PROFESSIONAL TESTER

Essential for software testers

SUBSCRIBE
It's FREE
for testers

April 2012 | £ 4 / € 5 | v2.0 | number 14

methods & frameworks

Including articles by:

Huw Price
Grid-Tools

Suri Chitti

Jim Holmes
Telerik

Paul Fratellone
MindTree

The long game

by Jim Holmes

Building long-term viability into test automation

How **Jim Holmes** avoids making a rod for his own back

The importance of functional test automation at the GUI level, and yet how difficult it can be, is a topic that's been close to everyone in testing for many years and will remain so for some time yet. Thinkers about testing have made many wonderful contributions to a growing body of literature that provides a great deal of help. I'd like to make special mention of Lisa Crispin (see http://lisacrispin.com), Elisabeth Hendrickson (http://testobsessed.com) and the just-released book *Experiences of Test Automation* by PT contributor Dorothy Graham and Mark Fewster (ISBN 9780321754066).

In this article I will discuss two current and growing challenges being faced now by testers: dealing with dynamic content, and deciding what and what not to automate. I have chosen them as examples of how test automators should focus on the lifetime of their tests as much as their initial value.

Deadly delay

Dynamic content is presentational layer information that's added to a web document at some time after it has completed loading, that being triggered by some event. Various methods exist of which AJAX ("Asynchronous JavaScript and XML") is the most popular. A user action, for example selecting a menu item, causes a call to the server for more information that adds or populates an element without reloading the page. That call causes a problem for automation because the client has no way of knowing how long it will take to complete, which varies with system and network delays and latencies. An automated test that interacts with the elements, to verify their content or in order to make inputs for another test purpose, must handle the delay or will likely fail to execute.

The most obvious way to prevent that is to place a fixed delay into the test, aiming to make it pause until the next object with which it interacts is ready. However it's hard to guess the period to use: too short and the test, although it runs OK right now, may fail later due to change of environment or environmental conditions, increasing the need for intervention and hand-holding. Too long, and execution is slowed unnecessarily, usually significantly so as the delay will probably have to be replicated many times within this and other tests. Worst, the effort and difficulty of maintaining the tests when the SUT or test requirements change is increased.

All things come to those who wait

A better approach is to use the wait facility offered by virtually all test automation frameworks and tools. Testers typically come across dynamic content in two forms: an element with which the test needs to interact but that has not yet loaded can be handled using *implicit wait*. Elements that

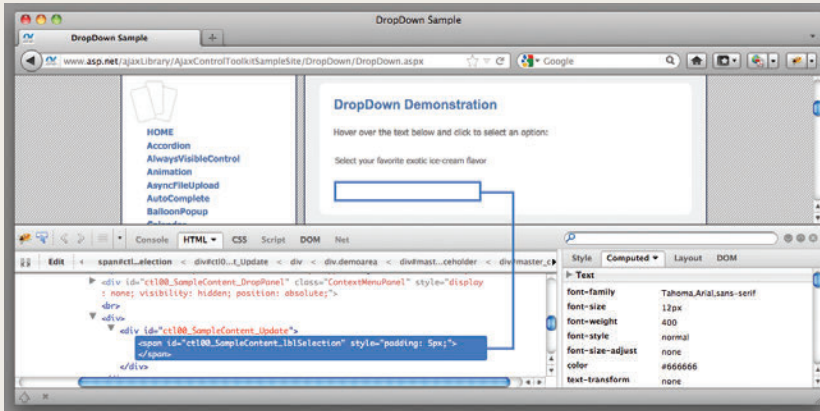

Figure 1: ASP.net AJAX DropDown extender with empty <span>

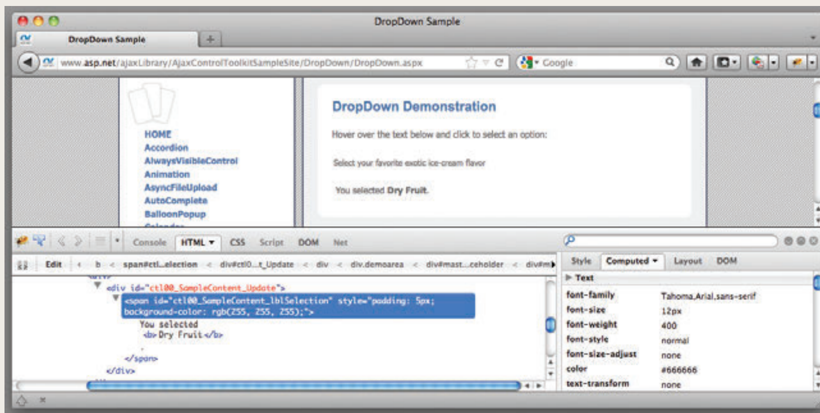

Figure 2: DropDown with populated <span>

```
[Test]
public void Implicit_wait_example ()
{
    IWebDriver browser = new FirefoxDriver();
    browser.Navigate().GoToUrl(
        "http://localhost/AJAXDemo/DropDown/DropDown.aspx");
    browser.FindElement(By.Id("ctl00_SampleContent_TextLabel")).Click();
    browser.Manage().Timeouts().ImplicitlyWait(TimeSpan.FromSeconds(10));
    browser.FindElement(By.Id("ctl00_SampleContent_Option1")).Click();
    Assert.IsTrue(browser.FindElement(By.Id("ctl00_SampleContent_lblSelection"))
                     .Text
                     .Contains("Dry Fruit"));
    browser.Quit();
}
```

Figure 3: WebDriver script for implicit wait

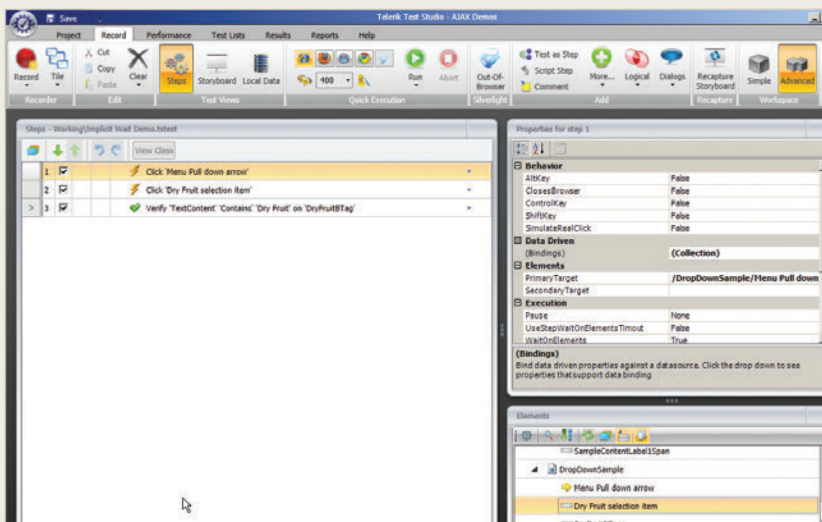

Figure 4: Implicit wait in Telerik Test Studio

have loaded but whose content has not require *explicit wait*.

Figure 1 shows an example of the first of these scenarios, a needed element not yet present (this page and the one discussed below are available online at http://asp.net/ajaxLibrary/AjaxControlToolkitSampleSite: this one is the DropDown page). That element is a message ("You selected…") which appears only after the user makes a selection from the dropdown menu. Examining the Document Object Model (DOM) of the page using Firebug (http://getfirebug.com) reveals an empty <span>. Figure 2 shows the DOM after the user makes a selection: the span is populated with some text including a <b> element containing text showing the item selected. An automated test script that validates that text to show that the correct item (the one selected) is displayed must deal with the delay before it proceeds.

For example, WebDriver (http://seleniumhq.org/projects/webdriver) provides implicit wait using the Timeouts method on an IWebDriver's Manager. Figure 3 shows part of a script for the DropDown example page written in C# using WebDriver's FireFox driver. After clicking the dropdown, the script polls the message element (lblSelection) and does not continue until it contains the text "Dry Fruit". If that does not happen within 10 seconds, a TimeoutException is thrown.

Some tools handle implicit waits automatically with no additional steps. Figure 4 shows the same test in Telerik Test Studio.

Explicit content

The second scenario is exemplified in figure 5, the *CascadingDropDown* example from the same site. Here the elements – the dropdowns themselves, in this case implemented with simple HTML <select>

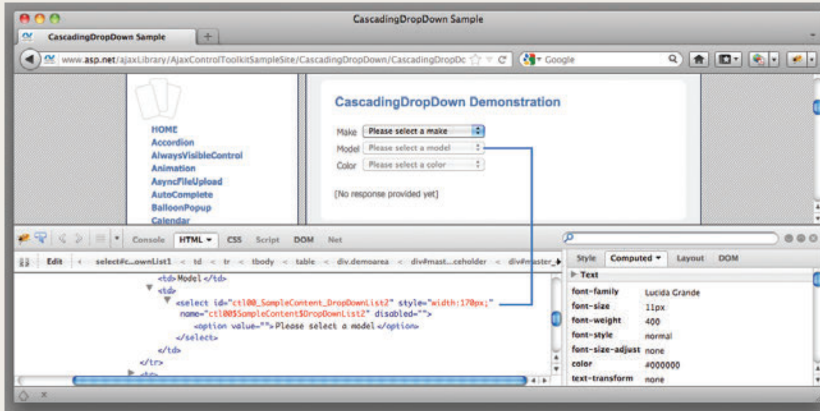

Figure 5: CascadingDropDown

```
var listOfMakes = browser.FindElement(By.Id("ctl00_SampleContent_DropDownList1"));
WebDriverWait wait = new WebDriverWait(browser, TimeSpan.FromSeconds(10));
wait.Until<IWebElement>((d) =>
{
    return d.FindElement(By.XPath(
        "id('ctl00_SampleContent_DropDownList1')/option[text()='Acura']"));
});
var makeOptions = new SelectElement(listOfMakes);
makeOptions.SelectByText(make);
```

Figure 6: Wait class

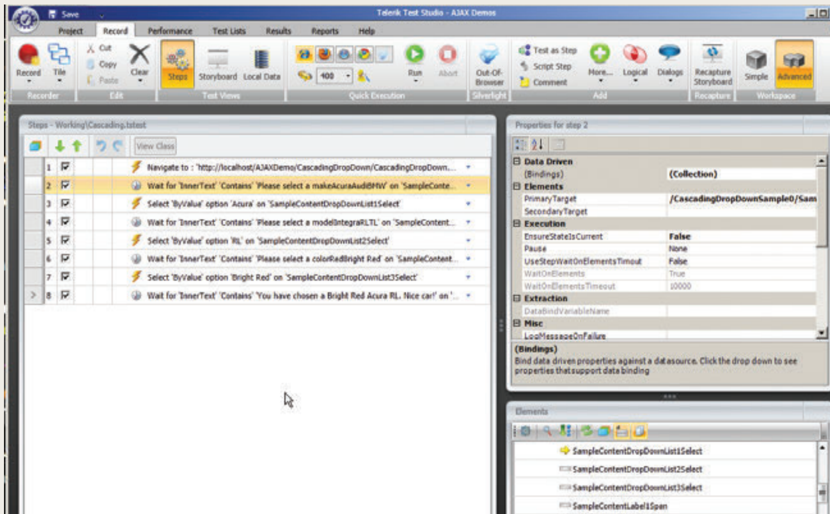

Figure 7: Explicit wait in Test Studio

tags – load with the page but only the top one is populated. When a selection is made from it, the second dropdown is populated with items dependent on that selection. The third dropdown is similarly dependent on the selection from the second.

In this case the framework cannot easily detect when content is loaded into the element. An extra step is needed to define explicitly a condition that must be satisfied before proceeding. In WebDriver create a “wait” class then invoke its Until method, passing the condition for which to wait, defined via a lambda expression (see figure 6). Most tools require a similar two-step approach: wait for the condition then make the interactions. Figure 7 shows it done in Test Studio.

Approaching scripting with a mind to stable actions around dynamic content can save a lot of time and trouble later. The approach applies also to testing many thick-client desktop applications that use the same model of retrieving dynamic content via service calls.

To v or not to v

Automated (and for that matter manual) tests that interact with GUIs will probably always be somewhat brittle and hard to maintain compared to lower-level tests. Evolution of frameworks and tools has made creating them easier: automatic recording of user actions, in particular, has made great strides forward. The downside is that it can be used, easily, to create too many low-value, high-maintenance tests, so that testing spirals out of control. It's important to have a strategy for deciding what and what not to automate.

High-value tests that are worth automating usually include “the show me the money path” – the transaction from which the application owner receives revenue. High-risk functionality, for example permissions control and regulatory compliance, are also strong candidates: where failure could lead to liability or other severe consequences, automation for regression testing is a must, however difficult. Where the system under test depends on third-party components or services, tests involving them should be automated: not to test them, but to detect failure-causing defects in the SUT caused by updates or outages. In testing, if not in finance, past performance is a guide to future performance: regression defects found previously tell you clearly what tests to automate to detect them if they happen again.

It's well understood that tests which will be executed few times should not be automated, although which ones those are is not always easy to predict. But experience teaches us which attributes

of the test item are best left to manual, and especially visual, testing. Figure 8 shows a grid for which I've designed automated tests (see also http://demos.kendoui.com/web/grid). The user accesses grouping and sorting functionality by dragging and dropping column headers. I've concentrated on that functionality.

The grid is on a page with many alignments, icons, styles and layout tweaks. I've specifically avoided verifying those in my tests, not because of the extra time it would take me to implement, but because the tests would then be brittle and take up too much of my time when the page changes: I say “when” because these things always change all the time. Instead I use a simple script that

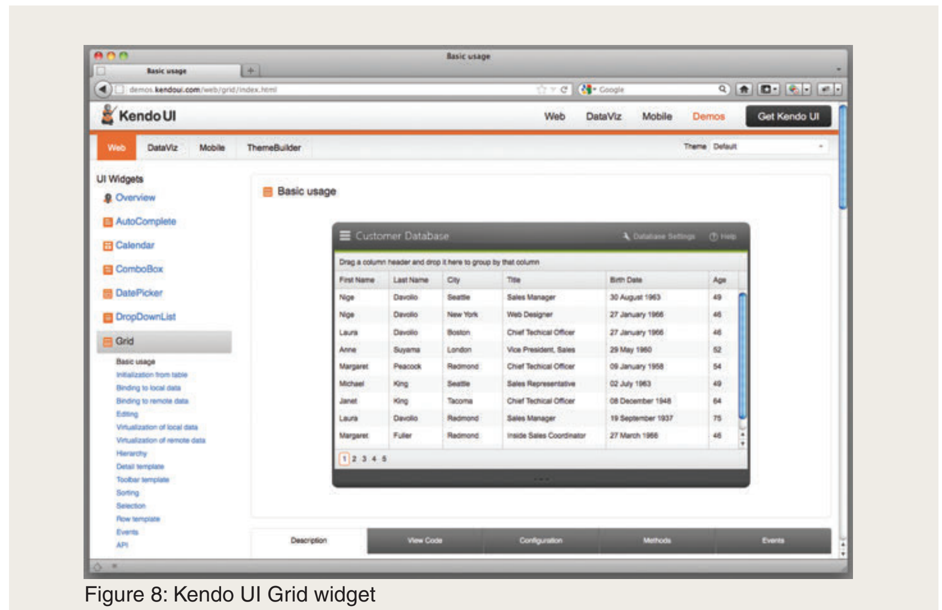

Figure 8: Kendo UI Grid widget

displays the page alongside an image of how it was supposed to look last time I checked and compare the two visually (and carefully). If I see a difference I raise an incident or, if I'm not sure, ask the web designers whether it's deliberate. When necessary, I replace my reference image with one of the current correct page ■

Jim Holmes is an evangelist for Telerik Test Studio. A free trial is available at http://telerik.com/automated-testing-tools

---

SUBSCRIBE
It's FREE
for testers

professionaltester.com