

10 TIPS ON HOW TO DRAMATICALLY REDUCE TEST MAINTENANCE



INTRODUCTION

Creating powerful functional user interface test suites is only partially about getting the tests right. Test automation software needs to be built with the same care as our production systems (test code is production code). This means maintainability and flexibility need to be at the forefront as we build out those suites.

Unchecked test suite maintenance issues aren't just frustrating, they take time away from initiatives that deliver value to your customers. In "The Art of Unit Testing," Roy Osheroove writes about a project his team was struggling with. Six months into the project his team was spending 75 percent of their day fixing failures in their test suite and only 25 percent working on the actual system. After mounting frustration with a lack of progress the team's client terminated the project and fired the team.

How do teams get into such a spot? They do it by missing out on crucial test suite design and implementation issues. User interface tests aren't designed with flexibility in mind—each small change to the UI breaks a significant number of tests. Execution times aren't considered when building tests, so suites take hours or even days to complete, creating ridiculously long feedback loops. Complex UI actions using AJAX or in-browser JavaScript cause intermittent test failures, resulting in increasing distrust across the organization.

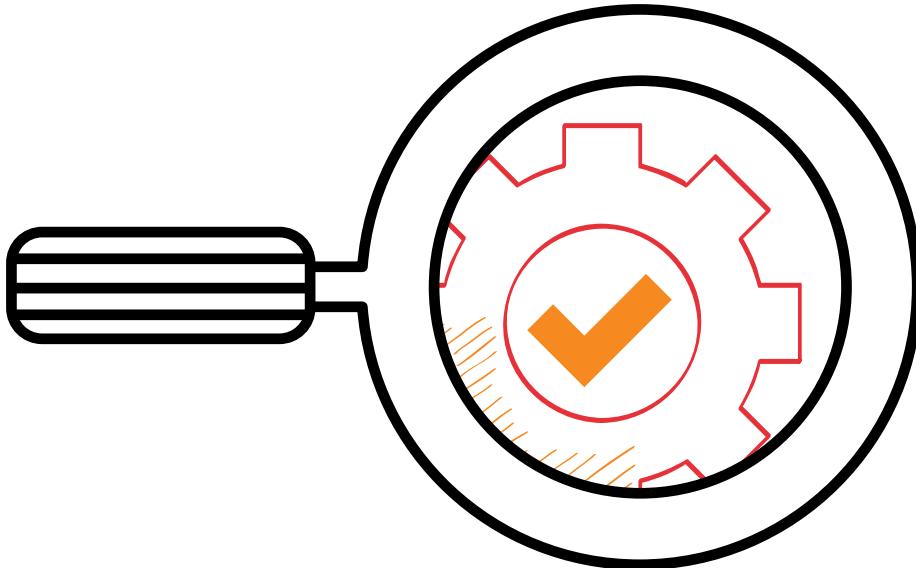
Many teams struggle to get reliable, low-maintenance test suites in place. Here are ten tips to help you keep your test suites much more maintainable.

In the grey boxes that complement this paper you will find a showcase on how Telerik Test Studio QA team takes advantage of the 10 tips to create better test automation.

Telerik Test Studio QA team

- ✓ Testing a WPF application [Test Studio]
- ✓ Leveraging an agile development process
- ✓ Both software developers and QAs take part in quality assurance
- ✓ Three major product releases per year and internal builds every week
- ✓ Thousands of customers using the product, including ESPN, Dell, Symantec, GE and more

1. FOCUS ON AUTOMATING THE RIGHT THINGS



What's the easiest test to maintain? The one you didn't write.

UI automation is hard. It's a difficult domain to work in, the tests are slower to write than other tests, they're slower to run and they're harder to keep working. If you want to keep your maintenance costs as low as possible, carefully consider what you're writing your UI tests for.

Focus your UI automation efforts on high-value features. Talk with your stakeholders and product owners. Find out what keeps them awake at night. I'll bet it's not whether or not the right shade of grey is applied to your contact form. I'll bet it's whether customers' orders are being billed correctly. I'll bet it's whether sensitive privacy or financial information is at risk of being exposed to inappropriate visitors to your site.

Automate tests around critical business value cases, not around shiny look-and-feel aspects of your application.

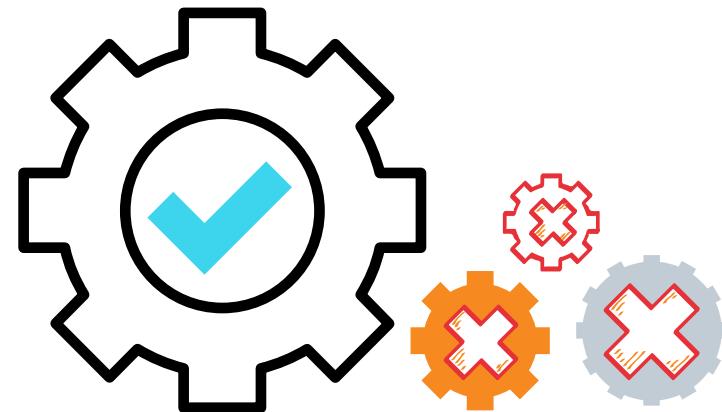
Write your tests for the correct features. It will cut your maintenance costs dramatically.

2. DON'T AUTOMATE LOW-VALUE OR STABLE FEATURES

Avoiding automated tests around look and feel isn't enough. Teams need to understand how to **avoid creating automated tests around low-value features or features that are relatively stable.**

Let's take a contact form, for example. It's handy and often an important communication channel for companies. That said, a contact form is a **feature that rarely changes**, and they're fairly simple in their function. Instead of writing automation around these sorts of features, leave these as areas you quickly explore and confirm occasionally during your release cycle.

Understanding which features to avoid writing automated tests for helps you ensure your test suite doesn't take on unnecessary maintenance costs.



3. CUT YOUR CONFIGURATION MATRIX

Here's a common scenario: You need to test your application's client side on multiple browsers on multiple operating systems. You also need to test the server side against different combinations of operating systems and databases.

This may look something like this:

COMPLETE CONFIGURATION MATRIX					
	IE9	IE10	IE11	FF	Chrome
Win7	X	X	X	X	X
Win8		X	X	X	X
Win8.1			X	X	X
Win10			X	X	X
Server2012			X	X	X

That's 28 different client-side combinations you'd think needed to be tested. You can easily extract the same complexities around database and server OS configurations.

Do you really need to test all combinations? Do some careful thinking around the high-value data pairs on this grid. **Look at adoption rates of browsers on particular operating systems, or usage rates in your existing systems.** You'll likely find you can dramatically reduce your matrix to something closer to the one below, which holds only 17 combinations.

By the way, these grids are fabricated and meant as examples. Please don't take these as gospel for your own configuration testing.

OPTIMIZED CONFIGURATION MATRIX					
	IE9	IE10	IE11	FF	Chrome
Win7		X		X	
Win8				X	
Win8.1					X
Win10				X	
Server2012					X

Do you have more complex data combinations? Are you using three or more variables with multiple values in them? If so, run, don't walk to <http://pairwise.org> and read up on combinatorial testing. Combinatorial testing can help you dramatically shrink your required test case data while keeping great coverage.

Complex test matrixes lead to significant maintenance costs for updating and executing. **Keep your matrixes small and effective.**

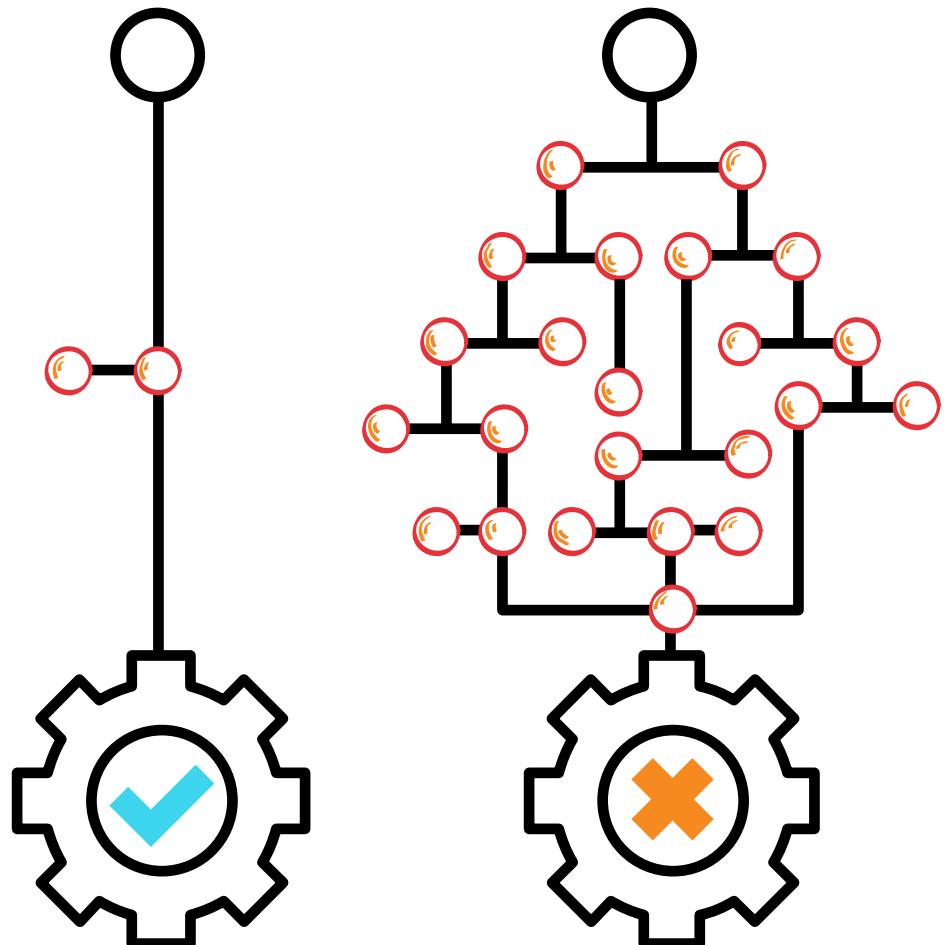
4. AVOID COMPLEX TEST SCENARIOS

Maintainable test suites focus on simplicity and flexibility. Just because you can create tests with multiple nested conditional statements (IF-THEN-ELSE) doesn't mean you should. Just because you can create complex data-driven scenarios that mix behaviors doesn't mean you should. As stated earlier, test code is production code. That means we should use the same concepts for writing tests that are simple and focused.

Avoid nested conditionals. Conditionals bump up the cyclomatic complexity of software. This makes tests harder to debug, and harder to read.

Data driving tests, while a powerful concept, can also be easily misused. Avoid trying to create one test to handle multiple behaviors based on different sets of data. **Data driving is best used when fleshing out coverage for one scenario,** not trying to validate different behaviors.

Complexity will kill your maintenance costs. Not just for your test suite, for your system overall. Avoid it.



5. KEEP TESTS GRANULAR AND INDEPENDENT

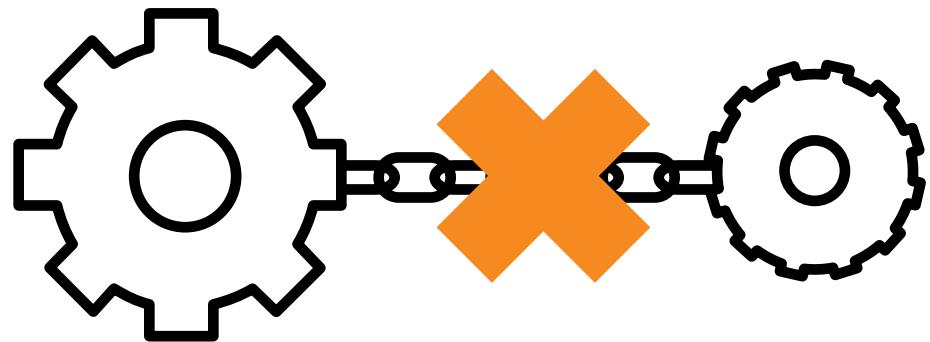
It can be alluring to think of stringing multiple tests together into one long scenario. It can be equally alluring to use a test to set state, data or configuration for tests executed later in the same set. It's alluring. ***It's also dangerous and wrong.***

Good automated test cases are specific—they check one thing and they check it well. Mixing test cases into a longer scenario can often be risky.

Worse yet is the notion of sharing test's data, state or configuration across tests. This lesson has been learned the hard way over years of frustration in the test automation domain. Unexpected interactions and subtle side effects can cause great disturbance. It's hard to track down these conditions.

While shared state/data can be frustrating in single-threaded, serial test execution, it's guaranteed to cause you grief in distributed/parallel execution environments. There's simply no way you can rely on ordered test execution in these conditions.

Save yourself long-term maintenance headaches. Take the time to ***ensure each test is granular and independent.***



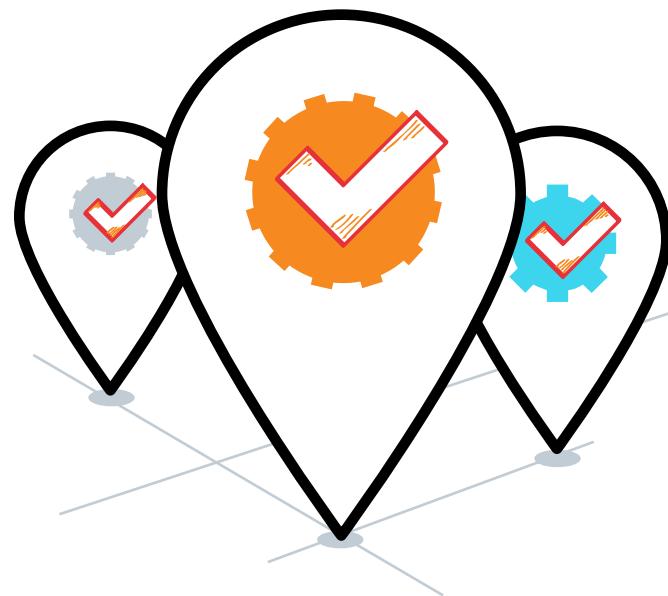
6. LEARN YOUR SYSTEM'S LOCATORS

To paraphrase the real-estate mantra: locators, locators, locators. Developing a great locator strategy for your system will greatly reduce your team's maintenance efforts for your automation suite. ***Brittle locators are the cause of 73.5 percent of all intermittently failing tests.*** [1] Avoid high maintenance costs by working early to understand which approaches work best in your environment and which aren't good choices.

Many factors play into how a web page renders elements on the page: the web server hosting the application, the technology stack the application is built on, the UI designer's approach and the developers themselves. You'll need to become extremely familiar with how all these factors interact and where you can leverage them to adjust how locators are created.

Your best friends in creating a great locator strategy are the developers on your team. They'll be able to work with you to provide good ID values wherever possible and help out with other approaches when IDs aren't appropriate.

Determining a great locator strategy may be your biggest timesaver in maintenance costs. Invest your team's time wisely early in your project.

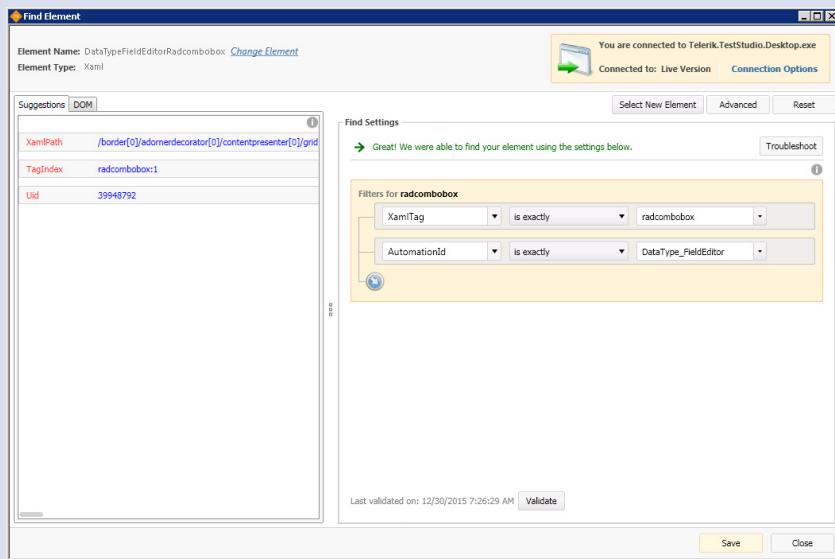


HOW THE TEST STUDIO TEAM DOES IT

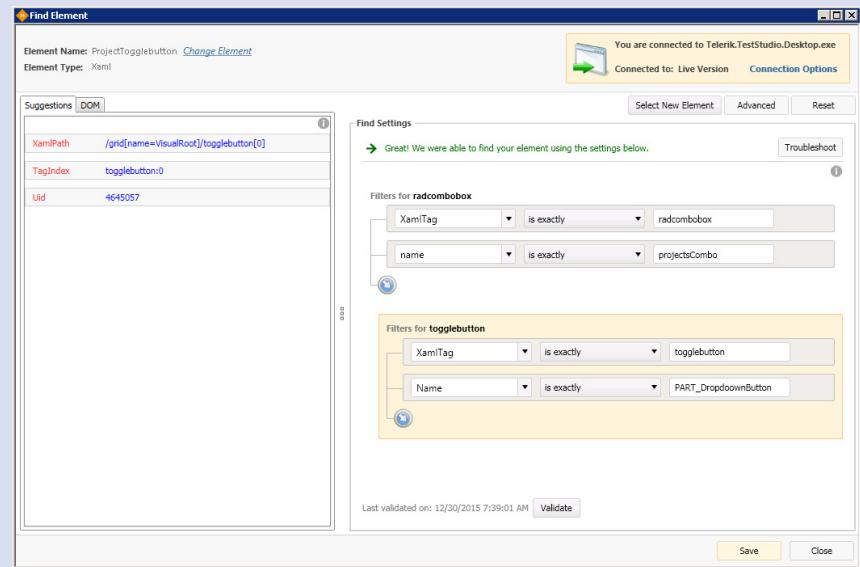
Locator Strategies

Test Studio is a Windows Presentation Foundation (WPF) application. WPF (and Silverlight) locator strategies are nearly identical to that of HTML-based applications: Start with good IDs and work backwards from there.

First and foremost, our QA team works closely with the Test Studio developers to ensure XAML Automation IDs are added to UI controls for the most reliable and hassle-free find logic. This lets the Test Studio recorder easily pick up solid, stable locators. It also enables us to easily make manual adjustments.



As with any application, sometimes IDs/Automation IDs aren't available for the element that's needed. When this is the case, we use the Test Studio chained find expressions—we try to identify the parent element, then search within it for the element that's needed:



Test Studio is a complex application. Sometimes our QA team encounters difficult location scenarios, like when no Automation ID is available and several elements of similar types exist on the page. In this case, we usually drop to coded steps to handle these scenarios. The coded example below is a common case for the Test Studio QA team in which we use a `Find.ByTextContent` expression to locate an element. The rest of the expression moves up to that element's parent (a `GridViewRow` and then back down in to find the `CheckBox` control.

```
public class BT_ConfigureAutoSubmitON_Step : BaseWebAiiTest
{
    [Dynamic Applications Reference]
    [CodedStep(@"Select TeamPulse bugtracker checkbox")]
    public void BT_ConfigureAutoSubmitON_Step_CodedStep()
    {
        System.Threading.Thread.Sleep(1000);
        var chkboxTP = this.Applications.TelerikTestStudioexe.
            Manage_Bug_Tracking_
ProvidersGridViewRadgridview.
            Find.ByTextContent("TeamPulse Server").
            Parent<GridViewRow>().
FindByType<CheckBox>();
        If (chkboxTP.IsChecked == false)
        {
            chkboxTP.User.Click();
        }
    }
}
```

7. LEARN YOUR SYSTEM'S ASYNCHRONOUS ACTIONS

Asynchronous activities on a page or app view will cause significant maintenance impacts if teams aren't ready to deal with them correctly in their automation scripts. Asynchronous events in an application update the UI without an otherwise blocking action. Automation scripts have to deal with these situations appropriately or suffer maddening¹³⁵.

Most teams are familiar with AJAX for asynchronous callbacks from the client to the server. What automation teams need to keep an eye on are the client-side asynchronous actions that are becoming extremely prevalent in modern applications. JQuery, Node, Ember, Backbone and many other popular JavaScript frameworks are often used to modify page content on the fly without any traffic leaving the client.

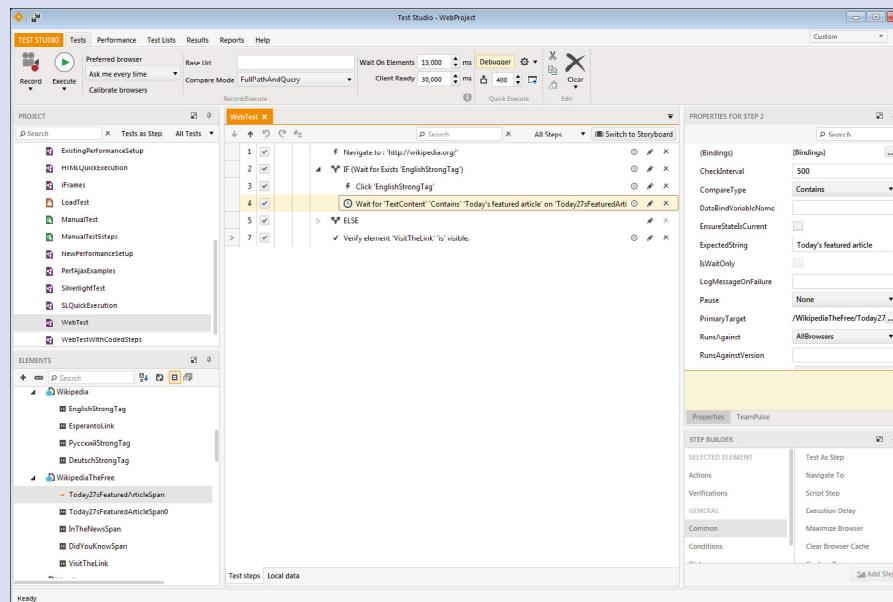
Subtle timing differences between environments and execution runs will cause frustrating intermittent test failures—a maddening hit on maintenance costs as teams struggle to deal with brittle tests.

Dealing with asynchronous operations effectively means learning how your system and applications handle UI updates via those async operations. **Your teams will need to learn to avoid hard-wired constant delays religiously.** They'll need to **learn which actions or elements to latch on in a dynamic, flexible fashion.**

Your teams will also need to learn to **spend more time collaborating between testers and developers.** People creating automated scripts simply can't treat the page or view like a black box. They'll never be able to reverse engineer the way complex events or multiple asynch operations are handled on the page. Developers can quickly point out a page's flow and they can also assistance by creating helper functions or adding specialized, hidden elements to the DOM.

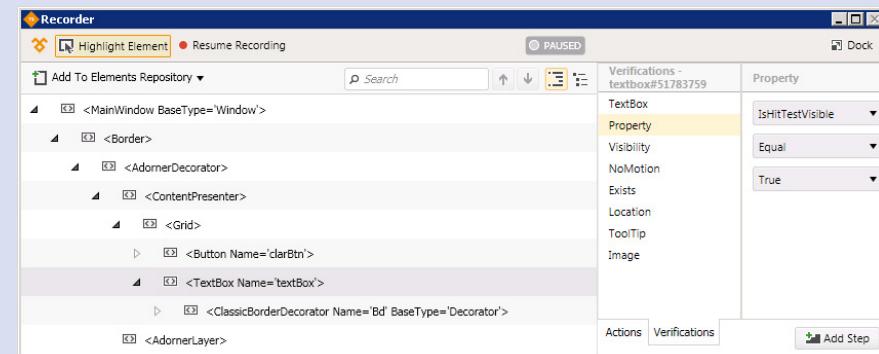
HOW THE TEST STUDIO TEAM DOES IT—ASYNCHRONOUS ACTIONS

Our Test Studio testers approach asynchronous actions by leveraging explicit Wait operations—those with a specific condition such as existence of an element or text—before any Actions or Verify operations. Verification steps have an implicit wait, but the verification step will fail if the condition is not met immediately. Explicit Wait steps help us delay the execution until the explicit condition defined in the step is met, or until a defined timeout period expires.

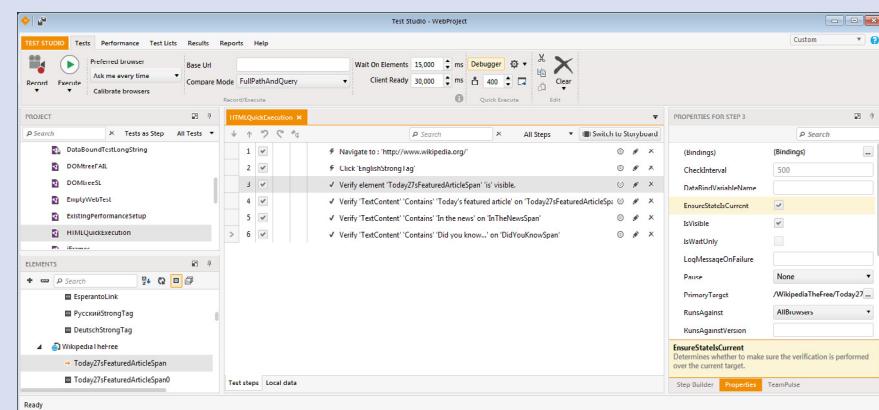


This pattern of wait, then act/verify, is a rock solid pattern that saves us, the Test Studio product testers, lots of maintenance and troubleshooting headaches. Close coordination with the Test Studio developers helps the QA team understand when to use `WaitForExists`, `WaitForVisible` or other explicit wait conditions. Our Test Studio developers have intimate knowledge of the view loading cycle and are able to work with us to properly handle any asynchronous situations.

XAML applications like the Test Studio one bring additional complexity to automation. XAML applications often stack UI layers atop one another in the same view. This leads to conditions in which all elements from every layer are reported as both Existing and Visible. Test Studio developers have implemented custom properties such as `IsHitTestVisible`, which indicate whether an element is visible on the top UI layer. This greatly eases automation since we can easily build custom validations using the advanced Test Studio Recording options:



The Test Studio `EnsureStatesCurrent` property can also help these situations by refreshing the DOM/XAML tree if state changes after the page/view's initial load.



8. LEVERAGE CODE WHERE NEEDED

Functional UI tests are already slow and prone to brittleness. Don't compound those issues by performing setup or teardown actions in the UI during tests. Instead, work with developers to create a backing infrastructure that can handle these tasks.

As an example, consider a test retrieving a user from a Customer Relationship Management (CRM) system. If we're retrieving a user from the CRM, we first need to create that user. A script creating a user via the UI might take 20–30 seconds to perform. The “retrieve” action might take 15 seconds to log on, query the system, and validate the data. We've doubled or tripled the length of the test by simply creating a user before doing the actual test. That may not seem like a huge impact, but it adds up when you're working with larger suites. 30 seconds of waste across 300 tests is ***over two hours of wasted time.***

Rather than doing these actions at the UI, use your system itself. Work with developers to create a backing infrastructure of helper classes and methods that call into appropriate web services, internal APIs or even directly to the database. Using system-level calls can cut these sorts of actions from 20-30 seconds to fractions of a second. That gives you hours of execution time back, and you'll find those APIs are much less brittle than the UI itself.

Moreover, the same principles of modularity applied to system development should also apply to test suite development. Abstraction, reuse, good design—all these principles apply equally to test automation suites. ***Create helper classes to handle repetitive actions. Refactor out duplication and complexity.***

Treat your test suites like production code—because they are.

HOW THE TEST STUDIO TEAM DOES IT—LEVERAGING CODE

The Test Studio QA team works hard to follow good design principles for our automation suites because we realize automation's highest cost isn't the initial creation of scripts, it's the maintenance costs.

We factor out common functions and tests to helper methods and classes for use by other tests. Multiple pre- and post- conditions are leveraged to help optimize where we think it makes sense.

Our QA team is also careful to not fall into the pit of Big Up Front Design (BUFD)—we work to keep initial implementations lean, and carefully iterate and extend helper infrastructure only when needed.

Finally, the Test Studio QA team makes use of web service calls and internal APIs, enabling them to quickly handle pre- and post- configuration actions. Instead of trying to handle setup and configuration via complex, long running tests, our QA engineers hit existing web service endpoints and internal APIs to leverage Test Studio functionality.

This is a huge maintenance saver for us, as the speed is tremendously faster, and the internal APIs are far less brittle than UI-level actions.

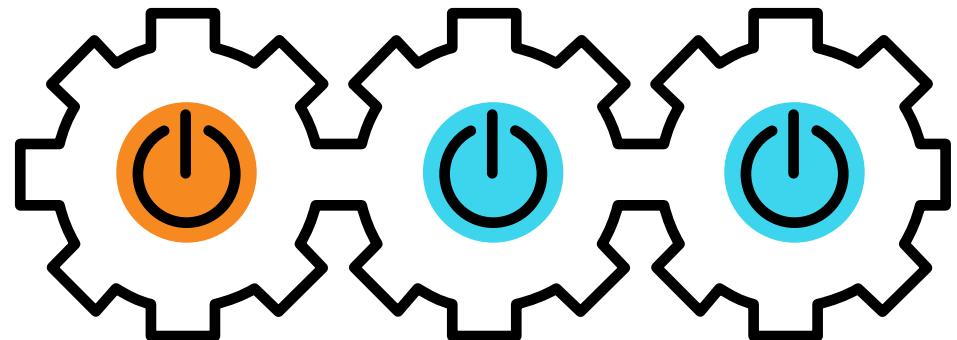
9. CONFIGURE YOUR SYSTEM FOR EASIER TESTING

“How do we automate testing of CAPTCHA?” is a question that frequently comes up. The answer? Don’t. Third party components like CAPTCHA or TinyMCE can be extraordinarily difficult to automate—and CAPTCHA is frankly impossible to.

There’s no sense in writing tests around third party components or services that have already been tested by that tool’s creators. In such cases it’s completely acceptable to ***work with your developers to create system switches that turn off parts of your system to make it more testable.***

Bypassing CAPTCHA, or swapping out TinyMCE for a simpler plain-text editor, are two examples of this.

Yes, it takes some design, work and testing to get these things done. However, the payoff is a system that’s dramatically easier to test, which means you’re saving significant time in your regression testing and hardening phases.



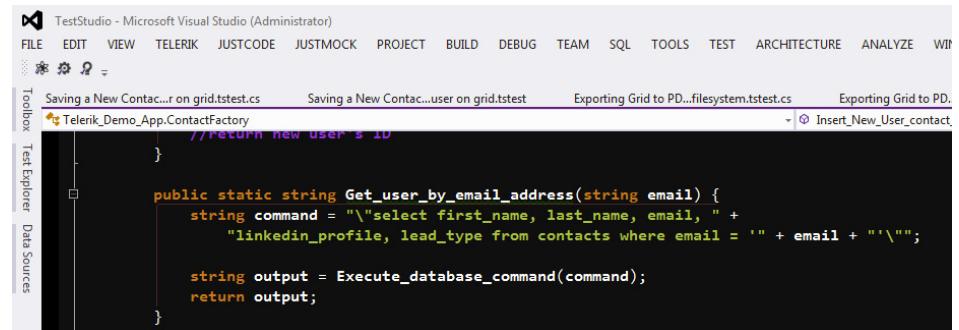
10. GO BEYOND THE UI FOR YOUR FUNCTIONAL TESTS

Good tests don't stop at the UI. How many times have we seen badly handled calls to the persistence layer resulting in a disconnect between what's shown on the UI and what's actually in the database?

Using an oracle or heuristic to perform this final validation is a sign of a high-value, well-written test. This true test checks not just the UI, but also the record of truth in your system: the database, a storage service, etc.

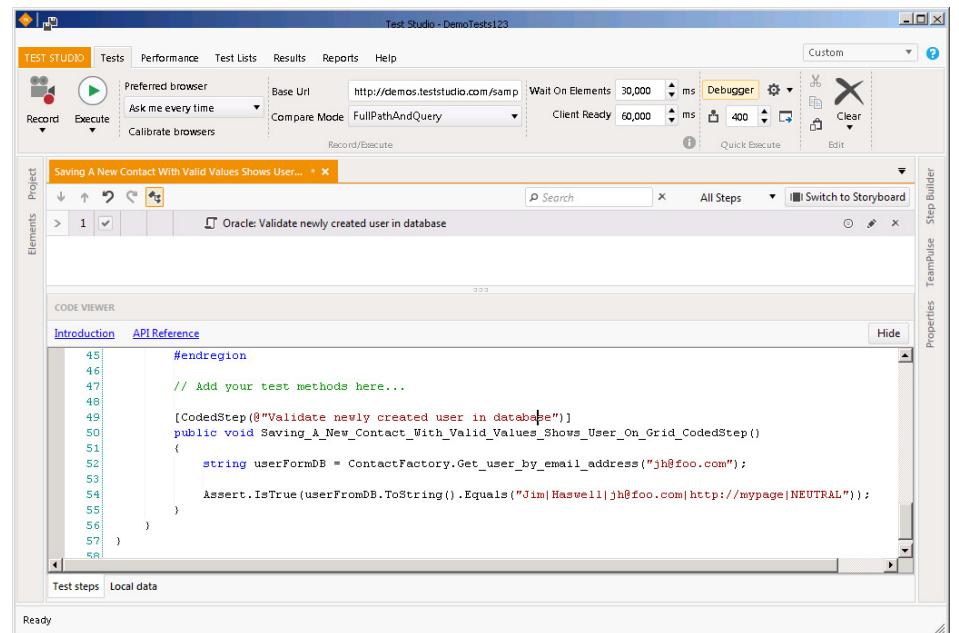
These oracles should be added into the same infrastructure you're creating to handle setup and teardown actions. Developers who live and breathe at the system level can quickly help create oracles using system-level calls. This gives automation script writers an easy way to quickly build high-value tests. Moreover, creating proper abstraction layers over the actual system-level calls cuts maintenance costs because your tests aren't impacted if a web service or internal API changes.

Earlier in this paper you saw how easy it was to bring developers and testers together to handle setup and teardown actions. It's the same simple collaboration for creating great test oracles. **Developers create the appropriate backing APIs, then hand off to the testers.**



```
    public static string Get_user_by_email_address(string email) {
        string command = "\select first_name, last_name, email, " +
                        "linkedin_profile, lead_type from contacts where email = '" + email + "'\'";
        string output = Execute_database_command(command);
        return output;
    }
```

Testers then finish their tests by pulling in the newly completed oracle APIs where needed.



```
    #endregion
    // Add your test methods here...
[CodedStep(@"Validate newly created user in database")]
public void Saving_A_New_Contact_With_Valid_Values_Shows_User_On_Grid_CodedStep()
{
    string userFromDB = ContactFactory.Get_user_by_email_address("jh@foo.com");
    Assert.IsTrue(userFromDB.ToString().Equals("Jim|Haswell|jh@foo.com|http://mypage|NEUTRAL"));
}
```

ABOUT TELERIK TEST STUDIO

Telerik Test Studio is a powerful, reliable test automation tool that helps you create maintainable test suites for a wide range of platforms and browsers. It inspires testers and developers to collaborate on building high-value test automation and increase team velocity.



I had to make a tough decision to dedicate time when I had virtually no time and projects / priorities being tossed at me seemingly non-stop. I decided to carve out a small amount of time to discuss our needs with Telerik's ALM consultants who took me on a brief tour. In a matter of hours, I could see the light at the end of the tunnel and today our product quality had improved tremendously, our team is more organized and I embrace projects as they come my way, knowing we have a system that works."

Jeff Freeman
Senior Consultant, NAV Canada



Test Studio Wins Gold at the ATI Automation Honors

ATI WinnerIt's an honor for us to have Test Studio recognized as the best commercial functional automated testing tool in the .NET category in the 5th Annual ATI Automation Awards.



Telerik named a Visionary in 2014 Gartner Magic Quadrant

Gartner recently released its Magic Quadrant for Integrated Software Quality Suites report. We couldn't be more excited about being named a "visionary" in the quadrant.



Telerik Test Studio

Standalone app and Visual Studio plugin for functional, performance and load testing