Open up the CommandClient solution in Visual Studio. This is one half of a sample C# chat application (just the client). You can see the complete project (though your version is very slightly modified) plus a detailed description of what’s going on in it here: <http://www.codeproject.com/Articles/12893/TCP-IP-Chat-Application-Using-C> - but understanding the underlying networking will not be too important for your task in this class. Instead, we’ll be using this code as example of a common problem in testing – our code is built to work with some other external thing (in this case the network) but we want to write unit tests for it in an isolated way. Also, as you’ll see, this code was not designed with unit testing in mind so it’s going to be reasonably interesting.

The function we want to test is SendCommandToServerUnthreaded. It’s in CommandClient.cs of the CommandClient project. It’s the code that takes a particular command object and sends it to the server, and it assumes that the server is running and that a connection has already been made. Of course, in our test, none of that will be true (indeed, we don’t even have the code for the server in this project). Take a look at the function itself and understand how it’s supposed to work.

*You won’t need to modify the function (or indeed any of the code in CommandClient.cs) for your tests to work.*

# Problem 1: Testing the UserExit command

You can see that the function takes a command object and outputs a bunch of bytes on a network stream. I’ve already started to write a test for it – take a look at the AdvancedMockTests in the CommandClientVisualStudio project. The test itself is named TestUserExitCommand.

This code uses VisualStudio’s built it testing framework rather than NUnit. They are similar (at least as far as you’ll need to use them) but by using VS framework it’s a little easier to get the debugger working for you if you have a problem.

Look at the test and the function itself and understand why the test is getting a null pointer exception. It has to with the network stream.

You can also see that I’ve written a fair amount of mocking code for a fake stream object that we can use to verify the encoding for UserExit is correct. There’s only one problem: my mock is designed to replace the networkstream, but that variable is private and the way it’s set within the object is not something we want to screw around with.

**Use reflection to set the private variable networkstream.** Once you do that the test should pass.

# Problem 2: Testing the UserExit command without Mocks

So this particular application is not ideal for mocks. Why? It’s a bit brittle. Note the large number of different calls to Write and Flush. As long as the correct data is written to the stream, it’s not really important that they be separated. For example, the whole chunk of data could be written with a single call to Write and Flush, and it would still work correctly. BUT, our mock based unit test would fail.

Instead of using a Mock in this case, it would probably be better use to something like a MemoryStream. That’s an object that looks like stream, yet keeps all its data in memory. If we could replace the networkstream with a Memory stream (hint: you can find documentation for MemoryStream here http://msdn.microsoft.com/en-us/library/system.io.memorystream.aspx), we could have it write the data using whatever combination of Write and Flush it wished, then pull the final result out and make sure it was correct with some basic asserts. That would be a less brittle test.

**Fill out the TestUserExitCommandWithoutMocks test to solve the problem without mocks and using MemoryStream. You’ll still have to use reflection to set the private variable though.**

# Problem 3: The Semaphore Under Normal Operation

So you’ll notice that the SendCommandToServer uses a semaphore. If you don’t know what that is, don’t worry, you’ll talk about it in your OS class. One key thing to know here though – if you acquire a semaphore (that’s what the WaitOne() function does) it is absolutely essential that you eventually release it (that’s what the Release() function does) or Bad Things happen.

So, we want to write a unit test that ensures that the SendCommandToServerUnthreaded command both acquires and releases the semaphores To do that, we’ll have to mock both the semaphore and the network stream. When we mock the semaphore, we’ll have to deal with a situation that’s a little different than what we’ve done in the past though...WaitOne and Release both return values. Here’s how you do that in the Rhino framework:

//within your mocks.Ordered block

Expect.Call(fakeSemaphore.WaitOne()).Return(true);

**That’s all you need to know. Write the TestSemaphoreReleaseUnderNormalOperation test that ensures the function both acquires and releases the semaphore.**

# Problem 4: The Semaphore Under Error

So it’s great that we do what we’re supposed in when the function is working normally, but usually where you have problems is when the function errors. For example, the call to networkstream Flush() might return an error. So we want to simulate that condition by making our mock stream throw an exception under that circumstance. Here’s how you do that in the Rhino framework:

//in your mocks.Ordered block

fakeStream.Flush();

LastCall.On(fakeStream).Throw(exception);

**That’s all you need to know. Write the TestSemaphoreReleaseUnderExceptionalOperation test that ensures the function both acquires and releases the semaphore.**