GETTING OFF ON THE RIGHT FOOT

WITH YOUR TEST AUTOMATION PROJECT

Getting Off on The Right Foot
with Your Test Automation Project
is brought to you by

Telerik
Test Studio

User Interface (UI) test automation can add tremendous value to any software team's development and testing processes; however, too often teams pass by UI automation for a number of reasons. In this whitepaper you'll discover what some of the problems around UI automation are, and you'll learn specific approaches to avoid those problems. You'll be able to move your focus from chasing intermittent test failures to adding value to your project's overall quality strategy.

Jim Holmes

Jim Holmes has around 25 years IT experience. He is co-author of "Windows Developer Power Tools" and Chief Cat Herder of the CodeMash Conference. He's a blogger and evangelist for Telerik's Test Studio, an awesome set of tools to help teams deliver better software. Find him as **@aJimHolmes** on Twitter.

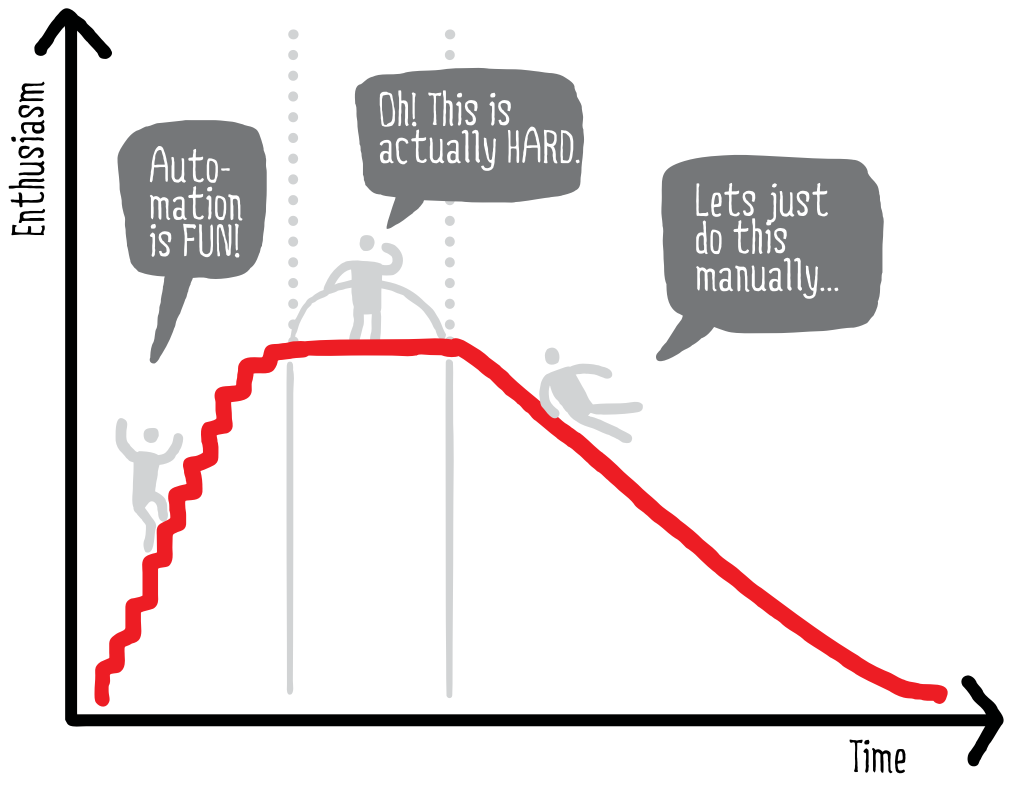
TOO MANY AUTOMATION EFFORTS STRUGGLE OR FAIL OUTRIGHT

UI test automation has long been a bane to software project teams. Smart groups of people are challenged with suites of automated tests that have become extraordinarily brittle and require far too much time to maintain and update when the system under test changes.

There's a common pattern many teams follow when starting off with test automation. Regardless of who you talk to in the automation industry, they'll have a story (Or two. Or three.) that mimics the timeline shown.

Teams start off with automation and are excited about learning a new approach for testing. A few tests are written and the team sees some success. After a few weeks though, the team starts to feel friction around intermittently failing tests. The team is also struggling to make their chosen automation tool work properly in their environment.

Enthusiasm continues to wane as the problems and maintenance work piles up. Eventually some teams abandon their automation efforts as a sunk value proposition and return to huge lists of manual regression tests.



COMMON PROBLEMS

Make no mistake, UI automation is a very difficult problem domain. The technologies used for applications' UIs can be hard to work with, the tools for automation can be difficult, and UI development practices can add to the burden.

Intermittently failing tests are the leading problem for teams. A developer and/or tester works hard to automate a test around a new feature, only to see the test failing in the testing environment. Troubleshooting in the tester's environment leads only to frustration when the test passes locally.

Brittle tests also cause rampant headaches for teams: one small change to the UI, sometimes a change that doesn't impact the visible page, can cause tens or even hundreds of tests to fail. Fixing the failing tests can take hours or days due to critical information being scattered across every test.

Additionally, tests occasionally have an extraordinary amount of duplication in them because testers have copied the same test over and over while simply changing a few input parameters and the expected output condition.

IDENTIFYING ROOT CAUSES

The common problems listed above tie back to a number of common root causes. For example, intermittently failing tests nearly always trace back to two issues: badly defined element locators, or synchronization issues when dealing with dynamic content such as AJAX or JQuery-like frameworks. Additionally, talented developers writing automation scripts too often forget basic software engineering/craftsmanship principles such as Single Responsibility Principle (SRP) and Don't Repeat Yourself (DRY).

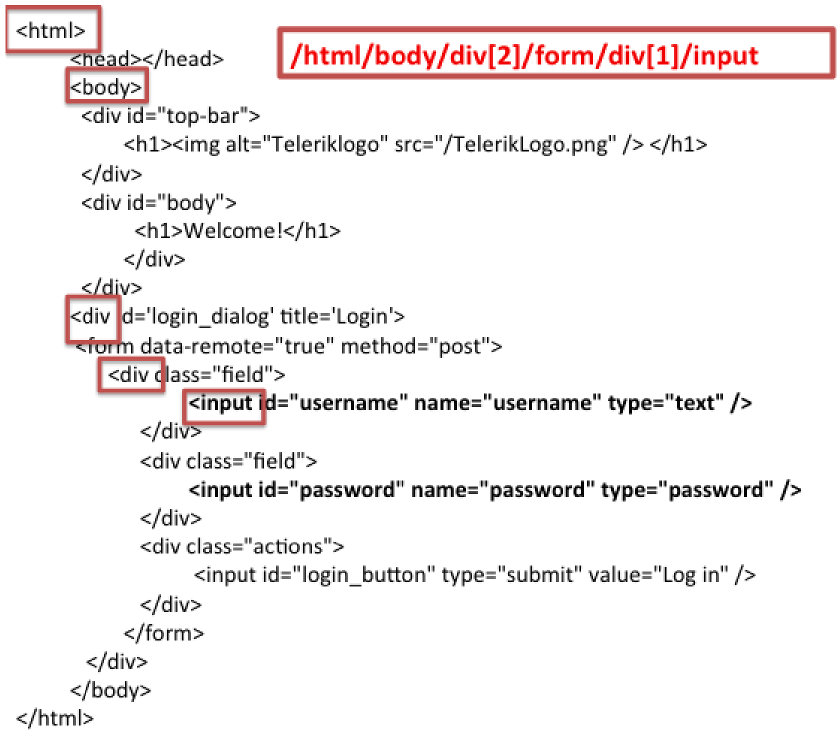
Getting a solid handle on these specific areas will ensure your test suites are providing value to your team instead of sucking the life and morale out of your project because of their maintenance costs.

Locators, locators, locators

Automation drivers, frameworks, and toolsets need to understand how to find elements on the page to interact with. “Locators” or “Find Expressions” are common terms used for the mechanics of how **the automation tool** is able to find one specific element on a page. Locators can be based on a number of factors around where and how the element is displayed in the page’s Document Object Model, or DOM.

Too often automation test writers will use inflexible locators that cause tests to break when a small change is made to the page. Many times this can be traced back to selecting an overly complex XPath-based locator instead of an ID attribute, or if no ID is available, then at least another method, or at the minimum a better-crafted XPath.

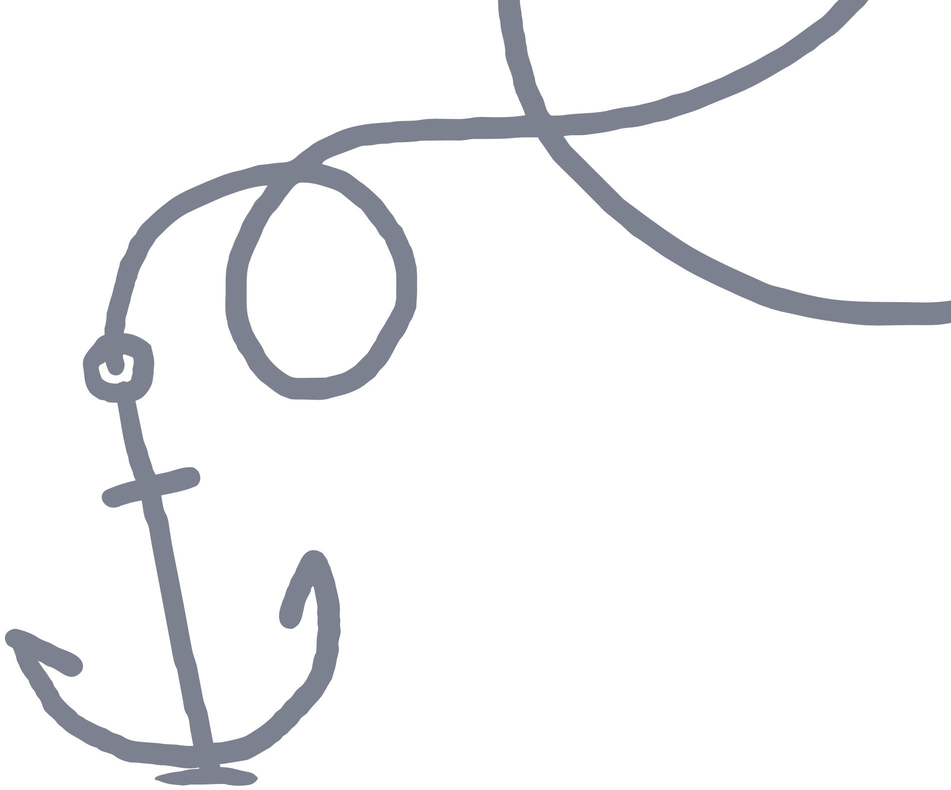
As a practical example, consider the following graphic that shows an absolute XPath which starts at the document’s root and narrows down to the input field for a username. This overly complex XPath is extremely brittle and will break if any element is added anywhere to the page above the username. It will also break if the username field is moved.

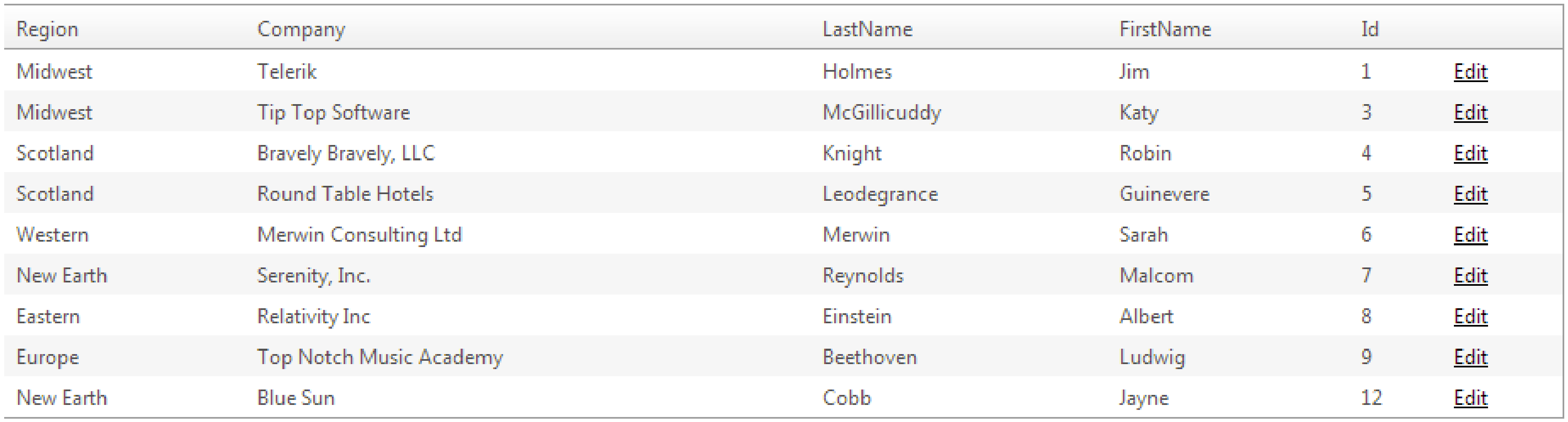


```
<html>
    <head></head>
    <body>
     <div id="top-bar">
         <h1><img alt="Teleriklogo" src="/TelerikLogo.png" /> </h1>
     </div>
     <div id="body">
          <h1>Welcome!</h1>
         </div>
     </div>
    <div id='login_dialog' title='Login'>
     <form data-remote="true" method="post">
       <div class="field">
             <input id="username" name="username" type="text" />
          </div>
          <div class="field">
             <input id="password" name="password" type="password" />
          </div>
          <div class="actions">
              <input id="login_button" type="submit" value="Log in" />
          </div>
        </form>
     </div>
    </body>
</html>
```

Position-Based Locators

Another problem teams inflict upon themselves are locators tied to an element's specific position on the page. We're not talking about X-Y coordinates, but rather things like row or column order. If we're trying to work with an edit test for the record holding the Jayne Cobb person shown below, then we shouldn't have to worry about the test failing if the table's sort order changes. Adding or moving columns in the table also shouldn't break the test. (Although you may want separate tests verifying the proper sort and column order!)





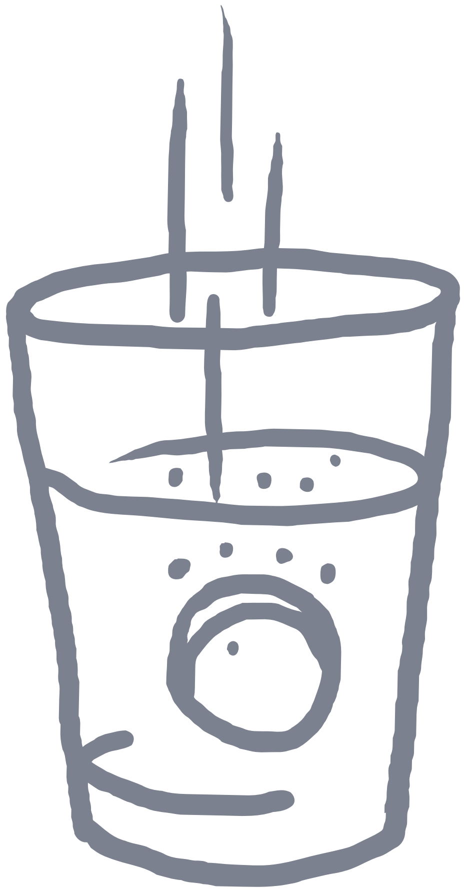
Region	Company	LastName	FirstName	Id	
Midwest	Telerik	Holmes	Jim	1	Edit
Midwest	Tip Top Software	McGillicuddy	Katy	3	Edit
Scotland	Bravely Bravely, LLC	Knight	Robin	4	Edit
Scotland	Round Table Hotels	Leodegrance	Guinevere	5	Edit
Western	Merwin Consulting Ltd	Merwin	Sarah	6	Edit
New Earth	Serenity, Inc.	Reynolds	Malcom	7	Edit
Eastern	Relativity Inc	Einstein	Albert	8	Edit
Europe	Top Notch Music Academy	Beethoven	Ludwig	9	Edit
New Earth	Blue Sun	Cobb	Jayne	12	Edit

Dynamic Content, Big Headaches

AJAX and frameworks/tools like JQuery give us amazingly responsive web pages, but they also make it extremely hard to deal with timing issues for our test automation scripts. Our human eyes guide us to waiting until an AJAX call completes, or JQuery has finished rendering a new control on a page. Unfortunately, automation scripts don't work the same way. Subtle synchronization and timing issues cause scripts to fail because the needed elements aren't currently on the page - the page is still **waiting for that AJAX** or JQuery call to finish their work.

Forgetting Good Software Design Principles

Test code should be treated like production code. Because it is production code! We should use the same careful design approaches in our **test software** as in the systems we're testing. Avoiding duplication of locators is critical to a sustainable test suite. Ensuring we're creating granular tests which can be reused as composable blocks is just as critical. Failing to follow good design principles in test suites will guarantee exploding maintenance costs as the tests and system evolve.



APPROACHES FOR SOLVING PROBLEMS

All the problems discussed above have proven solutions that can help teams get through these challenges. Understanding how to approach these problems is critical to a team's long-term success.

Avoiding Locator Duplication

Duplication in software explodes complexity and maintenance costs. Avoiding this duplication is critical as you evolve your test suite. It's especially critical for your element locators. You can't spend hours tracking down tens or hundreds of duplicate locators when, not if, your UI changes.

Many commercial UI automation tools such as **Telerik's Test Studio** or HP's Quick Test Pro handle locator centralization for you. Those tools use variants of a repository to ensure locators are defined only once, with all tests referencing that central element repository in some fashion. Updating the repository ensures all tests get updated locator information as well.

If you're writing your automation in coded solutions such as WebDriver, Watir, or some other API, then there are a number of approaches to simplify and handle locator definition. Teams have long used centralized dictionaries to store name/value pairs for locator names and definitions. External settings files have also seen modest success, with each test having to load locators from this external file. Those approaches have worked well in the past; however, over the last three or four years a new approach has gradually evolved: the page object pattern.

Page Object Pattern treats each page, or section of a page, as a unique class in code. Properties and methods from the page's class represent elements and services of the page such as logging on or error messages. The Page Object Pattern is a natural extension for developers familiar with good object-oriented development. Moreover, the various open source automation APIs have libraries and frameworks that ease the effort around creating page objects. For example, Jeff Morgan's Pages gem is a great addition to those writing WebDriver tests in Ruby. WebDriver's various bindings also include support for page objects in their native APIs.

However you're working with your tests, it's critical to ensure you're approaching your locator definitions in a centralized, non-duplication fashion.

Supporting Flexible Locator Strategies

Wherever possible, testers should generally prefer to use ID values for defining element locators. Per the HTML specifications, IDs are unique on a valid HTML page. This ensures the automation script can quickly locate the desired element simply by scanning the DOM for that ID. It's fast, it's extremely flexible.

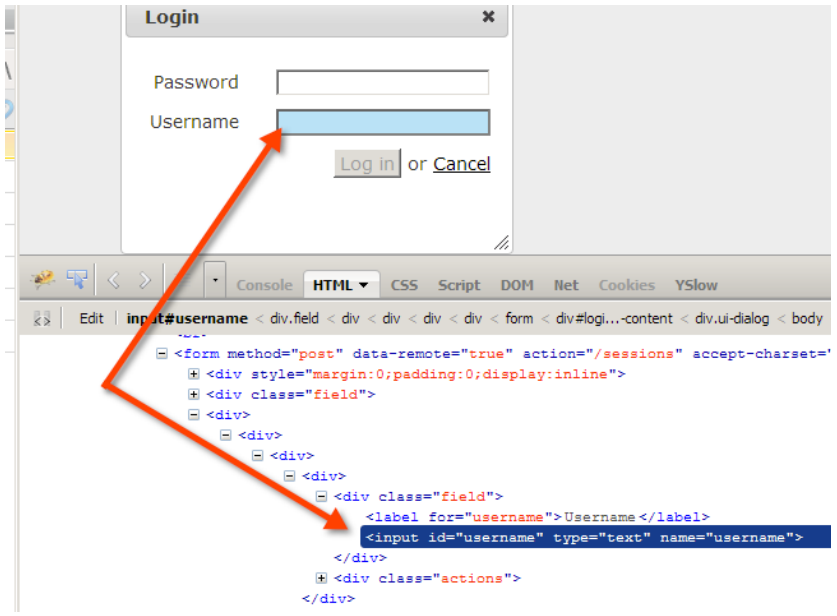
Unfortunately, some systems may not lend themselves to simple locator strategies. Frameworks and platforms may automate how they create ID values, for example, and they may do it in a dynamic nature. In such cases it's often possible for a developer to at least specify a prefix or suffix to the element.

If the developer can do that much, then it's a snap to create a find logic searching for that unique value. Most commercial automation tools, and several popular open source APIs, support defining ID-based locators via some form of "ends with," "begins with," "contains," or similar approaches. This allows you to handle situations where a dynamically generated ID has a unique suffix such as this:

```
ctrl100_ctrl009_div_ctrl300_username.
```

In some cases you may not be able to use an ID for your locators. You may be working with a legacy UI which wasn't designed with testability in mind. You may be working in a system which supports multiple widgets on a page, eliminating the ability to have unique IDs on those widgets.

These situations call for alternative approaches. You can look to name or class attributes, or you can carefully craft an XPath expression. Not all XPath is evil! It's a tool which used wisely can be extremely beneficial. For example, it's a snap to tie an input field to a neighboring label with XPath. Consider the following figure which shows a logon screen. The associated DOM section is highlighted below it in Firefox's Firebug.



```
input#username < div.field < div < div < div < div < form < div#logi...-content < div.ui-dialog < body
⊟ <form method="post" data-remote="true" action="/sessions" accept-charset='
    ⊞ <div style="margin:0;padding:0;display:inline">
    ⊞ <div class="field">
    ⊟ <div>
        ⊟ <div>
            ⊟ <div>
                ⊟ <div>
                    ⊟ <div class="field">
                          <label for="username">Username</label>
                          <input id="username" type="text" name="username">
                      </div>
                    ⊞ <div class="actions">
                </div>
```

A simple, flexible XPath expression can be used to define the locator for that input field:

```
//label[text()='Username']/../input.
```

Locator strategies are completely unique to every system simply because each application's UI is so wildly different from each other. You'll need to learn how to create good locators in your own environments using the general principles shown in this section.

Solving Positional-Based Locators

In the table example shown earlier, it's important to not rely on locators hard-wired to a specific row or column. Instead, understand how your particular automation tool or API can interact with the page. Look to create dynamic locators by querying objects to find elements underneath them match particular criteria.

Here's a snippet of an example in C# using WebDriver:

```
IWebElement table =
browser.FindElement(By.XPath("table_id"));
IWebElement targetRow = null;
IList<IWebElement> rows =
table.FindElements(By.TagName("tr"));
foreach (var row in rows)
{
    if (row.Text.Contains("Cobb"))
    {
        targetRow = row;
    }
}
```

This approach ensures we're still able to find the target row regardless of where it appears in the table. This block of code would still find the target row if it was in row one or 11.

Likewise, clicking the Edit link shouldn't be dependent on which column it appears in. Given the block above, we can use a similar approach to find the Edit anchor by querying the target row we'd already located:

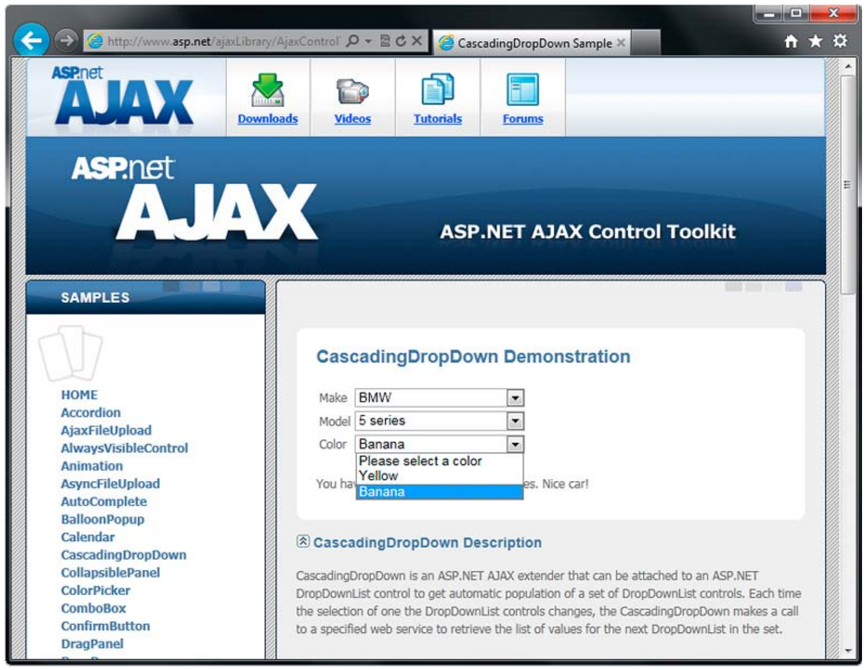
```
IWebElement editLink =
aRow.FindElement(By.LinkText("Edit"));
```

Commercial tools offer up similar features, usually both via coded solutions and native tool functionality as well. At the end of the day, you need to understand how your tools or API work, and leverage those features to ensure you're crafting great locator strategies.

Resolving Dynamic Content Without Headaches

Locators are the single most important thing to understand in your system; however, dynamic content is a close second. The problem lies in the inability of automation tools to detect when an AJAX call or JQuery event is changing the content of the page. This problem spans all web automation tools from **Test Studio** to Selenium WebDriver. Events which cause a page load or refresh get handled by all modern automation tools, but the dynamic events are a different issue.

The common, tried-and-proven patter for **creating rock-solid tests in dynamic content situations** is to use explicit waits for the condition needed by the next step. A great example of this is from Microsoft's ASP.NET AJAX control toolkit examples. The following figure shows a cascading menu system. Each menu selection causes an AJAX call back to the server, which returns the items for the following menu based on the choice the user just made.



CascadingDropDown Demonstration

Make: BMW
Model: 5 series
Color: Banana

You ha[...]es. Nice car!

CascadingDropDown Description

CascadingDropDown is an ASP.NET AJAX extender that can be attached to an ASP.NET DropDownList control to get automatic population of a set of DropDownList controls. Each time the selection of one the DropDownList controls changes, the CascadingDropDown makes a call to a specified web service to retrieve the list of values for the next DropDownList in the set.

That tiny little server callback is what causes automation workers serious grief. It's dynamic, it's impacted by network conditions, and it's always slightly different timing.

Using explicit waits before interacting with a newly updated element are the key to saving your team's sanity in these situations. In the example above, you'd create an explicit wait for the Make dropdown to fully populate with its options, then select the particular option you want for that pass. The Model option list would get the same treatment: an explicit wait for the exact contents to load, followed by the selection action.

This pattern of an explicit wait coupled with an interaction is a tried, proven strategy for every dynamic content situation. The pattern is the same regardless of whether you're waiting on content or controls to appear on the DOM, or even an existing control to change its state (inactive to active, eg).

Implementing waits in your test scripts is completely dependent on the tool or API you're using. Telerik's Test Studio uses a Wait step; WebDriver utilizes the WebDriverWait class in the support namespace. Other tools and APIs have similar features.

Explicit waits eliminate the frustrating intermittent failures due to synchronization issues around dynamic content.

Helping Maintainability by Supporting Good Design

You've seen how good design for storing element locators helps teams create maintainable tests by eliminating duplication around element locator definitions. The same concept of test or method reusability is just as critical to good test case creation. The ability to compose elaborate tests from smaller building blocks ensures teams aren't wasting valuable time updating the same functionality across multiple tests when a part of the system's workflow changes, for example.

Design each test such that it's granular, specific, and doesn't rely on other test cases first. This enables you to reuse functionality such as logging on to a system or entering customer data.

Composability and reuse don't have to be at the page level, either. You can look to break down complex forms into small pieces of functionality designed to handle one specific responsibility such as customer identification, ticketing, etc.

Careful reuse of functionality lets you write more accurate, thorough tests at a much faster rate - and at the same time dramatically decreases maintainability costs.

KEEPING YOUR AUTOMATION SUITES SANE, STABLE, AND MAINTAINABLE

The approaches in this paper aren't a magic panacea for everyone's automation woes. UI automation is an incredibly hard problem, and it's completely different for each application. You'll have to learn the fundamentals of your application works, and you're still responsible for ensuring you're creating solid automation tests.

Spend time learning how to get past the basic domain problems like locator strategy, dynamic content synchronization, and support for good test case design. That leaves you more time to focus on the real problem, which is how to deliver more great value to the projects you're working on.

Test Studio

Test Studio is an automated testing tool that offers an intuitive, codeless and productive way to test any application! Complex AJAX, Silverlight and WPF scenarios, MVC, client-side functionality, JavaScript calls, data-driven testing – we cover them all. Test management and failure resolution are brought to a new level, making you times more productive. Even more, the slick yet simple UI will have you testing like an expert in minutes.

Download
a fully functional free 30-day trial