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, but every time we wanted to do a release we’d call a code freeze and wait for the QA team to do a full, manual regression test through the UI.

After 8 years and 7000 unit and integration tests, you might think that we’d have a pretty solid release process, right? Yet it was far too frequent an occurrence that all 7000 tests would be green and a QA person would click through the happy path of some feature and find a defect that would have been embarrassing at best if a customer had found it, and a complete show-stopper at worst. In many cases we hadn’t changed the feature in question, but it had become broken as an unintended consequence of a seemingly unrelated change elsewhere in the system.

Eventually, we were forced to accept 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 UI tests give us something that no other type of test can. They give us confidence that all of our code will work correctly when a user hits the site in a browser, which is what really matters.

Unit tests tell us that our classes and methods return the correct values when called in isolation using mocks.

Integration tests tell us that those classes and methods will correctly modify the database or filesystem when called from the test runner.

But only UI tests go the final mile and confirm that the browser can parse the HTML, build the DOM, and execute the JavaScript so that the user can accomplish the task they came to do.

(click for “too bad they’re so painful to write!”)

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 started to suck less and less.

And then, late last year, one of my developers 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, we’ve solved enough of the pain points that our programmers no longer look for excuses to *avoid* writing them.

I kind of feel like we were wandering in the wilderness for a long time, and at first we struggled to survive. We were cold and hungry and without hope. But over time we figured out how not just to survive but to actually thrive, and then at some point we bumped into someone *else* wandering in the wilderness. And even though we were dirty and wearing rags and still occasionally struggled to find food, in comparison to that *other* wanderer we realized exactly how far we’d come. And rather than watch that poor soul struggle the same way that we did, we decided to share our experience in an attempt to elevate the state of the art for “wilderness wanderers” everywhere.

(click for “Dangerous to go alone”)

And that’s the point of this talk.

We’ve learned some lessons about what works and what doesn’t and we’ve identified some things that work well for us. My goal today is to share these lessons with you in the hope that you can avoid some of the pain that we experienced in our journey.

(click for agenda)

In this session you’ll see:

1. How we decide when to write UI test, and when some other approach is better
2. How we use the PageObject pattern to write our UI tests
3. How we create and manage test data
4. How we design UIs and components to be UI-testable
5. Miscellaneous tips and tricks for handling common problems

What you will NOT see are introductory slides about getting started with WebDriver or Selenium. 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 it doesn’t matter how much of a Selenium master you are; if you don’t approach UI testing with the right strategy, it’s going to hurt and you’re really going to struggle when things start to get complex.

(click)

I do need to manage expectations here. If you’re hoping to learn the Secret Of The One True Way of UI Testing, you’re going to be disappointed. There *is* no one true way of doing this.

There are lots of things that my team does that may not work for you. That’s cool. I’m not here to show you the *only* way to approach UI testing, I’m just here to show you how *we* do it, and *why* we do it that. Even if you decide to do things differently, you can at least learn from our mistakes as you set out to discover your own patterns.

**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 a UI test.

(click to fade in)

And the simple rule is that you should write a UI test *only* when a cheaper, faster, easier test can’t sufficiently prove that the software is working as needed.

Or, put another way, the surest way to guarantee that your UI tests *aren’t* fun to write is when you’re using 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 cost 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.

Our strategy is to cover as much as we can with the cheaper, low-level tests and then write higher level tests that close specific gaps in coverage.

What does that mean? Let’s take it level by level.

(click for unit)

We use unit tests for anything that is easy to isolate into a specific method or group of classes.

For obvious reasons it’s silly to spin up an entire browser and click through your site just to validate a computation, when it’s much easier to do that as a unit test.

Likewise, if the code you want to test lives in a single component with minimal interactions with other parts of the app, it’s probably unnecessary to use the browser.

(graphic: unit, no integration tests)  
  
The problem is that unit tests only tell us that we have tiny bits of code that work correctly in isolation. That’s it; they don’t give us any confidence whatsoever that those individual units will work correctly as an integrated system. These tests could be green even if the database doesn’t exist yet!

One way to increase the level of confidence we get from the test suite is to hit an actual database. Some teams call *any* test that hits a database an “integration” test but I think it’s helpful to differentiate between “data tests” and “integration tests”.

(click for Data tests)

In our nomenclature, a “data test” is basically a unit test of the data access layer itself. These tests target very small units of code, they just happen to involve a real database. We still don’t know if the application as a whole will correctly integrate those tiny units, but at least know the queries they would generate are correct, and that helps move the confidence needle a bit.  
  
On the other hand, they cost more as well because we have to set up real data to run them, and because hitting a real database makes them slower to execute.  
  
(click)  
  
Data tests are ideal for testing stuff that happens in the database, and that’s about it. We use these to validate calculations that can’t be unit tested, and if we’re doing TDD on a data access component.

(click for integration)

To get even more confidence in the system, however, we have to start testing those units in concert with each other and not in isolation. That’s where integration tests come in.

In practical terms, data tests and integration tests are very similar; both call some piece of code directly in order to test it, and both use a real database without any mock objects.

The main difference is that a data test typically calls some low-levelmethodin the data access layer that usually does not call into any additional components. An integration test typically calls some higher-level method in the application code that *does* have other dependencies.

(click for integration test graph)  
By calling into code at a higher level in the stack, integration tests more closely simulate what actually happens in production, and green integration tests give us even more confidence in the system.

However, integration tests are often more costly to write because they require additional setup work than unit or data tests. For example, when you test a data access method you just need to worry about setting up the database. But if you write an integration test against a controller action method, you might need to set up the database *and* create a user object to be the logged in user *and* some session state object.  
  
(click)  
  
We primarily use integration tests when there are business rules that involve multiple components, and when we don’t really care about the UI. For example, we have tons of code that handles form posts and does all kinds of data validation and business processing. That code can be validated by examining the changes that are made, or not made, to the database, so we can achieve the desired level of confidence without involving the UI.

We *could* and *do* write unit tests for much of this logic, but we’ve found that with a complex system there are tons of things that can go wrong at runtime that a unit test with mocks will never identify. So even when components are unit tested, we layer in some integration tests to give us the extra degree of confidence.  
  
(click for UI tests)

UI tests finish out the trend line. These are the most costly tests we write because they require the most effort to set up and are the slowest to execute, but they also give us the most confidence in the system.  
  
We focus our UI tests 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 for testing pyramid)  
  
You may have seen the Testing Pyramid before, which tells you to create a lot of unit tests, fewer integration tests, and even fewer UI tests.

I don’t like thinking of it as a pyramid though because the pyramid suggests that there is a “proper” proportion of one type of test to another, and that type of thinking hasn’t been useful to us.

(click for jigsaw)

Instead, I like to think of tests as intentionally designed jigsaw pieces that work together as a whole, with limited overlap, and where each type of test fills in the specific gaps created by other types of tests.

The actual proportion of each type of test could change from system to system and feature to feature, and as long as you’re using each type of test for the right type of thing, there’s no right or wrong number of UI tests.

**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? [TODO]
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?

(click for arrows)

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.

**(click for PageObject)**

Like many teams doing UI testing, we organize our tests using the Page Object pattern. This pattern has us create a PageObject class for every page that we want to test, and the job of that PageObject is to provide an API that allows the test code to interact with that page.

This is an example of a PageObject for a Login page.

(click – properties highlighted)

The first thing that a PageObject does is expose public properties representing the HTML elements on that page. For instance, a Login form would obviously contain fields for entering a username and password and a button to submit the form.

(click – FindBy)

We use Selenium WebDriver which gives us this handy FindsBy attribute which allows us to declaratively map each element to the DOM. There are many different lookup strategies you can use, although the vast majority of our use cases are handled with ID and class names. We’ll talk more about that later.

(click – method)

PageObjects also provide public methods that represent the “services” that a page offers. Since the whole point of a login form is to allow someone to log in, the Login page object might contain a “LoginAs” method that accepts the username or email address and password and then automates the tasks of assigning those values to the HTML elements and submitting the form.

(click – app centric)

The key point here is that a good PageObject gives you an *application-centric* API for writing tests, rather than an *HTML-centric* one. They should 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.

This not only makes the tests easier to read and understand, but it promotes reuse and reduces duplication. If the HTML structure changes over time, you only need to update the affected PageObjects and not every single test that works with those elements.

(click for “not just for pages”

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.

PageObject is a great high-level pattern to follow, but out of the box it doesn’t necessarily encourage you to write super clean tests. We’ve come up with a couple of micro-patterns that we layer on top of PageObject that really help.

(click for base object)

The first is that we create a base class that all other PageObjects derive from, and this base class is what maintains the reference to the WebDriver object itself. This is the object that actually drives the browser.

Tracking this in the base class makes it easier to replace common boilerplate code that deals with WebDriver.

(click for GetInstance)

Here’s an example. The base class provides this static factory method called GetInstance. We pass in the type of PageObject to create and it instantiates that type, sets the WebDriver instance, and then calls this InitElements() method that is provided by Selenium. InitElements() is the key thing here – this is what actually populates all of those public properties with references to the underlying HTML elements.

A lot of WebDriver sample code shows tests calling this InitElements() method explicitly, which we think is ugly and unnecessary. Pushing that stuff into a base class helps us keep each test lean and mean.

(click for example of GetInstance)

This is what it looks like from the calling side. We generally will construct the PageObjects in our setup method, and that lets us call them in the body of the test without needing to pass that WebDriver instance around.

(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. We have a lot of menu items so 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.  
  
Just like with unit tests, we want failures to be meaningful and actionable. Having 50 tests fail because of some totally unrelated change elsewhere in the system is the exact opposite of a “meaningful and actionable” failure.  
  
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 “at least one menu test”)

It’s still extremely important that at least ONE test navigates through the full UI to ensure that all of the navigation elements are functional and go to the right place, but you only need ONE TEST to do that. All of the other tests for that feature should just go straight to the page in question and start doing their work.

(click for GoToPage implementation)

Our general implementation strategy looks like this. The base class defines this abstract “Relative Url” property that all PageObjects must implement.

Each PageObject then includes a GoToPage() method that utilizes that relative URL to do the navigation. Simple.

(click for “why not in base class”?)

You might be wondering why the GoTo method is implemented in the PageObject and not the base class, since the only thing it depends on is that relative URL which is also defined in the base class.

(click for GoTo w/ args)

The answer is that some pages have required URL parameters arguments and we don’t want to expose a parameter-less GoTo method if it won’t result in a valid navigation.

By defining the GoTo method within each PageObject, we have the freedom to fully tailor its signature to that specific page.

For instance, this page requires some sort of ID to be passed in, and optionally accepts a keyword filter. By using optional arguments here we can very clearly indicate that in order to navigate to this page you *must* pass in an ID, and you *could* pass in a filter. Any attempt to call this method without the ID will result in a compilation error. And if in the future we add a *new* required parameter, the compiler will very quickly help us identify the tests that would need updated as a result.

(click for comic)

**Setting up test data**

Once we have a PageObject written and it’s easy to get to the page in question, it’s time to use it to write an actual test. This brings up a whole set of new challenges because we have to start dealing with test data.

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 application process that a person has to go through to become a licensed practitioner in a field such as nursing or real estate.

As you might imagine, these application processes can be very complex. Applicants need to fill out lots of different forms that collect lots of different data, they need to pay fees, other people might need to conduct background checks or review the applications in different ways, and all of this stuff needs to be configured in our system. And because those data entry forms are so critical to the application process, 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 one of those things where a “best practice” just doesn’t exist, and that every decision we would make would be flawed in one way or another. This is a big part of why I say that UI testing is “trade-offs all the way down”.

(click for 2nd comic slide)

Since there is no “best practice” that *always* works, I can’t tell you exactly what you *should* do in your own tests. Instead, I want to lay out the different decisions you have to make, the trade-offs that each decision involves, and our advice for negotiating those trade-offs.

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. 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 form as part of the test itself.

(click – pre-existing)

Let’s say we decide to render a pre-existing form. That will make our test very easy to write because there’s little to no setup cost because the data already exists, but how does the developer remember which pre-existing data to reference?

(click for test + index)

To make our jobs easier, we’ll want some sort of list or index of all of the pre-staged data entry forms that are in the test database so that programmers can easily locate a suitable record from that index. And in the beginning, when the system doesn’t have many features, the list of pre-staged data will be small.

In this example there’s a list of constants, each representing the ID of an existing record with some defining characteristic. If I want a form that collects a fee then I can load this one, if I want a form that has a multi-tabbed UI then I load this one, and if I want a form that requires manual review or audit then I can load this one. Simple enough.

(click for expanded index)

But things rarely stay that simple. Over time, that list is going to grow. Now we have a form that collects a fee on a single-tabbed UI, one that collects a fee on a multi-tabbed UI, one that collects a fee and requires an audit, one where the user can upload a single file, and one where the user can upload multiple files.

I made all these things up 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 will become to manage and pick between them.

And what happens when we have a test that needs to collect a fee, AND accept a file upload, AND utilize a multi-tabbed UI? None of the pre-existing data provides that exact setup.

We could go into the test database and create a new data entry form matching our requirements and add it to this list. But that effort has a cost to it. Someone has to load up the current test database, create the new data, and then update the snapshot or backup file in source control so that it’s available to other team members. That’s painful and time consuming, and the whole point of using pre-existing data was to *avoid* costly data setup.

(click for load-edit-modify)

We could also start with one of the existing records, but modify it to fit our 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 helps reduce the number of permutations of pre-existing data but it’s a really bad idea in practice because now you have tests that are essentially making permanent changes to shared global data, and that’s bad. This makes the tests hard to understand, because you never really know exactly what state the test data will be in when the test runs, and harder to troubleshoot. You might have a test that fails when you run the whole suite, but then when you run that test in isolation it works fine.

(click for “last resort”)

In general, I am not a fan of this approach. I think it seems easy and simple at first, but it comes with lots of hidden costs and complexity. At the same time, though, there *are* some scenarios in which the trade-offs may be worth it, so you should include this approach in your toolbox even if you don’t use it often.

(click for rules)

I have established 4 rules on my team for using pre-existing data.

1. Rule one is that we can use pre-existing data for UI tests ONLY. All of our data and integration tests create their own data, and I’ll talk about that in a minute.
2. Rule two is that UI tests can only rely on pre-existing data if it is legitimately difficult to create on a per-test basis. Our default posture is to create data on the fly, and we enforce that through team standards and code reviews.
3. Rule three is that UI tests should rely on the least amount of loosely-defined data as is necessary to avoid the difficult part of creating data on the fly. For instance, going back to those data entry forms in my system, creating the base form takes a lot of stuff, but configuring the specific fields on a form is fairly simple. So rather than have a large number of highly specialized pre-existing forms in the system, we just have one or two that are designed to be augmented by individual tests.   
     
   For example, if there’s a test that needs an arbitrary data entry form to hold a couple of highly-specified input fields, that test would start with that loosely defined form and would then add fields to it in the test setup. The idea is to stage the *least amount of data possible* to avoid the most costly part of data setup, and push as much of the data specialization into the individual tests.
4. Earlier I said that the “load existing data and then modify” approach is a bad idea. That’s why the final rule is that UI tests that use pre-existing data cannot modify the shared data in any way that would break other tests. It’s fine if a test loads up that generic data form and adds some fields to it, as long as that doesn’t prevent *other* tests from doing the same thing. But it’s NOT OK if a test deletes that pre-existing data form because that will obviously break other tests.

(click for “if you *must*”)  
  
I want to be clear: using pre-existing data sucks. It’s a bad idea. You’re depending or modifying global data in each test and that’s an approach full of pitfalls and gotchas. If you do it, you should feel a little bit dirty.

However, “trade offs all the way down”. There *are* cases where we have deliberately chosen those trade-offs versus other ones. And in those cases, aggressively following these rules has definitely helped us manage those trade-offs.

(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.

The primary benefits of this approach are that (1) each test is totally independent of the others, and (2) tests are easier to read and maintain because all of the test context is described in the body of the test; there’s no global state that we’re making assumptions about.

The drawbacks here are equally significant. One issue is that every test that creates its own test data will leave behind permanent remnants after each test run. Depending on how much data is being created, that might be a problem. We tend to run our automated tests against the same local databases that we use for manual testing and we really don’t like seeing them fill up with reams and reams of junk data. Our data and integration tests get around this by wrapping each test in a database transaction that gets automatically rolled back to revert its changes, but there’s no easy way to do that with a UI test.

(click for difficult ctor)

Another issue is that creating test data in the body of each test might be very difficult. If I made 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.

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.

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.

(click)

But for today, the point I’m making is that if you decide to have your UI tests create their own data, do not create that test data by hand. Doing that is a huge mistake. I don’t care if you create your own data helper library like we did or if you use some other tool, as long as it keeps the tests short and tidy and they only need to specify the values that matter to the test at hand.

(click for caching)

There’s one other problem that you might run into when creating data in the body of each 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 accessing the data through the UI.

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.

Summary

To summarize, using existing data can make individual tests easier 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. Staging the data in the first place can be a pain, and it can be really difficult to manage and classify all of those existing entities in a usable way. And if your tests need to *modify* data 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, but it can also make tests harder to write if you have a large or complex object graph. Also, every time you run the test suite you’ll end up with a ream of newly created junk data in the database. To mitigate those issues, invest in a library of data creation helpers to simplify the test authoring and invest in some tools or utilities that can easily restore the database to a known clean state on demand.

For example, we have a single command line utility that we can execute that will delete our local database, restore it from the most recent snapshot in source control, and then re-apply any migration scripts from the local working copy. That allows us to quickly and easily discard all of that junk data and get back to a clean state.

Those helpers and automation scripts aren’t free and may take a little bit of effort, but they are crucial to making this approach succeed over time.

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

As much as I hate relying on pre-existing data, there’s one thing that I hate even more. And that’s 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)

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 Writing UI-testable code)**

The last thing I want to share with you is some techniques that we use to make our systems more easily testable through the UI.

The first technique is basically a way of *avoiding* UI tests by designing your code to be unit or integration testable instead, because it’s easier to unit test a view model than UI test through the browser.

(click for view)

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

For example, let’s say that if a user has an active license then their license end date should be displayed on their My Account page. If they aren’t licensed then we show some default text otherwise.

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

(click for UI test)

First we 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 utilize a PageObject to access the HTML element representing the email.

(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.

(click for refactored view + test)

Alternatively, we could move that logic into a method on the view model and then test that method directly with a unit test, which is much easier to write and way faster to execute.

(click for TestCase)

Extracting code into unit tests really pays off when you’re testing multiple permutations of something because you can parametertize the test, and now we have a single test that covers both the positive and negative cases and will execute in a fraction of the UI test.

You can’t always get away with this. This works best when you’re testing HTML that’s rendered on the server, rather than on the client, and unit tests won’t detect if the 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 runtime 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 logic where it can be unit tested instead.

(click for CSS classes)

The second technique I want to talk about has to do with how your UI tests interact with the DOM.

When we first started writing UI tests, people tried to write the test 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 inside of a the first span inside of a td that is part 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.

(click for refactored)

As a result, we started using UI-test-specific markers which look like this: “selenium-foo-bar” and “data-selenium-baz=’42’”.

The “selenium” prefix means that these CSS classes are not to be used for styling, and the data elements are not to be used in any JavaScript (other than script executed by a UI test). Developers are encouraged to add these markers to a page whenever it simplifies a DOM lookup for a test.

I’m not saying that we *never* use normal CSS classes in a test, it just depends on the purpose of the test. If the whole point of the test is to verify that an element with a specific class exists, then obviously you should just test for that class.

But if there is no natural CSS class or ID that makes the test easy, then don’t do something fragile like testing a complex dependency chain or searching for a specific piece of text when you could search for one of these markers instead.

Test harnesses

The next technique I want to talk about is test harnesses.

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 see it 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 every single component that allows that component to be executed in isolation from any other 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 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.

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

(click for #2)

Second, test data is a royal pain. If you create your test data up front then tests are easy to write, but maintaining all the pre-built data can be very difficult. If you create data within each test then your tests are harder to write but easier to maintain.

My advice is to invest in a set of data creation helpers to make this easier, and automate the task of restoring your database to a known clean state so that you can regularly clear out all of that junk data.

(click for #3)

Third, if you find yourself using the same UI components on multiple pages, consider creating a test harness for that component. This will not only make it easier for other programmers to understand that component, but this makes UI testing simpler by isolating the component from the application logic.

(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.

**ALSO: Please remember to provide feedback through the CodeMash app about this session.** If you don’t know what to say, I recommend Googling for synonyms of “awesome” and then using them to describe me.

Thank you for your time and have a great rest of CodeMash!

**Parking lot**

Use the “Locator Tree of Life”

<http://www.3qilabs.com/wp-content/uploads/2012/02/selenium-locators-apple-tree-diagram-dashed-fixed.png>

<http://3qilabs.com/best-practices-for-watir-and-selenium-locators-the-locator-tree-of-life/>

<https://screenster.io/selenium-locators-best-practices-7-helpful-tips-for-ui-testers/>

Finding stuff on the page

* Use the tree of life –
  + unique, test-specific IDs, classes
  + keep xpath and CSS locators short. Aim for the closest relatives; immediate parents and children are generally OK, grandparents and grandchildren are riskier, anything else is too brittle. (Graphic: a selector w/ shaded areas indicating “green zone”, “in moderation”, and “danger zone”)
  + When selecting on parent-child structures, choose an ancestor or descendant that has a unique ID. In many cases, is better to target a grandparent via a specific ID than a parent with non-unique attributes. (Less brittle – if a dev modifies an element with a testing ID they will probably search to see what uses it. If you’re targeting an element with no unique identifier then it’s far more likely a developer will change that markup and have no idea it impacts a test)
* Develop a fail-safe lookup algorithm (