Moving to scenario-based unit testing in .NET

Wednesday, 10 June 2009
http://www.davesquared.net/2009/06/moving-to-scenario-based-unit-testing.html
David Tchepak (dave@davesquared.net, http://davesquared.net)

Table of Contents

Moving to scenario-based unit testing in .NET

From my observations most developers (myself included) start writing tests using one testcase class per class under test. Due to an unfortunate attribute naming choice, NUnit users may know this as "fixture per class", and implement it as one class marked `[TestFixture]` holding all the tests that relate to one production class.

I think the reason this approach is so common is because introductory testing and TDD examples tend to focus on smallish utilities like stacks or basic calculators, where fixture setups and object interactions tend not to get out of hand. In my experience, taking the knowledge learned during from basic examples and trying to apply them to Real Life™ ends in pain, frustration, nausea, alcoholism and, in rare cases, spontaneous human combustion. Even worse, it may cause developers to abandon writing unit tests altogether!

There are many alternatives for organising tests (testcase class per fixture, feature, method, etc.), but the one I've found easiest to adopt and use is a testcase class per *scenario*. We've been using this at work across two teams for a few months now, and while not perfect it does seem to make writing, reading and maintaining tests relatively easy. In fact, I'm 99% sure that learning to write tests in this way initially (rather than the usual testcase class per class way) would avoid most of the problems we face when first trying to adopt unit testing and test driven development.

The aim of this post is to describe what on earth I'm talking about, as well as how to move from testcase class per class to scenario-based testcase classes, and also outline some of the unresolved problems we've run into with this approach.

A note on terminology

I'm probably drastically misusing terminology here when I talk about scenario-based testing. The way I'm using the term "scenario" seems quite different to scenario-based testing as described in Wikipedia, but I don't know a better name for it. The word "scenario" is used a lot when talking about Behaviour Driven Development (BDD) as well, but I wouldn't be so presumptuous to think this approach qualifies as BDD. It doesn't really seem to match the Dan North or Dave Astels definitions. Instead, I'd appreciate it if we ignore the terminology and just focus the basic ideas -- you can substitute whatever terminology you like! :)

The example

Let's look at the traditional way of introducing testing concepts -- a contrived example. Say we've been contracted by KAOS, an private company with a strong focus on research and development. Their latest project is something they casually refer to as The Doomsday Device. KAOS, being an agile sort of organisation, have given us some simple stories:

- The screen should display a welcome message to the agent using the Doomsday Device.
- The user can click a button to fire the Doomsday Device
- After firing, the user cannot click the button again until the Doomsday Device has finished its firing sequence

Seems fairly easy. We'll use a Model View Presenter style approach so we can test this logic without actually firing the device. Siegfried, the product owner, has assured us that the consequence of any bugs could be quite dire. He stresses "Vee are KAOS agents -- vee don't write bugs here!".

Testcase class per class

We start off by creating a `DoomsDayDevicePresenterFixture`, and then begin writing tests and filling in passing implementations. Here is what our fixture looks like when we're done:

```
/* Fixture per class example */
```

```
[TestFixture]
public class DoomsDayDevicePresenterFixture {
    private IDoomsDayDeviceView stubView;
    private IDoomsDayDevice stubDoomsDayDevice;

    [SetUp]
    public void SetUp() {
        stubView = MockRepository.GenerateStub<IDoomsDayDeviceView>();
        stubDoomsDayDevice = MockRepository.GenerateStub<IDoomsDayDevice>();
        new DoomsDayDevicePresenter(this.stubView, this.stubDoomsDayDevice);
    }

    private void RaiseLoadedEventOnView() {
        stubView.Raise(view => view.Loaded += null, this, EventArgs.Empty);
    }
    private void RaiseFireButtonPressedOnView() {
        stubView.Raise(view => view.FireButtonPressed += null, this, EventArgs.Empty);
    }
    private void RaiseFiringCompletedOnDevice() {
        stubDoomsDayDevice.Raise(device => device.FiringCompleted += null, this, EventArgs.Empty);
    }

    [Test]
    public void ShouldWelcomeEvilAgentWhenLoaded() {
        const String agentsName = "Siegfried";
        stubDoomsDayDevice
            .Stub(device => device.CurrentOperatorsName())
            .Return(agentsName);
        RaiseLoadedEventOnView();
        Assert.That(stubView.WelcomeMessage, Is.EqualTo("Welcome KAOS Agent " + agentsName + ". What w
    }

    [Test]
    public void ShouldEnableFireButtonWhenViewLoaded() {
        RaiseLoadedEventOnView();
        Assert.That(stubView.FireButtonEnabled, Is.True);
    }

    [Test]
    public void FireButtonShouldFireDoomsDayDevice() {
        RaiseLoadedEventOnView();
        RaiseFireButtonPressedOnView();
        stubDoomsDayDevice.AssertWasCalled(device => device.Fire());
    }

    [Test]
    public void ShouldDisableFireButtonWhileFiring() {
        RaiseLoadedEventOnView();
        RaiseFireButtonPressedOnView();
        Assert.That(stubView.FireButtonEnabled, Is.False);
    }

    [Test]
    public void ShouldEnableFireButtonWhenFiringCompleted() {
        RaiseLoadedEventOnView();
        RaiseFireButtonPressedOnView();
        RaiseFiringCompletedOnDevice();
        Assert.That(stubView.FireButtonEnabled, Is.True);
    }
}
```

Our tests all pass. Hooray!

What's wrong with this?

What's wrong with this testcase class? Well, nothing really I guess. It's manageable enough for now. However when we start adding features we may find the lack of cohesion between the tests causes us pain (at least it always has for me. I'm quite happy to acknowledge I could just be doing it wrong).

The main source of the cohesion problem is the setup used for each test. The testcase class has its own setup method, but the first few lines of each test performs some additional setup. In somes cases, such as the `ShouldDisableFireButtonWhileFiring()` and `FireButtonShouldFireDoomsDayDevice()`, this setup code is repeated. These two tests are really asserting two facts about the same situation (firing the device), but you can't easily see that from the test code. Test code can be a great way of communicating how your code works, but this lack of cohesion in the setups used can make it really difficult to parse out this information.

Having setup information all over the place can make the test code hard to maintain. Changes like requiring a dependency to provide new data at a different time can cause large numbers of tests to break, especially when they've ended up with setups that don't truly reflect the contexts they use. Well factored tests can be a great enabler of change. Fragile tests have the opposite effect.

One thing you can't see from this example, and to me one of the most important drawbacks, is the impact of this approach on test driven development. Most people I know who have learned/tried to learn TDD (especially when learning without a mentor) have done some of the examples, decided to apply it to a real project, sat down to write the first test and... have absolutely no idea where to start. Should we first test the presenter has wired up the correct events on the view? Should we construct a presenter and see if its null (don't laugh, I've seen it recommended!)? Knowing what tests to write to drive development was probably my greatest hurdle in learning TDD -- it is natural to keep thinking in terms of how we want objects and methods to work, rather than what we want the design to do, which robs you of the design benefits TDD can provide.

Testcase class per fixture?

The setup required for a series of tests is generally called the *test fixture* -- basically, all the stuff that needs to be in place for the assertions in a test to apply and run successfully. To make managing these tests easier we could break them out into a series of testcase classes per fixture. But then we end up with another problem -- our fixtures overlap. Let's think this through for a moment and see if we can find out why this could pose a problem.

All the tests start with a call to `RaiseLoadedEventOnView()`, so we could just whack that in the setup and have a fixture for all our tests the rely on the loaded view. Except for the `ShouldWelcomeEvilAgentWhenLoaded()`, which needs to stub out a value before we raise the loaded event. We could stub the value in the setup method as well, but then we lose the benefit of knowing which functionality relies on having that value available. Following this path leads to an increase in the amount of setup code, most of which doesn't apply to all the tests. This can end up obscuring what we are actually testing.

If we were going to go by fixtures, what would we name them? We have one fixture that stubs an agent name and uses a loaded view, another that just needs a loaded view, two that share a fixture where we have a loaded view and a pressed button, and another that has a loaded view, pressed button, and a firing completed event. Based on fixture alone it's hard to come up with good names here, which is generally a sign that we have the wrong class break down.

When I first looked at moving from testcase class per class, grouping tests by fixture was my first stop. I found I had trouble finding clean slices through the fixtures that would neatly group everything. Even when I thought I had it right, my next test would require a change to a fixture that invalidated the breakdown I had chosen. It seemed like I was getting closer to a good approach, but it was still missing something. It wasn't until some people much smarter than me (although less bloggish :)) got together and came up with a better solution that everything started to click into place.

Testcase class per scenario

The change in thinking was to stop worrying about specific fixtures and setups and to group things by scenario. Bit anti-climatic I know, but the change in thought process seemed to fix the majority of the problems we were facing. By concentrating on scenario, and how our class under test should behave in that scenario, the fixtures naturally became more cohesive. The delineation between scenario-specific fixture setup and test logic became obvious. Fixtures and tests were easy to name, and trivial to write. For the first time I no longer struggled to come up with what to test next. Instead I just picked a scenario and started asserting facts about how my class should behave in that scenario and what state it should have.

Of course, this really just ends up as another way of breaking things down by fixture, which I've spent the entirety of the last section complaining about. But I've found the change in thinking indispensable in getting this to work. It's very hard to establish up front exactly what fixtures you're going to need (which results in a lot of churning of test code), but it is generally fairly obvious what scenarios your objects need to work under. In a way, the nice fixture break down is a nice side-effect that comes out of having well-structure scenarios.

I'm intending to post later on how to start from scratch using this approach, but seeing we've come this far lets look at the steps we can use to break our current testcase class into scenarios. Hopefully this will illustrate why I like this approach so much.

Refactoring toward scenario-based tests

The first step is to identify some scenarios in our existing testcase class. Any name referring to a time or event is a hint about the scenario the test relates to. The word "when" is a dead give-away. Scanning through our current tests I can see two that relate to *when the view is loaded* (`ShouldWelcomeEvilAgentWhenLoaded()` and `ShouldEnableFireButtonWhenViewLoaded()`). We also have `ShouldDisableFireButtonWhileFiring()`. When is the device firing? Looking at the code in that test, it is *when the fire button is pressed*. The `FireButtonShouldFireDoomsDayDevice()` also seems to relate to this scenario. And lastly we have `ShouldEnableFireButtonWhenFiringCompleted()` -- so *when firing completed* sounds like a good scenario for that one. (You don't have to identify all the scenarios up front, just enough to get you started.)

The second step is to pick one of the identified scenarios and create a testcase class for it. We can then proceed in a couple of ways. One way is to start moving tests one at a time into our new file that relate to the new scenario. Another is to copy and paste the entirety of the old testcase class into our new file, and start removing things that don't relate to the scenario. We repeat this until all our old test cases are in scenarios.

The final step is to factor out the duplication between our testcase scenarios into helper methods in a common base class or two. You can probably get away with doing this as you go, but I'd advise against doing it too prematurely. Wait until the code in two or three scenarios is screaming to be consolidated, and then do it. Otherwise you can wind up extracting commonalities that aren't really there, and we go back to losing cohesion between our setups and tests.

Let's have a look at the *when view is loaded* scenario. I'll create a new subfolder in our test project called `DoomsDayDevicePresenterScenarios`, then create a `WhenViewLoaded.cs` class. I'm quite a fan of underscore-overload (cue shameless post plug), so I'd prefer to use `When_view_loaded`, but I'll try and skip potentially heated debates for now. :)

The code below is how it looks after refactoring out some duplication:

```
namespace DaveSquared.Kaos.Tests.DoomsDayDevicePresenterScenarios {
    public partial class DoomsDayDevicePresenterScenario {
```

```
        [TestFixture]
        public class WhenViewLoaded : ScenarioBase {
            const string AgentsName = "Siegfried";

            [SetUp]
            public void SetUp() {
                CreateDoomsDayDevicePresenterAndDependencies();
                stubDoomsDayDevice.Stub(device => device.CurrentOperatorsName()).Return(AgentsName);
                RaiseLoadedEventOnView();
            }

            [Test]
            public void ShouldWelcomeEvilAgent() {
                Assert.That(stubView.WelcomeMessage, Is.EqualTo("Welcome KAOS Agent " + AgentsName +
            }

            [Test]
            public void ShouldEnableFireButton() {
                Assert.That(stubView.FireButtonEnabled, Is.True);
            }
        }
    }
}
```

Look how beautifully simple those tests are -- both one line assertions. And it reads nicely too: when view loaded, should welcome evil agent, and should enable fire button. You'll notice I've committed one of the sins I mentioned in my rantings about testcase class per fixture -- I've stubbed out a value in our setup that doesn't relate to all our tests. There are a couple of reasons that I don't care in this case. First, our setup focuses on what is required for this scenario, and the current agent's name happens to be important to this scenario, even if not to all the tests. I'm still getting good cohesion vibes. Second, I can use constants and properties with decent names to make sure the expected state of the scenario is clear. Thirdly, it just doesn't seem to cause me any troubles in real life, unlike when I tried breaking things down by fixture alone.

You may have noticed a weird partial class thingoe happening here. This was suggested by a colleague to make scenarios more discoverable from within Resharper. If all scenarios related to our presenter are inner classes of the `DoomsDayDevicePresenterScenario` partial class, then Resharper's "go to type" command (`Ctrl + T` on my configuration) will let us select the partial class, then list all the scenarios for us to jump to. It adds some code noise, and isn't strictly necessary, but does make navigating around your tests easier.
Let's move on to our next scenario.

```
public partial class DoomsDayDevicePresenterScenario {
    [TestFixture]
    public class WhenFireButtonPressed : ScenarioBase {
        [SetUp]
        public void SetUp() {
            CreateDoomsDayDevicePresenterAndDependencies();
            RaiseLoadedEventOnView();
            RaiseFireButtonPressedOnView();
        }

        [Test]
        public void ShouldFireDevice() {
            stubDoomsDayDevice.AssertWasCalled(device => device.Fire());
        }

        [Test]
        public void ShouldDisableFireButton() {
            Assert.That(stubView.FireButtonEnabled, Is.False);
        }
```

```
    }
}
```

Compare this to how one of these tests used to look.

```
/* plus [SetUp] method code */
[Test]
public void FireButtonShouldFireDoomsDayDevice() {
    RaiseLoadedEventOnView();
    RaiseFireButtonPressedOnView();
    stubDoomsDayDevice.AssertWasCalled(device => device.Fire());
}
```

Even though this is a trivial example, the new version is much clearer and cleaner. You'll have to take my word for it, but this applies even as you get to more complicated, real code. You tend to push more into the scenario setup (which is exactly what the setup is for) and the tests themselves stay trivial.

Let's look at the final scenario, and at our base class which we have used to keep duplication in check (although not eliminated, as we'll discuss later).

```
[TestFixture]
public class WhenFiringCompleted : ScenarioBase {
    [SetUp]
    public void SetUp() {
        CreateDoomsDayDevicePresenterAndDependencies();
        RaiseLoadedEventOnView();
        RaiseFireButtonPressedOnView();
        RaiseFiringCompletedOnDevice();
    }

    [Test]
    public void ShouldEnableFireButton() {
        Assert.That(stubView.FireButtonEnabled, Is.True);
    }
}

public class ScenarioBase {
    protected IDoomsDayDeviceView stubView;
    protected IDoomsDayDevice stubDoomsDayDevice;

    protected void CreateDoomsDayDevicePresenterAndDependencies() {
        stubView = MockRepository.GenerateStub<IDoomsDayDeviceView>();
        stubDoomsDayDevice = MockRepository.GenerateStub<IDoomsDayDevice>();
        new DoomsDayDevicePresenter(this.stubView, this.stubDoomsDayDevice);
    }
    protected void RaiseLoadedEventOnView() {
        stubView.Raise(view => view.Loaded += null, this, EventArgs.Empty);
    }
    protected void RaiseFireButtonPressedOnView() {
        stubView.Raise(view => view.FireButtonPressed += null, this, EventArgs.Empty);
    }
    protected void RaiseFiringCompletedOnDevice() {
        stubDoomsDayDevice.Raise(device => device.FiringCompleted += null, this, EventArgs.Empty);
    }
}
```

Issues with this approach

Duplication in setup methods

All the `[SetUp]` methods in our scenarios have some duplication -- creating the subject under test and raising basic events. Why don't we factor all this into the base class? There's a few points to consider here. First, do we really want to hide this duplicated setup code? Scott Bellware questions this under the subheading Reuse: Friend or Foe in an article on BDD for CoDe Magazine:

> "Specification code is intended to document the behaviors of the system, and this often means leaving duplicated code in-place to support the learnability of the specs... If you do move common context code to a base class, do so with care for how youâ re impacting the learning experience of the code"

Having the setup steps repeated to a degree in each scenario (although with the logic behind each step encapsulated as methods in the base class) is great for documenting the scenario, and also helpful with scenario's with fixtures that diverge from the more standard cases (an example of a divergent fixture is one that tests exceptions thrown from a constructor, so we can't have a constructed object setup already).

On the flip side, developers have good reason to be wary of duplication, having Don't Repeat Yourself drummed into them from the moment they are first exposed to inheritance. You can do some really clever things in terms of context base classes. One of the cleverest of these I've seen is JP Boodhoo's developwithpassion.bdd approach, which has the downside of looking completely foreign to people who speak C#, but has the upside of being really nice once you get used to it.

If the duplication bothers you, you can have a general context base class with an existing `[SetUp]` method that provides hooks like `BeforeSetup()`, `AfterDependenciesCreated()` etc. This lets your scenarios hook into whichever parts of the setup chain it needs to and do things like stub out calls. The real trick comes when you want to chain together contexts, which is something JP's approach gives you, and something which you get for free with tools like rspec (using nested blocks).

This is something the purist in me is still questioning, but in practice we've just been leaving the duplication in and have had little to no problems with it, mainly because the duplication is contained to a small, related area, and because the logic itself is neatly encapsulated in a base class.

What goes into the base class?

Anything that can conceivably apply to every scenario. Fields for dependencies, methods to create a subject under test, common operations like raising events etc. You can eliminate some of this by using an automocker, but this can hide design smells like your SUT having too many dependencies.

Tests that apply to multiple scenarios

Sometimes you'll come across a test that should really apply across all the scenarios, an invariant if you will. There's a few approaches for dealing with this. You can put the test in the base class, and it will automatically run against all inheriting scenarios. I hate this approach -- it means you can get multiple failures from one problem, and I don't like the lack of clarity you get with tests being pulled in mysteriously from a base class. You can also create a specific scenario for these things, calling it `Always`, or if you like the when-convention, `WhenEver` :).

I really don't like either of these approaches. I've become increasingly suspicious of this actually being a test smell. If you have logic that doesn't depend on a specific scenario, then this could be a cue to extract this into another class to isolate it properly for testing. You can then check the subject under test uses the other class correctly, without worrying about the

behaviour changing across multiple scenarios.

We've used all three approaches, with my preference being for isolating the behaviour properly. The `WhenEver` style approach feels a bit dirty but hasn't caused much grief. The inherited test thing has had mixed results.

Using in conjunction with other organisational methods

Sometimes you end up sprouting classes or writing utilities that don't seem to have a home or fit a particular scenario. In these cases falling back to lumping everything in the one fixture can work just fine. As this post tried to show, it's pretty easy to move to scenario-based testcase classes in future once some scenarios start emerging or when the single fixture starts giving you grief.

Conclusion

Writing tests around scenarios is a way organising your tests which, in my opinion, makes test code easier to write, helps identify common fixtures, and improves the readability and maintainability of tests. Writing testcase classes per scenario ends up breaking down tests by common fixture, but focusing on the scenario makes this breakdown occur more naturally and more maintainably. It is also pretty straight forward to refactor existing tests into this format.

If anyone actually made it this far, I'd love to hear any comments you have on this, both so I can get valuable feedback, and also to lavish you with compliments for actually getting through this epic :). This technique has really dramatically improved how I practice TDD and approach testing, so hopefully it will also help someone else out there.