**11/29/2017**

**Head First Design Patterns**

**Project 4 Report**

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

**Table of Contents**

[Table of Figures 3](#_Toc499720011)

[Adapter Pattern 4](#_Toc499720012)

[References 13](#_Toc499720013)

# **Table of Figures**

[Figure 1. Adapter Design 4](#_Toc499719938)

[Figure 2. Adapter and Target Design 5](#_Toc499719939)

[Figure 3. Initial Failed Test 6](#_Toc499719940)

[Figure 4. Initial Dog Class 7](#_Toc499719941)

[Figure 5. Initial Beagle Class 7](#_Toc499719942)

[Figure 6. Initial Wolf Class 7](#_Toc499719943)

[Figure 7. Initial WildWolf Class 8](#_Toc499719944)

[Figure 8. WolfAdapter Class 9](#_Toc499719945)

[Figure 9. Refactored Wolf Class 9](#_Toc499719946)

[Figure 10. Refactored WildWolf Class 10](#_Toc499719947)

[Figure 11. Refactored Dog Class 10](#_Toc499719948)

[Figure 12. Refactored Beagle Class 11](#_Toc499719949)

[Figure 13. Successful Test 12](#_Toc499719950)

# **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.



Figure 1. Adapter Design

The adapter pattern “converts the interface of a class into another interface the clients expect” (Freeman 243). 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.



Figure . 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.

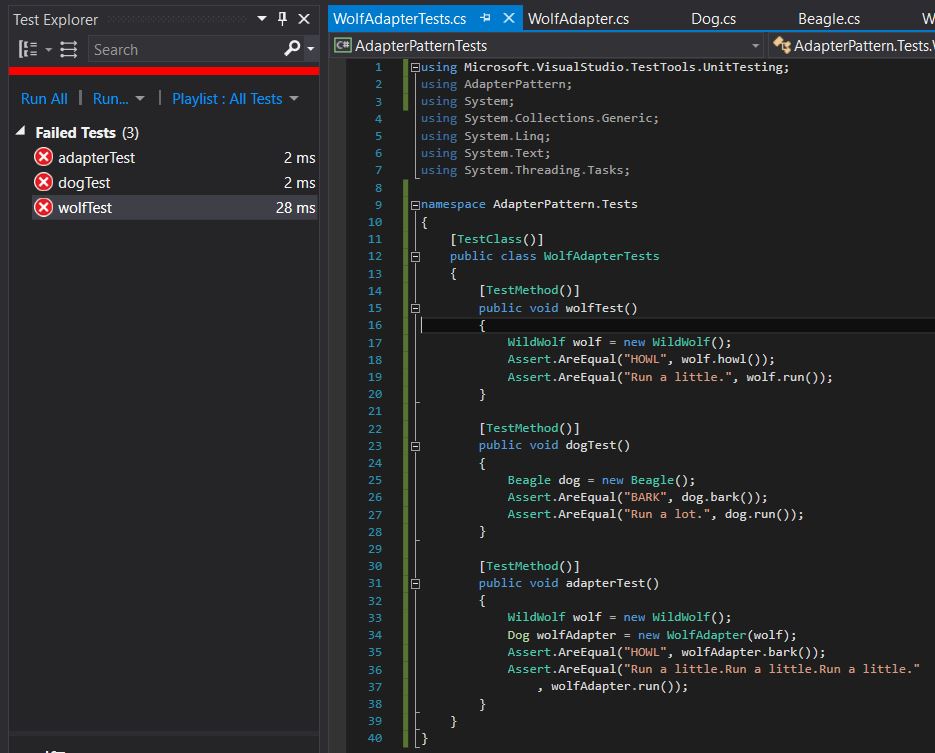


Figure . Initial Failed Test

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

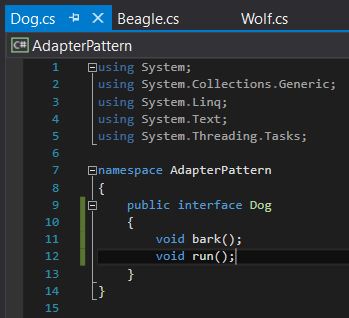


Figure . Initial Dog Class

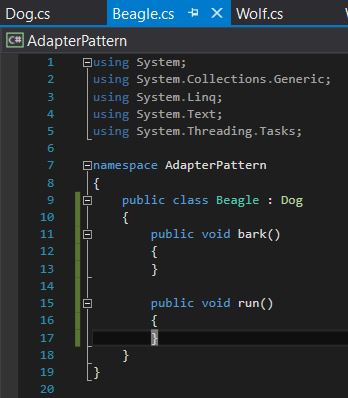


Figure . Initial Beagle Class

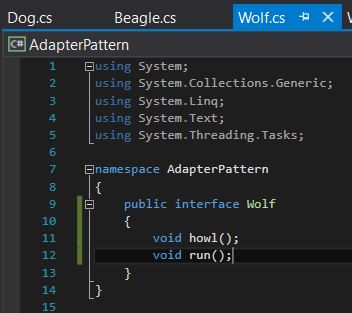


Figure . Initial Wolf Class

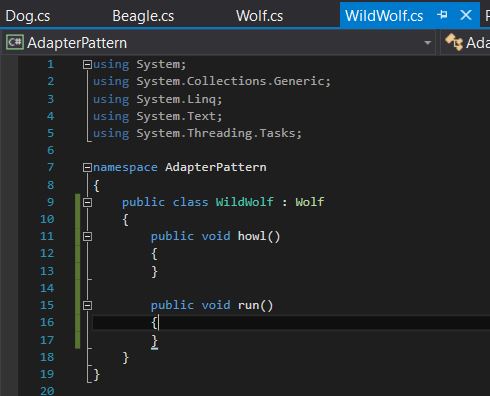


Figure . Initial WildWolf Class

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.

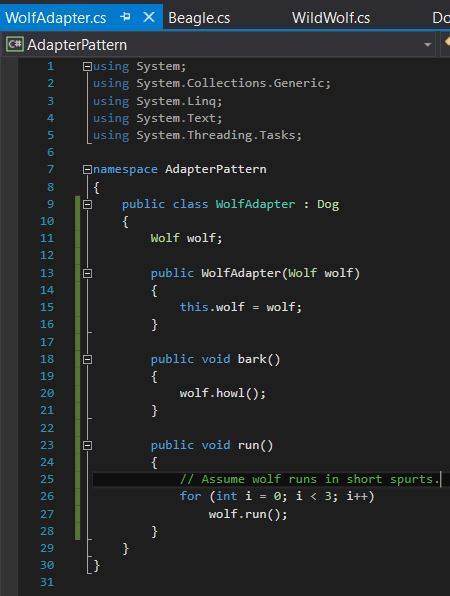


Figure . WolfAdapter Class

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.

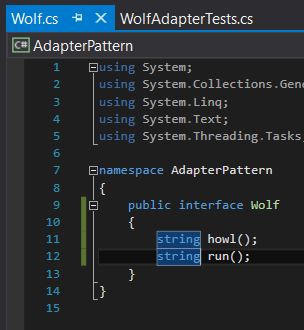


Figure . Refactored Wolf Class

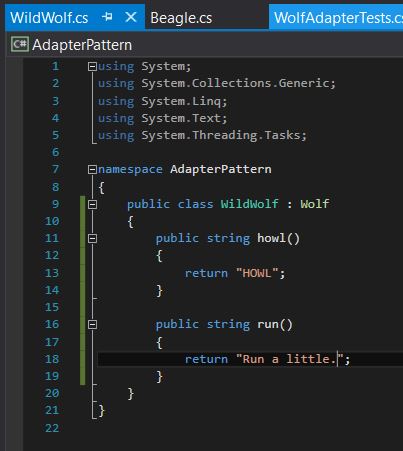


Figure . Refactored WildWolf Class

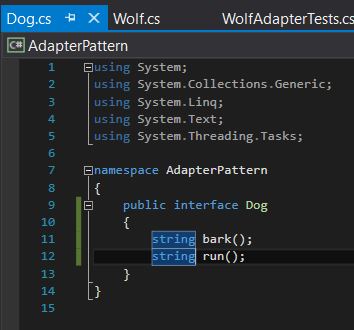


Figure . Refactored Dog Class

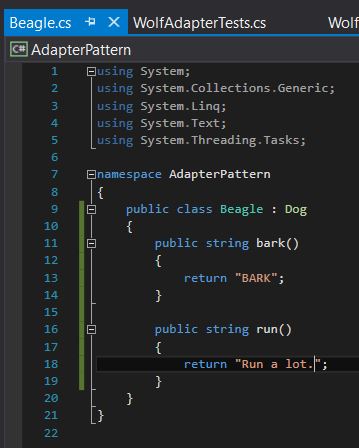


Figure . Refactored Beagle Class

With all of 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.

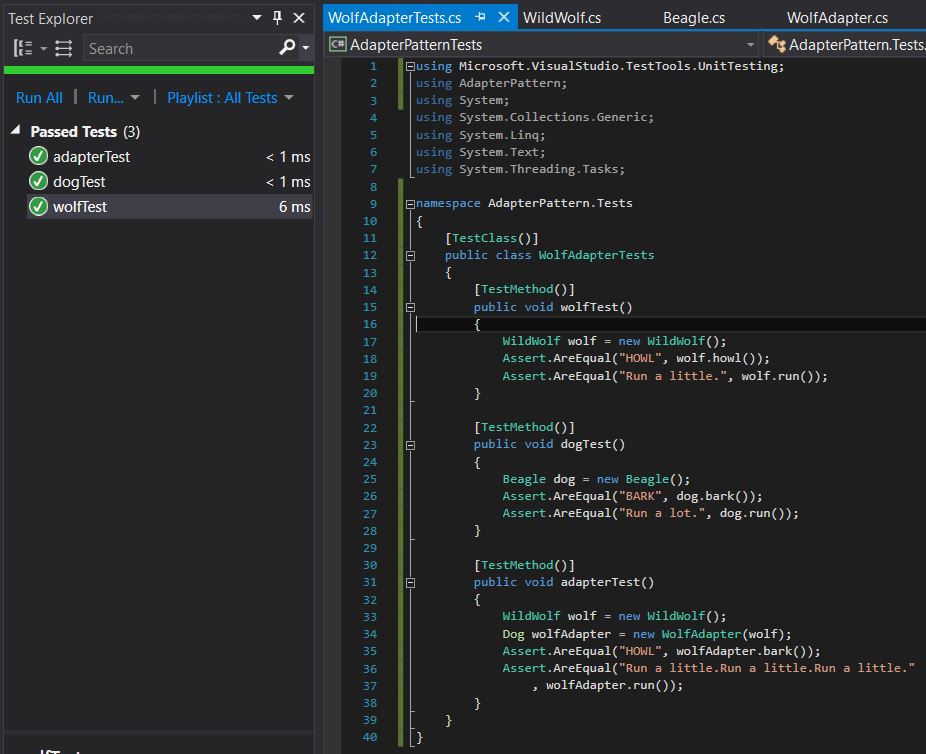


Figure . Successful Test

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.

# **References**

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/