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 doing things that make your tests hard to read, hard to write, and hard to maintain. You’re doing things that will frustrate you, waste your time, and waste your employer’s money. You may not even recognize the anti-patterns that you’re following, 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 your tests would still suck in one particular area: the test setup.

This is something a lot of people overlook when writing tests, and that is a huge mistake. The techniques that I’m going to show you today have literally been the reason that writing tests is still a financially viable exercise for my team and our six year old application. I started these patterns years ago and they are still paying dividends today.

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.

Second, this is not a talk about mocking or stubbing or any specific testing framework or library. Those things are all important, they just aren’t my focus today.

Lastly, this is not a talk about a specific language or platform. The code I’m going to show you is in C# and uses NUnit. If you’re one of those cool kids using xUnit or SpecFlow or Javascript or Ruby or whatever else, don’t worry. The real point of this talk is patterns and principles and you should be able to apply them to lots of different tech stacks.

**Click for “Story time…”**

To get started, I want to tell you a short story about why I’m here and where the content for this talk came from.

This story begins 8 years ago when I had just joined my current employer. We were in the middle of 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.

I don’t remember why we picked 70% exactly, but the general idea was that 100% coverage seemed unreasonable and that we thought it was a good starting point to have this soft, comforting safety blanket of tests wrapped around the most important 70% of our code.

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.

The first problem that I identified was 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.

The second problem was that the tests that *did* get written were a nightmare. In one particular case I needed to make a minor adjustment to a feature. The feature itself was complex, but the new change was 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 became clear that even though my change was simple, even 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 I’m going to show you the patterns that we created to help that team write more and better tests.

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 some of the basic setup patterns that work really well for in-memory unit tests, and then we’ll finish by looking at how to extend those patterns to your integration tests when you need to set up data in a database or external system before running your test.

We are NOT going to talk about mocking, stubbing or how to write testable code. Those are important topics but they are out of scope for this session.

**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 referred to as “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.

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 the filesystem, etc. When I say “setup” I’m referring to anything that you do prior to running the actual logic that you’re testing.

**Anti patterns intro slide**

The first step in writing better setup code is to recognize a few anti-patterns, that you’re probably following today, that are making your tests harder to write, read, and maintain than they need to be.

On my team, I identified four specific things and practices that were causing our problems.

**Easier to describe scenarios in words than code**

The first anti-pattern is when it is significantly easier to describe business conditions using words than with code.

For instance, 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 in the system that don’t involve a workflow in one way or another.

Because these things are so foundational, we gets tons of requirements like “*When a workflow is <configured like this> then the system <should do that>*”. That’s simple for an analyst to say, but in code these workflows are comprised of lots of smaller objects and are controlled by a lots of configuration settings. If the objects aren’t constructed in a logically consistent way, we get runtime errors. That’s why that first piece of code I showed you was so gnarly; even though the requirement itself was easy to describe, actually putting the system in the desired baseline state was a lot of work.

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 next anti-pattern 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 anti-pattern 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.

**“Noise” values obscure what’s really important**

The third anti-pattern is when you’re specifying a lot of explicit values in your setup code, but those values don’t actually impact the outcome of the test. Just like all of those excess dependent objects, irrelevant primitive values are noise that obscure what’s really meaningful.

(**click)**

Imagine that you have code that fails if the Customer email address is null, or if some integer field is left at its default value of 0. When setting up a test that executes that code path, even if it’s not really central to the test itself, 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. This means other programmers have to work much harder to determine what they can change without affecting the test.

In other words, in order for tests to serve as useful documentation, they have to clearly communicate the conditions under which their assertions hold true. They can’t clearly communicate that if there’s no way for another programmer to differentiate that values that are significant from those that aren’t.

**Re-use of setup code -> restrictive inheritance**

The final anti pattern is using inheritance to share setup code between tests.

I often find that there’s a certain amount of boilerplate setup code that’s useful in multiple tests. For instance, you might create an Order, a Customer, and a few Line Items and link them all together. 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.

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 replacing or overriding parts of the shared data in the body of each test, but that’s ugly and error prone.

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 to summarize, poor setup code makes tests hard to write and maintain.

Tests that are hard to write don’t get written.

Tests that are hard to read waste our time.

Tests that are hard to change tend to get broken. And tests that break and are hard to change or understand have a tendency to get “fixed” with the delete key.

So how do we 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 it well into 4 broad guidelines.

**“4 keys to success” – no “new”**

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 maintain
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 default values to anything that wasn’t otherwise specified.

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 or not the caller provided a value. You can do that will 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”.

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 unique values easy is a class called ShortGuid. This is basically a shorter, URL-friendly 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?

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.

**Good setup code is highly expressive**

The first principle is that tests should be “highly expressive”. What does that mean? The dictionary defines “expressive” as “effectively conveying thought or meaning”, so this means that the setup code should clearly convey the scenario that’s being created and why. We can’t understand the assertions the test makes unless we also understand the context in which they are being made.

It’s also important that readers can come to that understanding quickly. We have thousands of tests and we spend a lot of time each day reading them; if we can’t skim the test code and understand them, we very quickly begin to waste a lot of time.

And lastly, when the tests are highly expressive then it’s easy to look at similar tests and spot what’s different between them. This is especially important when you’re dealing with complex systems because you may have a large number of tests that cover very similar scenarios with small, but important, differences. Being able to quickly identify the things the tests have in common, and the things they don’t, is really important.

Here’s an example. Quick, what’s the point of this code?

It’s hard for me to tell too, and I wrote this.

(click)

Now how about this? It does the same thing, but I’ve removed all of the extraneous noise and all that you’re left with is a simple, concise expression of my intent. This is the sort of readability that I’m talking about when I say that test code should be highly expressive.

**Entire test should fit on 1 screen**

Another way to write expressive code is just to write less of it. My general guideline is that tests should not take up more than 1 screen of code. The purpose of this constraint is to keep me hyper-focused on the clarity of my setup code; if I can’t describe the test data in just a few statements then I’m probably writing the setup code at too low a level.

It’s like refactoring a method into smaller and smaller pieces: just keep asking yourself, could I describe this with less code without losing any of the true essence of what I’m trying to do?

This is not a hard and fast rule, but I’ll show you some techniques in a few minutes that help me meet this goal the majority of the time.

**Avoid intermediate objects**

Another way to fit a test onto a single screen is to write more concise code, and one way to do that is to avoid what I call “intermediate setup objects”.

For example, sometimes we create one object only so that we can use it to create something else, and we never refer to that first object again. In this example, I need a Customer to create an Order.

This results in my writing two physical statements in order to accomplish one logical operation, which is the creation of an Order that I can use in my test. It’s not immediately clear if the Customer is reused elsewhere. Is it safe to change?

(click)

Moving the instantiation of the Customer *inside* the Order constructor allows us to express one logical, meaningful operation as 1 code statement. Avoiding intermediate objects like this makes the code more concise and improves clarity, making the test much easier to skim over and quickly understand.

**Make clear *why***

Of course, making it easier to understand *what* you’re doing is only half the battle. Good, expressive setup code also communicates *why* we’re doing it. Specifically, it should help the reader understand why or how a particular piece of test data will affect the test’s outcome.

(click)

As an example, this test asserts that product reviews are sorted in a specific way. The setup code intentionally adds data in reverse order than it should be output because if we insert the data in the same order we expect to get it back out, how do we know the sorting code actually works? So in this test there’s a comment that draws attention to this significance.

It’s worth noting at this point that I have very different rules about comments in test code than in production. In production, we try very hard to make comments unnecessary so we’d probably do an extract method refactoring to pull this code into a well-named method and then remove the comment.

I’m less likely to extract small, one-off methods in test code however. We tend to put multiple tests in the same physical file and so extracting a lot of single-use methods clutters up the files and makes them hard to manage, especially at scale.

Another reason that we avoid comments in production code is that they often get stale when the logic changes but comments aren’t kept in sync. In my experience, however, setup code has a much lower rate of change than application code. Once a test is written, other than to make minor changes to stay in sync with application code, it’s very rare for the setup logic to undergo a significant enough change that these comments get stale.

Multiple data setup

Another way that I use setup comments is to describe how data are related to each other for the purpose of a test. In this example you can see that I’m creating three Activities, each with a different status, and then I’m using one of them to set up some additional data. Any competent programmer could read the code and figure out those relationships, but a short comment makes my intent explicit. It makes the setup logic way easier to understand and maintain for a negligible amount of extra effort.

TEST NOTE

The final way that I use setup comments is to explain why a test exists in the first place.

Here’s another example from a real project. In this particular feature the UI *always* posts a JSON string to the server in a specific field, but the value of that JSON string is only meaningful if the user had performed a specific action in the UI. We could tell that the user had performed that action if some other data was also present.

So in other words, this test was important but only because of some very specific implementation details. It’s quite possible that at some point we might re-write that UI and render this entire test pointless, and we’d want to be able to understand that we could safely delete the test (especially if it was failing!).

However, there’s no good way to describe that particular nuance in the test name. In case like this I’ll add a comment to the top of the test that adds the necessary context. As a personal convention, I always prefix these types of comments with “TEST NOTE”. Following this convention helps me spot features that are too complex, or where the test name might be insufficient.

I use this style of comment very rarely; out of approximately 2000 tests, less than 30 use this. But in those 30 cases, these comments are extremely valuable and following this naming convention has been very useful.

**Highlights what matters**

The next principle is that “good setup code highlights data that impact the test’s outcome”. The goal is to clearly differentiate the data that matter to the test and those that don’t. This makes it significantly easier for programmers to read and maintain the tests.

(click)

One way to do this is to replace key values with named constants. This has three main benefits: it draws attention to those values and visually reinforces their importance, it makes the code read more like English, and it makes it easier to spot differences between otherwise similar tests.

In this particular example we’re really just aliasing a specific enum value to a named constant. The main benefit in this case is that visually, the uppercase constant sticks out a little more than the enum name. If you have two tests that are very similar, and they assert against different enum values, this technique can make those tests easier to compare/contrast. It’s perhaps a minor benefit, but in my opinion a bunch of “minor benefits” added together yields some significant benefit.

(transition) In this case, the name is just communicating its *value*. This technique is even more effective when the name communicates its *purpose* in the test.

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.

Of course, giving things good names applies everywhere, not just to constants. Just like in production code, give test objects names that communicate their purpose. If your test revolves around the fact that a Customer is disabled, don’t just call it “customer”; call it “disabledCustomer”.

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” in them and it’s a completely unnecessary waste of someone else’s time.

Tests should tell a story about some rule or expectation you have of the system, and those stories are easier to understand when the actors have meaningful names. It doesn’t take a lot of effort and if you practice this consistently, especially in concert with the other tips I’m mentioning, it can absolutely improve the quality of your codebase.

(transition)

Obviously it’s important that meaningful data stand out. It’s also important to downplay data that aren’t as meaningful.

A common example of when this is useful is when you’re testing code that does some sort of search or filter. In that case, it’s just as important to ensure that the code returns the things it should as it is to ensure it does NOT return what it shouldn’t.

My convention in this case is to use the term “distractor” when naming the objects that *shouldn’t* get returned. This pattern communicates that the object itself is not important, as long as it can be differentiated from the target. And by using this name consistently, my co-workers can immediately recognize the point of these objects. This sort of instant pattern recognition is another key benefit of expressive code.

(next)

Another way to downplay data that don’t impact the test outcome are with consistent dummy values. In this example, the constructor takes 3 arguments but I only really care about 1 of them. I can indicate this by passing values that clearly communicate that they are irrelevant to the test outcome.

In general, you want to avoid passing default values like null or 0. Not only can this lead to null reference exceptions if the code isn’t null-safe, but it makes it very hard to differentiate between the value you passed in and an uninitialized property that has its default value.

I also like to avoid using numbers like 1 or 2. These frequently appear in code and tests as meaningful values and I want arbitrary numeric values to stand out more. I personally use 42. The actual value doesn’t matter, the key is that you use that value consistently as a way of communicating your intent.

**Avoids inheritance**

The third principle of good setup code is to avoid inheritance, at least for the purposes of creating test data.

(click)

The first reason is that inheritance makes it hard to tweak shared data for each test. Imagine that you have 5 tests covering the same piece of code. The setup for those tests will be very similar; for the sake of example, let’s say that any 2 tests will have 90% of the same setup needs. The problem is that each test could have a *different* 90% in common. One test might need to specify an Order’s ship status, another might need to change the Customer’s name. This *can* be handled with inheritance but it’s not clean and it’s certainly not elegant.

(click)

To re-use setup logic we need to get it out of the base class and into something more easily shared. In a few minutes, I’ll show a clean and elegant technique for doing this without inheritance.

(click)

Just to be clear, base classes *can* initialize shared services or stub out certain types of behavior. The key is that you want to avoid creating *data* in the base class.

**Resilient to change**

The 4th principle of good setup code is that it is resilient to changes in the main app. Simply put, tests should be *adding* value, not subtracting it by demanding constant attention.

If you’ve ever made a simple change to your app code and then found a hundred compilation errors in your test projects then you know exactly how damaging this can be.

(click)

In the same vein, we want to be able to easily refactor our test projects. Just like our app code tests will evolve over time. We’ll adapt new patterns, add new libraries, and move things around. I’ve seen smart developers be sloppier in test code than in “real” code and it’s a huge mistake. Tests should be malleable and easy to manage over time.

**Recap**

We just identified 4 principles of good setup code: it is highly expressive, it differentiates the data that impact the test outcome and those that don’t, it avoids using inheritance as a way of setting up shared data, and it doesn’t require a lot of effort to maintain over time.

Next we’re going to look at some concrete patterns and practices for adhering to these principles in your own tests.

**Patterns and Practices**

**Avoid “unexpected equality”**

First, avoid creating objects that have properties equal to *null* or specific hardcoded values because this can lead to what I call “unexpected equality”.

For example, let’s say you create two different Customers from the helper, one called “target” and one called “distractor”. You then do whatever you’re trying to test, and finally you make an assertion that your method under test returned a result that’s equal to the target customer’s email.

The assumption here is that the test will fail if the code returns the distractor customer’s email address. 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”.

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)

One way to create unique string values is with a class called ShortGuid, which you can find online. It’s a simple utility that compresses a normal GUID into a 22-character string by Base64-encoding it. The result is short, URL-friendly, and unique.

In this example, every string property on the Customer will be unique unless the caller specifies a specific value to use.

(click)

To create unique integer values, I created a class called the IdSequencer which you can get from my GitHub page. Every time you call .Next() it returns a new value, and it keeps track of all of the values it’s assigned. In a little bit I’ll show you how this class is implemented and why it’s useful to keep track of the values that have been assigned.

I always assign unique integers to ID fields. I may or may not assign unique integers to other fields, depending on what they are.

(click)

It’s also important that each Test Helper is concerned with creating a single type of object only. If your object has a dependency on something else, delegate to another helper to create that dependency.

Here’s an example of an Order Helper that allows its Customer to be passed in as an argument. If the caller doesn’t pass a Customer then the customer argument defaults to null, and the helper delegates to the CustomerHelper. This way we end up with a fully-specified object, and each helper stays focused on doing one thing only. This is really important and will help you maintain and organize the code as the number of helpers grows.

(click – like an Object Mother)

Generally speaking, my helpers have a single method called Create. I purposefully make this as generic as possible because the whole point is that the caller specifies whatever data it cares about.

In some cases it does make sense to create specialized helpers like an Object Mother. This is useful if you notice common patterns developing in your setup code. For example, let’s say you have an e-commerce platform and many of your tests deal with orders that have multiple payments. Rather than calling multiple helper methods to create those payment objects, you could create a helper like this one that wraps that pattern into a single method call. This means you have less code to write and makes it easier to find and refactor related tests.

Do this sparingly, however. This has all of the drawbacks as the Object Mother pattern so you do need to be careful, but it can be a powerful tool when used in moderation.

After some experimentation, my team has settled on Test Helpers as our go-to pattern. It’s flexible, easy to implement, and easy to extend for integration tests. We’ll come back to the integration test bit in a minute, but first I want to talk about using helpers for more complicated setup.

**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’re testing a product review feature and you have this test which ensures that objectionable reviews get rejected. First it creates a product, then a customer, then creates two flagged reviews by that customer for that product. Finally, it tries to submit a new review & asserts that it gets rejected.

This isn’t a *bad* test, but it could be better. Wiring up these relationships by hand is tedious and sacrifices expressiveness and leads to copy/paste reuse.

(click)

In these situations we use a pattern we call a Scenario. This is essentially a façade that simplifies the usage of multiple Test Helpers towards a common goal. It provides a convenient way of tracking relationships between test data.

This last point is pretty important. If you just have a bunch of disparate objects you need to create then just create them individually. But if there is a common set of *related* data that you often create in a similar way, then a Scenario object is useful.

(click)

Scenarios are also useful for reusing complex arrangements without inheritance. Chances are that if your feature requires complicated setup, you’re going to have multiple tests. And if you have multiple tests for the same feature, you may want to re-use the setup code between tests. Scenarios are specifically designed to support re-use without using inheritance.

So, what does a Scenario object look like?

(click)

The biggest difference between a Scenario and a Test Helper is that a Test Helper is a static factory class and a Scenario is something that you instantiate. Instead of customizing the result via method arguments, you customize it via constructor arguments.

(click)

After you instantiate it, the Scenario exposes its key data as instance properties. In this case, the Scenario contains a Product and three different Reviewers, each in a different state.

By naming these properties well, and by commenting them if necessary, you make it very easy to share complex setup across multiple tests while maintaining expressiveness and keeping your tests flexible and easy to maintain.

(click)

Here’s that example again of the product review test that is NOT using a Scenario. Again, it’s not a terrible test, but look what happens when we re-write it using a Scenario.

(click)

Instead of explicitly creating two flagged reviews, I’m specializing the scenario by passing a single, declarative argument into the constructor.

Also notice how much less noisy this code is without the Customer and Product initializations. In this case there’s nothing about those objects that impacts the test outcome so we can delegate their instantiation to the Scenario and just use its references to them.

To repeat this test with a different number of flagged reviews, I just need to copy/paste a single line of code and change 1 argument. This is an excellent example of how expressive code makes it easy to add new tests to existing code.

(click – drawbacks)

Scenarios do have many of the same drawbacks as Object Mother. This makes sense, because a Scenario is basically a Mother for a *group* of objects rather than a single one.

It can get painful when tests need to start customizing data. There are ways to manage this, but in general if you find yourself needing to heavily customize a Scenario then you’re probably better off creating those objects individually.

Also, it can be hard to change a Scenario object once many tests start using it. The more tests you have that make assumptions about the data in a Scenario, the less flexible it becomes.

(click)

These drawbacks are mitigated somewhat by the fact that Scenarios are rarely reusable in broad terms. By their very nature, Scenarios represent tight coupling between multiple objects. This coupling reduces their ability to be reused in different contexts, which means you don’t tend to reuse them as frequently as you might reuse a single-object Mother.

(click)

I tend to use Scenarios when a group of related tests share complex setup, and when there is a core set of objects that must be created but that don’t generally impact the test outcomes. I rarely use Scenarios in a wider context than that.

**Summarize Object Construction**

OK, let’s take a second and summarize the object creation patterns we just talked about.

(click)

First, avoid constructing test objects by hand. If you need to create a single object, consider the Test Helper pattern (or Object Mother or Data Builder)

(click)

Use Scenarios if you need multiple objects AND they are related to each other. Scenarios are also a good alternative to inheritance for sharing or reusing setup code.

(click)

I recommend that you put these helpers in your Test project, rather than your core application project. If you have multiple test projects, consider creating a “Test Library” project so that you can reuse these helpers across all your tests.

(click)

Lastly, I want to acknowledge that creating these helpers DOES have a cost. Adding them to a large, legacy project can be especially painful. I’ve been there, done that, and it was worth it.

On an existing legacy project I recommend that you start with helpers for your smaller objects first. Start with the simple things and build up to your larger, more composed objects over time. If you start out with the most complex objects then it could turn into a rabbit hole because every of all of the dependent helpers you’ll have to build out too.

On a greenfield project, create these as you go even if it seems like overkill. In the beginning you may not think you get a lot of value, but if your objects are small and simple then it won’t cost you much to do this either. In my experience software rarely stays both small *and* simple over time, and these patterns will pay dividends as your software gets larger and/or more complex.

**Order brought to chaos**

Here’s that nasty chunk of setup code I showed at the start…

(click)

(click)

Here’s that same chunk of code, cleaned up and rewritten to use Test Helpers. I determined that many of the data being created were irrelevant and pushed those into helpers with good defaults. Everything that remains is meaningful to the test.

We’re still creating a lot of objects, but it’s manageable now. I said it before and I’ll say it again, the single most important thing is to build a good helper library like this.

**Advanced techniques – Integration tests**

Until now we’ve been talking about in-memory objects only. Eventually you’re going to want to save your test data to a database so that you can test your data access code, or so that you can automate some of your full-stack system tests. In this final section I’ll show you how to use Test Helpers in integration tests.

(click)

The goal is to be able to use the same object creation patterns to create data in memory as we do to put sample data in the database.

(click)

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

First, you have to deal with foreign keys. You can’t just new up a Customer and an Order in memory and then save them to disk. You have to new up the entire object graph, save objects to the database in the correct sequence, and then update all of the ID properties to reflect any database-assigned key values.

Second, you have to deal with column constraints. Some fields will reject NULL, others have max length constraints that your in-memory sample data might violate.

Lastly, you need to clean up that test data when the test run is over. We don’t want the test database littered with junk records because it can impact performance, because it makes it hard to find specific data while debugging, and because it’s just plain unsightly if you use the same database for manual testing as you do for automated tests.

(click)

These challenges are a pain, but they are manageable with a few extra additions to your Test Helper classes.

The solution we use is to add a *Save()* method to your Test Helpers. Before I show you how that works, I want to mention a few things: first, this feature assumes you’re using an ORM of some sort. If you’re NOT using an ORM then the concept still applies but you’ll need to find a different implementation. Second, these code samples are for NHibernate, but they should work for Entity Framework or whatever else you’re using with a few adjustments.

Let’s take a look at how the Save method is implemented in my helpers. There are four things I want to call your attention to:

(click – Save() method implementation)

First, note that like all of the methods on my helper classes, Save is static. This means I need to pass in the ORM interface as an argument. It’s certainly possible to use Dependency Injection but I’ve never found it to be worth the effort.

(click)

Second, note that the Save method delegates to other helpers to save dependent objects. This is an important pattern because foreign keys often require us to create dependent objects before we can reference them. In this case, we can’t save an Order unless it has a reference to a valid Customers. Delegating Save calls like this helps keep the helpers clean and focused on a single object type.

(click)

Third, the Save method is responsible for resetting any Id values that were assigned by the IdSequencer. This is important because many ORMs use the ID property to determine if it should issue an INSERT or UPDATE query. If we tell the ORM to save an object that has a non-zero ID it may think the object represents a database record that already exists and will therefore issue an update, which of course won’t do anything. Resetting the ID to zero causes the ORM to issue an INSERT statement.

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

(click)

Fourth, I delegate to the ORM to insert or update the record.

Generally speaking, 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.

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 can be a real pain to update the baseline backup every time the schema changes. Another reason is that I tend to use a single database for unit tests and for manual testing. It really sucks to spend a bunch of time crafting data for a manual test and then accidently lose it because you run a test that resets it back to a clean state.

(click)

Another option is to wrap each test run in a database transaction and then roll that transaction back when the rest run is over.

We handle this via a custom attribute that we can add to our integration tests. Any test that has this attribute is automatically executed inside of a transaction that is discarded when the test finishes.

This is a custom NUnit attribute that we wrote. I’ve posted the details to GitHub, but note that I wrote this five years ago. There are probably newer and better ways of doing the same thing.

**A quick recap**

Let’s recap:

Good setup code:

1. Is highly expressive, which makes the tests easier to write, read and understand
2. Calls attention to the data values that impact the test outcome, and downplays the data values that don’t
3. Avoids inheritance as a way of sharing setup code
4. Does not require a lot of upkeep as the software changes

And how do we achieve these goals?

1. Short, clean code
2. Delegate object creation to some sort of helper library

**Parting words of wisdom**

I’d like to wrap up with a few pieces of advice.

(click)

If something is hard to test, or if the test helpers are hard to write, it’s probably too complex. Don’t be clever in your tests or your test helpers; change the design of your code so that the cleverness is unnecessary.

Remember: today’s “clever trick” is tomorrow’s throat punch from a ticked off co-worker!

Basically, clean, simple and elegant tests are the keys to success.

(click)

You can download these materials from my Github account and if you have any questions or feedback of any sort, you can reach me on Twitter or email.

THANK YOU!