Welcome to “UI Tests Are Fun To Write (If You Write Them Right)”.

Let’s start with the obligatory audience participation survey. Quick show of hands: how many of you are currently writing automated UI tests in your projects? Anyone here that *used* to write them, but gave up? And the last question, how many of you *enjoy* writing UI tests at least as much as you enjoy writing other types of tests?

One thing that I think everyone that raised their hand will agree with is that automated UI tests tend to suck. They often *aren’t* fun to write. They are hard to set up. They’re slow to execute. They can’t be run in parallel. They make permanent changes to the database or file system which sometimes prevent the tests from being re-run without resetting to a known clean state.

(click for window blinds)

And if you don’t write them right, they can be extremely brittle. Even a tiny CSS change can break a whole suite of tests. And because these tests take so long to run, triaging and fixing broken tests can be very time consuming.

(click for “why bother”)

For those reasons, and many more, my team resisted writing UI tests for a very long time. We’ve been producing a large and complex SaaS product for about 8 years. We’ve been writing unit and integration tests since the beginning, and we’re up to about 7000 unit and integration tests.

You might think that with thousands of unit and integration tests, that we’d be able to very quickly release new code into production.

But if you thought that two years ago, you’d have been wrong. Unfortunately, no matter how many unit and integration tests we wrote, we kept having experiences like this.

(click for boat)

All too frequently we’d have a bunch of green checkmarks in the test runner, and then a user would go and do something fairly innocuous, and they’d hit some sort of problem in the UI layer that none of those tests was able to detect.

Unit tests tell us things like “*if* the electrical system is working, *then* the lights will turn on when you press this switch”. That’s great to know when testing the switch but doesn’t really tell us that the boat will float when placed in the water.

Integration tests tell us things like “the engine correctly causes the propeller to spin at the desired speed”. Again, that’s really helpful when designing the propeller system, but again it doesn’t tell us whether or not the boat is going to actually float.

Eventually, we accepted the fact that unit and integration tests alone are insufficient to support frequent releases of a large or complex app. UI tests are absolutely necessary for us to maintain a stable velocity over time.

(click for “in the browser”)

And that’s because only UI tests give us the confidence that our code will work correctly when a user hits the site in a browser. Only UI tests give us that reliable confidence that the boat will in fact float.

(click for “too bad”)

Unfortunately, there’s a difference between *knowing the path* and *walking the path*.

Our first attempts at UI testing were painful. We made lots of mistakes, we wrote lots of bad tests, and we spent lots of time struggling to keep them green.

But we kept at it. We kept looking for better patterns and different ways of doing things, and over time our UI tests got better.

And then, late last year, one of the new developers on my team said something that inspired me to write this talk.

(click for quote)

We were talking about our UI testing patterns and he said “when I first started writing UI tests, I hated them. Now, I kind of like them. They’re fun”.

This quote is the reason I’m up here today. It told me that, even though we still struggle with many aspects of our UI tests, and there are lots of imperfect aspects to our approach, we’ve solved enough of the pain points that new programmers joining our team enjoy learning how we do things and no longer look for excuses to *avoid* these types of tests.

(click for “Dangerous to go alone”)

I kind of feel like my team went off into the UI Testing wilderness for a long time, and at first we struggled to survive. We were cold and hungry and every tiny success we had was really hard fought. But over time we figured out how not just to survive but to thrive in that wilderness, and then one day we hired this new developer who was just starting their own journey into start same wilderness. And even though we were dirty and wearing rags and still occasionally had to eat bugs in order to survive, in comparison to that *other* wanderer we realized exactly how far we’d come and all the things we’d learned. And even though we weren’t “wilderness survival experts”, the lessons we’d learned could still help people who were a little less far along on their own journey.

And that’s the point of this talk.

We’ve learned some lessons and have some patterns that work well for us, and I want to share them in the hopes that your early journey through this wilderness is a little less painful than ours was.

(click)

Remember though, that I’ve figured out how to survive in one particular wilderness. I’m not an expert in all the types of wildernesses there are, and that means that if you’re hoping to learn The One True Way of UI Testing, you’re going to be disappointed. I don’t think that silver bullets exist in UI testing, and that if you do this for a long time it’s basically a case study in trade-offs. Every decision that you make has consequences and impacts the choices you have in your next decision.

You might see things today that simply will not work for you. That’s cool. I’m hoping that if I tell you *why* we’ve made the choices we have, that it will help you make the right decisions for your own projects.

(click for agenda)

That’s probably enough stage-setting, so let’s get into some details.

In this session you’ll see:

1. How we decide when to write UI test, and when not to. I’ll talk about ways to refactor your code to *avoid* writing UI tests.
2. How we use the PageObject pattern to write and organize our UI tests
3. How we deal with test data
4. Miscellaneous tips and tricks that you may or may not think of as “fun”, but can certainly make these things less painful

What you will NOT see are introductory slides about how to actually get started with Selenium WebDriver, which is the API that we use to automate the browser. If you don’t already know how to automate the browser via code, that’s fine; you don’t need to know that to get value from this session. I’m going to focus on concepts and patterns and strategies because that’s what I think determines if you’re going to be successful with UI testing or not.

**(click) What needs a UI test, and when are they written?**

The first thing I want to talk about is when you should, and should not, write UI test.

(click to fade in)

And the simple rule is that you should write a UI test *only* when you can’t write a cheaper, faster, easier test that can give you the confidence that you desire.

Or, put another way, the surest way to guarantee that your UI tests *aren’t* fun to write is to use them to test the wrong things.

(click)

My team differentiates between these 4 types of tests, and each type of test involves a different trade-off between the cost to author or maintain that test and the confidence that it gives us about the system.

Unit tests are generally easy to write and cheap to execute, but because they tend to test things in isolation, they don’t give us a lot of confidence in the overall system.

UI tests give us lots of confidence about the system, but they’re hard to write and costly to execute. And there are varying degrees of trade-offs in the middle.

Whenever we set out to write a test, the first thing that we ask ourselves is “what is the lowest test type on this list that could conceivably test the code in question”? And if the answer to that question is “UI test”, then the very *next* question we ask is “how can we refactor the code to change that answer?”

(click for “Refactor UI code…”)

Here’s an example.

A common scenario we encounter is a requirement that some piece of information is hidden or visible based upon some other condition.

My application deals with licensing and credentialing, so the My Account might have a piece code that says that if the current user has an active license then the end date for their license cycle should be displayed. If the current user isn’t licensed, we show some default text instead.

If this rule is important enough that it justifies a test, then the only way to do that is through a UI test, and the UI test would look like this:

(click for UI test)

I’ll talk about some of these patterns in more detail in a little bit, but for now the point is that in order to test that one tiny piece of logic in the view code we’d have to create a new user in the database, save it, spin up the browser, log in as that user, navigate to the My Account page, and then use Selenium to determine if the cycle date is visible.

That’s a lot of setup work to test a tiny piece of logic!

(click for negative case)

We’ll probably also want to test the negative case, in which the dates do NOT show up if the user is not licensed, which means creating a second copy of this test and having a second UI test repeat the whole process.

This is NOT the appropriate way to test logic like that.

(click for refactored view + test)

A better approach is to move that logic out of the view and into a method on the view model, and then write unit tests against the model.

This test is thousands of times faster than the UI test because it executes entirely in memory, and it’s easier to write because you’re not dealing with test data or login credentials or anything else.

(click for TestCase)

Extracting code into unit tests really pays off when you’re testing multiple permutations of something.

In this case, I’ve written a single test that has been parameterized to cover both the scenario when the user IS licensed, and when they’re not. This allows me to reuse the same setup code which makes these tests easier to write and maintain.

To be fair, you *can* write parameterized UI tests as well, but it gets hairy if those tests have to create or modify data because you end up with multiple tests, each trying to make the same permanent changes to the database. Unit tests don’t have any of those constraints.

(click for “Use unit tests for…”)

The general point I’m making is that any time that you can refactor your code so that you can unit-test a view model, rather than UI test your DOM, that’s probably something that you should do. It makes the tests significantly cheaper to write and maintain.

This technique isn’t *always* a good approach. A code-based unit test won’t detect if a certain string is being output, but then hidden with CSS, or removed from the DOM with JavaScript. That’s where your critical thinking comes into play – if the feature in question is on a highly dynamic page with lots of client-side complexity then maybe a UI test makes sense.

But if the feature in question is a simple HTML details page then you’re much better off keeping your views simple and stupid and putting formatting and calculation logic where it can be unit tested instead.

(click for “data/integration tests”)

If we want more confidence in the system than we get from a unit test, the next best thing we can do is get rid of all the mocks and start testing things against a real database and in collaboration with each other.

Many teams will say that any test that hits a database is an “integration test”. On my team, I differentiate between “data tests” and “integration tests” because it helps us be more explicit about coverage.

In our nomenclature, a “data test” is basically a unit test of the data access layer. These are low-level tests that typically execute a single class or component using a real connection string.

Integration tests are higher-level tests that generally execute multiple components at a time.

(click)

There are all kinds of things that you can test for using data or integration tests that don’t require spinning up a browser:

* Database changes
* File system updates
* Email delivery

(click for circled biz logic)

In fact, the vast majority of our tests that validate true business-logic are data or integration tests. We deliberately write that logic so that we’re able to test it without exercising the UI.

(click for UI tests)

That brings us, finally, to UI tests, which are the most costly tests we write, but also the ones that give us the most confidence that things are going to work “for real” when a user shows up with a task to accomplish.

I already said that we use unit tests against a view model to check for calculations and visibility of certain data, and that we use data and integration tests to validate most of our business logic.

So what does that leave for UI tests to focus on?

Well, we try to only write UI tests for things that cannot be tested any other way. And if you think about it, there are *lots* of things that are difficult to test without standing up the full web stack, hitting it with a browser, parsing the HTML, and executing the JavaScript.  
  
(click)   
  
Let’s say you have a simple Edit page. Someone clicks on a link and hits an endpoint and the system returns an HTML form. The person makes changes to the form and clicks submit, and then the form gets POSTed to another endpoint where the form data is processed in some way.  
  
Think about all of the things that could go wrong:

* + Maybe the code that renders the form doesn’t initialize its default values to match the thing being edited, so the form renders but the fields are blank.
  + Maybe the form uses a jQuery plugin for one of the fields, but there’s a syntax error in some JavaScript and it causes the JS engine to abort before the plugin is fully initialized.
  + Or maybe someone changed a CSS class on a field, but didn’t update the JS to match so the plugin doesn’t actually do anything.
  + Or maybe the <form> tag has a typo in the action element and points to an invalid location, or maybe the endpoint it posts to has been renamed but the form wasn’t updated
  + Or maybe the field names on the form don’t match up with the payload the endpoint is expecting, so even though the user’s changes are sent to the server, they end up getting ignored.

(click for “use UI tests for”…)

These sorts of things are impossible to catch with a compiler or code-level test, so that’s what we focus UI tests on.

We rarely, if ever, write UI tests to validate back-end business rules; it’s all about ensuring that the browser can parse the HTML, build the DOM, execute the JS, and respond correctly to the user’s input.

**(click) How are tests structured?**

Once you’ve decided that you have something worth testing via the UI, you need to write the actual test. Every UI test needs to consider 4 things:

1. What baseline data needs to exist before the test can run?
2. What user identity will be used to perform the test?
3. How does the test navigate to the “starting page” or context from which the test begins?
4. Once the browser is on the target page, how do we write good, clean, maintainable test code?

There are interesting things to consider, and problems to solve, hidden within each of those concerns.

I think the best way to talk about these is in reverse, starting with the test code itself.

Like many teams doing UI testing, we organize our tests using the Page Object pattern. The point of this pattern is to create an application-specific API to write your UI tests against.

**(click for empty page object)**

Following this pattern, we create a PageObject class for every page or UI component that we want to test. In this case, I’ll be showing you the PageObject for a Login page.

Now, the point of a PageObject is to create an API that your UI tests can use to interact with that page. The idea is that by writing tests against this API they’ll be cleaner, have less duplication, and be less tightly coupled to any one specific automation framework.

(click – add properties)

The first thing that we do in these PageObjects is create public properties representing the HTML elements on that page.

For instance, this is a Login page so it has a username field, a password field, and a button to submit the form.

How you actually *implement* these properties will depend on the specific browser automation tool that you’re using. We use Selenium WebDriver which gives us

(click – FindBy ID)

this handy FindsBy attribute which allows us to declaratively map each property to the DOM using different types of selectors.

Our most commonly used selector type is an ID selector like you see here, but we also frequently use

(click – FindBy CSS)

CSS class name and CSS Selector strategies as well.

I’m going to talk more about writing good selectors in a little bit, for now the main point is that the PageObject exposes its HTML elements as public properties, and if you’re using WebDriver you can use the FindsBy attribute to easily bind those properties to the DOM. All we have to do is worry about the selector and Selenium WebDriver handles the rest.

(click – app centric)

Doing this means that PageObject will provide an *application-centric* API for interacting with the page, rather than an *HTML-centric* one.

An application-centric API is focused around what the elements *mean* and how they can be used, whereas an HTML-centric API is tightly coupled to how those elements are actually built.

Application-centric APIs are better because they allow a software client, like a test, to do anything that a human being could do, but without requiring that client to know anything about the specific HTML structure in use.

If the HTML structure changes over time, we really don’t want large numbers of tests to break. The application-centric model encapsulates the HTML structure in a single place where it’s easy to maintain.

(click for “methods, not properties”)

When we first started using PageObjects, our test code looked like this. Each test would interact with the PageObject’s properties to modify HTML elements, and then would make the necessary assertions.

Simple, right? We’ve definitely decoupled this tests from the HTML *structure*, but it’s stillcoupled to the specific front-end *implementation*.

(click for 5 sample tests)

Let’s imagine for a moment that I have five different tests I want to run for the login page:

* Invalid credentials results in an error message showing up on the page
* Valid credentials for a “locked” account displays an error message
* Valid credentials for an Admin user causes redirect to the Admin dashboard
* Valid credentials for a Normal User causes redirect to the default dashboard
* If a user that hasn’t changed their password in over 180 days triggers Password Reset process

Actually, before I move on, pop quiz: can anyone tell me why these are really bad example of UI tests?

(click for X-ed out tests)

It’s a bad example because these are business logic tests, not UI tests. Things like “when the account is locked, don’t let user log in” should be a unit or integration test instead.

In a real-world test of a login page I’d probably have a single test that ensures that a failed login results in an error message being displayed, but I’d enumerate all of the different reasons that could cause the failure, and the specific error messages that are appropriate for each one, as lower level tests.

(click to remove the X’s)

Anyways, let’s pretend for a moment that I wasn’t being lazy in my slides and that these ARE good example of UI tests you’d want to right.

Each of these 5 tests needs to do the same 3 things: put a username in the username field, put a password in the password field, and then click submit. If we copy and paste these 3 lines in each of those 5 tests, that’s a lot of duplication.

What happens if someone in the legal department gets paranoid about cookie usage and wants us to put some checkbox on the login page that requires the user to acknowledge our privacy policy before they are even allowed to log in?

Any change to the set of steps that has to happen in order to log in has to be duplicated in all of these different tests.

(click – method)

Over time, we’ve started adding *behavior* to our PageObjects as well.

In the case of the LoginPage object example, we has a “LoginAs” method that takes the username and password and then encapsulates whatever steps need to be performed. Now, any test that wants to log in can do that with one simple method call.

In fact, we’ve started making most of our PageObject properties *private* because it forces us to expose all of the things that a page does as method calls instead. This makes the tests more expressive, more intent-revealing, and easier to maintain.

~~(click for navigation)~~

~~Here’s another useful tip about PageObjects – whenever a PageObject triggers a navigation, you can have the method return an instance of the page object representing the destination page. This lets you write tests that span multiple page requests very easily.~~

~~In this example, the LoginAs method performs a navigation and then returns a~~ *~~HomePage~~* ~~object that represents the default landing page.~~

~~In a little bit I’m going to talk a little more about multi-request tests. I don’t recommend that you write a~~ *~~lot~~* ~~of multi-page tests, but sometimes they’re worth the effort and this is a really elegant way of writing them.~~

(Click for GoToPage)

Another thing that we do on all of our PageObjects is to include a “GoToPage” method, which makes it easy for tests to navigate directly to that specific page or context.

When we first started writing automated tests, we did *everything* the exact same way that a user would. Every single test basically had to log in, navigate through the site menus to get to the page under test, and *then* perform the test itself. This made tests tedious to write and it made them take longer to execute on account of all those intermediate page renders.  
  
It also made the tests extremely brittle. At one point in our history we had something like 50 UI tests for the admin area of our site. Each of those tests was coded to click on the top-level “Administrator” link in the navigation bar. That would navigate to the admin index dashboard, and then each test would branch off from there by clicking on *another* link.  
  
Our app is extremely configurable, and one of the things that you can change is the labels of the navigation buttons. At one point, during a manual regression test, someone changed the text of that “Administrator” link to “Admin”. The UI tests were hooked up to that same database, so all of a sudden all 50 tests started failing, even though the app was still functioning perfectly.  
  
The lesson here is that every single interaction that you automate through the browser is a potential point of failure, so you really only want to automate those things that are completely necessary for your test case.

So as a result of that debacle, we started including the GoTo method on all of our PageObjects so that we can navigate directly to the page in question, rather than automating clicks against the navigation menu.

(click for GoToPage implementation)

The other benefit of the GoToPage method is that it lets us document the required URL parameters for each page.

The login page, for example, has no required parameters but it does support an optional “redirectTo” parameter that overrides the default redirect following a successful login.

The member details page, on the other hand, *requires* that you specify the ID of the member you want to view.

Baking those details into the GoToPage() method ensures that our tests are never navigating to pages without the required data. And if we happen to *change* the set of required or optional parameters for a page, static analysis tools and the compiler will very quickly help us determine which UI tests need updated accordingly.

(click for “not just for pages”)

I want to wrap up this section on PageObjects with one last comment: Even though this is called the “PageObject” pattern, it doesn’t *have* to be used at the page level. You can create a PageObject for **any significant element** in your UI.

For example, we have this concept in our app called a Comparison Rule and we use them on many different features. The comparison rule editor is implemented as a modal popup that is displayed, collects some complex information from the user, and then saves that data as JSON to a hidden form field.

Even though that modal popup isn’t a “page”, we still encapsulate the functionality exposed by that modal into its own PageObject so that it’s easy to reuse. In this example, we call a method on the main PageObject and that method **returns an instance of the comparison rule editor PageObject**. The test can then interact with that modal to do whatever it needs.

You can apply this pattern for a single page application as well. In a single page app you still have separate contexts or views that the page can be in, so you could create a PageObject for each of those contexts.

(click for “setting up a UI test”)

A few slides ago I stated that every UI test needs to consider these 4 things.

The PageObjects that we create handle or facilitate three of the four:

* They provide the API that the test code uses to automate the browser and make the assertions
* They handle the navigation to the page in question using the *GoToPage()* pattern I showed you
* Reusing the *LoginPage* object gives us an easy way to specify which user context each test should use

~~(click for sample test)~~

~~If we put those things together it looks like this.~~

~~First we instantiate the PageObjects, which is easy to do. We use a static factory initializer because there’s a little bit of boilerplate setup we do in a generic way, but that’s not strictly necessary.~~

~~Once we have the PageObjects we use the~~ *~~LoginPage~~* ~~object to log in and establish a session with the site. This is important – we want these tests to be as independent and isolated as possible, so~~ **~~every UI test does a fresh login~~**~~. I’ll come back and talk about this in a little bit.~~

~~As a little bit of syntactic sugar, the~~ *~~LoginPage~~* ~~object exposes different versions of the Login method so that we can specify which type of user to log in as without having to embed usernames and passwords in every single test.~~

~~Once we’re logged in, we use the~~ *~~GoToPage()~~* ~~method to navigate to the page in question, and then we consume its page-specific API to do whatever needs done.~~

~~And in many cases, we’ll wrap the assertions up into the PageObject as well because It’s not uncommon that we reuse similar assertions in multiple contexts.~~

(click for 4 steps again)

That brings us to the very first thing that you’ll need to think about when writing a UI test, which is what data needs to exist in the system in order for your test to execute, and how does that data get there?

**(click) comic - Setting up test data**

The answer is that it’s complicated. And it’s even *more* complicated if you have a complicated data model to begin with.

As an example, the main product that my team develops is in the licensing and credentialing space. One of the things that we can do in our product is configure the workflow that an applicant goes through become a licensed practitioner in a field such as nursing or real estate.

As you might imagine, these workflows can be very complex. Applicants need to fill out an application form that collects lots of different data, they need to pay different types of fees, other people might need to conduct background checks or review the applications in different ways, and all of this stuff needs to be configurable. All that configurability means that we have a large and relatively complex data model. But because those data entry forms are so critical to our system, we decided we needed to write UI tests for them.

As we started writing more and more UI tests for those features we very quickly realized that test data management would be a crucial to keeping our costs under control.

(click for code)

As an example, pretend that we want to write a UI test for one of the data entry forms that my system can display to an applicant.

Obviously, one of those things has to exist in the database before we can write the test, so the first decision that we have to make is whether our test should render a pre-existing form that’s already been configured in the database, or whether that test should create a brand new data entry form as part of the test itself.

(click – pre-existing)

Let’s say that developers on our team have a local database that’s built up from a standard baseline that is checked into source control and already has sample data in it.

If that database already has a data entry form created, then we can just create a constant that references the ID of that pre-existing record, and then the test is super easy to write. The data already exists so there’s no real setup to do, and we can jump right into the test itself.

(click for constant #1)

In the beginning this will be easy to do because the amount of pre-staged data will be very small.

Unfortunately, things rarely stay simple over time. Eventually we’re going to write more tests, and some of those tests are going to need different things.

(click for constant #2)

Maybe there’s one test that deals specifically with fees and needs to have one specific configuration,

(click for constant #3)

and maybe there’s another test that deals with a very specific UI layout,

(click for constant #4)

And yet another test that covers a specific workflow process.

The programmers writing those tests will do the logical thing and they’ll create new data entry forms in the baseline database, they’ll push the updated snapshot or test data script into source control, and they’ll add new IDs to our list of constants.

(click for constant #5)

If your application is complex enough, and the number of possible permutations of this data is large, then it won’t take long before this gets out of hand.

Let’s say we have a long list of IDs like this and a new developer joins the team and needs to write a test. This new developer has a very slim chance of recognizing if the data she needs is already represented here, so we’ll start to see more and more duplicates within our pre-built data.

Now, I made up these specific scenarios for this example, but the need to highly specify the data context for a test is quite common in a complex system, and the more “pre-existing special cases” that you create, the harder it’s going to be to maintain that data.

(click for load-edit-modify)

Another approach would be to maintain a smaller set of pre-existing special cases, and then have each test modify one of those pre-existing things to match the test-specific needs.

For example, we could write a bit of SQL that takes an existing record with *some* of the characteristics we need, modify the other properties as needed, and then continue with the test.

This does reduce the number of permutations of pre-built data, but it’s a really bad idea in practice. Each test is now making permanent changes to shared global data, and that’s bad. You might get different results if you run the tests in different sequences, and it’s really hard to manage these tests because you never really know exactly what state the test data will be in when the test runs.

(click for “each test should”)

For those reasons, I am a diehard believer that each test should create its own test data. It should make the fewest possible assumptions about the state of the database, and it should be precise and explicit about which properties of that data are relevant to the test.

(click for scrawl)

My team would tell you that I will absolutely die on this hill. I think that this is crucial to managing a large and complex test suite over time.

(click for “create” test)

Let’s go back to that pretend test we’re trying to write, and instead of using pre-existing data, let’s have the test create the data that it needs on the fly. Here’s an example of what that might look like:

* We’ll need to create the data entry form,
* we’ll need to customize it to match the requirements of the test
* and then we’ll need to save it to the database so that the UI test can access it

This is great because each test is now totally independent of the others, but there’s a problem here.

(click for difficult ctor)

If I were to make this sample code a little more realistic, then creating a data entry form would require this thing called a Credential as well as the Member that owns it.

(click for difficult ctor #2)

But it turns out that a Credential requires this thing called an Interval and a thing called a Board, and a Member requires a few Address records, and each of those things may have their own dependencies, and by the time that we’ve satisfied all of the constructors we’ve had to specify a ton of data that is totally irrelevant to the outcome of our test.   
  
(click for noise)  
  
These values that we had to specify are noise; they make the test harder to write and harder to read by obscuring the data that *do* matter to the test outcome. Those noise values also make our tests brittle because if any of these constructors get changed in the future, this test is going to need updated as well, even if those changes have no logical bearing on the test outcome.

(click for difficult ctor #3)

And of course, all of this code here is only for creating the in-memory object graph; we still need to push this thing into the database, which means we might need to deal with foreign key constraints and make sure that things are created in the correct sequence.

(click for FooHelper)

To address these issues, we’ve created a library of data creation helpers that make it much simpler to construct test data. This is what they look like:

1. For every entity Foo in our system there is a class called FooHelper that exposes a Create method.
2. The create method exposes every piece of data that can be customized as an optional argument. When tests call the Create method, they only specify values for the properties that actually matter to the test outcome.
3. The helpers provide reasonable defaults for any property that wasn’t explicitly specified in the test and then return a fully-constructed object, with all of its dependencies in a valid state.  
     
   (click for 2nd highlight)
4. Data, integration,and UI tests call a second helper method, Save(), to persist those in-memory objects to the database. This Save() method is what ensures that things are saved in the correct sequence to satisfy all of the foreign keys, and that all of the in-memory objects get updated to reflect any primary key values that are assigned by the database.

We call these objects “Test Helpers” and the really nice thing about this approach is that we use the exact same technique to create test data in *all* of our tests. We can create in-memory objects with one line of code and persist them with another. And when we add new things to our data model, we don’t need to update tons of tests; we just update the helper, provide a default value for the new thing, and that’s it.

This isn’t the *only* way to create test data. There are libraries and frameworks that will give you a generic API for building test objects, and you might be able to use your existing ORM to save those things to the database. For us, rolling our own helpers has allowed us to better deal with the complexity of our data model because we can more easily provide default values that represent real-world scenarios and we can tailor the API to suit our needs.

(click: w/ Test Helpers)

I could go on and on about this stuff, and I actually have a 60 minute talk on just this data helper pattern that we use. For today though, the key points are:

1. Don’t new up test data by hand; wrap the test data creation in some API
2. When you create test data, only specify those values that actually impact the test. Let your data creation helpers provide default values for everything else.

This image here is from a blog post by Steven Hicks and he uses the concept of Checkov’s Gun to describe this last point.

Checkov’s Gun is a principle in dramatic writing that says that all elements in a story must be relevant in some way. One of Checkov’s quotes boils down to “if in the first act you have hung a pistol on the wall, then in the following act it should be fired”.

In terms of testing, if you specify a value when creating test data, that value should impact the test. If it does not. This is really, really important and helps keep your tests short, clean, and readable.

(click for caching)

There’s one other problem that you might run into when creating data in the body of each UI test, and that’s caching. If the website is caching data, and your tests are making changes to that data behind the scenes, you’re going to need some way to refresh that web cache before your test starts making web requests.

We’ve addressed this by creating an endpoint in our app that will refresh the cache, and we’ve added a “refreshCache” method to our base test class. Any test that creates data of a cached type can call this method to issue an AJAX request to that endpoint to synchronize everything.

(click for comparison)

Here are your two options when it comes to test data.

Using existing data can make individual tests easy to write, as long as the data you want to use already exists in your test database, but it can make tests harder to maintain over time. Adding new test data to the baseline database can be a pain, and it can be really difficult to manage the pre-existing data at scale. Plus, if your tests need to *modify* or delete data in any way, then you run the risk that one test changes data that could impact another test.

Writing each test to create its own test data is *usually* the better approach. It makes the tests easier to read and understand and avoids all the headaches of global data. It can make tests harder to write if you have a large or complex object graph, but a good library of data creation helpers can mitigate that. Also, every time you run the test suite you’ll end up with a ream of newly created junk data in the database, so you’ll want to write some teardown methods or create some scripts that can easily restore your database to a known clean state on demand.

We’ve been doing it this for years and it’s one of the best decisions we made on this project. The Test Helpers and automation scripts weren’t free and take some effort to maintain, but they are absolutely worth it. I don’t think any specific pattern that we follow contributes more to our success than this.

(click: Antipatterns (don’t try this at home)

Using pre-existing data is one anti-pattern that will make your UI tests not fun to write. Another anti-pattern is using the side effects of one test as the starting point for another.

For instance, say your app lets users create and edit Widgets. It might seem like a really good idea to write the tests like this:

(click)

1. First, write a test that navigates to the New Widget form and submits it, to create a new Widget.
2. Next, write a test that navigates to the Edit screen for the widget that you just created, and validates that the edit screen works.
3. Finally, write a 3rd test that deletes that Widget

And truthfully, if those were the only 3 tests that you wrote for that feature, it’s a decent and pragmatic approach. This nice, clean little package is self-contained, covers a logical progression within the application, it doesn’t depend on pre-existing data, and it cleans up after itself at the end.

The problems creep in when you start adding additional tests and that nice, clean little package gets bloated.

(click)

Presumably, Widgets have some purpose in the system beyond the CRUD screens, so next you’ll probably want a test that goes to a completely different part of the system and uses that new Widget in some way. Why not just insert that test between the Edit and Delete cases, since it’s still just building on top of what already exists?

(click)

TODO – break slide into pieces

And then later you add a new property to Widgets called “Widget Type”, and this property has some major impact on the UI. So then you modify the Create test so that it creates a Widget of Type 1. After the test that uses the Type 1 widget you insert a new test that *modifies* the Widget and changes it to Type 2, and then uses the Type 2 widget in the app. And then you still end with the Delete.

In terms of overall test coverage this is pretty good, but it’s no longer a nice, neat little package. And if we keep following this pattern as we add more and more features it very quickly becomes a brittle, hard to maintain ball of mud.

(click for issue 1)

Tests written like this are hard to understand because the only way to understand the starting point for one test is to understand the cumulative effect of all the tests that came before it.

(click for issue 2)

If any of these tests fails for any reason, every test that runs after it could fail as well. This sort of coupling between tests makes the suite as a whole much harder to maintain.

(click for issue 3)

This approach also prevents you from running your tests individually. UI tests are slow and you *will* find yourself wanting to run a smaller subset of them in order to shorten the feedback cycle. But if you’ve written your tests like this, then your only real option is to run them as a group.

My advice is that you should never chain feature level tests together. If you want to do an end-to-end scenario test that covers a multi-page use case, that’s great! But do it as a single, self-contained test, and use it only to validate that end to scenario; you should still go ahead and create individual, self-contained feature tests that cover all the different steps along the way.

(click for tips & tricks)

At this point I’ve shown you how we use the PageObject pattern to organize our tests and how we use the Test Helper pattern to manage test data.

For the last part of this session I have a few other miscellaneous tips and tricks that have really made a big difference for us.

The first has to do with how you bind your PageObject properties to the DOM.

When we first started writing UI tests, people tried to write the test code without making any changes to the UI code. When the HTML was complex, and the test needed to be very specific about what is was referencing, we ended up with PageObject references like this.

(click for ugly CSS)

This is looking for a link that is the direct child of the first span that is the direct child of a td that is the direct child of a table row with a specific class.

Hopefully you can already see why this would be brittle, because even a tiny change to the HTML structure, such as changing this span to a div, could break the test.

To avoid things like this, we encourage developers to modify the UI code to make it easier to test.

(click for refactored)

One way of doing that is to create specific IDs, classes, and data attributes that are used ONLY for UI tests.

When we do this, include a “selenium” prefix which indicates that these IDs, classes, and data elements are not to be used for styling or any functional javascript. This prefix means these things are ONLY to be used by the tests.

We encourage complete separation between the things that you use for styling, the things that you use for functionality, and the things that you use for testing because it makes it far less likely that a designer is going to inadvertently break your tests when they update the styles, or that another programmer will break your tests when they refactor the client-side javascript.

Every once in awhile you may run into some scenario that makes it hard to use test-specific markers. In that case, if you can’t bind your PageObjects directly to a selenium-specific marker, you should pick a locator that is going to be the most resilient to change.

(click for tree of life)

To do this, you can use this concept called the Locator Tree of Life.

The locator tree of life is basically a way of thinking about locators that encourages you to write more maintainable code. The idea is that fruit at the bottom of the tree is tasty and healthy, but as you move higher and higher the fruit becomes bitter and you should look for alternatives or eat it very sparingly.

The locators at the bottom, such as ID, classname, and data attribute locators, tend to be easier to use and more resilient to change.

(click for code sample)

For example, let’s say I want to make some changes to a piece of code using an HTML ID. If I do a code search for that ID, I will quite easily locate any tests or other code that reference it, and I can then determine if those other files will need updated as a result of my change.

(click for sample #2)

Things get more difficult when we’re working with locators higher up the tree. Here’s another piece of HTML and a path-based locator. Nothing about this HTML fragment suggests that adding a <div> tag to the table cell is going to break anything, and there’s no text search that I can run that’s going to help me figure it out.

The lack of an obvious, explicit link between the HTML structure and the assumptions made in the tests is what makes this sort of locator extremely brittle.

(click for X’d out locators)

As a general rule, we avoid locators based on paths, element text, index or position within a group, and all of the other complex stuff that CSS makes possible.

Combining these low-hanging fruit with test-specific names or values gives you tests that are very self-documenting, very intent-revealing, and very resilient in the face of changes to the HTML.

Test harnesses

The next technique I want to talk about is the concept of a test harness for your UI code.

Because my system is so dynamic and configurable, it’s not always easy to get to a page or context where a given feature is used.

For instance, we support lots of different types of data input fields on our data entry forms, and some of them are pretty complex. But the only way to execute that code in the browser is to create an application process, add a data entry form to it, add a data field of the necessary type, and then log in and begin that form as a user. That’s a lot of work and the vast majority of it has nothing to do with the test itself, it’s only necessary so that we can navigate to a page where the code we want to test is actually executed.

To make this easier, we create test harnesses for all of our UI components.

The first thing we do, of course, is to “componentize” our features in the first place. This is a really important design concept irrespective of UI testing and provides lots of other benefits, and it also makes things easier to test.

The idea is that whenever you have a nontrivial piece of UI, you should design or package it as a “component” or helper so that it can be more easily re-used. Even if the UI is heavily coupled to the functionality of a single page, and you don’t think you’d ever re-use it on a different page, wrapping that UI into a component makes it easier to reason about and makes it easier to test.

(click)

It really doesn’t matter what stack you’re using, there’s a way to encapsulate your UI stuff into reusable pieces.

* In ASPNET MVC you can wrap an HTML Helper around it (example)
* You can turn it into a jQuery plugin
* Most front-end frameworks and libraries like React, VUE.js and Knockout support components as a 1st class design concept

(click for test harnesses)

The second thing we do is create a page for each component that invokes it in isolation from the business logic.

This is a screenshot of an actual test harness we created recently. We added a 3rd party address verification service to the system and we created a component that wraps that 3rd party library and adapts it to our needs. We then created this test harness which contains a section on the left for tinkering with the configuration, and then a section on the right that renders the component using that configuration.

This is super helpful for manual testing, but it’s also really easy to automate because we don’t have to deal with any test data, or navigating through any complex set of steps just to get to a page that contains this component. We can navigate directly to the test harness, set the configuration we want to test, and then test it.

These test harnesses also serve as technical documentation for developers which has also proven to be quite useful.

(click for strategy)

Obviously, just like with unit tests, testing a component in isolation doesn’t actually tell us that it will work *for real*, so our practice is to write at least 1 test that covers the component in some sort of in-app happy path scenario. The purpose of this test isn’t to confirm any specific piece of functionality, but just to make sure that everything is wired up together.

Once we have that, all other tests are against the harness.

**(click for takeaway #1)**

To wrap up, I want to summarize the main things I want you to remember when you leave here.

First, you should create a deliberate test strategy for each feature, and you should endeavor to validate business rules and logic with lower-level tests. Write UI tests only to plug specific gaps in the strategy. And if you find yourself testing what feels like “business logic” via the UI, look for ways to refactor the code so that you can unit test that logic instead.

Make this strategy a first class part of your process. This is something you should be talking about regularly with your team.

(click for #2)

Second, use the PageObject pattern to create an application-centric API for your tests to consume.

By encapsulating the DOM locators and page functionality within these objects you can isolate your test logic from the HTML implementation and make your tests much easier to maintain.

(click for #3)

Lastly, each test should create its own test data, rather than relying on pre-existing data.

This might seem a little painful at first, but it gets a lot easier if you create a library of data creation helpers that you can use for ALL of your tests, unit, integration, and UI.

And when you use those helpers to write your tests, make sure that you only have to explicitly specify the values that actually impact the test outcome. Your helpers should create logical defaults for everything else.

(click for final slide)

And that’s it! Here are those 3 takeaways for your screen shotting pleasure, plus the best ways to get ahold of me if you have questions, comments, stock tips, etc.

**Thank you for your time, and have fun testing!**