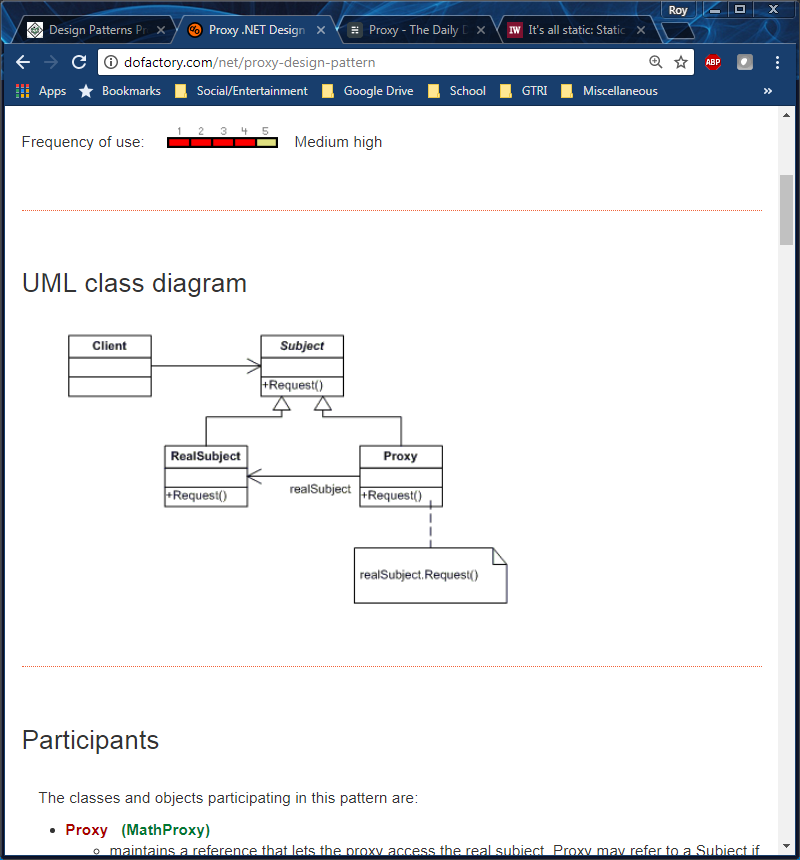
Proxy Pattern

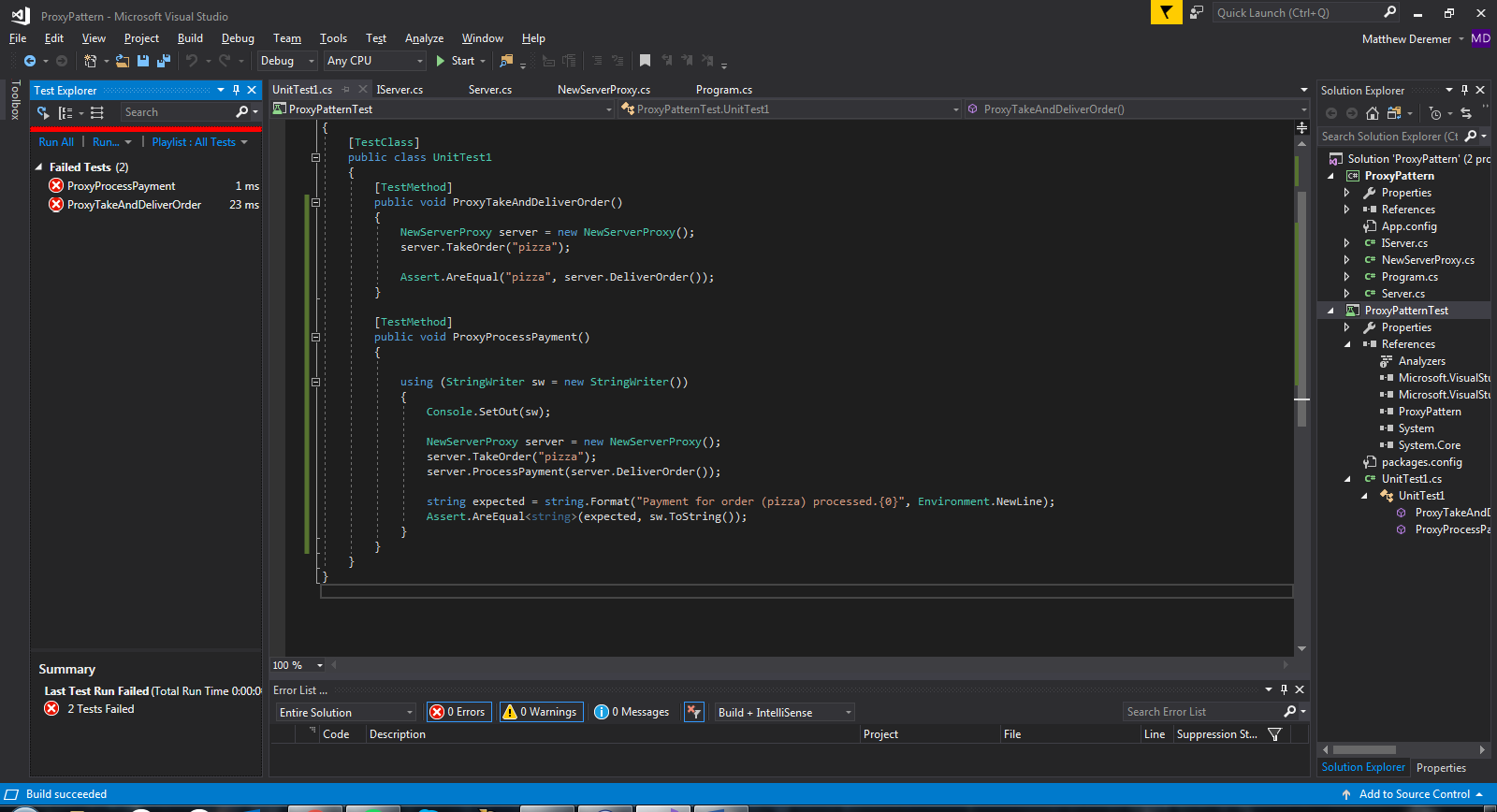
We’ve all seen examples of the proxy pattern even if we didn’t know it at the time. For example, take a receptionist. When you contact an office in search of an employee, the receptionist answers the call. The receptionist can perform many of the same actions as the employee, such as answering the call, 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 X for a general structure of the Proxy Pattern.



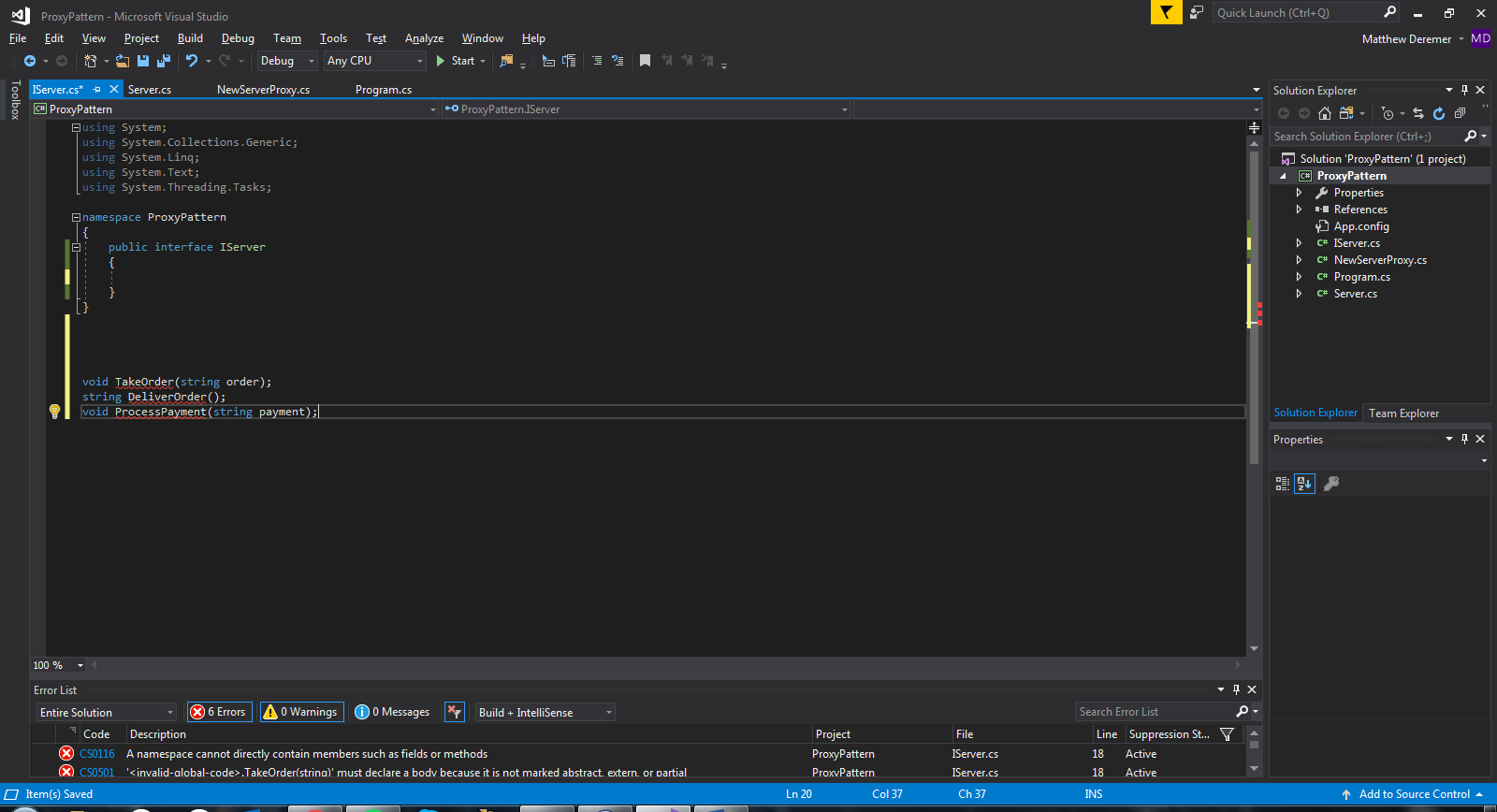
Proxy Class Diagram

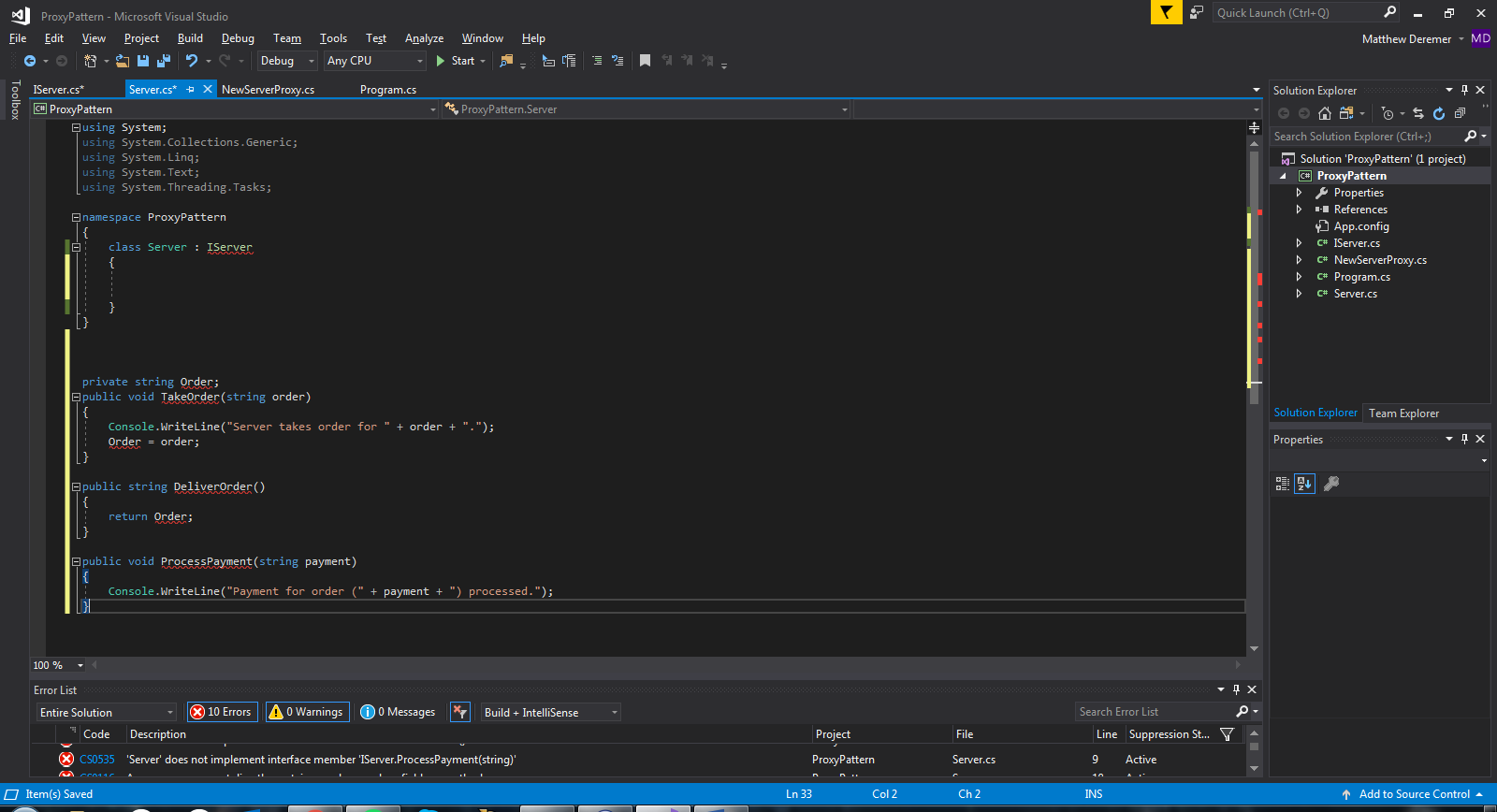
For the given example, the receptionist acts as the Proxy while the employee acts as the RealSubject. The Proxy Pattern is useful because it controls who or what has access to an object. Additional functionality is then added when access to the object is granted. This pattern is practical in the real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

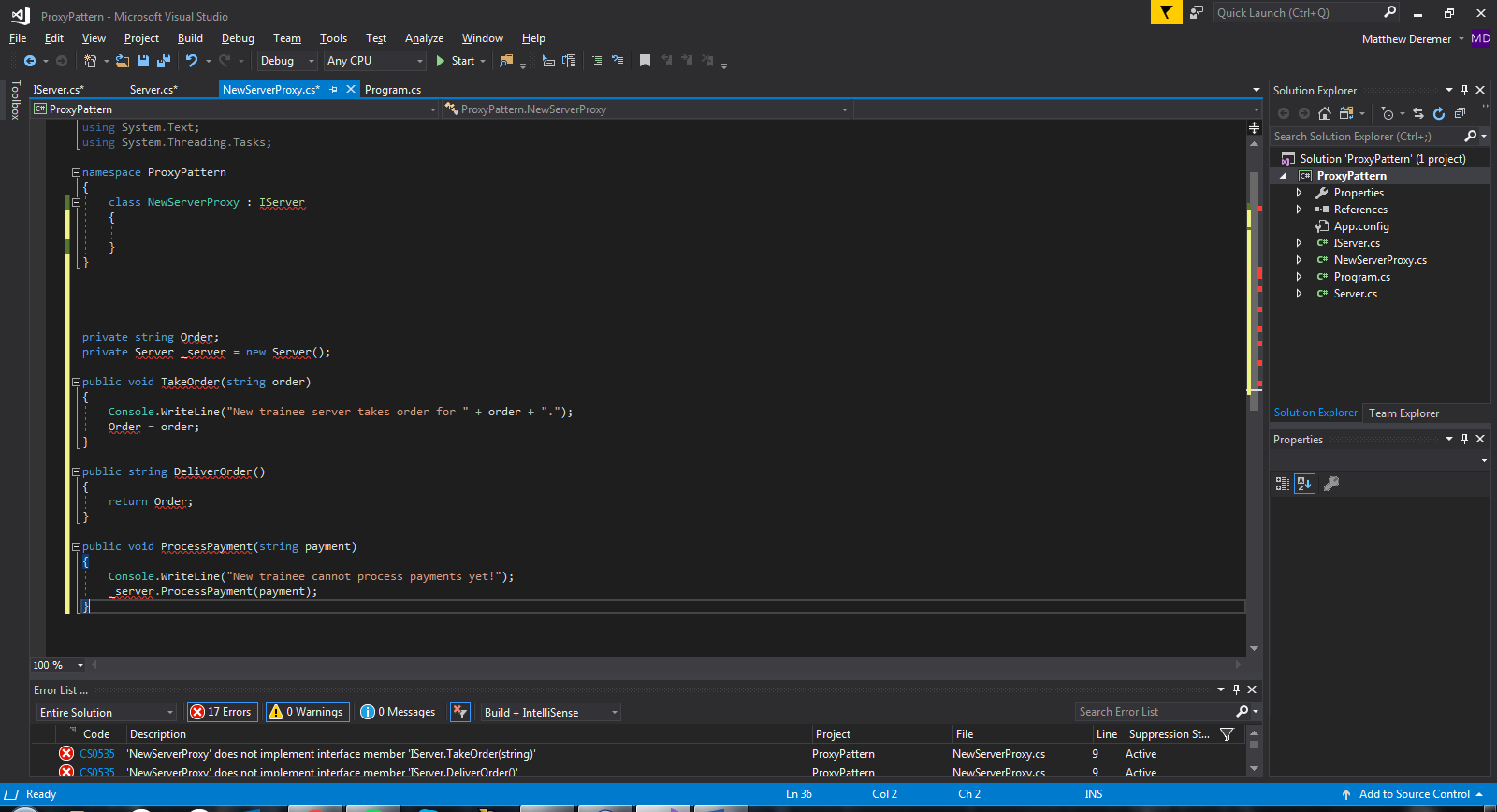
To demonstrate the Proxy Pattern we have designed a waiter application where an experienced waiter will serve as the RealSubject and a training waiter will be the Proxy. We also have an interface to connect the RealSubject and Proxy which will be the Subject. Before setting up the classes, we wrote a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



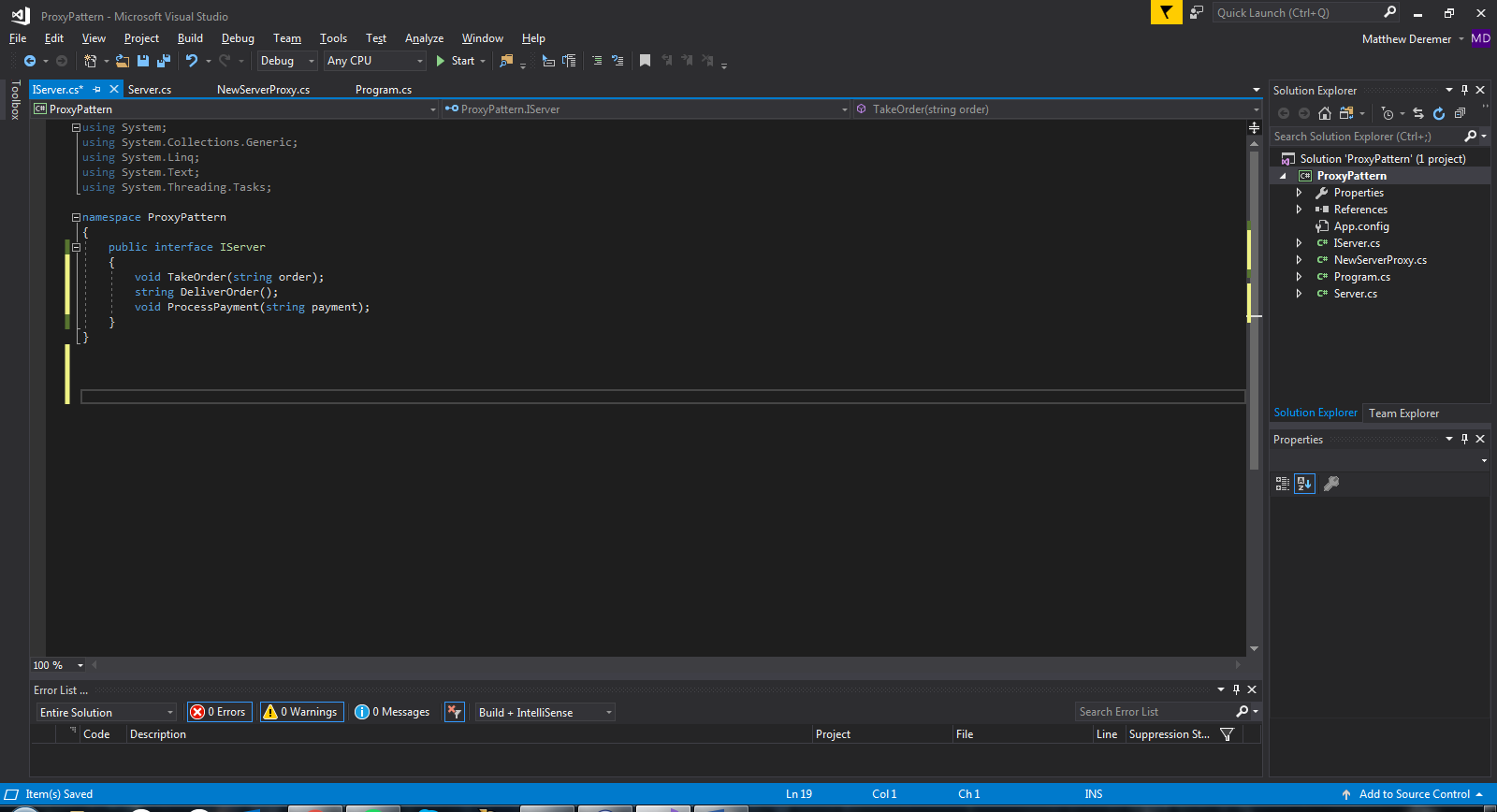
After writing the test suit, we outlined the Subject, Proxy, and RealSubject classes which can be seen in Figures X through X, below.



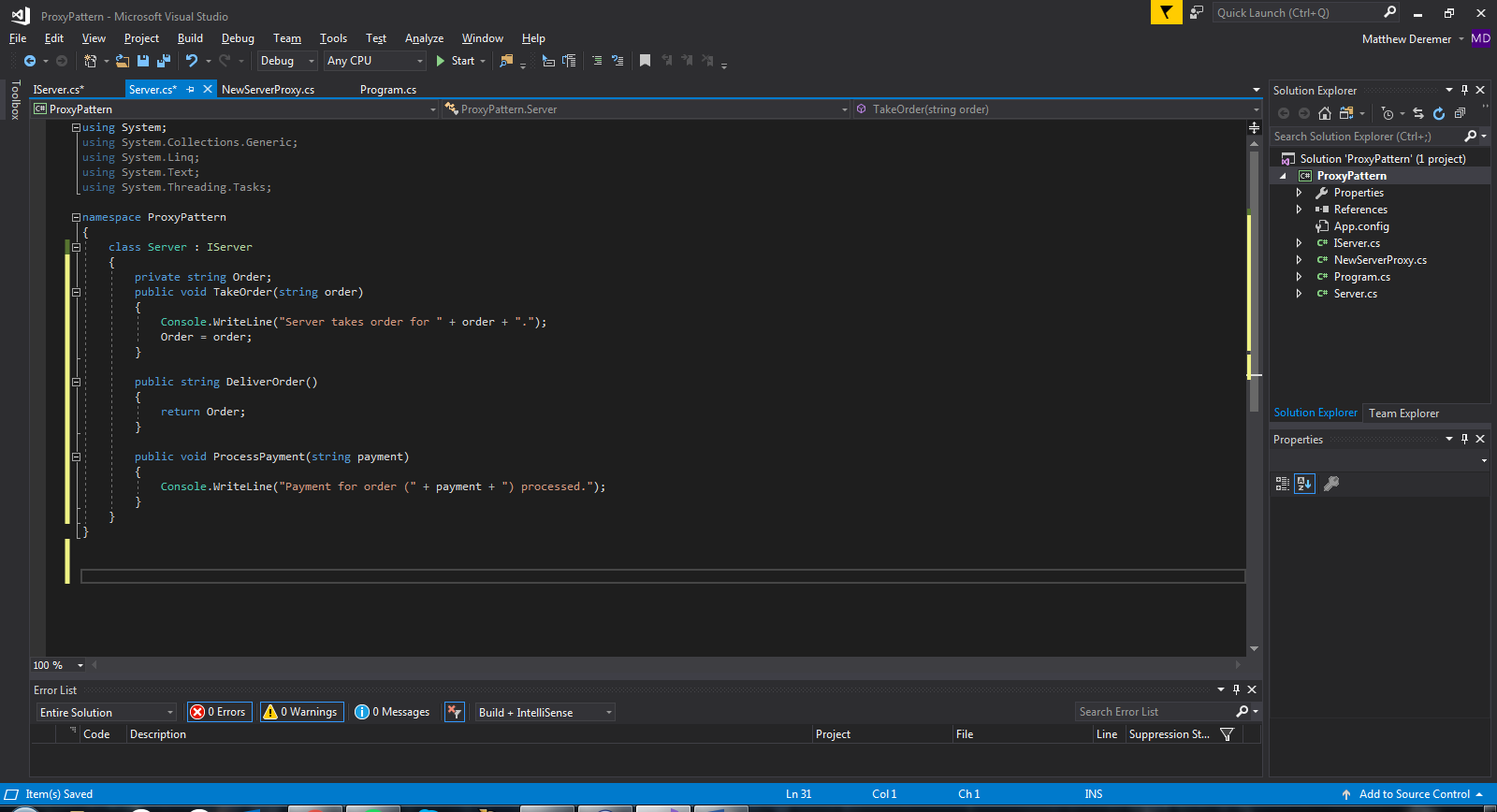




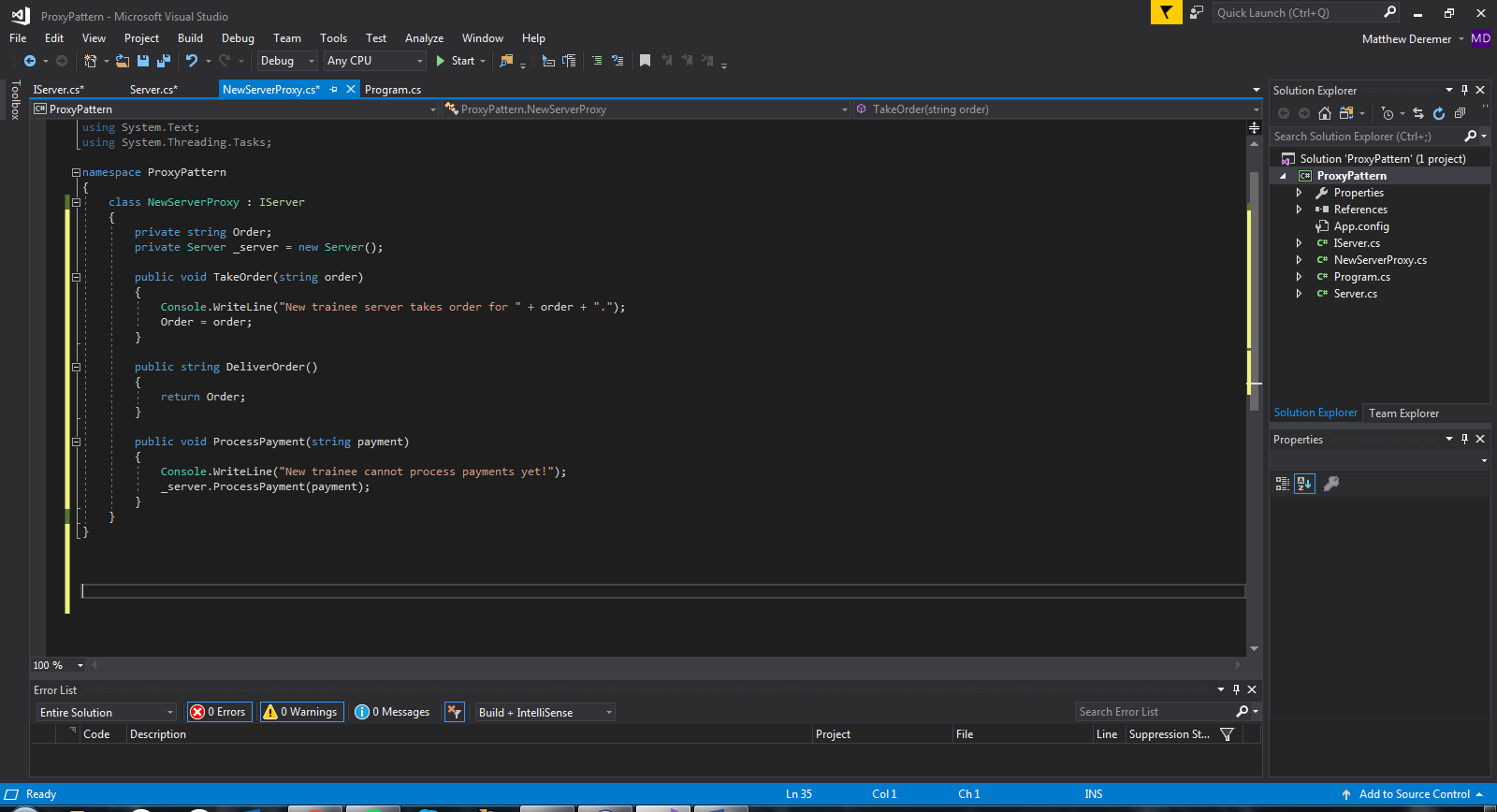
After outlining each class, we needed to implement the interface that would be used to connect the RealSubject and Proxy together. As seen in Figure X, below, the interface has three methods. TakeOrder(), DeliverOrder(), and ProcessPayment(). The Proxy is able to take and deliver an order but cannot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



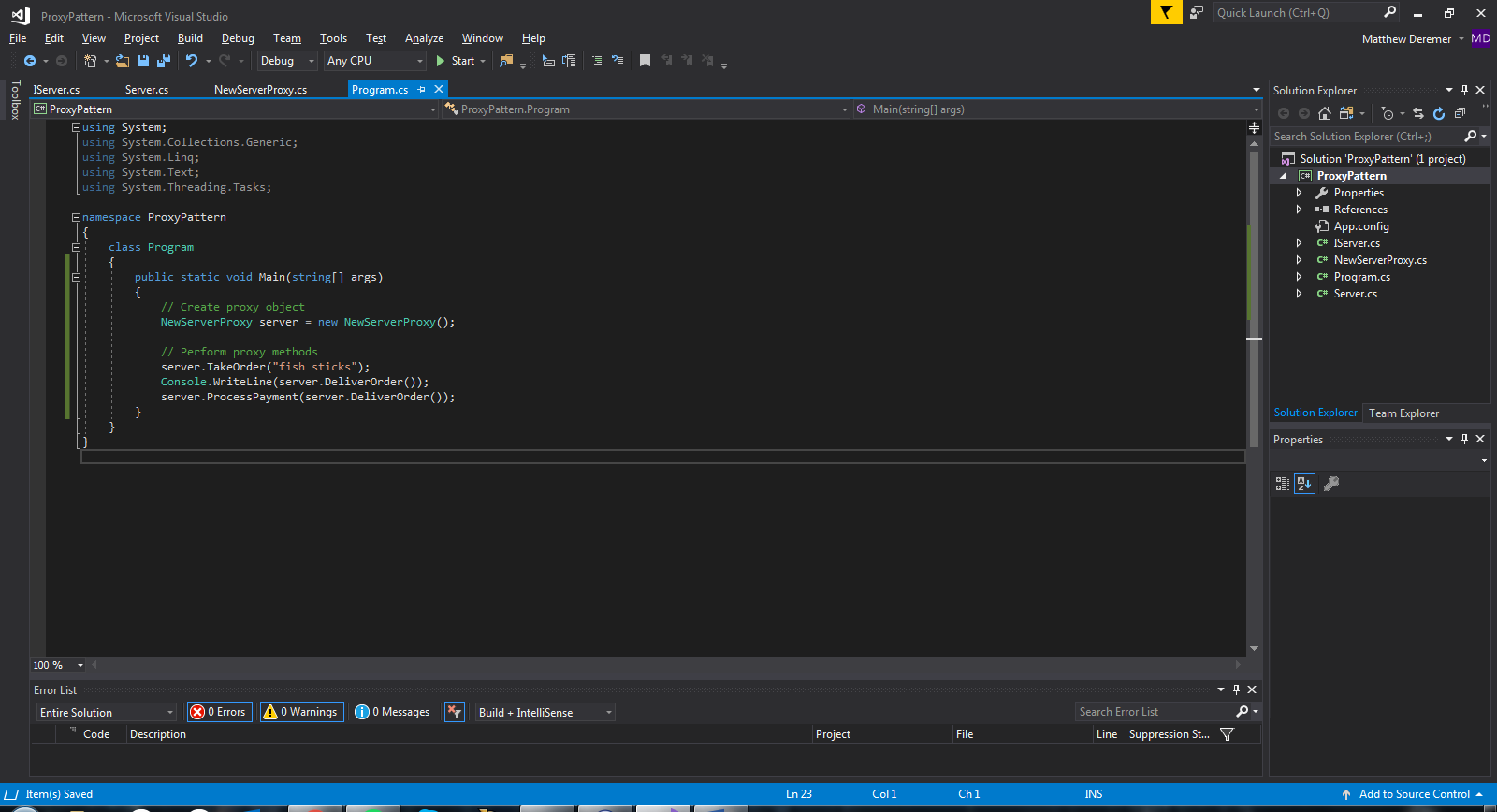
Once the interface or Subject had been implemented, we wrote the code for the server or RealSubject. As shown in Figure X, below, the server implements all the methods of the interface.



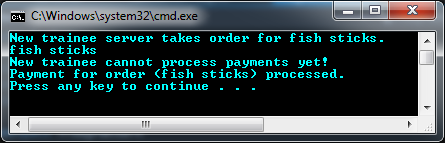
The final class for the Proxy Pattern is the new server or Proxy. As shown in Figure X, below, the Proxy implements its own TakeOrder() and DeliverOrder() but when ProcessPayment() is called, it must create a server object.



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 X for the main method.



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



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 X, all tests have passed.

