Adapting Task Asynchronous Programming Methods to Existing Interface Contracts

The Task Asynchronous Programming model (TAP) provides a standard way to write async methods and make them testable. Tests can await async methods that return Task or Task<T>. That ensures that assertions are not tested before the actions have completed. But some existing idioms we need to support do not allow us to follow the TAP model. These APIs mandate a different signature.

Many of us have adopted the Model-View-View Model (MVVM) pattern for the testing benefits it provides. There is less logic in the code behind for the view classes. The ViewModel logic can be validated with automated tests. One important practice in the MVVM pattern is creating classes that implement the ICommand interface and databinding objects of those types to the command buttons in the views. This enabled us to test our command actions by testing our ViewModel code. When we always wrote synchronous code, it worked great. But now that many of our command handlers start asynchronous operations, it's harder to test because of an API mismatch. The ICommand.Execute() method is synchronous and has a void return. When a command handler starts an asynchronous action, we need some other convention to determine when our asynchronous work has completed, and we can begin asserting our expectations. The asynchronous operation may not have finished before Execute() returns.

In this article, I'll show techniques to write testable async code in this very common scenario where following the TAP model is difficult. You'll be able to extend these techniques to other APIs where you need async methods that follow older conventions and aren't TAP compliant. You be able to write testable async methods in any scenario.

Mismatches in APIs – Exploring Solutions and Why they Fail

The mismatches are caused by the void Execute() method in the ICommand interface. You've heard that Windows 8 mandates that any operation that may take more than 50ms should be done asynchronously. That means many developers are writing Windows 8 command handlers that have async Execute methods. Those methods are async void methods, because that is mandated by the ICommand interface. But that practice causes many problems. You've probably heard guidance that discourages async void methods. Async void methods cannot be awaited. The code calling an async void method cannot tell if the method has completed its work yet, or if its work completed successfully, or faulted because the method threw an exception. That lack of returned information, via a Task, is what makes async void methods hard to test, and what leads to guidance that discourages async void methods.

That mismatch makes writing MVVM command handlers in Windows 8 difficult. The command handlers often must be async, because the command may take more than 50ms to complete. To conform to the ICommand interface, command handlers must return void. Let's walk through a small sample and demonstrate the problems you'll encounter, and how to overcome them.

I wrote a Windows 8.1 application that has one command button that performs an asynchronous action. To simulate an action that might take time, this command handler uses Task.Delay to wait up to 5 seconds, then adds a message to a list. Here's the async void command handler:

```
public async void Execute(object parameter)
{
    var delay = generator.Next(1000, 5000);
    await Task.Delay(delay);
    owner.Messages.Add(string.Format("Waited {0} for something to happen", delay));
}
```

For purposes of the demonstration, I'm awaiting the Task.Delay method call before adding the messages. I want to simulate a long running operation that reports its results only after the long running operation has completed. One test for this method might be to ensure that it adds a message to the list of messages:

```
[TestMethod]
public void TheShinyRedButtonAddsMessages()
{
    // Arrange:
    var underTest = new CommandHandlerViewModel();
    var initialMessageNumber = underTest.Messages.Count;

    // Act:
    underTest.CommandHandler.Execute(null);

    // Assert:
    Assert.AreEqual(initialMessageNumber + 1, underTest.Messages.Count);
}
```

If you ran this test, you'd be surprised to find that this test fails even though the code appears to be implemented correctly. That's because the command implementation has not finished its work before the Asserts are tested. The Asserts fail because the method has not finished its work and added the message yet.

As it is written, this code can't be easily unit tested. You can't know when the async operation is complete, and therefore you can't know when you should execute the asserts. The Execute method returns void, so you can't query a task to determine if the work has completed. You can't simply change the signature to return a Task, because that's not the ICommand interface definition.

One strategy I've seen developers use to create tests for asynchronous command handlers is to introduce a Task property so that tests can determine if the command has completed its work. The example below demonstrates this technique, and shows why it can be brittle and many developers have problems because of the subtle interactions between tasks and their continuations.

```
public async void Execute(object parameter)
```

```
{
    var delay = generator.Next(1000, 5000);
    owner.ShinyRedButtonTask = Task.Delay(delay); //Returns Task here since no
'await'
    await owner.ShinyRedButtonTask; // Then 'await' task here
    owner.Messages.Add(string.Format("Waited {0} for something to happen", delay));
}
```

Then, you can write an asynchronous test as follows:

```
[TestMethod]
public async Task TheShinyRedButtonAddsMessages()
{
    // Arrange:
    var underTest = new CommandHandlerViewModel();
    var initialMessageNumber = underTest.Messages.Count;

    // Act:
    underTest.CommandHandler.Execute(null);
    await underTest.ShinyRedButtonTask;

    // Assert:
    Assert.AreEqual(initialMessageNumber + 1, underTest.Messages.Count);
}
```

Note that my test is an asynchronous method that returns a task. Modern test engines have been enhanced to await the completion of the work for any Task-returning tests. This enables us to write async tests for async methods. I'll return to this concept later to demonstrate what happens if you accidentally create void returning async tests. If, instead, this method were void returning, the test engine would not know that the test method may still have work to do.

While the faux Task strategy works, it leaves a lot to be desired. The Task property is used only for testing. Someone reviewing the code would wonder why that property is part of the view model. It doesn't seem to be used. It's also very brittle: Both the test and the code under test are awaiting the same Task. This introduces a race condition where the code under test may still execute after the test code finishes. By awaiting the task in two different methods, you have two different continuations to execute after the task finishes. They may execute in either order. You may get false failures because of the order of execution of the continuations. If the continuation in the test executes first, the test will fail. If the continuation in the view model executes first, the test will pass.

A small change to fix this is to refactor the execute code so that the Task member variable encompasses all the work:

```
public async void Execute(object parameter)
{
    owner.ShinyRedButtonTask = ExecuteWork();
    await owner.ShinyRedButtonTask;
}

private async Task ExecuteWork()
```

```
{
    var delay = generator.Next(1000, 5000);
    await Task.Delay(delay);
    owner.Messages.Add(string.Format("Waited {0} for something to happen", delay));
}
```

You may look at this version, and decide that you really don't need to await that task in the last line of the Execute() method. After all, Execute is void returning. So you may think about deleting that line. That would change the runtime behavior in ways that can be harmful. Async void methods behave very differently from synchronous void methods when an exception gets thrown in the asynchronous work.

In the original version, when an exception gets thrown, the windows runtime terminates the application. That's because an async void method generated an exception. This is a difficult part of the language, and there was quite a bit of debate about how to handle async void methods when the async work generated an exception. Task returning async methods have a clear protocol for exceptions: when an exception is thrown, the task enters the faulted state. When code awaits a faulted Task, the exception is thrown at the site of the await. Exceptions are observed, and appropriate action can be taken. There's no corresponding implementation that would work for async void methods. They don't return a Task, so the task can't enter the faulted state. The method may have already returned before the exception is thrown, so they can't throw the exception in the calling context.

Another major issue is that exceptions indicate serious error conditions, and it's better to terminate the program than to support constructs where exceptions cannot be observed. What actually happens is that compiler-generated code injects an exception into the synchronization context that created the task. In a Windows 8 application, that means injecting an exception into the main message pump, which terminates the message processing thread. Yes, this is somewhat harsh. But the alternative was to create code that silently ignored all exceptions from async void methods. That alternative was worse. You can configure a global unhandled exception handler to do logging or possibly decide if you can keep the application alive.

When you change the command handler from an async void method that awaits another asynchronous method to a synchronous method that starts asynchronous work and does not await the task, you've created code that silently ignores any exceptions. Modify the ExecuteWork method so that it sometimes fails:

```
private async Task ExecuteWork()
{
    var delay = generator.Next(1000, 5000);
    await Task.Delay(delay);
    if (delay > 3500)
        throw new InvalidOperationException("This is a sample exception");
    owner.Messages.Add(string.Format("Waited {0} for something to happen", delay));
}
```

Try this version of the sample yourself and see. There will be times when you press the button, and it appears that nothing happens. In those instances, the delay was long enough to cause the exception, but nothing was reported. The command simply didn't complete. In a production application, it may have left the application in an unsafe state. You can verify this by running the application under the debugger and configuring the debugger to break on all exceptions. You'll see the exception gets generated, but there is no code that observes it.

You could continue down the current path and sprinkle await expressions using that Task member variable introduced earlier. But in a real world application, that would quickly become unwieldy. You may have many commands. That would mean many tasks that should be awaited at each user interaction. You may have many different UI actions. That would mean many locations to add multiple await expressions. It would be difficult code to understand and maintain.

A better reusable solution

A better way to deal with the mismatch is to separate the object that implements the ICommand interface from the object that performs the asynchronous work. The object implementing ICommand provides the bridge between the contract expected from ICommand and the standard TAP model. The actual command methods are TAP-compliant methods that can be unit tested using standard techniques. The ICommand handler can be tested independently, and can be tested with less concern about the TAP protocol.

I'll go through a few refactoring steps to get from the existing code to the final version. I've already extracted a method from the ICommand Handler's Execute method. The extracted portion represents the operations that are specific to this command. I named the extracted method ShinyRedButtonHandler(), because it handles the command for the shiny red button.

The Extract Method refactor command in Visual Studio understand the TAP model. Because the code I'm extracting contains await expressions, the extracted method returns a Task. The automated tools help me make TAP compliant code. ~~Notice [not sure where to notice? Maybe show code here or provide name of new method? I'm struggling reading this and the next few paragraphs scrolling up and down over 2-3 pages to look at code] that the automatic refactoring tools did extract an async Task returning method. The extracted method is TAP compliant.~~

Now, let's go back into the command handler and make this command handler safe even though it must be an async void method. I need to ensure that this async void method does not ever throw exceptions. Since the extracted method is a TAP-compliant method, this is not too hard. In the Execute command handler, the code awaits the call to the extracted ShinyRedButtonHandler() method, and that await must be in a try / catch block. I am catching all exceptions at this time. In the catch clause, I'm adding the exception message to the list of messages. In a production application, you'd want to follow your logging process, and determine if the error is one from which you can recover, or if you should notify the user and exit.

```
public async void Execute(object parameter)
{
    try
    {
        owner.ShinyRedButtonTask = ExecuteWork();
        await owner.ShinyRedButtonTask;
    } catch (Exception)
    {
        // A production app should probably use a different logging scheme:
        owner.Messages.Add("Async work failed");
    }
}
```

That's the basis for the solution. The Execute command handler, the void returning method, is now responsible for conforming to the ICommand interface, and the extracted method, the ShinyRedButtonHandler() is responsible for executing the specific work of this command. The extracted method, the ShinyRedButtonButtonHandler() is TAP compliant.

I can now write tests using the extracted method to ensure that it's doing its work. I can change the ShinyRedButtonHandler()method from private to public. I'll write tests against the extracted method. Now, I don't need the Task variable in my view model for testing purposes:

```
[TestMethod]
public async Task TheShinyRedButtonAddsMessages()
{
    // Arrange:
    var underTest = new CommandHandlerViewModel();
    var initialMessageNumber = underTest.Messages.Count;

    // Act:
    var commandHandler = underTest.CommandHandler
        as CommandHandlerViewModel.AsyncCommandHandler;
    await commandHandler.ExecuteWork();

    // Assert:
    Assert.AreEqual(initialMessageNumber + 1, underTest.Messages.Count);
}
```

As I mentioned earlier, notice that the test method is marked with the async modifier, and it returns a task. That way, the test runner ensures that the returned task runs to completion before declaring that the test passes. If you create fire and forget tests (async void), by not awaiting the tasks and not declaring your async tests as async Task returning methods, the test methods may return before the task completes. Your test indicates that it passes, even if I remove the code that adds the message to the list. The reason the test passes is because the test runner assumes the test has run to completion as soon as it returns. None of the Asserts have been tested!

Due to the possible false positives, I strongly recommend that you make sure you run your async tests when they expect to fail, using TDD, or a similar strategy, as part of your development process. If you only see tests for code that already works, you can easily miss mistakes like this.

I still have tighter coupling than I'd like to have between the command handler and the view model. You can see that in my test where I cast the command handler to its specific class type. That should be hidden from the ViewModel's client classes. After extracting the specific code for this command, let's perform some more refactoring and separate the general code for implementing an async ICommand handler from the specific actions of that command. The only code specific to a given command is the Name method that I originally extracted from the ICommand handler. The only requirement for that method is that it must be a TAP-compliant method (that is, it returns a Task). That means any method or lambda expression that represents a Func<Task> will work. Let's refactor the code a bit more and construct the Command Handler with a Func<Task> to represent the specific work for a given command. As part of that work, I'll move the extracted method from the nested command handler to the view model class.

At the same time, I'll add a second lambda expression, this time an Action<Exception> to represent the work that should be done if the command handler throws an exception.

After that change, the command handler no longer needs a reference to the view model. The final code is shown in the following listing:

```
public class AsyncCommandHandler : ICommand
{
    private readonly Func<Task> commandAction;
    private readonly Action<Exception> onError;

    public event EventHandler CanExecuteChanged;

    public AsyncCommandHandler(Func<Task> commandAction, Action<Exception> onError)
    {
        if (CanExecuteChanged != null)
            CanExecuteChanged(this, EventArgs.Empty);
        this.onError = onError;
        this.commandAction = commandAction;
    }
    public bool CanExecute(object parameter)
    {
        return true;
    }

    public async void Execute(object parameter)
    {
        try
        {
            await commandAction();
        }
        catch (Exception e)
        {
            onError(e);
        }
    }
}
```

```
public class CommandHandlerViewModel : INotifyPropertyChanged
{
    public ObservableCollection<string> Messages { get; private set; }

    public ICommand CommandHandler { get; private set; }

    private readonly Random generator = new Random();
    public CommandHandlerViewModel()
    {
        Messages = new ObservableCollection<string>();
        CommandHandler = new AsyncCommandHandler(
            () => ShinyButtonCommandHandler(),
            (e) => Messages.Add(e.ToString()));
    }

    public async Task ShinyButtonCommandHandler()
    {
        var delay = generator.Next(1000, 5000);
        await Task.Delay(delay);
        if (delay > 3500)
            throw new InvalidOperationException("This is a sample exception");
        Messages.Add(string.Format("Waited {0} for something to happen", delay));
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

```
[TestMethod]
public async Task TheShinyRedButtonAddsMessages()
{
    // Arrange:
    var underTest = new CommandHandlerViewModel();
    var initialMessageNumber = underTest.Messages.Count;

    // Act:
    await underTest.ShinyButtonCommandHandler();

    // Assert:
    Assert.AreEqual(initialMessageNumber + 1, underTest.Messages.Count);
}
```

Examine how this separation facilitates testing. The view model class now has a Task-returning, TAP-compliant method, ShinyButtonCommandHandler(), that implements the command. In a production application, you could write as many tests as you need to verify that it did what was expected based on other view model properties when the command was executed. Make those tests Task returning async tests, and they will provide great test coverage for your command implementations.

The command handler itself is a bit trickier to test, because it is still an async void method. It's a solvable problem because it has only a single responsibility: to call the Func<Task> method when Execute() gets called. We can rely on some rules in the C# language to write those tests in such a way that the tests always execute synchronously.

First, here's a method that verifies that the configured Func<Task> gets called when Execute gets called. The Func<Task> that I've used for this test executes synchronously, and always returns a completed

Task. That way, when Execute calls the Func<Task> delegate, the Execute method continues its execution synchronously. Even though my test uses a Task returning method, the execution is synchronous; all the logic is properly exercised.

```
[TestMethod]
public void CommandHandlerCallsConfiguredFunction()
{
    // Arrange:
    bool calledMethod = false;
    var underTest = new AsyncCommandHandler(() =>
    {
        calledMethod = true;
        return Task.FromResult(true);
    },
    (e) => { });

    // Act:
    underTest.Execute(null);

    // Assert:
    Assert.IsTrue(calledMethod);
}
```

This next test verifies the failure path. The configured delegate always returns a faulted task. When the Execute method awaits that task, the exception contained in the faulted task gets thrown in the context of the Excecute method. The Execute method then enters~~processes?~~ the catch clause, and the test ensures that the error handler is called. Again, this test executes synchronously with other tests because the task returned by the delegate has already finished. It finished by faulting, but it is finished.

```
[TestMethod]
public void CommandHandlerCallsConfiguredErrorFunction()
{
    // Arrange:
    bool calledMethod = false;
    var tcs = new TaskCompletionSource<bool>();

    var underTest = new AsyncCommandHandler(() =>
    {
        return tcs.Task;
    },
    (e) => { calledMethod = true; });

    // Act:
    tcs.TrySetException(new InvalidOperationException("Testing a failure"));
    underTest.Execute(null);

    // Assert:
    Assert.IsTrue(calledMethod);
}
```

In this article, I demonstrated how you can build reusable and testable async code even when a required contract mandates a void returning method, or some other non TAP-compliant structure. I used the ICommand interface to demonstrate because that’s a very common use case. The same concepts would apply to any other interface you must work with. There are a lot of good reasons to follow the TAP model for async methods. When you must deviate from those recommendations, make small adapter classes that isolate the mismatch between an existing contract and other methods you write, which should follow the TAP model. You’ll create more testable code, with better maintainability, and avoid many of the common pitfalls associated with async programming.