Welcome to Patterns of Effective Test Setup!

This talk is about writing tests. My assertion, if you’ll pardon the pun, is that you are making 4 specific mistakes that make your tests hard to read, hard to write, and hard to maintain. Those mistakes are causing you frustration, they are sucking the joy out of doing TDD, they are wasting your time, and they are wasting your employer’s money. You may not even know that you’re making these mistakes, but that doesn’t make them any less costly.

It’s not your fault, though; lots of really smart people have written lots of really smart articles and books about how to write testable code and how to isolate the logic you want to test and how to use tests to drive the design of your code. But even if you were doing everything right, just like all those smart people said to do, I assert that you might *still* be making these 4 mistakes.

That’s because people rarely think about test setup and test data as an area of necessary improvement, despite the significant impact setup has on the overall test quality. The techniques that I’m going to show you today have literally been the reason that writing tests is still financially viable for my team and the seven year old application that we’re building. These techniques have continued to pay dividends as the app has grown larger and much more complex over time.

(click for “Are you in the right place”)

Before I get into the good stuff, I want to set some quick expectations about this session.

First, this is not a Testing 101 session. I’m assuming that everyone here is familiar with at least the basics of unit and integration testing. If you don’t have prior experience then you may not have a frame of reference for some of the techniques I’ll discuss.

Second, this is not about mocking or stubbing or about how to write testable code, and the techniques I’ll show you are not specific to any testing framework or language. All of those things are crucial to your overall testing success, but I’m going to focus entirely on how you arrange the data that you need for your test.

Even though these techniques are not specific to any given language, I happen to work in C# and most of our tests are written in NUnit, so that’s what you’re going to see on the screen. If you’re one of those cool kids using xUnit or SpecFlow or Javascript or Ruby or whatever else, don’t worry. There’s very little that you’re going to see that couldn’t be easily adapted to your stack.

**Click for “Story time…”**

So why is test setup so important? To answer that, I want to tell you a short story about why I’m here and where these ideas came from.

This story begins 8 years ago when I had just joined my current employer. We were just beginning our agile transformation and everyone was super excited about having “user stories” instead of “requirements” and “story points” instead of “estimates”. In the midst of all that agile euphoria, we decided to require tests for 70% of the code in this new project.

Most of the team was new to testing and felt that 100% coverage was unreasonable, but everyone agreed that having that soft, safety blanket of tests around the most important 70% of the code was a good starting point.

The project starts out great, everyone’s writing tests and shipping features and things are going pretty good. But a few months later, after the code had started to get a little complex and we’d started revisiting features to add new functionality, I began to realize that something was wrong with our tests. I’d really shotgunned the Agile kool-aid and was expecting this transformational impact from testing, but the tests just weren’t delivering that value.

As we continued to modify existing code for new features, we found that despite our 70% test coverage requirement, many important tests were missing. It turns out that as the code got more and more complex it got harder and harder to write and maintain tests. So rather than testing the *most important* 70% of the code, developers ended up testing the *70% that was easiest to test*. And as you can imagine, that left a lot of important code uncovered.

One of the main reasons that tests were getting hard to write is that the application was growing more and more complex, and it was requiring more and more effort just to describe the starting point for a given test. So people began writing tests for the superficial aspects of the code and were avoiding writing tests for the really meaty business logic.

The best way to show how bad it was is with an example. In one particular case I needed to make a minor adjustment to a feature. The feature itself was complex, but the new change was relatively simple and I didn’t think it would take much time. Before writing any new code, however, I wanted to learn more about how the feature currently worked and I wanted to write a failing test. So I opened up the file containing the tests and my heart sank when I saw this:

**(click – code sample)**

and this….

(click)

And this.

That’s

(click) 29 string, integer and Boolean values being initialized,

(click) 7 different objects being created and

(click) 75 lines of code to understand.

(click) CRAP! And even worse, this is just the SHARED setup code for the test suite! Each individual test in the suite had more code like this, and each test depended on different portions of this mess.

It was clear that even though my change was simple, just modifying the existing tests would be difficult, let alone adding new ones. I went back to the team, increased my estimate, and spent way more time than should have been necessary implementing that change.

And this is not an isolated case! We have thousands of tests in our projects and we spend countless hours reading those tests and trying to make sense of stuff like this.

This sucks! But there is a better way.

**What’s on the agenda?**

Today we’re going to start by taking a deeper look at the anti-patterns and practices that you may already be following, and why they are so harmful.

Then I’ll show you four patterns and techniques that you should start doing for your in-memory unit tests, and then we’ll finish by looking at how to extend those patterns to your integration tests as well.

I tend to talk pretty fast so there will likely be some time at the end for questions, but please feel free to interrupt me at any time if you need to.

**What is “setup”?**

Before we go any further, let’s define what we mean by “setup code”. Most test frameworks allow you to share setup code between multiple tests. This is sometimes called “fixture setup” or “test suite” setup. The code I just shared is an example of fixture setup code.

In addition to the shared setup code, each individual test can set up its *own* test data. This is often called the “arrange” part of the test.

When I say “setup”, it’s a shorthand that refers to *any* code that creates test data. Test data can be objects in memory, it can be data that you put data into a database, it can be files you create on disk, etc. When I say “setup” I’m referring to anything that you do prior to running the actual logic that you’re testing.

**Mistake #1**

If the first step in getting better is to admit that you have a problem, then the first step in writing better setup code is to recognize 4 mistakes you’re making today and why they’re so harmful.

The first mistake is allowing your project to get to the point where it is significantly easier to describe business conditions in words than in code.

For example, in my main app, one of our core domain concepts is a thing called a “workflow”. There are very few things that a user can do that don’t involve a workflow in one way or another, which means that many of our requirements deal with workflows in different states.

(click for example)

It’s pretty simple for an analyst to say something like “*When a workflow is <configured like this> then the system <should do that>*”. In practice, though, actually creating a workflow in that state is complex; a workflow is composed of lots of smaller objects that work together, and they have to be set up in a logically consistent way to represent real-world code paths and to avoid runtime errors.

When it’s much easier to describe a scenario in words than in code, you end up with the setup nightmare I showed you a few slides ago. And if you can’t easily put your software into common states for testing purposes, then you’re either going to pull your hair out when you write tests, or you’re going to stop writing them.

**Dependencies you don’t care about**

The second mistake people make when setting up tests is constructing all of your object dependencies by hand.

In almost every system there are tests that only care about a *portion* of an object. A test about an Order’s SHIPPING STATUS may not care about its line items, or a test about a Customer’s ADDRESS may not care about their name.

But it’s not always possible to create objects and specify ONLY what you care about. In C# for example the object’s constructor may require things that are necessary to the domain model, but don’t actually matter to that specific test.

**click**

For example, in this test, all I need is a shipped Order.

**click**

But apparently an Order needs a Customer…

**(click)**

And a Customer needs some Addresses…

**(click)**

And by the time I’ve satisfied the constructor, I’ve written a whole lot of code when I really only care about two things: the shipping status, and whether or not the order accepts new items.

**(click)**

All of these things that I created, but that don’t actually influence the assertion I’m making, are noise. Writing a test like this is painful, but it’s also painful to *read* these tests. You have to work hard to filter the signal from the noise so that you can understand it.

Some objects that I deal with have 4, 5 or even 6 layers of composition. Object A uses B, B uses C, etc. If we had to deal with this for every single test, we would be writing way fewer tests. It would make me crazy to do this every day.

**(click)**

This mistake can also make your test code brittle.

What happens when the Order, Customer or Address constructors get modified? If you’ve ever made a simple change to your application, and then spent the next two hours cleaning up compilation errors and test failures in totally unrelated parts of the suite, then you’ve felt this pain.

**Mistake #3**

The third setup mistake is specifying a lot of explicit values in your setup code, when those values don’t actually impact the outcome of the test.

Imagine that you have code that fails if the Customer email address is null, or if some integer field is left at its default of 0. When setting up a test that executes that code path, you have to initialize those properties to avoid those failures. Those values that you set, which DO NOT MATTER to the test, are impossible to distinguish from other values that DO matter to the test.

(click for example)

When other programmers read your code, they have to spend time figuring out which values are part of the test scenario and which are arbitrary. Does this assertion *only* apply to customers that are in the “PasswordReset” state? Or does it apply to all customers?

And if you’re writing shared setup code, it can be hard to identify which values can be changed without impacting other tests using the shared setup.

**Mistake #4: Inheritance**

The fourth mistake I see in setup code is using inheritance as a way of sharing logic between multiple tests.

I often find that there’s a certain amount of boilerplate setup that’s useful across multiple fixtures. For instance, you might create an Order, a Customer, and a few Line Items and link them all together in a meaningful way. This arrangement could be useful when testing *any* of those objects or any number of related business features, so naturally we’d want to make that setup logic reusable.

(click for example)

A quick and easy way of doing that would be to create a base class that does the setup and then derive multiple fixture classes from it.

There are two problems with this. First, inheritance is a very restrictive way of achieving reuse. In C# you can only have a single base class, and there’s just no good argument for requiring that your Customer tests and your Order tests derive from the same base.

Secondly, as your system evolves over time, the needs of your tests might start to diverge. Maybe there’s one specific Customer test that needs to specify a distinct email address, or an Order test that requires that the Order not have any line items.

If the setup code is in the base class, it’s really difficult to manage those test-specific changes. You end up doing things like initializing data in the shared area, and then overriding parts of it in the body of each test. This just doesn’t scale over time. It’s confusing to read and it’s hard to maintain.

To properly reuse setup logic we need to get it out of a base class and into something more easily managed. I’ll show you what that looks like in a moment.

**There is a better way!**

So how do we avoid those mistakes and write tests that *don’t* suck up all our time, money and energy?

I’ve been writing tests for over ten years and I’ve boiled down all of my advice for doing effective test setup into 4 key practices.

**Key #1**

The first key to success, and the single most important thing you can do to improve your setup code, is to stop constructing test objects by hand. Instead, push object creation into some sort of helper method or object.

This gives you two benefits:

1. It often shortens your setup code, making it easier to write and read
2. It increases resiliency; if an object’s constructor changes, you potentially only need to update the helper method.

There are a couple of established patterns for handling object creation.

**Object Mother**

The first pattern that we tried is called Object Mother. The key idea behind this pattern is that you identify up front the different test data that you’ll need, and then you create static factory methods for each of those pre-defined states. For example, the “Order Mother” object might have a factory method for creating an order with an unpaid balance, or if you work with insurance, the “Policy Mother” object might create an insurance policy object with a specific combination of coverages.

Object Mother is a great way to get all of those noise values and objects out of your setup code, but it doesn’t really scale that well. As your software gets more complex you’ll need more and more pre-built objects in more and more pre-defined states. And as the number of pre-built objects and states grows it becomes harder to maintain them and harder for developers to choose between them.

(click)

Eventually, you end up with a mess like this. There’s one method for creating an order with different bill-to and ship-to addresses. There’s one for indicating that the credit card failed address verification. There’s one for specifying that the order was placed by a new customer.

There’s a ton of overlap here. What happens when someone needs an order that was placed by a new customer, and had failed the AVS checks, *and* had different bill-to and ship-to addresses? That exact scenario isn’t covered by any of these, so that developer would probably end up creating yet another method for their exact need. And that new method would probably have a lot of duplication when compared against the ones that already exist.

Object Mother is a really easy pattern to implement if you only need a couple of course-grained pre-built objects. We needed a lot more control over our test data, so we quickly outgrew this pattern.

(click, Data Builder)

The next thing we tried was a pattern called Data Builder.

Rather than a factory that returns pre-built objects, Data Builder lets you create customized objects in the body of each test. It’s common for this to be accomplished via a Fluent API that exposes the things that can be customized.

The general structure in this pattern is that initialize the builder itself and then start calling methods to customize various parts of the object. Those methods are chained together and at the very end you call a Build() method which returns your fully built object.

These can be very simple, or they can get pretty complex as you see here where we’re creating both an Order and Customer with customized properties.

The main benefit of the Builder pattern is flexibility because it lets you can create the precise data that you need for each test, and that makes it a much better fit for larger or more complex applications.

(click)

However, I’m really not a big fan of the Fluent API. It’s verbose and adds a lot of noise, as you see here. The green arrows are pointing to the significant data that I’m creating and the red circles are basically the “noise” that we get from the fluent API.

All this noise code means that the setup code is harder to write, read, and maintain. And on top of that, actually implementing the Fluent API is tedious. It requires a lot of boilerplate code that in my experience just isn’t worth the hassle.

(click, Test Helper)

Eventually, we created a hybrid of these two patterns that combines the static factory class of Object Mother with the customizable nature of a Data Builder, minus the Fluent API.

We call this the Test Helper pattern. It’s a terrible name, but it’s been a really useful pattern for us and I think it’s highly applicable not only in C# but also in JavaScript or Ruby.

(click)

The first step in implementing a Test Helper is to create a static factory method like Object Method, but give it a generic name like “Create” and expose all of the data that you want to customize as method arguments.

This gives us a flexible, extensible mechanism for creating data that’s specific to each test, without all the overhead of that fluent API.

**Key #2: Only what matters**

The second key to success is to make your test data as expressive and as readable as possible by specifying *only* the values that are significant to the test outcome.

Like I mentioned earlier, when someone is reading your setup code and they see a string or integer literal value, they have to figure out whether that specific value is relevant or not. If you create a customer and set their tax exempt status to TRUE, does that mean that the assertion you’re making only applies to tax exempt customers? Or was that just an arbitrary value you picked because you had to pick something?

The goal of any good test is to communicate how the system will behave given a specific set of inputs, and your setup code can’t effectively do that if it’s full of noise values that dilute that message.

**(click – liberal use of defaults)**

If your language supports the concept of “optional parameters” then our Test Helper pattern is easily adapted to follow this guideline.

Basically, you just take the static factory method and you specify default values for just about everything.

Then when you call the method, you only specify those specific values that you care about.

**(click – behind the scenes 1)**

Behind the scenes, the helper itself is responsible for assigning meaningful default values for everything that the caller left empty.

For primitive properties I generally make all of the arguments nullable. You *could* specify an actual value in the default, but sometimes it’s helpful to know whether the caller provided a value or not. You can do that with null, but not if your arguments are given a default value in the argument declaration.

**(click – behind the scenes 2**)

If you’re building an entire object graph, you can expose children or dependencies as arguments as well. If the caller leaves them null, just delegate to the relevant helper to create them. This approach keeps each helper focused on a single object type while still supporting very rich and complex compositions.

**(click – accidental equality)**

There’s one potential gotcha with this approach. You should be very careful when creating objects with hardcoded values as their defaults. This can lead to something I call “unexpected equality”.

For example, let’s say you create two different Customers from the helper, you pass them into some method, and finally you assert that the method returned a result with an email address equal to customer #1.

The assumption here is that the test will fail if the code returns the email of customer #2. But if the CustomerHelper object sets a default email address of NULL, or some hardcoded static value, then this test will pass even if the logic is faulty. This is what I mean by “unexpected equality”; if I create two separate objects, I don’t expect them to pass an equality test.

By default, I prefer to make all values unique. I want to *force* programmers to be explicit if they want things to be equal.

**(click – short guid)**

One thing that makes it easy to assign unique values is a class called ShortGuid. This is basically a shorter, URL-friendly, base64-encoded GUID, and you can get the code from this link.

Whenever I’m creating a name or a title or something, I use a ShortGuid as the default. It guarantees that no two objects I create will share the same value, unless I explicitly set them up that way.

**(click – id sequencer)**

This issue of unexpected equality also applies to integer values. For instance, if I create two customers from the customer helper, I wouldn’t expect them to have the same ID value unless I explicitly assign it that way. So how do we assign a unique ID to each integer?

Instead of a random number, I created a static class called the IdSequencer. It basically starts a counter and hands out a unique value each time I call “next”. Any time I have a helper that creates something with an ID property, I expose the ID as an argument and then I default it using the sequencer.

This technique is actually really, really helpful if you want to use these helpers for integration tests too. You’ll see that in a few minutes.

**Key #3: Scenarios**

Test Helpers are great at returning single objects. But what if you need of keep track of multiple objects AND their relationships?

For example, let’s say you have an ecommerce site, and one of your business rules is that all orders of heavy equipment, from new customers, with a different bill-to and ship-to address, must go through a verification process to prevent fraud and expensive shipping mistakes. To write that test, you’ll have to create a customer with no previous orders, assign different bill-to and ship-to addresses, create an order containing a heavy equipment item, and attach the customer to the order.

(click)

That test would look like this. Here’s the customer helper, where I’m creating separate addresses, and here’s the order containing a heavy equipment item.

This isn’t a *bad* test, but it could be better. Wiring up all this stuff by hand is tedious, and it works against our goal of being able to easily and concisely describe the context for a given test.

And if you have multiple tests that need minor variations on this setup, this leads to a lot of copying and pasting.

(click)

In these situations I use a pattern that we call a Scenario. This is essentially a façade that wraps the coordination of multiple Test Helpers towards a common goal and makes your setup code cleaner and more readable.

You’ll notice that while a Test Helper is a static factory, a Scenario is something that you instantiate. You can still customize the result but you do it with constructor arguments and not method arguments.

The reason for this difference is that a Test Helper returns one of our core domain objects, and we don’t want to litter our app code with constructors that exist only for testing. The factory pattern works great to isolate the Test Helper logic from the core objects.

For scenarios, though, we’re actually creating *multiple* objects, and we need a handy way to keep track of all of those objects. If we implement the Scenarios as brand new classes, then we can use instance properties of those classes to expose pointers to the objects the tests will care about. You could still use static factory methods if you wanted to, but it saves a little code to just use the constructor instead.

(click)

This is what the Scenario itself looks like. In general, everything that the Scenario creates that a test might need to easily get a reference to is exposed as instance properties.

(click)

Here’s another example of a Scenario. In this case, we’re creating multiple orders for a single customer, each with different characteristics. The Scenario exposes each order as a distinct property so that the test code will be very clear in terms of its intent.

(click – split payment example)

And here’s another example of a Scenario. This one creates an Order that has two payments associated with it, one mastercard and one VISA. If you were creating this data manually then you’d need to make sure that the order consisted of line items that add up to $85. But by extracting the logic into a helper, you don’t have to worry about it; the helper can contain a little bit of logic to make sure that everything it creates is logically consistent with itself.

(click – drawbacks)

I recommend the Test Helper pattern without reservation. There’s literally no good argument not to. But there are a few drawbacks to the Scenario pattern that you should consider.

First, you’ll need to figure out for yourself the correct balance between specialization and customization. If you create a large number of Scenarios, each highly specialized for a specific use case, then you’ll generally find that each individual is scenario easy to maintain over time, but you’ll end up creating a lot of them with chunks of duplicate logic. If you create a smaller number of general purpose Scenarios that can be highly customized via arguments then you’ll create fewer objects, but it may be hard to refactor a Scenario because it might be used in lots of different ways by lots of different tests.

That balance is hard to predict in advance. My rule of thumb is that if I’m only creating two or three objects in a test, I’ll usually just call the Test Helpers directly. But if I have more than that, AND I’m reusing that logic in more than two or three tests, I’ll extract out a Scenario.

Also, remember that Scenarios are helpful because they are facades that encapsulate and abstract the coordination of multiple objects at the same time. If you need a lot of Scenarios, that might be a code smell indicating that those facades would be more useful in the core app itself. For example, if we go back to that e-commerce scenario I just showed you, you could argue that a better design might be to create an object or service that encapsulates all of the information about a given Order, and then base your business rules on that abstraction instead. If you go that route then you may need fewer Scenarios in your unit tests, but they are still really handy when it comes to integration tests.

**Key Practice #4: Tell a story**

The fourth key to effective test data setup is to tell a story with your data.

At the core of their essence, tests are valuable because they help us understand our software. And in order to fully deliver that value, they have to be designed to effectively convey information when they are read. You could write the most bassackwards and incoherent tests possible and the computer could still figure out what to do, and whether your assertions are true. But that won’t help your poor coworker who opens that file a month later and needs to make a change. CPU cycles are cheap; your coworker’s time is not.

When it comes to setup code, there are a few simple practices that I recommend you consider.

(click)

First, names matter. Give your test data names that communicate how they contribute to the baseline state that you’re setting up.

In this example, the actual values associated with the constants are irrelevant; they are just two arbitrary values that we need to differentiate. Note how the names of “original value” and “new value” give extra clarity to the assertion and support the name of the test.

(click)

Or if your test revolves around the fact that a customer is disabled, don’t just call it “customer” or “c”; call it “disabledCustomer”.

(click)

If your test contains multiple objects of the same type, differentiate them! It sounds obvious, but I still see smart, experienced developers writing tests with variables called “customer1” and “customer2”. That is a completely unnecessary mistake that reduces the clarity of your setup code.

In this case, I’m testing some search logic. To write a complete test, I need assert both that the code DOES return something that matches the filter and DOES NOT return something that doesn’t match. I could have named these things “customer1” and “customer2”, but then the assertions wouldn’t convey as much meaning as they do now. Good names make it clear that one of these records is supposed to match, and the other is not.

(click)

Here’s one final example of how good names can convey meaning. This test is asserting that a specific feature returns data ordered in a specific way.

If I create the test data in the same order in which I expect them to come back out, then it’s possible that the sorting code isn’t do anything at all, and the test is passing by coincidence. A better test would be to create the data in a *different* sequence than it should come out, because then the test will only pass if it’s actually applying some sort logic.

Here’s one way to write that test. This isn’t terrible, but it’s not telling as clear of a story as it could.

(click)

Here’s that same test, but where I’m assigning those dates to named variables that make my intentions far more clear.

On a really simple test like this, it may not matter. But if you get in the habit of doing this, it will pay off as your setup logic gets larger and more complex

(click)

My last tip about “telling a story” is use clear and consistent “dummy” values. Ideally, if a value doesn’t matter to the test then it shouldn’t be set explicitly, but sometimes for various reasons we *do* need to assign a value, but the value itself doesn’t matter.

In these cases, I generally avoid *null*, 0, 1, or 2. It’s not uncommon for those values to have special meaning for a given property or type, and I want my arbitrary selections to be obvious as such. So I tend to use strings that identify themselves as irrelevant or arbitrary, and I generally use 42 as my go-to dummy integer. It doesn’t really matter what values you use, just be consistent. The whole point is that you want to send a clear signal to whomever is reading your code that the value doesn’t matter, and consistency helps make that clear.

**Click for recap**

That brings us to the end of section 2 of this talk. For a quick recap, here are the 4 keys to writing effective setup code:

1. Stop creating objects by hand
2. Create your helpers to use default values, so that your tests only have to specify those values that actually impact the outcome
3. If you have a complex piece of setup that you want to reuse, put it in a Scenario object
4. Tell a story with your setup code and test data. Make the code as clean and expressive as possible for the most value over time.

As an example of how powerful these techniques can be, let’s first revisit that nasty chunk of setup code I showed at the start…

(click)

(click)

All three screenfulls of it.

(click)

Here’s that same chunk of code, cleaned up and rewritten using Test Helpers. I determined that many of the objects being created were just dependencies, and nearly all of the literal values were irrelevant to the test outcomes. After pushing all of that stuff into helpers we’re left with just this.

There’s still a lot of room for improvement, but this is manageable now.

**Advanced techniques – Integration tests**

Until now we’ve been talking about in-memory objects only. And the usefulness of those patterns to you probably depends on how your code is structured. If you have a standard object oriented web app then helpers for creating objects will be really useful. If you’re doing a lot of functional programming, or if you tend to abstract things away into services that are easy to mock out, then you may not have as much of a need to create complex object graphs in memory.

Regardless of your app’s design, however, eventually you’re going to want to test your data access code, or you’re going to need higher level integration tests to do some “real world” verification of your system. And when that time comes, it would be really nice to use the *same* object creation patterns to put real data into a real database as we do to create data in memory.

(click)

Unfortunately, that’s a little easier said than done.

First, you have to deal with foreign keys. Your app may not care if you create a Line Item by itself, but you can’t save the Line Item to the database without an Order. And maybe you can’t create an Order without a customer. It’s the same issue we had with constructor dependencies, but in the database. This means you have to new up the entire object graph and then save objects to the database in the correct sequence. And if your database is responsible for generating primary key value, you have to go back through all of those objects and update their IDs once everything has been saved.

You also have to deal with column constraints. Some columns will reject NULL, others may have max length constraints that your in-memory test data could violate.

Lastly, you may want to clean up that test data when the test run is over. I run my automated tests against the same database I use for manual testing. I don’t want that database filled up with junk data because it can impact performance, waste disk space, and it’s just ugly.

These things make integration tests difficult, but it’s manageable with a few extra additions to the Test Helper pattern.

(click)

The first thing is to add a *Save()* method to your Test Helpers classes. In our system, since every other Test Helper method is static, we make the Save method static as well and we pass in the database connection as an argument. We use NHibernate and this ISession thing is basically the database gateway object. If you use Entity Framework then you might pass the db context here, or you might pass a raw ADO.NET connection or whatever other object that you need to talk to the database.

You could do some dependency injection if you wanted to, but I created this pattern way back before we had a good DI framework in place and it’s never bothered me enough to change.

It’s important to note that there is a Save method on each individual Test Helper that supports integration tests. There is no global or generic Save method for reasons that you’ll see in a moment.

(click)

To deal with foreign key constraints, each Save method delegates to other helpers to save dependent objects. In this case, we can’t save an Order unless it references an existing Customer ID. By delegating to the Customer helper to create that record we keep each individual helper clean and focused on a single type.

(click)

Third, the Save method is responsible for dealing with primary key values that are assigned by the database. This is where that IdSequencer object comes in handy. Many ORMs use the ID property to determine if they should issue an INSERT or UPDATE query. If our helpers always assigned IDs of 0, then we’d always be creating new records in the database, but then we’d have that “unexpected equality” issue I talked about earlier. But if we assign a random, non-0 value, then the ORM will try to issue an UPDATE.

Using the ID Sequencer addresses both issues. The sequencer knows which values it’s handed out, so before we save our object we do a check to see if the ID currently in use was assigned by the test helper. If it was then we reset it to 0, and cause an insert. If it wasn’t, then that means we’re dealing with an object that already exists in the database and we do nothing, and cause an update.

Note that we don’t need to reset ALL values that were assigned by the IdSequencer, only entity IDs. Only properties that map to primary keys.

(click)

Finally, the helper calls out to the ORM to insert or update the database. If you’re not using an ORM then the exact details might change, but the general pattern should hold up.

The important thing is that each helper should know how to save the objects that it creates. Keep in mind that the implementation of Save will be driven by your Create methods – you don’t necessarily need to be able to save any arbitrary object, only those configurations created by the helper.

If you follow this pattern then you can basically write a bunch of unit tests with in-memory data, copy and paste the setup code from one of them into an integration test, add a few calls to this Save method, and you’re done. It’s pretty sweet when it comes together.

The final thing I want to talk about is how to prevent this test data from lingering in the database when the test run is over.

One possibility is to reset the database to a known state at the start of each test run. This works, but I don’t recommend it. One reason is that it’s a massive pain to maintain that baseline backup every time the schema changes or new data is added. Another reason is that I use the same database for unit tests as I do for manual testing. It really sucks to spend a bunch of time crafting data for a manual test and then lose it because you accidently ran an integration test that wiped the slate clean.

(click)

Another option is to start a database transaction when each test starts, and then roll that transaction back when the test is over.

Years ago I wrote a custom NUnit attribute called Rollback that does this for us. Any test that has this attribute is automatically executed inside of a transaction that is discarded when the test finishes. The implementation for this is on my GitHub, but at this point it’s like seven years old. There are probably newer and better ways of doing it now, but you’re welcome to copy my approach if you’re using NUnit.

**FAQ #1**

I’ve given this talk a few times and a couple of questions often come up.

The first question deals with libraries that can create or populate objects automagically. There are a number of these in the .NET space including NBuilder, AutoFixture, AutoPoco, etc and people want to compare and contrast those approaches versus my hand-rolled helpers.

The simple answer is that no library that you download can make better decisions about your test data than you can. Early on, when your objects are simple and you don’t have lots of special cases in your system then sure, those libraries might seem useful. But eventually, as things grow and get complex, you’re going to need more control over how the defaults get assigned, or more control over how child objects are wired up, or whatever. And those libraries sometimes give you hooks that you can use to shoehorn in some special stuff, but if you’re stretching that library to its limits to do what you need, you may be better off just writing your own library.

Basically, I look at it like this: when your app is small and simple, then the libraries are easy to use. But if your app is small and simple, then introducing my Test Helper pattern is super easy too. And since Test Helper is designed to deal with complexity, then you’re starting out with a solid foundation that will scale right along with your app code.

**FAQ #2**

I’ve also been asked for suggestions on introducing these patterns to existing systems that already have a lot of complexity in place.

And the answer to that question is that the absolute best time to introduce this pattern is at the very beginning. The second best time is right now. If you’re starting with a really complex system then it will take a bit of effort to get things moving, especially if you’re doing integration tests, but I promise you it’s worth it. My suggestion is to start with “leaf” objects that don’t have any children or dependencies and work from there. If you try and start with that huge, massive, ancient beast that lives at the heart of every legacy system then you’ll probably have a rough time.

**FAQ #3**

blah

**Closing**

And that brings us to the end of this session. Here are the 4 keys to effective test setup as well as links to these slides on Github, my website, and my twitter account. If you have any questions or comments I think we have a few minutes right now, or please feel free to seek me out online.

THANK YOU!