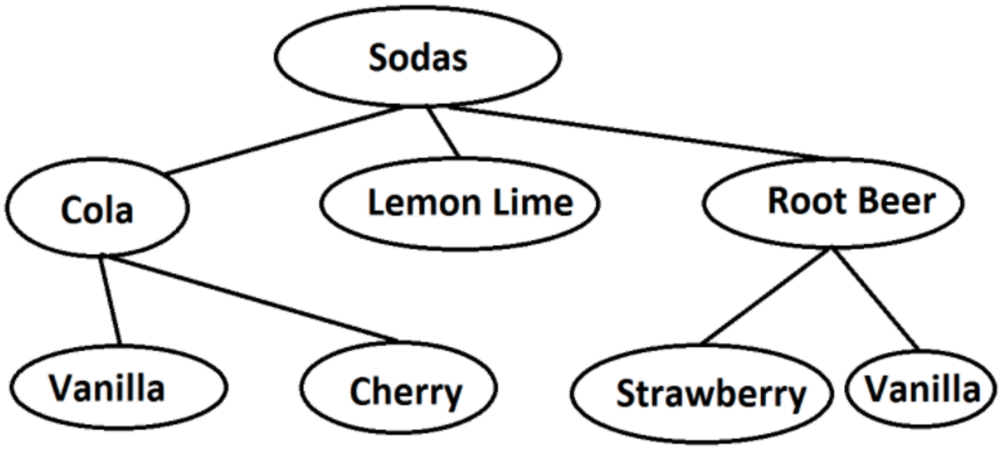
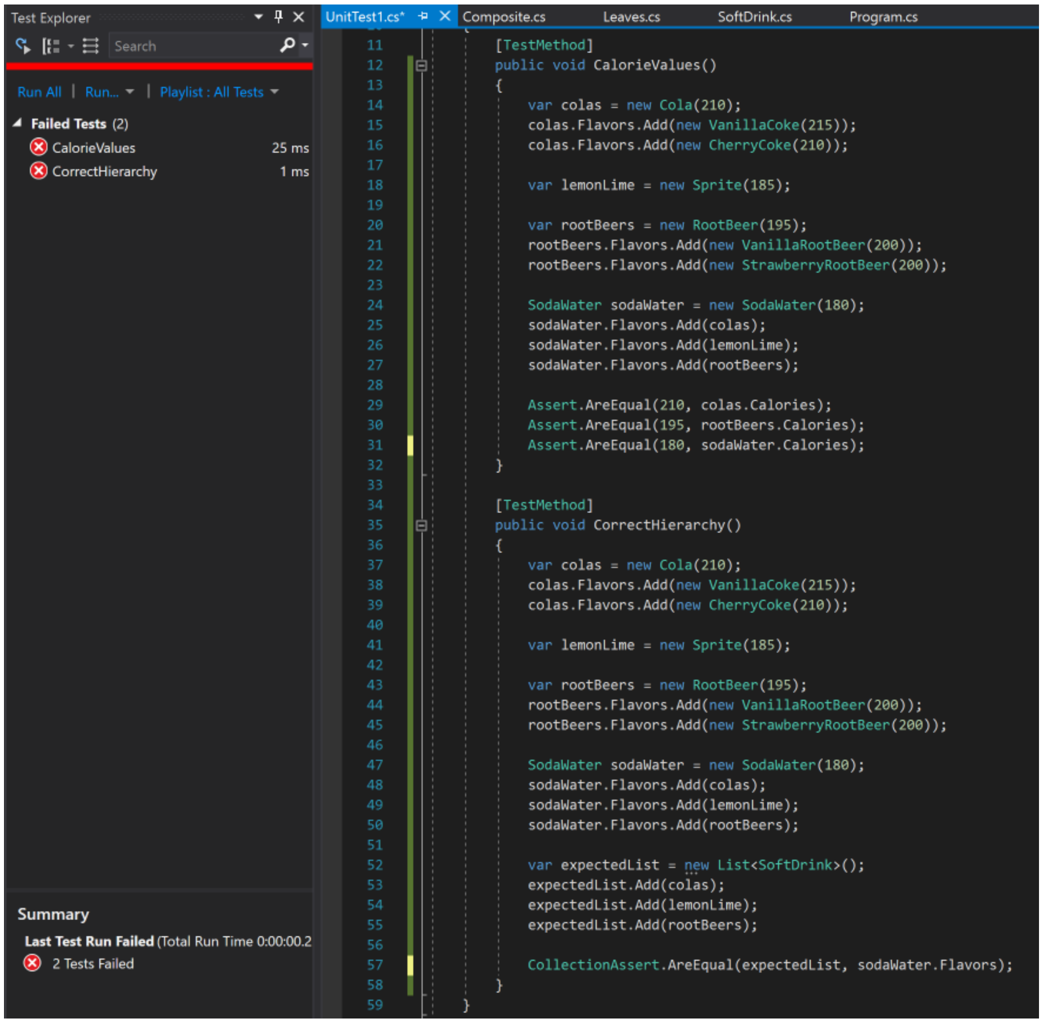


Figure 16: Freestyle Coke Model

A test suite was created to test the functionality we hoped to achieve which is shown in Figure 17, below.

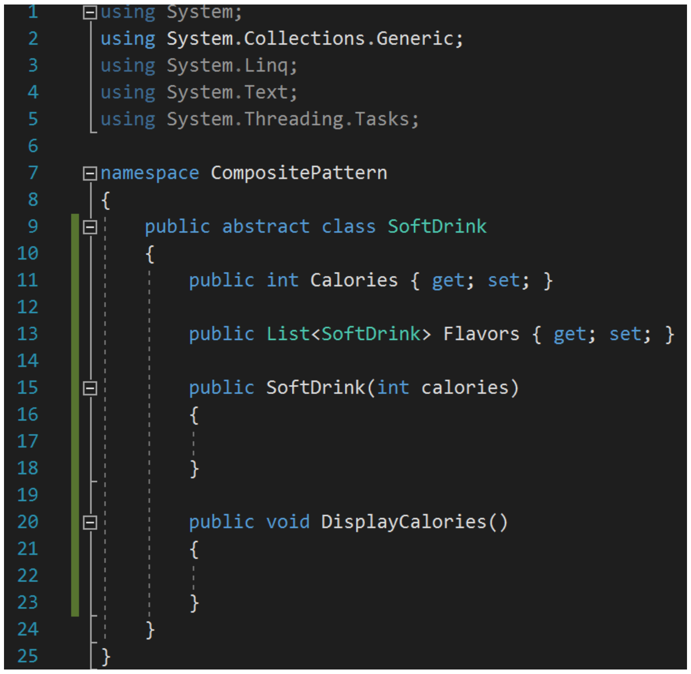


```
Failed Tests (2)
  CalorieValues        25 ms
  CorrectHierarchy      1 ms

Summary
Last Test Run Failed (Total Run Time 0:00:00.2
  2 Tests Failed
```

```
        [TestMethod]
        public void CalorieValues()
        {
            var colas = new Cola(210);
            colas.Flavors.Add(new VanillaCoke(215));
            colas.Flavors.Add(new CherryCoke(210));

            var lemonLime = new Sprite(185);

            var rootBeers = new RootBeer(195);
            rootBeers.Flavors.Add(new VanillaRootBeer(200));
            rootBeers.Flavors.Add(new StrawberryRootBeer(200));

            SodaWater sodaWater = new SodaWater(180);
            sodaWater.Flavors.Add(colas);
            sodaWater.Flavors.Add(lemonLime);
            sodaWater.Flavors.Add(rootBeers);

            Assert.AreEqual(210, colas.Calories);
            Assert.AreEqual(195, rootBeers.Calories);
            Assert.AreEqual(180, sodaWater.Calories);
        }

        [TestMethod]
        public void CorrectHierarchy()
        {
            var colas = new Cola(210);
            colas.Flavors.Add(new VanillaCoke(215));
            colas.Flavors.Add(new CherryCoke(210));

            var lemonLime = new Sprite(185);

            var rootBeers = new RootBeer(195);
            rootBeers.Flavors.Add(new VanillaRootBeer(200));
            rootBeers.Flavors.Add(new StrawberryRootBeer(200));

            SodaWater sodaWater = new SodaWater(180);
            sodaWater.Flavors.Add(colas);
            sodaWater.Flavors.Add(lemonLime);
            sodaWater.Flavors.Add(rootBeers);

            var expectedList = new List<SoftDrink>();
            expectedList.Add(colas);
            expectedList.Add(lemonLime);
            expectedList.Add(rootBeers);

            CollectionAssert.AreEqual(expectedList, sodaWater.Flavors);
        }
    }
```

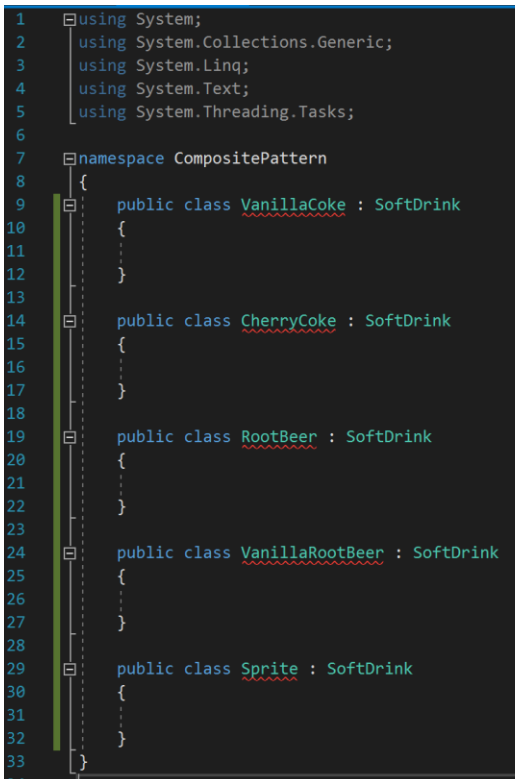
Figure 17: Failed Test

After creating the test suite, we created the outline for our abstract class to represent all soft drinks, the classes for different leaves, and the classes for composites. Figures 18 through 20 show the initial classes.



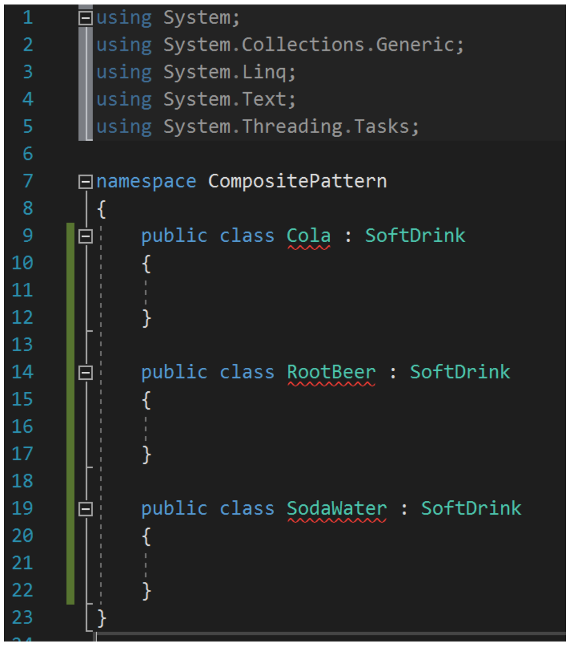
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CompositePattern
{
    public abstract class SoftDrink
    {
        public int Calories { get; set; }

        public List<SoftDrink> Flavors { get; set; }

        public SoftDrink(int calories)
        {

        }

        public void DisplayCalories()
        {

        }
    }
}
```

Figure 18: Initial Abstract Class



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

namespace CompositePattern
{
    public class VanillaCoke : SoftDrink
    {

    }

    public class CherryCoke : SoftDrink
    {

    }

    public class RootBeer : SoftDrink
    {

    }

    public class VanillaRootBeer : SoftDrink
    {

    }

    public class Sprite : SoftDrink
    {

    }
}
```

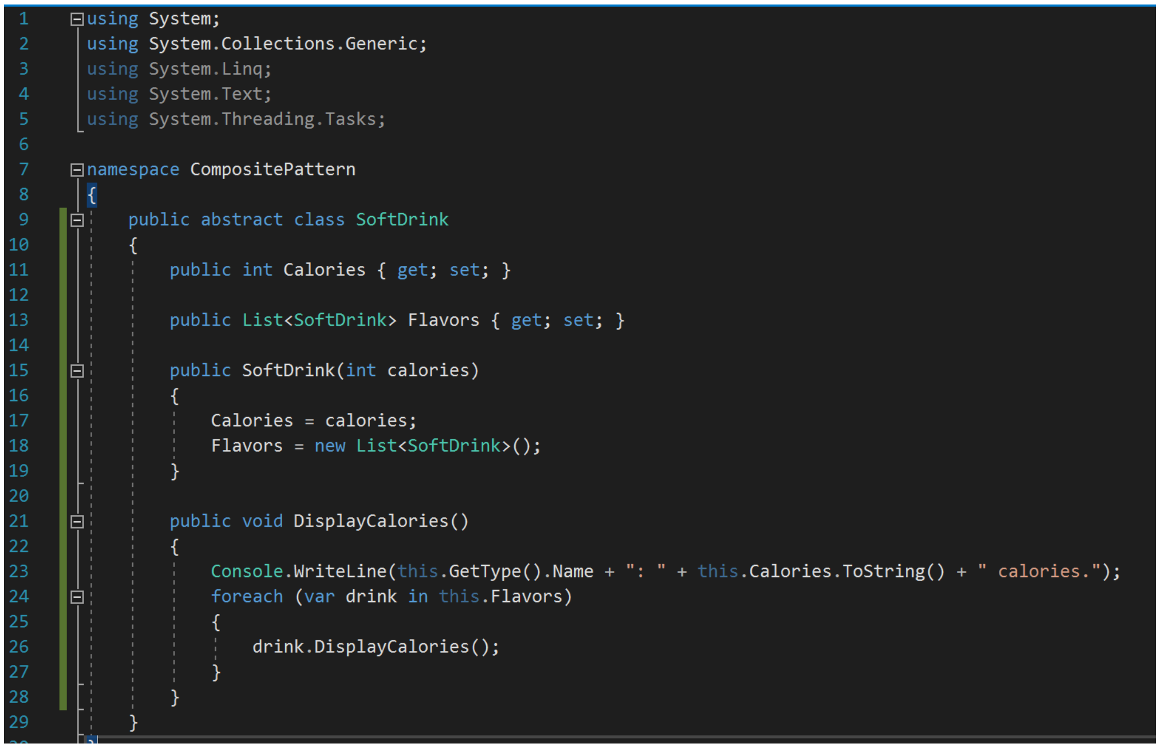
Figure 19: Initial Leaf Classes



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

namespace CompositePattern
{
    public class Cola : SoftDrink
    {

    }

    public class RootBeer : SoftDrink
    {

    }

    public class SodaWater : SoftDrink
    {

    }
}
```

Figure 20: Initial Composite Class

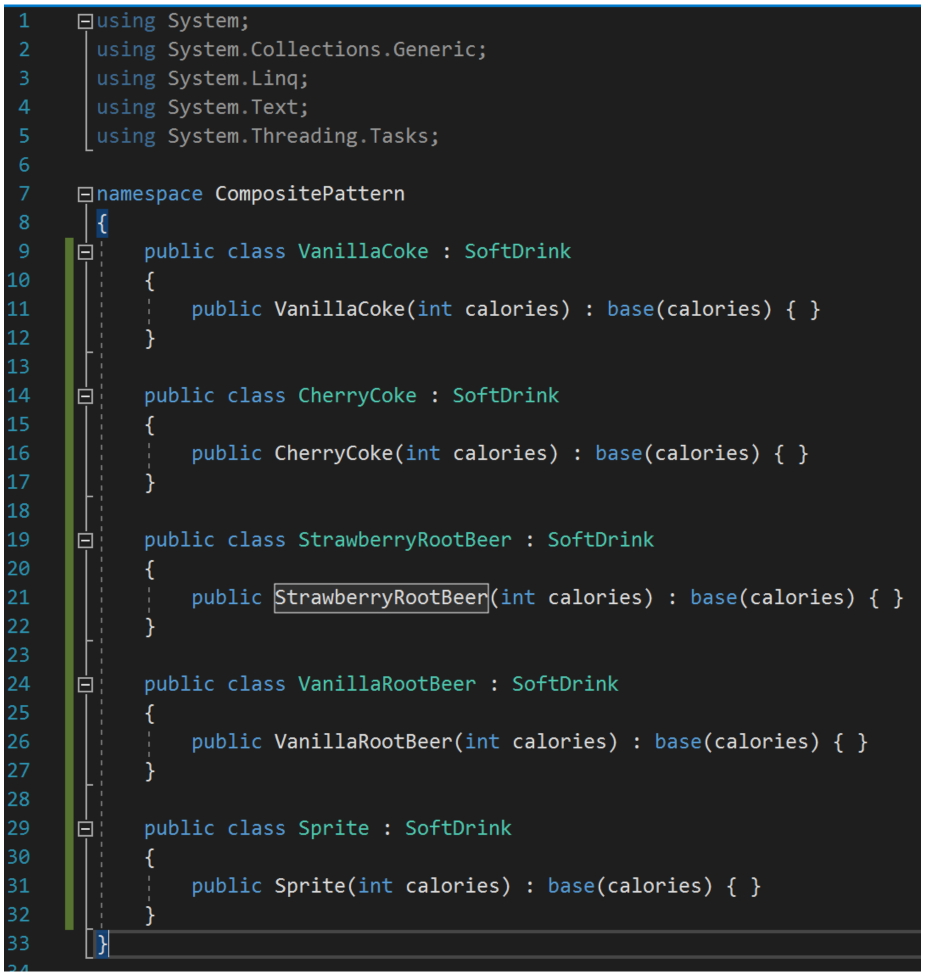
We then needed to implement the abstract class which included a method, DisplayCalories(), that is recursively called to print the number of calories in a soda for each node. The class can be seen in Figure 21, below.



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

namespace CompositePattern
{
    public abstract class SoftDrink
    {
        public int Calories { get; set; }

        public List<SoftDrink> Flavors { get; set; }

        public SoftDrink(int calories)
        {
            Calories = calories;
            Flavors = new List<SoftDrink>();
        }

        public void DisplayCalories()
        {
            Console.WriteLine(this.GetType().Name + ": " + this.Calories.ToString() + " calories.");
            foreach (var drink in this.Flavors)
            {
                drink.DisplayCalories();
            }
        }
    }
```

Figure 21: Abstract Class

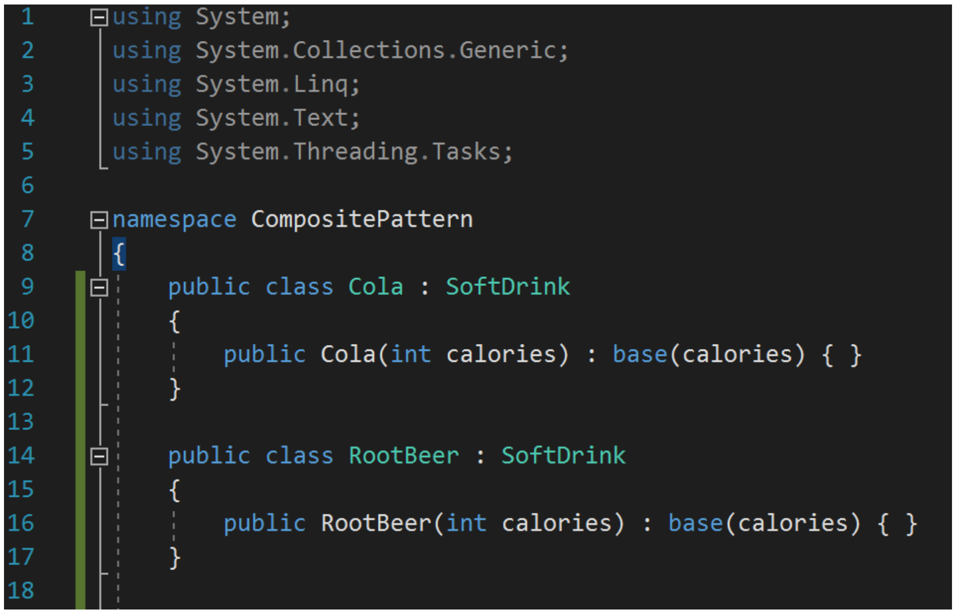
We then implemented the concrete classes for the different soda flavors as shown in Figure 22, below.



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

namespace CompositePattern
{
    public class VanillaCoke : SoftDrink
    {
        public VanillaCoke(int calories) : base(calories) { }
    }

    public class CherryCoke : SoftDrink
    {
        public CherryCoke(int calories) : base(calories) { }
    }

    public class StrawberryRootBeer : SoftDrink
    {
        public StrawberryRootBeer(int calories) : base(calories) { }
    }

    public class VanillaRootBeer : SoftDrink
    {
        public VanillaRootBeer(int calories) : base(calories) { }
    }

    public class Sprite : SoftDrink
    {
        public Sprite(int calories) : base(calories) { }
    }
}
```

Figure 22: Concrete Classes

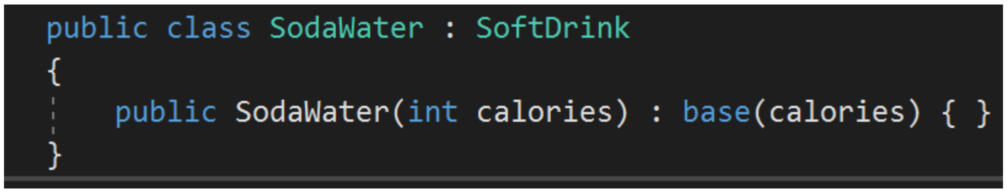
We then implemented the two composite components, Cola and RootBeer, which represent the objects with children as shown in Figure 23, below.



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

namespace CompositePattern
{
    public class Cola : SoftDrink
    {
        public Cola(int calories) : base(calories) { }
    }

    public class RootBeer : SoftDrink
    {
        public RootBeer(int calories) : base(calories) { }
    }

```

Figure 23: Composite Classes

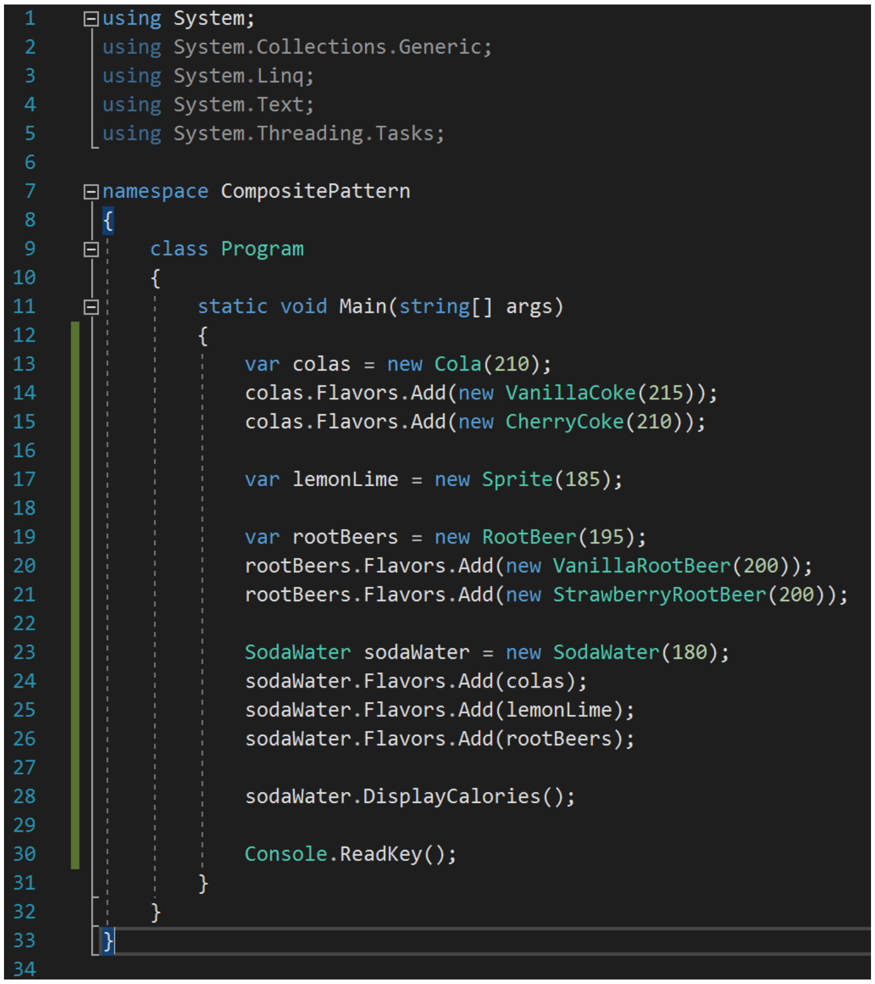
The last component to be added is a composite class to be used as the root node shown in Figure 24, below.



```
public class SodaWater : SoftDrink
{
    public SodaWater(int calories) : base(calories) { }
}
```

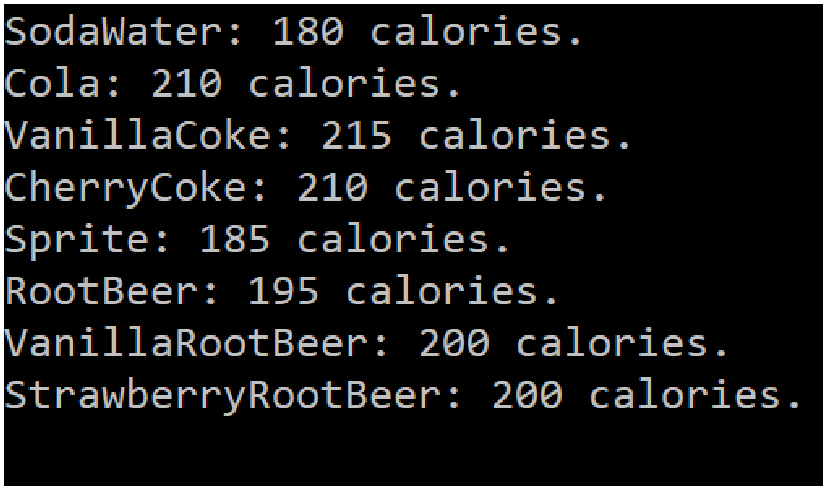
Figure 24: Composite Root Class

We could then utilize our hierarchy as shown in Figure 25 which yielded the output shown in Figure 26.



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

namespace CompositePattern
{
    class Program
    {
        static void Main(string[] args)
        {
            var colas = new Cola(210);
            colas.Flavors.Add(new VanillaCoke(215));
            colas.Flavors.Add(new CherryCoke(210));

            var lemonLime = new Sprite(185);

            var rootBeers = new RootBeer(195);
            rootBeers.Flavors.Add(new VanillaRootBeer(200));
            rootBeers.Flavors.Add(new StrawberryRootBeer(200));

            SodaWater sodaWater = new SodaWater(180);
            sodaWater.Flavors.Add(colas);
            sodaWater.Flavors.Add(lemonLime);
            sodaWater.Flavors.Add(rootBeers);

            sodaWater.DisplayCalories();

            Console.ReadKey();
        }
    }
}
```

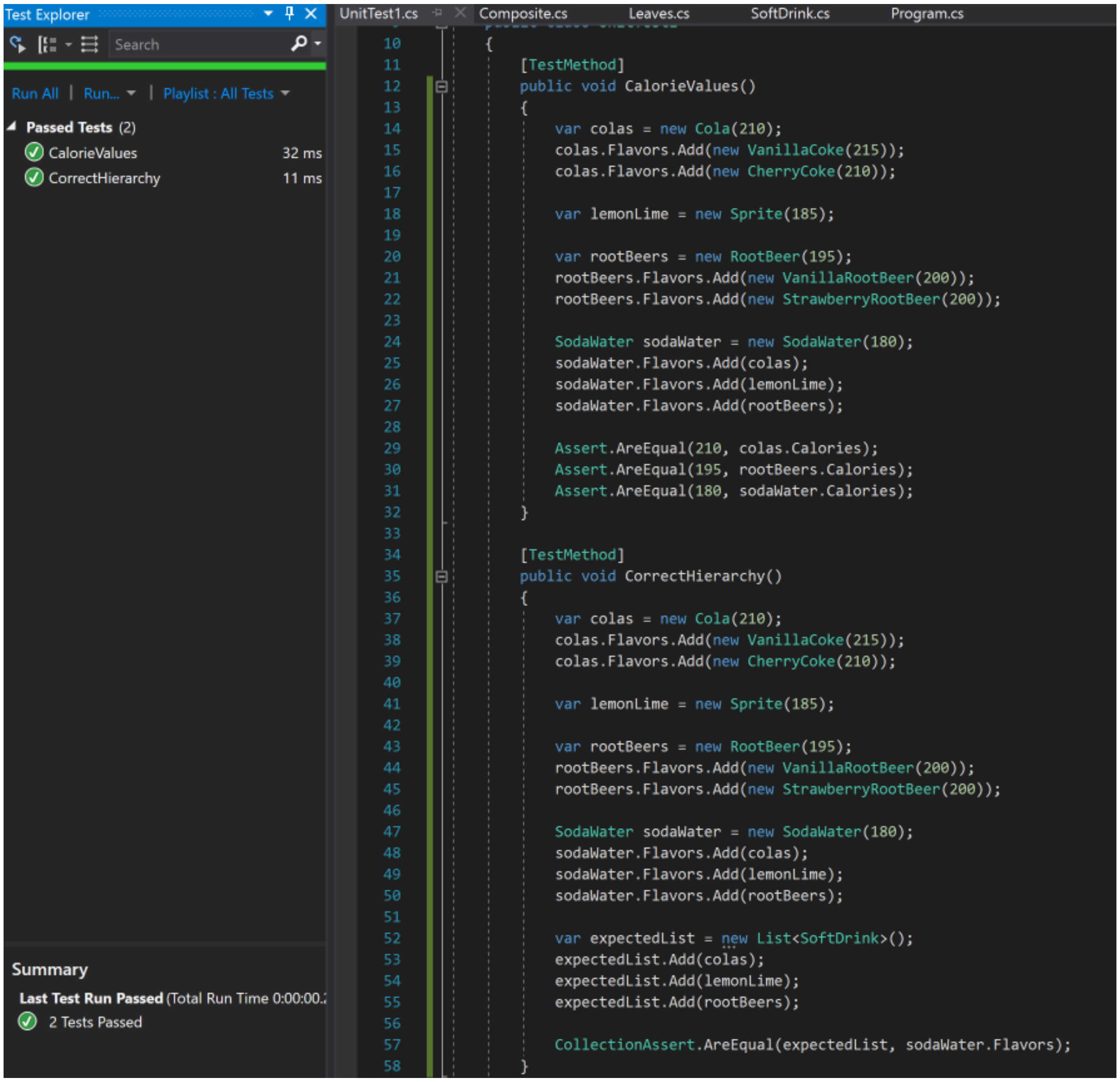
Figure 25: Composite Pattern Main



```
SodaWater: 180 calories.
Cola: 210 calories.
VanillaCoke: 215 calories.
CherryCoke: 210 calories.
Sprite: 185 calories.
RootBeer: 195 calories.
VanillaRootBeer: 200 calories.
StrawberryRootBeer: 200 calories.
```

Figure 26: Composite Pattern Output

Now that all the classes have been implemented, we can run our test suite and see that all test pass as shown in Figure 27, below.



```
Passed Tests (2)
  CalorieValues        32 ms
  CorrectHierarchy     11 ms

Summary
Last Test Run Passed (Total Run Time 0:00:00.
  2 Tests Passed
```

```csharp
{
    [TestMethod]
    public void CalorieValues()
    {
        var colas = new Cola(210);
        colas.Flavors.Add(new VanillaCoke(215));
        colas.Flavors.Add(new CherryCoke(210));

        var lemonLime = new Sprite(185);

        var rootBeers = new RootBeer(195);
        rootBeers.Flavors.Add(new VanillaRootBeer(200));
        rootBeers.Flavors.Add(new StrawberryRootBeer(200));

        SodaWater sodaWater = new SodaWater(180);
        sodaWater.Flavors.Add(colas);
        sodaWater.Flavors.Add(lemonLime);
        sodaWater.Flavors.Add(rootBeers);

        Assert.AreEqual(210, colas.Calories);
        Assert.AreEqual(195, rootBeers.Calories);
        Assert.AreEqual(180, sodaWater.Calories);
    }

    [TestMethod]
    public void CorrectHierarchy()
    {
        var colas = new Cola(210);
        colas.Flavors.Add(new VanillaCoke(215));
        colas.Flavors.Add(new CherryCoke(210));

        var lemonLime = new Sprite(185);

        var rootBeers = new RootBeer(195);
        rootBeers.Flavors.Add(new VanillaRootBeer(200));
        rootBeers.Flavors.Add(new StrawberryRootBeer(200));

        SodaWater sodaWater = new SodaWater(180);
        sodaWater.Flavors.Add(colas);
        sodaWater.Flavors.Add(lemonLime);
        sodaWater.Flavors.Add(rootBeers);

        var expectedList = new List<SoftDrink>();
        expectedList.Add(colas);
        expectedList.Add(lemonLime);
        expectedList.Add(rootBeers);

        CollectionAssert.AreEqual(expectedList, sodaWater.Flavors);
    }
```

Figure 27: Passed Test

As seen from this section of the project, the Composite Pattern is a useful way to implement a hierarchy where the client can apply similar functions to all parts. Developers should be careful when using this pattern as to what design they follow, uniformity or type safety. Uniformity allows the client to treat leaves and composites the same but the type can be lost as a leaf can perform a function only a composite should. It is better to follow the type safety design as this project has which implements the leaves and composites separately therefore preserving the type.

State Pattern

The State Design Pattern is used when there is one too many relationships between objects such that if one object is modified, its dependent objects are to be notified automatically. This pattern is used to alter the behavior of an object when its internal state changes. In this pattern, an object is created which represents various states and a context object whose behavior varies as its state object changes. See Figure 28 for the State Pattern class diagram.

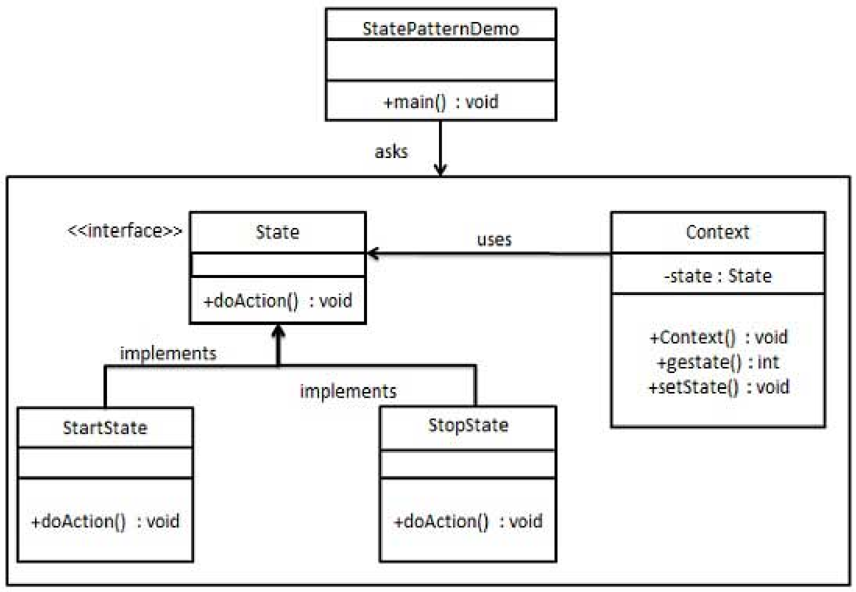


Figure 28. State Pattern Class Diagram

In the above diagram, the State interface defines an action and concrete classes to implement the State interface. Context represents a class which carries a State. The demo class, StatePatternDemo, uses Context and state objects to demonstrate change in Context behavior based on which state it is in.

Let's begin creating our own example of a State Pattern. For this example, we will create a class named ParkGark that might be used in a theme park simulation video game. Let's allow the park guest to have three different states: RoamingInPark, InQueue, and OnRide. To transition between states, we will write three methods: EnterQueue, GetOnRide, and ExitRide. See Figure 29 for a simple diagram of the three states and the required action to transition between each state.

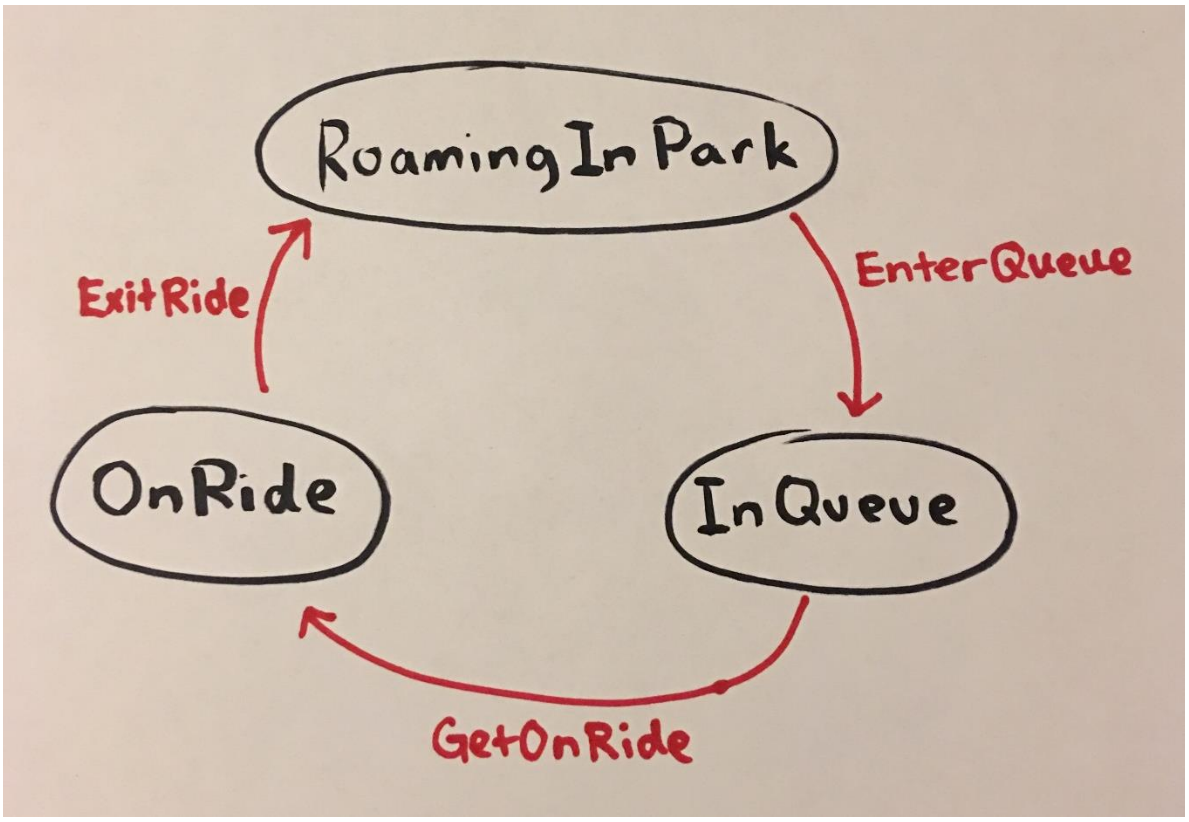
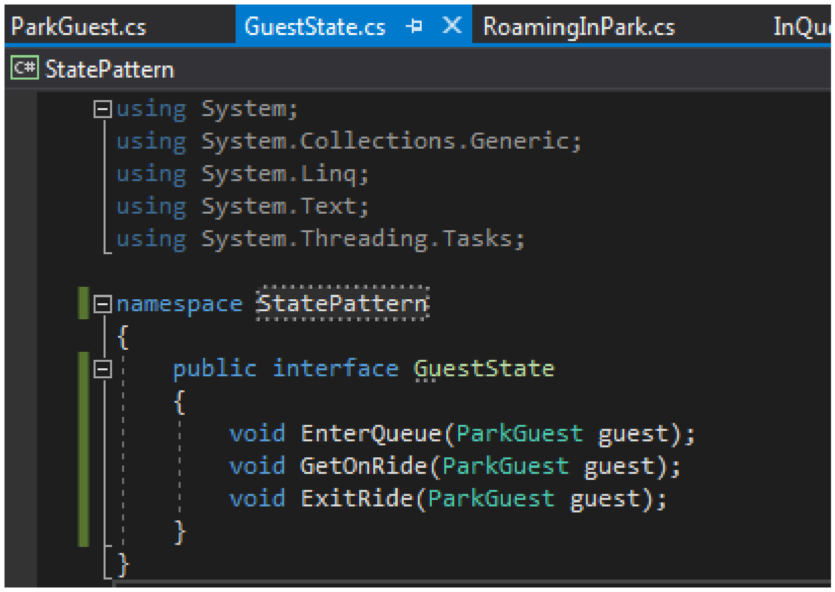


Figure 29. Diagram Sketch

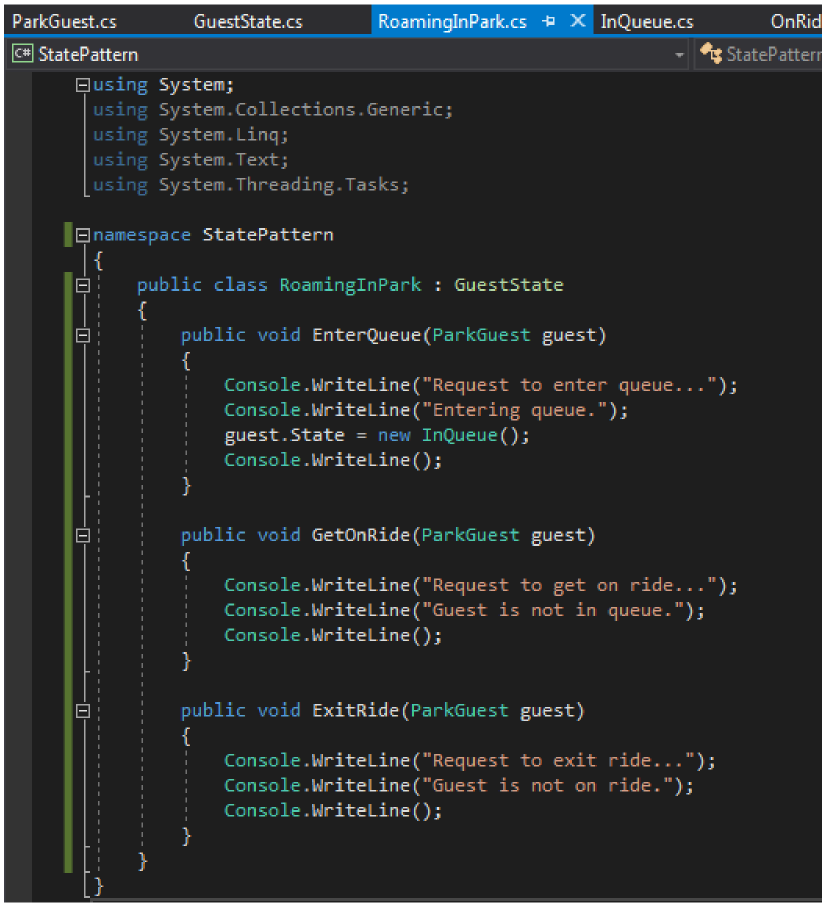
It’s time to begin writing this in code. First, we will create an interface for the state. See Figure 30.



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

namespace StatePattern
{
    public interface GuestState
    {
        void EnterQueue(ParkGuest guest);
        void GetOnRide(ParkGuest guest);
        void ExitRide(ParkGuest guest);
    }
}
```

Figure 30. GuestState Interface

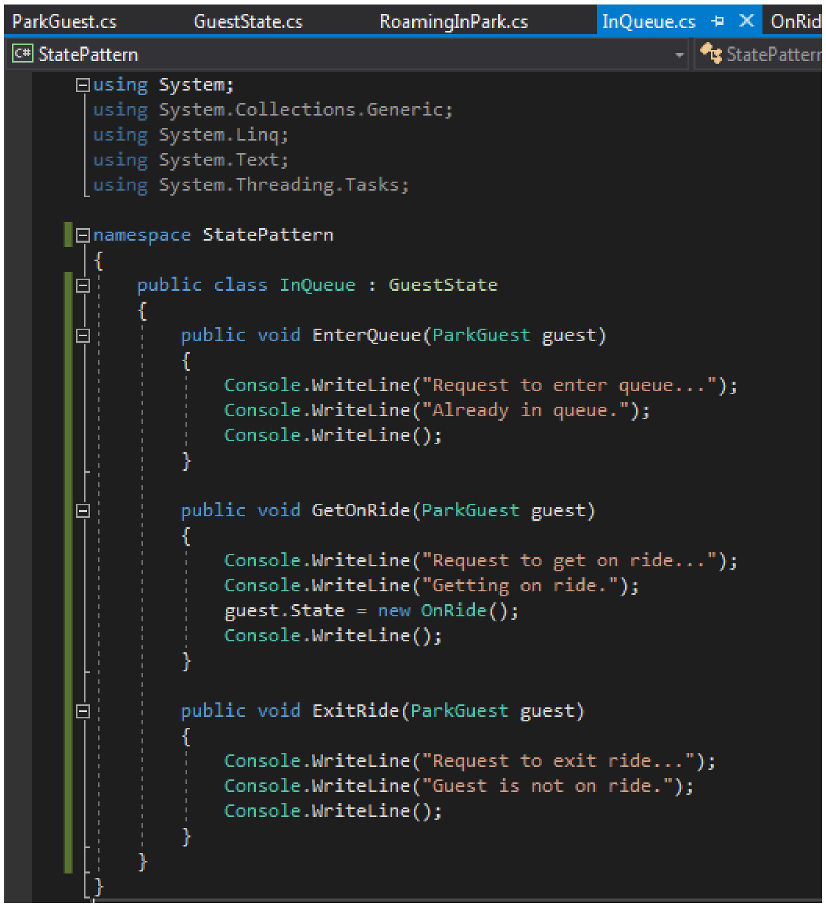
Let’s now create our three concrete classes to implement the state interface. See Figures 31 through 33.



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

namespace StatePattern
{
    public class RoamingInPark : GuestState
    {
        public void EnterQueue(ParkGuest guest)
        {
            Console.WriteLine("Request to enter queue...");
            Console.WriteLine("Entering queue.");
            guest.State = new InQueue();
            Console.WriteLine();
        }

        public void GetOnRide(ParkGuest guest)
        {
            Console.WriteLine("Request to get on ride...");
            Console.WriteLine("Guest is not in queue.");
            Console.WriteLine();
        }

        public void ExitRide(ParkGuest guest)
        {
            Console.WriteLine("Request to exit ride...");
            Console.WriteLine("Guest is not on ride.");
            Console.WriteLine();
        }
    }
}
```

Figure 31. RoamingInPark Class

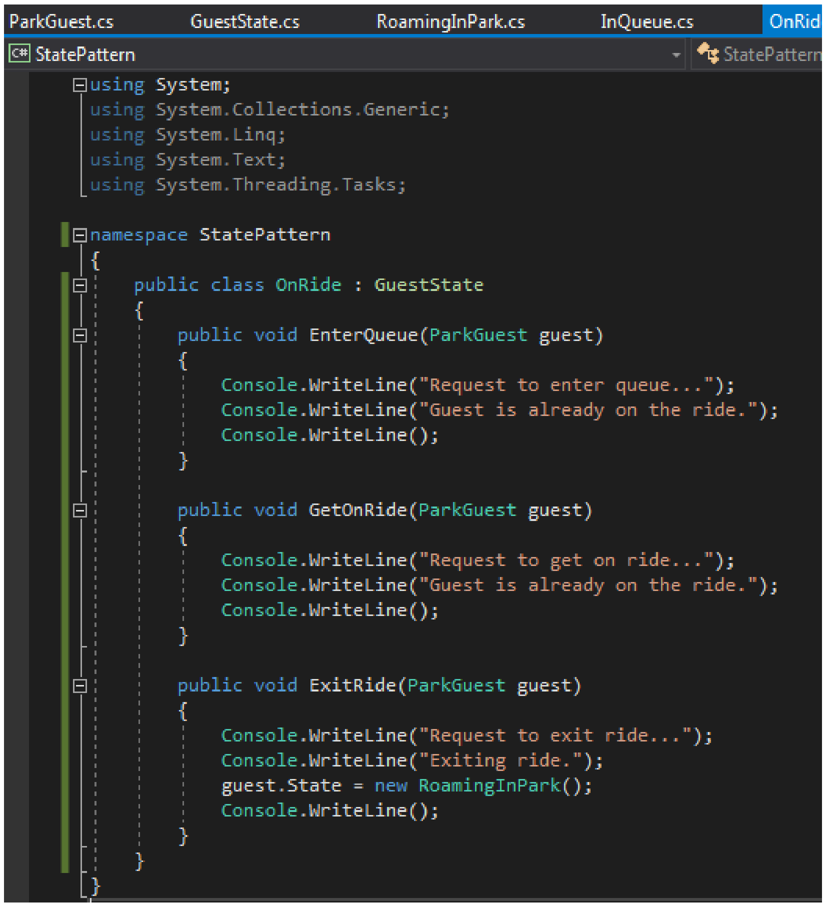
Notice how each method begins with a request. In the case of RoamingInPark, the only method that causes a state in change is the EnterQueue method. This causes a state in change from RoamingInPark to InQueue. The GetOnRide and ExitRide methods only return a message and do not result in a change of state.



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

namespace StatePattern
{
    public class InQueue : GuestState
    {
        public void EnterQueue(ParkGuest guest)
        {
            Console.WriteLine("Request to enter queue...");
            Console.WriteLine("Already in queue.");
            Console.WriteLine();
        }

        public void GetOnRide(ParkGuest guest)
        {
            Console.WriteLine("Request to get on ride...");
            Console.WriteLine("Getting on ride.");
            guest.State = new OnRide();
            Console.WriteLine();
        }

        public void ExitRide(ParkGuest guest)
        {
            Console.WriteLine("Request to exit ride...");
            Console.WriteLine("Guest is not on ride.");
            Console.WriteLine();
        }
    }
}
```

Figure 32. InQueue Class

In the InQueue class, the only method that causes a state in change is the GetOnRide method. This causes a state in change from InQueue to OnRide. The EnterQueue and ExitRide methods only return a message and do not result in a change of state.

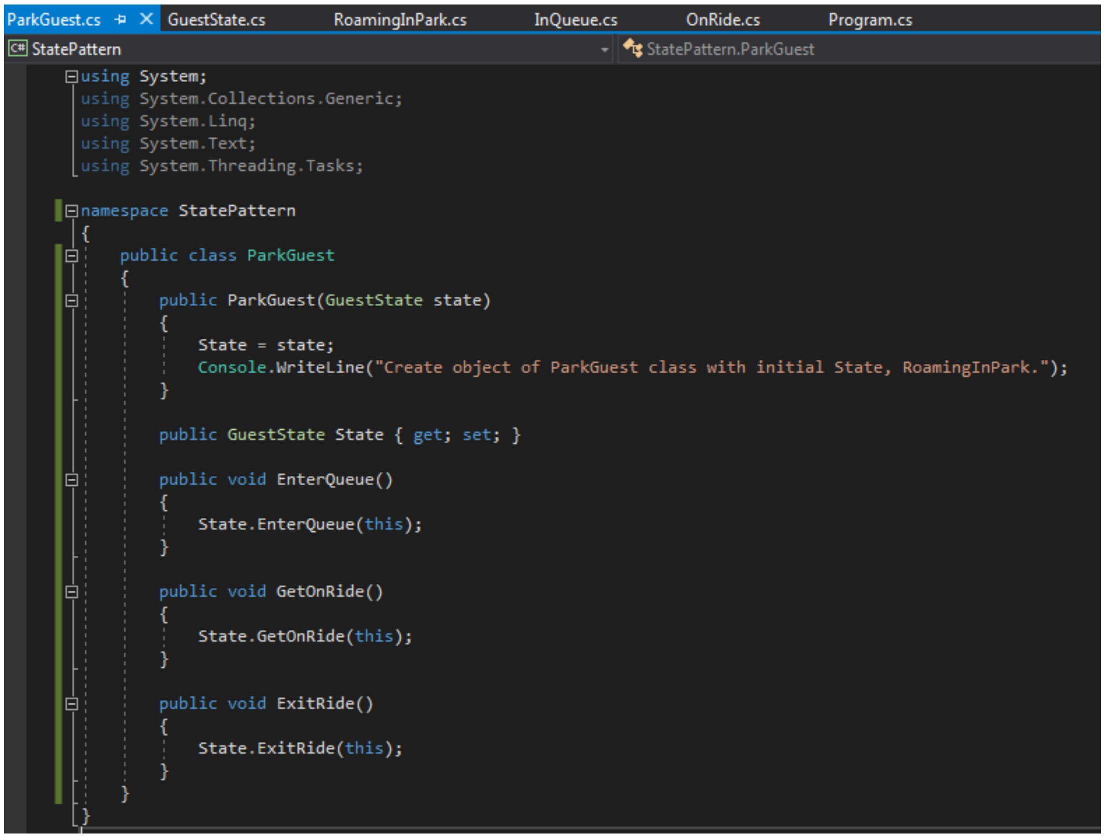


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace StatePattern
{
    public class OnRide : GuestState
    {
        public void EnterQueue(ParkGuest guest)
        {
            Console.WriteLine("Request to enter queue...");
            Console.WriteLine("Guest is already on the ride.");
            Console.WriteLine();
        }

        public void GetOnRide(ParkGuest guest)
        {
            Console.WriteLine("Request to get on ride...");
            Console.WriteLine("Guest is already on the ride.");
            Console.WriteLine();
        }

        public void ExitRide(ParkGuest guest)
        {
            Console.WriteLine("Request to exit ride...");
            Console.WriteLine("Exiting ride.");
            guest.State = new RoamingInPark();
            Console.WriteLine();
        }
    }
}
```

Figure 33. OnRide Class

In the OnRide class, the only method that causes a state in change is the ExitRide method. This causes a state in change from OnRide to RoamingInPark. The EnterQueue and GetOnRide methods only return a message and do not result in a change of state.

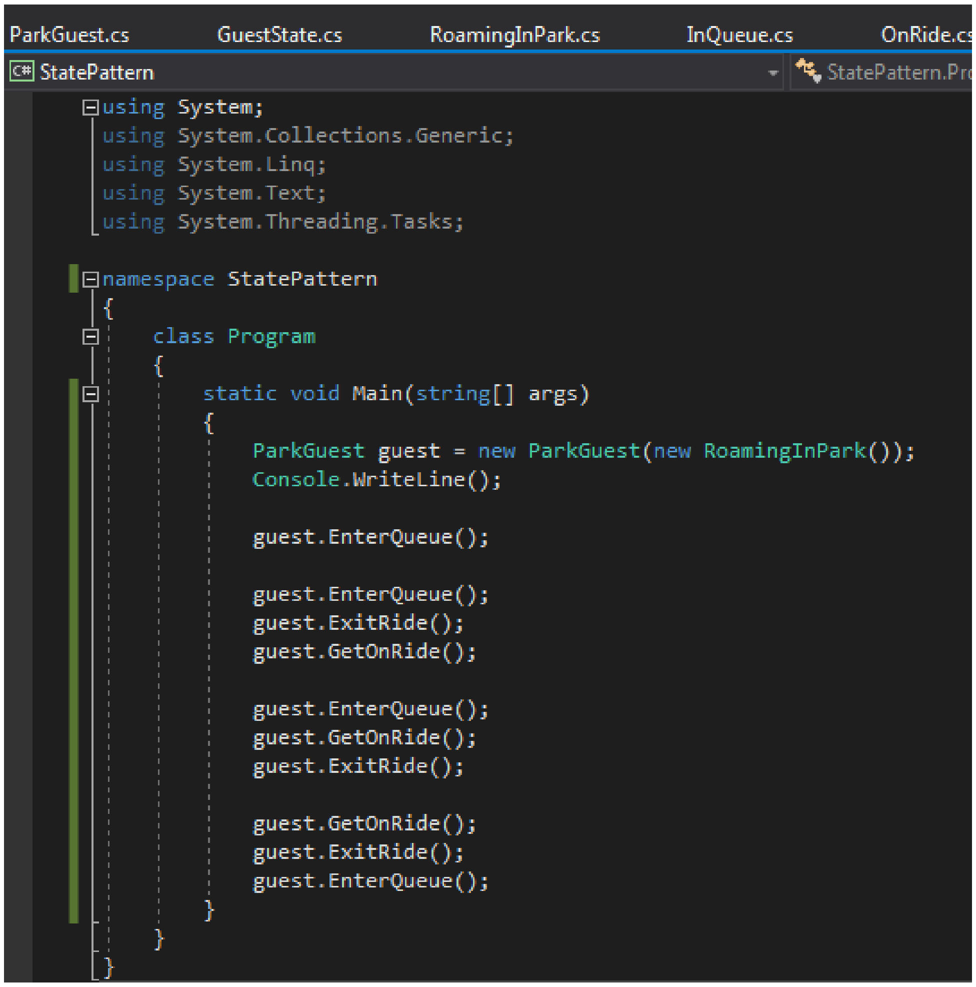
Now we need to create the ParkGuest Class as shown in Figure 34, below.



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

namespace StatePattern
{
    public class ParkGuest
    {
        public ParkGuest(GuestState state)
        {
            State = state;
            Console.WriteLine("Create object of ParkGuest class with initial State, RoamingInPark.");
        }

        public GuestState State { get; set; }

        public void EnterQueue()
        {
            State.EnterQueue(this);
        }

        public void GetOnRide()
        {
            State.GetOnRide(this);
        }

        public void ExitRide()
        {
            State.ExitRide(this);
        }
    }
}
```

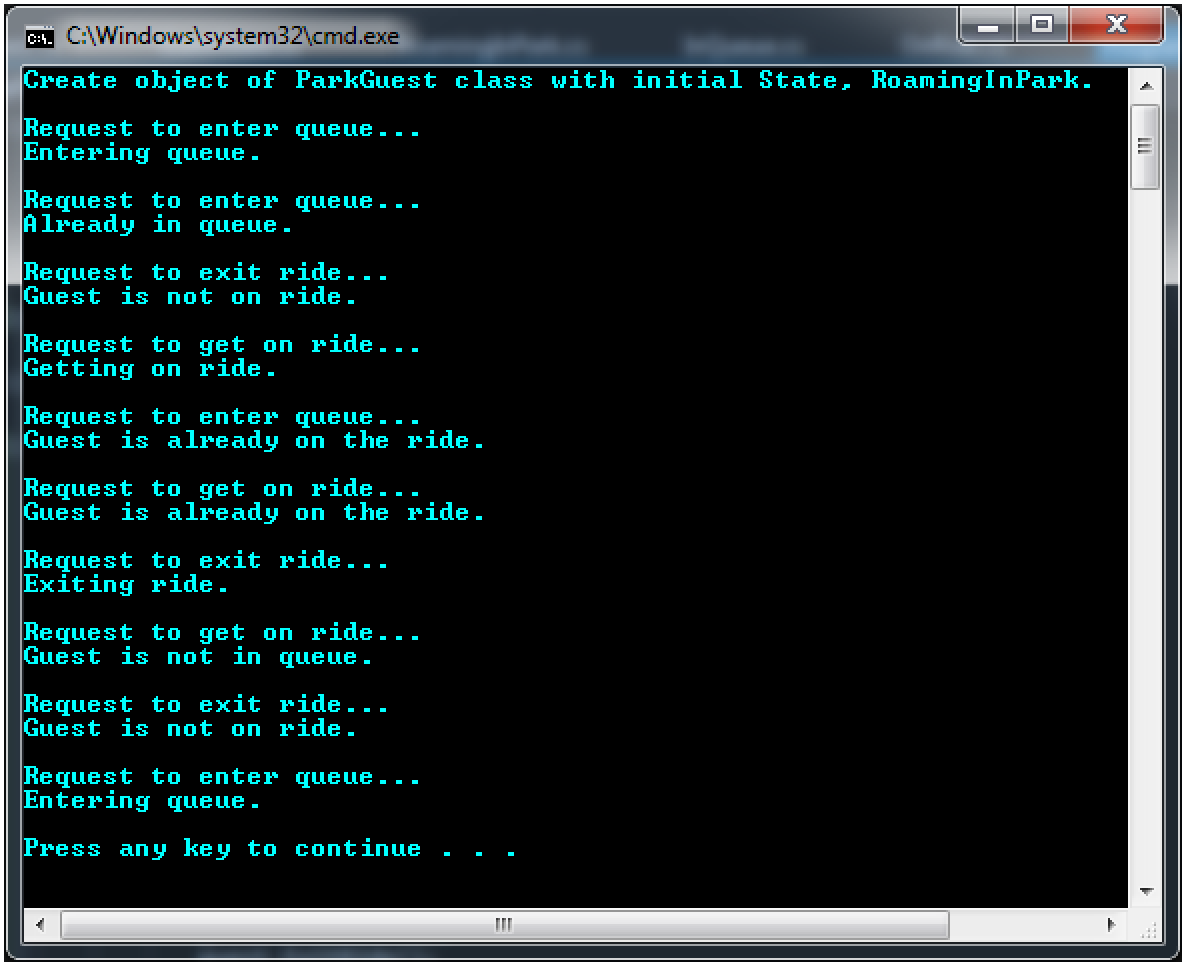
Figure 34. ParkGuest Class

Now that all our classes have been created, we can write code to test the behavior when GuestState changes. Follow along between the code and the output to verify it works properly. See Figure 35 and Figure 36.



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

namespace StatePattern
{
    class Program
    {
        static void Main(string[] args)
        {
            ParkGuest guest = new ParkGuest(new RoamingInPark());
            Console.WriteLine();

            guest.EnterQueue();

            guest.EnterQueue();
            guest.ExitRide();
            guest.GetOnRide();

            guest.EnterQueue();
            guest.GetOnRide();
            guest.ExitRide();

            guest.GetOnRide();
            guest.ExitRide();
            guest.EnterQueue();
        }
    }
}
```

Figure 35. Test Code



```
Create object of ParkGuest class with initial State, RoamingInPark.

Request to enter queue...
Entering queue.

Request to enter queue...
Already in queue.

Request to exit ride...
Guest is not on ride.

Request to get on ride...
Getting on ride.

Request to enter queue...
Guest is already on the ride.

Request to get on ride...
Guest is already on the ride.

Request to exit ride...
Exiting ride.

Request to get on ride...
Guest is not in queue.

Request to exit ride...
Guest is not on ride.

Request to enter queue...
Entering queue.

Press any key to continue . . .
```

Figure 36. State Pattern Output

The output shows each requested action and the result of the request. The code performs as expected, only changing state based on the sketched diagram in Figure 29.

Proxy Pattern

We’ve all seen examples of the proxy pattern even if we didn’t know it at the time. For example, take a receptionist. When you contact an office in search of an employee, the receptionist answers the call. The receptionist can perform many of the same actions as the employee, such as answering the call, setting up meetings, and providing information about the company. However, the receptionist cannot do everything the employee can do. In this case, the receptionist must make a request to the employee for the desired action. In this way, the receptionist acts as a proxy for the employee. See Figure 37, below, for the UML diagram of the Proxy Pattern.

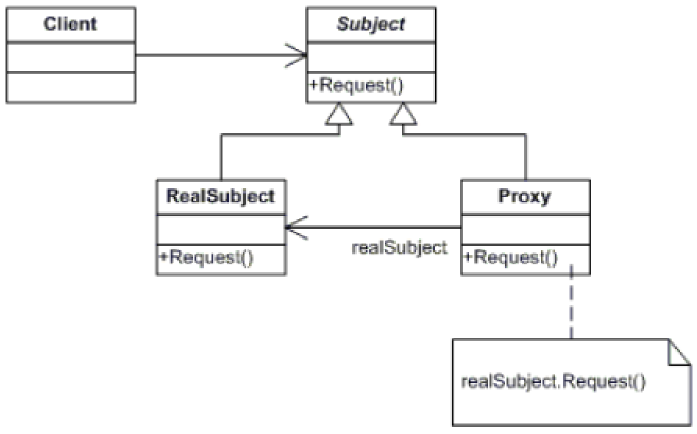
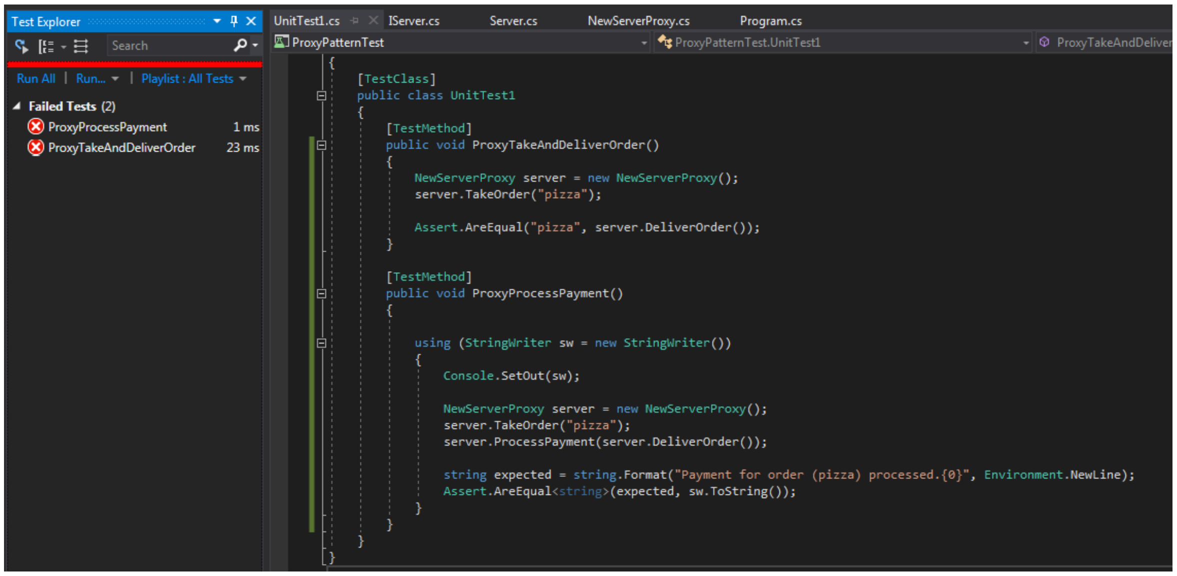


Figure 37: Proxy Pattern UML Diagram

For the given example, the receptionist acts as the Proxy while the employee acts as the RealSubject. The Proxy Pattern is useful because it controls who or what has access to an object. Additional functionality is then added when access to the object is granted. This pattern is practical in the real world because it can limit access to a server.

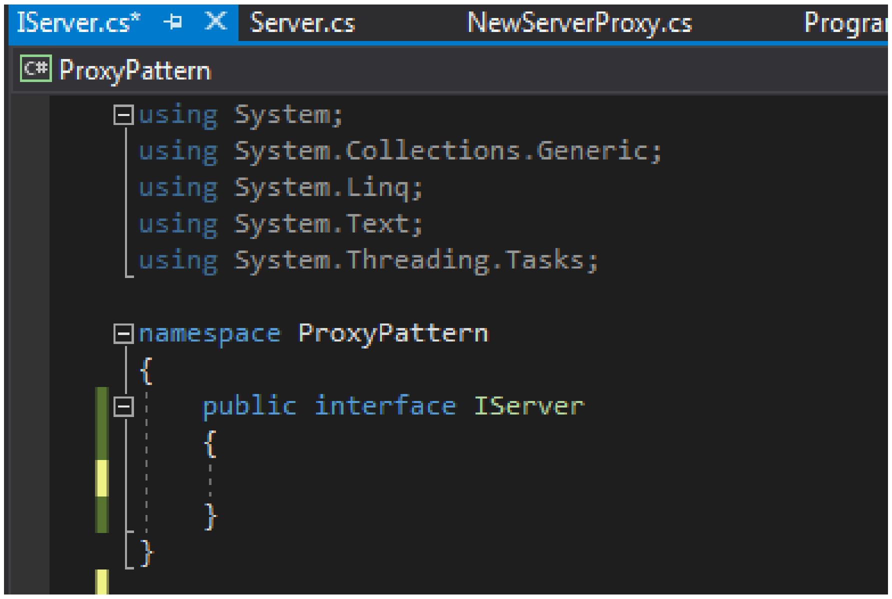
To demonstrate the Proxy Pattern, we have designed a waiter application where an experienced waiter will serve as the RealSubject and a training waiter will be the Proxy. We also have an interface to connect the RealSubject and Proxy which will be the Subject. Before setting up the classes, we wrote a test suite which can be seen in Figure 38, below.



```
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void ProxyTakeAndDeliverOrder()
        {
            NewServerProxy server = new NewServerProxy();
            server.TakeOrder("pizza");

            Assert.AreEqual("pizza", server.DeliverOrder());
        }

        [TestMethod]
        public void ProxyProcessPayment()
        {

            using (StringWriter sw = new StringWriter())
            {
                Console.SetOut(sw);

                NewServerProxy server = new NewServerProxy();
                server.TakeOrder("pizza");
                server.ProcessPayment(server.DeliverOrder());

                string expected = string.Format("Payment for order (pizza) processed.{0}", Environment.NewLine);
                Assert.AreEqual<string>(expected, sw.ToString());
            }
        }
    }
}
```

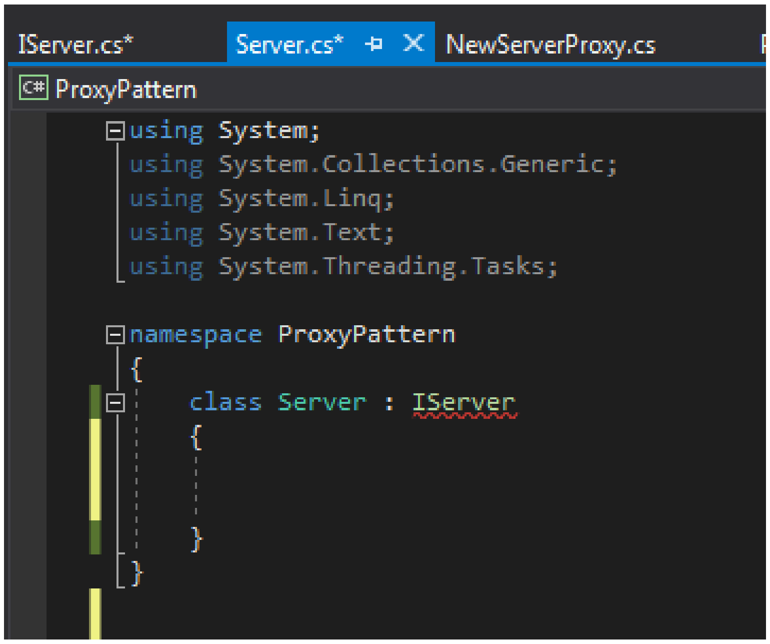
Figure 38: Proxy Test Suite

After writing the test suit, we outlined the Subject, Proxy, and RealSubject classes which can be seen in Figures 39 through 41, below.



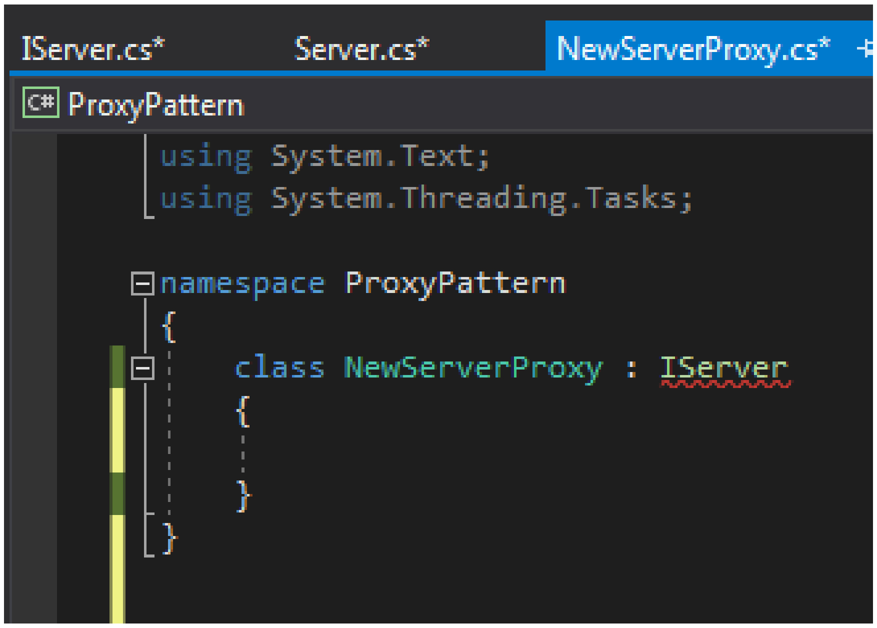
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProxyPattern
{
    public interface IServer
    {

    }
}
```

Figure 39: Subject Class



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

namespace ProxyPattern
{
    class Server : IServer
    {


    }
}
```

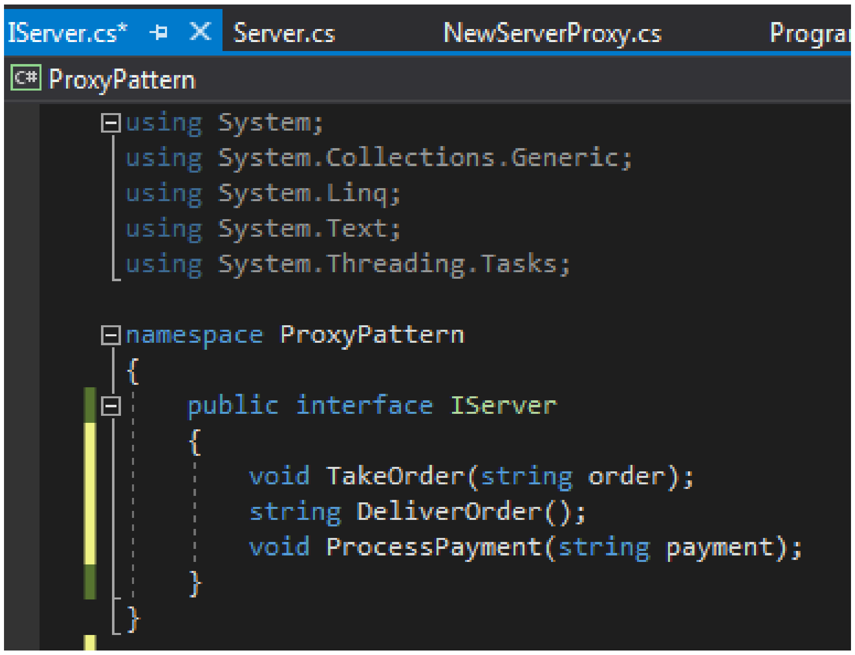
Figure 40: RealSubject Class



```
using System.Text;
using System.Threading.Tasks;

namespace ProxyPattern
{
    class NewServerProxy : IServer
    {

    }
}
```

Figure 41: Proxy Class

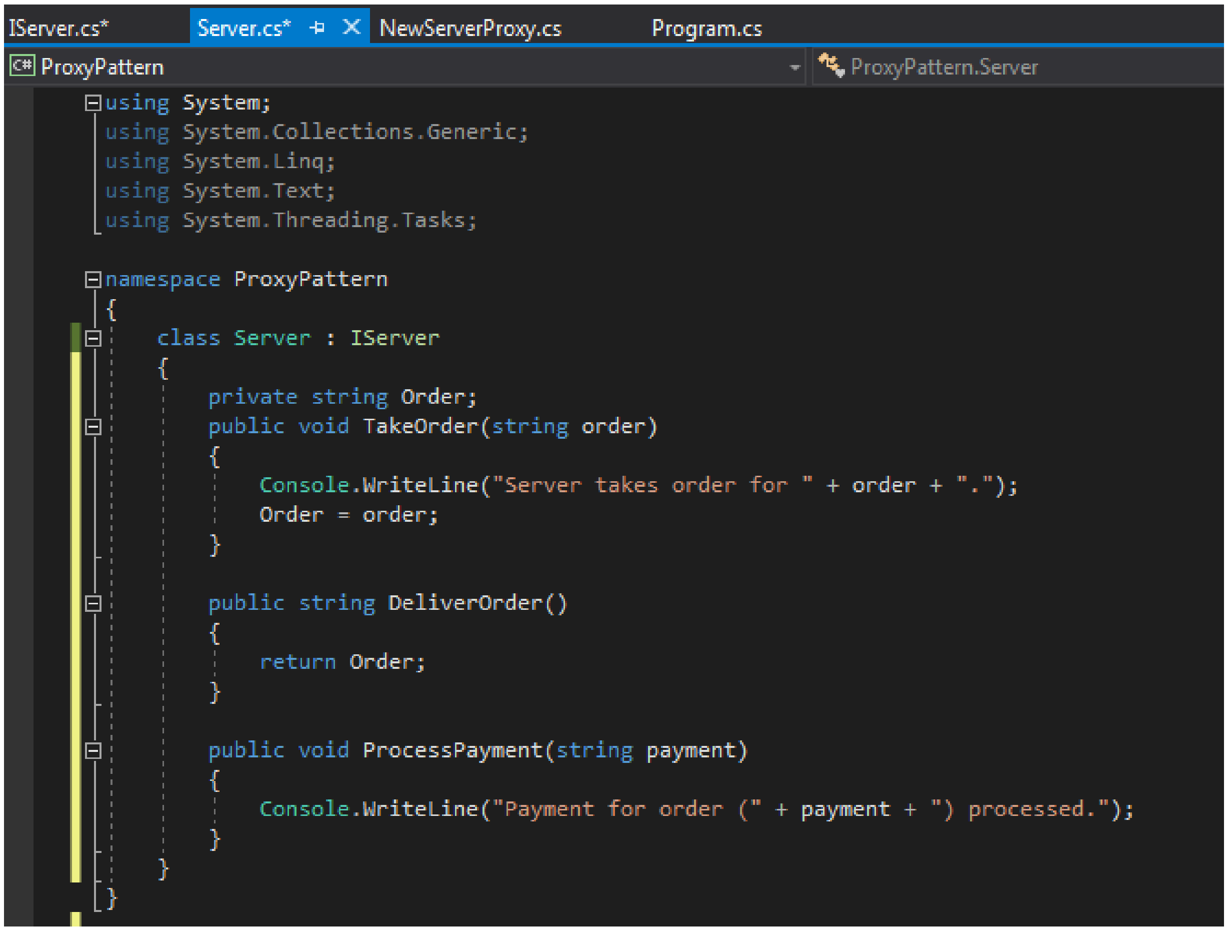
After outlining each class, we needed to implement the interface that would be used to connect the RealSubject and Proxy together. As seen in Figure 42, below, the interface has three methods. TakeOrder(), DeliverOrder(), and ProcessPayment(). The Proxy can take and deliver an order but cannot process a payment. Only the RealSubject can process a payment. The implementations of the Proxy and RealSubject will be shown next.



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

namespace ProxyPattern
{
    public interface IServer
    {
        void TakeOrder(string order);
        string DeliverOrder();
        void ProcessPayment(string payment);
    }
}
```

Figure 42: Implemented Subject Class

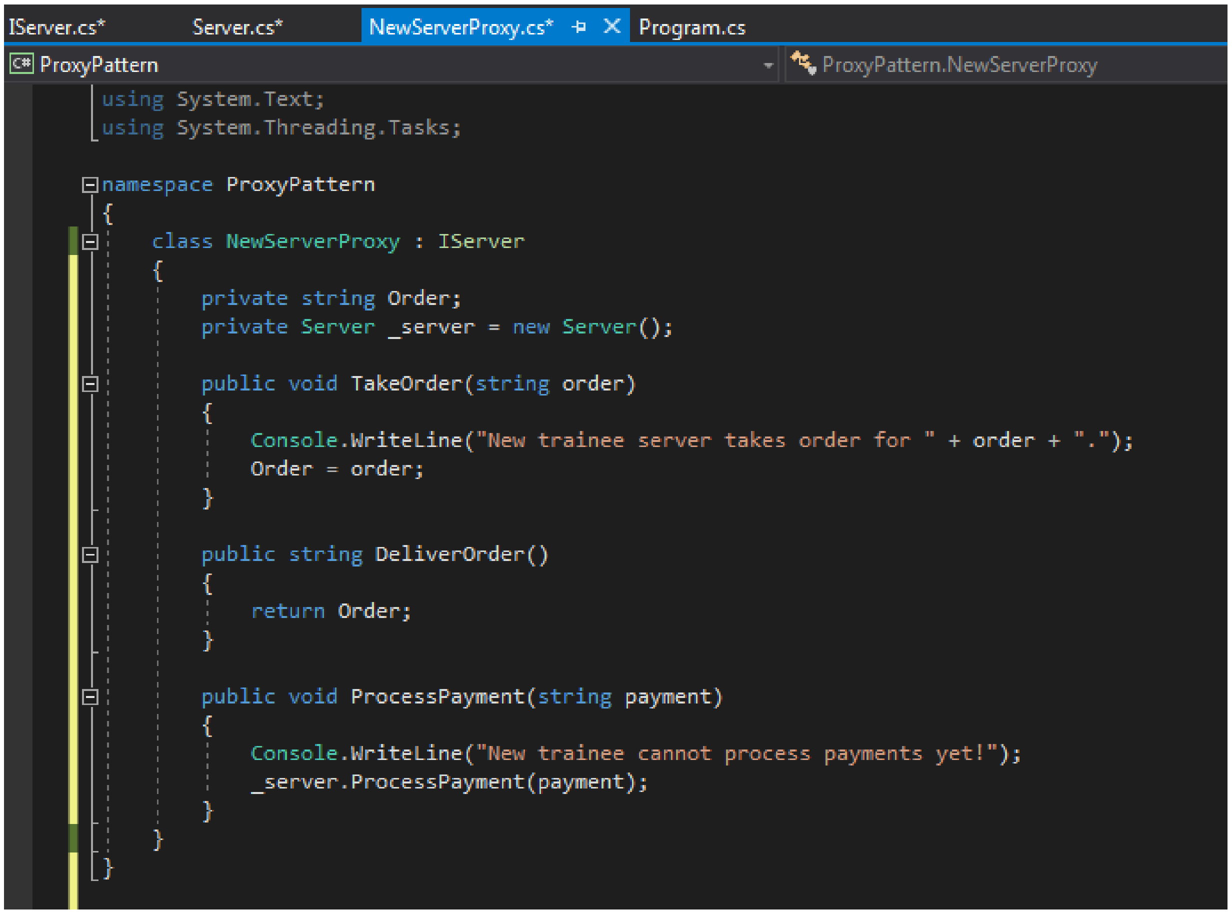
Once the interface or Subject had been implemented, we wrote the code for the server or RealSubject. As shown in Figure 43, below, the server implements all the methods of the interface.



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

namespace ProxyPattern
{
    class Server : IServer
    {
        private string Order;
        public void TakeOrder(string order)
        {
            Console.WriteLine("Server takes order for " + order + ".");
            Order = order;
        }

        public string DeliverOrder()
        {
            return Order;
        }

        public void ProcessPayment(string payment)
        {
            Console.WriteLine("Payment for order (" + payment + ") processed.");
        }
    }
}
```

Figure 43: Implemented RealSubject Class

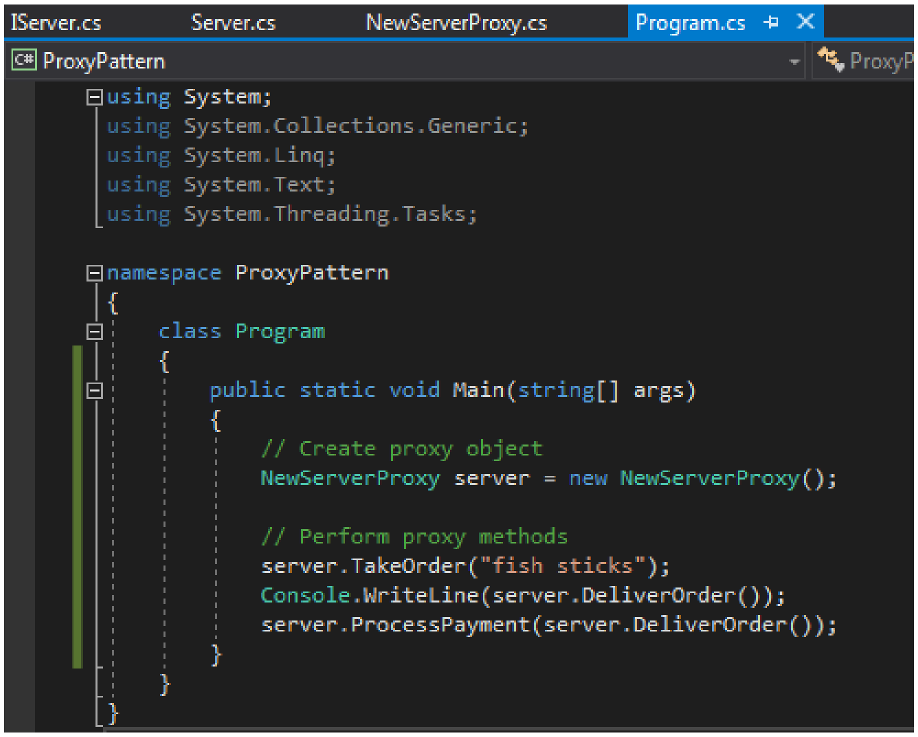
The final class for the Proxy Pattern is the new server or Proxy. As shown in Figure 44, below, the Proxy implements its own TakeOrder() and DeliverOrder() but when ProcessPayment() is called, it must create a server object.



```csharp
using System.Text;
using System.Threading.Tasks;

namespace ProxyPattern
{
    class NewServerProxy : IServer
    {
        private string Order;
        private Server _server = new Server();

        public void TakeOrder(string order)
        {
            Console.WriteLine("New trainee server takes order for " + order + ".");
            Order = order;
        }

        public string DeliverOrder()
        {
            return Order;
        }

        public void ProcessPayment(string payment)
        {
            Console.WriteLine("New trainee cannot process payments yet!");
            _server.ProcessPayment(payment);
        }
    }
}
```

Figure 44: Implemented Proxy Class

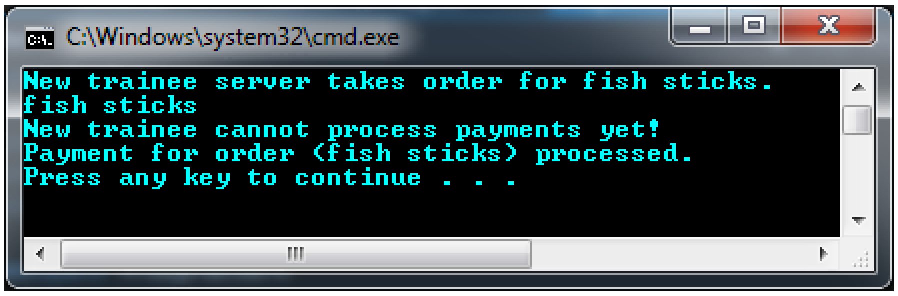
After finishing setting up our Proxy Pattern, we wrote a main method to demonstrate the pattern and write our output to the command prompt. See Figure 45, below, for the main method.



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

namespace ProxyPattern
{
    class Program
    {
        public static void Main(string[] args)
        {
            // Create proxy object
            NewServerProxy server = new NewServerProxy();

            // Perform proxy methods
            server.TakeOrder("fish sticks");
            Console.WriteLine(server.DeliverOrder());
            server.ProcessPayment(server.DeliverOrder());
        }
    }
}
```

Figure 45: Proxy Pattern Main

We started by creating a NewServerProxy object. All three methods were called to show how the proxy object handled each request. Figure 46, below, shows the output and describes how the pattern handled these requests.

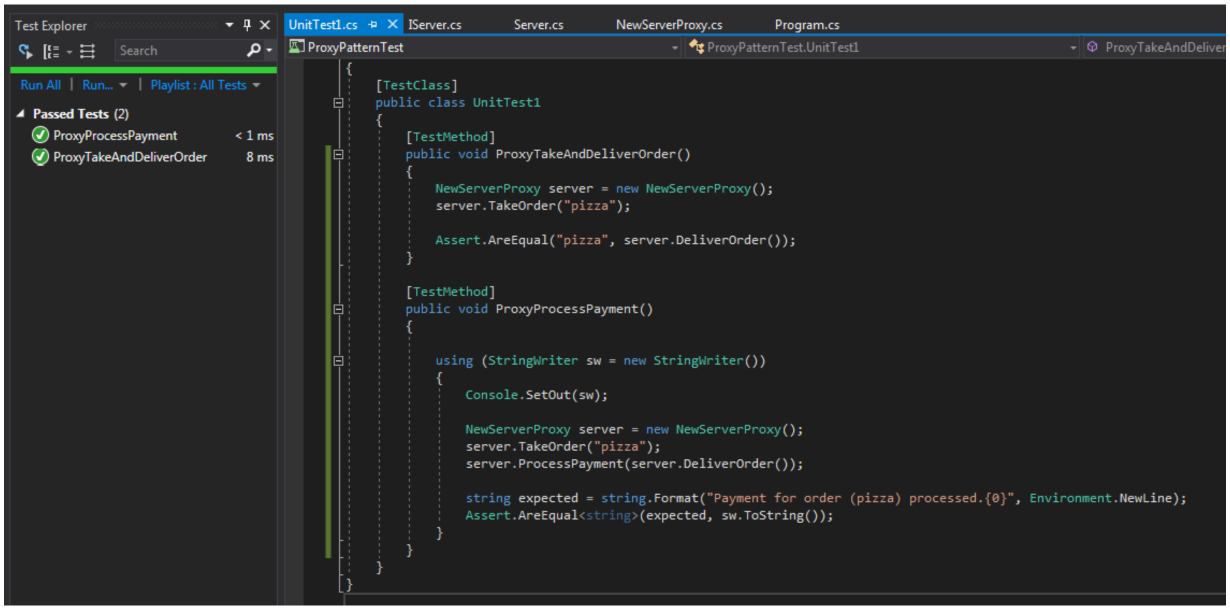


```
New trainee server takes order for fish sticks.
fish sticks
New trainee cannot process payments yet!
Payment for order (fish sticks) processed.
Press any key to continue . . .
```

Figure 46: Proxy Pattern Output

The proxy object, *server*, was able to complete the TakeOrder method and return the order as a string in the DeliverOrder method. However, when we request *server* to perform the ProcessPayment method, *server* was not allowed to complete that request. Instead, it passed the request on to a new object in the Server class, which was then able to perform the request and process the payment. This is a successful demonstration for how the Proxy Pattern works.

Now that all classes have been implemented, we ran our tests again, and as shown in Figure 47, all tests have passed.



```
Passed Tests (2)
    ProxyProcessPayment        < 1 ms
    ProxyTakeAndDeliverOrder     8 ms
```

```
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void ProxyTakeAndDeliverOrder()
        {
            NewServerProxy server = new NewServerProxy();
            server.TakeOrder("pizza");

            Assert.AreEqual("pizza", server.DeliverOrder());
        }

        [TestMethod]
        public void ProxyProcessPayment()
        {

            using (StringWriter sw = new StringWriter())
            {
                Console.SetOut(sw);

                NewServerProxy server = new NewServerProxy();
                server.TakeOrder("pizza");
                server.ProcessPayment(server.DeliverOrder());

                string expected = string.Format("Payment for order (pizza) processed.{0}", Environment.NewLine);
                Assert.AreEqual<string>(expected, sw.ToString());
            }
        }
    }
}
```

Figure 47: Successful Proxy Pattern Test

As this example of the Proxy Pattern has shown, a proxy can be a useful way to control access to an object. This pattern has many real-world applications such as logins to applications and banking software.

References

[1] E. Freeman, E. Robson, and M. Hendrickson, Head first design patterns. Beijing: OReilly, 2014.

[2] “Composite,” Composite .NET Design Pattern in C# and VB. [Online]. Available: http://www.dofactory.com/net/composite-design-pattern. [Accessed: 11-Dec-2017].

[3] “Proxy pattern,” Wikipedia, 26-Nov-2017. [Online]. Available: https://en.wikipedia.org/wiki/Proxy_pattern. [Accessed: 11-Dec-2017].

[4] BillWagner, “Interfaces (C# Programming Guide),” Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/. [Accessed: 11-Dec-2017].

[5] T. Point, “Design Patterns State Pattern,” www.tutorialspoint.com, 15-Aug-2017. [Online]. Available: https://www.tutorialspoint.com/design_pattern/state_pattern.htm. [Accessed: 11-Dec-2017].