Coded UI Testing Guide

For Large Projects and Teams

Wednesday, October 03, 2012

Visual Studio ALM Rangers
Casey O'Mara, Mathew Aniyan, Richard Albrecht, Tim Star, Richard Fennell, Christofer Löf

Microsoft Corporation

Visual Studio ALM Rangers
This content was created by the Visual Studio ALM Rangers, a special group with members from the Visual Studio Product Team, Microsoft Services, Microsoft Most Valued Professionals (MVPs) and Visual Studio Community Leads.

Table of Contents

Table of Figures

Table of Tables

Foreword by Mathew Aniyan

Coded UI Tests, which we released in Visual Studio 2010, has seen a lot of adoption. It provided a very effective and powerful model for functional UI Testing. It gives a lot of flexibility to the user for building her test projects. As the complexity of applications that are being tested increases, it becomes more and more important to follow a structured approach to building Coded UI Tests. Users have been asking for guidance on how to structure their Coded UI Tests so that it can scale to large projects and teams.

The Visual Studio ALM Rangers are a group of people who have a lot of experience in building real-life Coded UI Tests for large projects. They have come together to build this guidance which will help users with such critical questions as

1. Where in the SDLC should I think about Coded UI Tests?
2. How do I design my Coded UI Tests so that they are maintainable?
3. How do I structure my test projects so that large teams can work on them?
4. What are the best practices while building Coded UI Tests?

The detailed guidance provided in this document, along with its companion Hands-on-lab will be an invaluable resource for anyone who is using Coded UI Tests for testing their applications.

Mathew Aniyan (Senior Program Manager, Visual Studio ALM)

Introduction

Overview

> This guidance should be used in conjunction with documentation that accompanies the product and Microsoft Developer Network (MSDN) at http://msdn.microsoft.com.

Visual Studio does a fine job of generating code from action recordings or by using the Coded UI Test Builder while interacting with an application. This generated code will work well for testing a small application with a small QA team. If you are working on a larger team or application and want to share the responsibility of creating and maintaining the Coded UI Test assets, you should consider following the below guidelines. Using Multiple UI Maps, naming conventions, and a combination of generated and hand edited code will allow you to better organize your UI testing assets to make testing a large application with a team approach much easier. This approach will require some code on your part but, much of the glue to support multiple UI maps, promote code reuse, and better organization is straight forward and does not require an expert command of your .Net language of choice. You will also get guidelines on designing robust Test Automation Frameworks to handle larger engagements Capabilities of getting the test automation results reported as part of Test Manager (MTM) for better reporting.

Visual Studio ALM Rangers

Visual Studio ALM Rangers is a special group with members from the Visual Studio Product group, Microsoft Services, Microsoft Most Valued Professionals (MVP) and Visual Studio Community Leads. Their mission is to provide out of band solutions to missing features and guidance.

This guide is intended for Microsoft “200-300 level” users of Coded UI Test feature in Visual Studio. They are intermediate to advanced users of Coded UI Test features in Visual Studio and have in-depth understanding of the product features in a real-world environment. Parts of this guide might be useful to novices and experts, but they are not the intended audience for this guide.

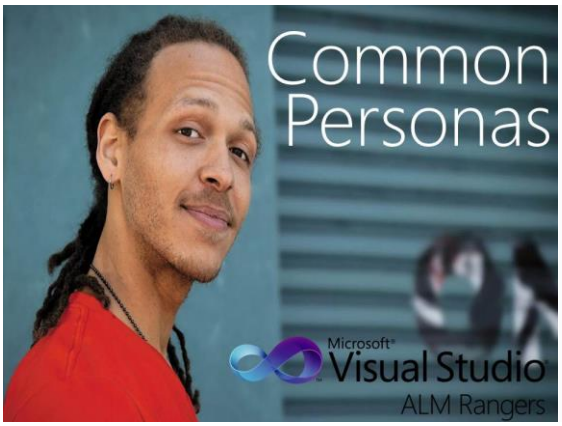
Understanding the Epics and Personas

Overview

This guidance is based on hypothetical customer profiles, personas, and scenarios (user stories). The intention is to demonstrate, in a realistic and convincing way, how personas leverage technology, along with this guidance, to perform a task and create a quantifiable outcome within their environment.

Personas

Refer to **Visual Studio ALM Rangers Personas and Scenarios** [1] for more information on the personas.



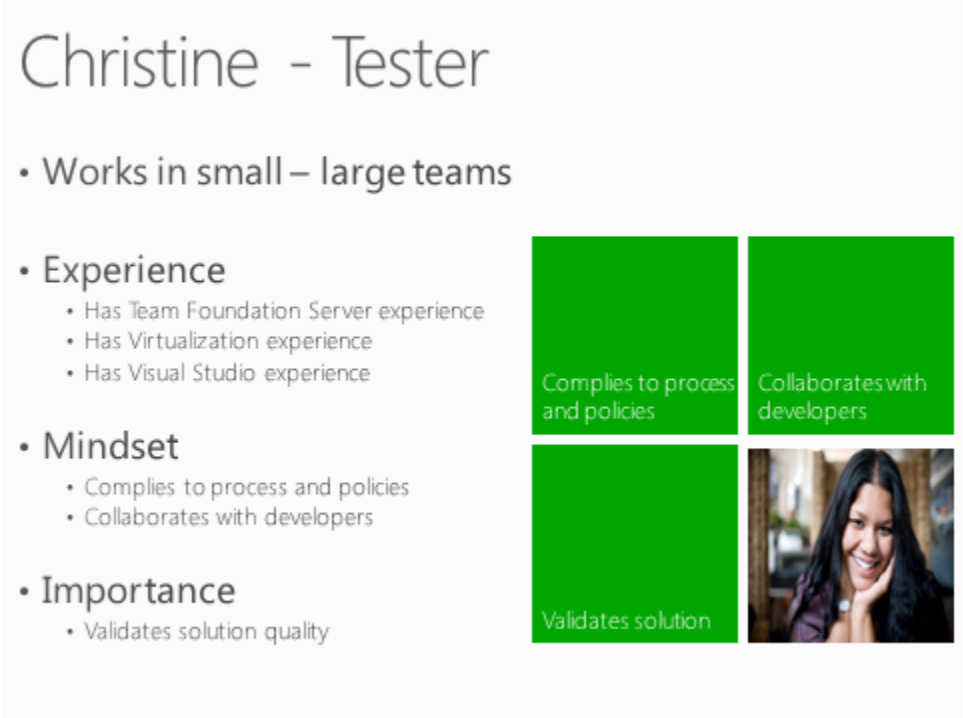
Customer types

Refer to **Visual Studio ALM Rangers Personas and Scenarios** for more information on the customer profiles.

[1] http://go.microsoft.com/fwlink/?LinkID=230942

Scenarios and Guidance Cross Reference

Christine - "Tester"



Scenario	Refer to page
I would like to understand how to deal with UI changes during the SDLC process	11
I would like to understand how to build Coded UI Tests that can be shared across UI components for testing	14, 23 and 26

Table 1 - Christine Scenarios to guidance mapping

Managing Coded UI in the SDLC process

Software Development Lifecycle (SDLC)

The SDLC or software development life cycle covers a vast many different processes such as: “Waterfall”, “Agile”, “Scrum”, “XP”; just to name a few.

For this process guidance we’ll reference a generic SDLC waterfall process with a few basic parts. I’ll use the following image as an example:

Figure 1 - Sample SDLC Phases “Waterfall”

Each of these phases can obviously impact your automation, but there are a few that can impact your automation the greatest in a waterfall SDLC. This may vary based on your implementation.

Initiate Delivery:

There shouldn’t be much automation change here unless you’re in a v.Next situation in which case this is the first place you should start to look at what could potentially go wrong in your automation solution. There isn’t a lot you can do until the changes come in, but you can start looking here first.

Requirements

Things start to firm up a bit with requirements. You should start looking at your user stories, features, and requirements and get some idea of how you’re going to plan your automation. If in a v.Next you should look at what is new work, and blending that in, while keeping a watchful eye on what impacts what you already have.

Design

This is where your objects start to come into play. Hopefully you can get a POC (Proof of Concept) or potentially a rough outline of objects / screens that you can start to plan your object map with.

Build

In build you’re really looking to start getting your hands on code drops, and as features/requirements are complete you should be hopefully putting your functions and methods together loosely. We’ll go into this more as we talk about automation impacts, and collision/collusion later in the guidance.

Stabilize

This is where you’re bound to see the most churn. It’s normally not in build where you think you’d see the most change, but it really tends to be in stabilize where you’re getting feedback, change requests, and bug fixes that you see the most problems. All of these changes are occurring while you’re trying to test the product and get it out the door again compounding problems of spending time fixing your automation (which is sometimes is more expensive than just running it manually) to get it out the door. This is where building and thinking about your automation up front becomes the key, because the number one killer of all automation is rework.

Deploy

By this point hopefully you’re past the churn stage but not out of the woods yet. You will still need to keep a close eye on those last minute change requests that tend to not be big sweeping changes, but usually cosmetic, and object oriented. So while much fewer of these are generated, you’ll find that most target your precious UI Map and objects.

Close

This is typically your maintenance, hotfixes, QFE's, and other assorted fixes. Keep an eye on these, but usually they are a fix to functionality, and hopefully not much more. You'll tend to see smaller, detailed bug fixes, and less risk factor bugs as you can't afford long cycles, and need to get something out to your users much quicker.

Automation Impacts

In some cases changes can have a minimal or very devastating automation impact. To minimize this you will want to get as ahead of the changes and as close to development as you can. In some cases test driven development will help you get ahead of these and insure that the code is meeting those test requirements.

ALERT | WARNING

To help reduce these automation impacts you'll want to try to compartmentalize your code in such a way that you create functions or methods that you can easily change in your code.

One way to do this is to use your recorded actions in Coded UI and save those out to be used in a test case. Take for example you're going to run a quick automation script that opens a website, does something, then verifies that what you expected to happen has occurred:

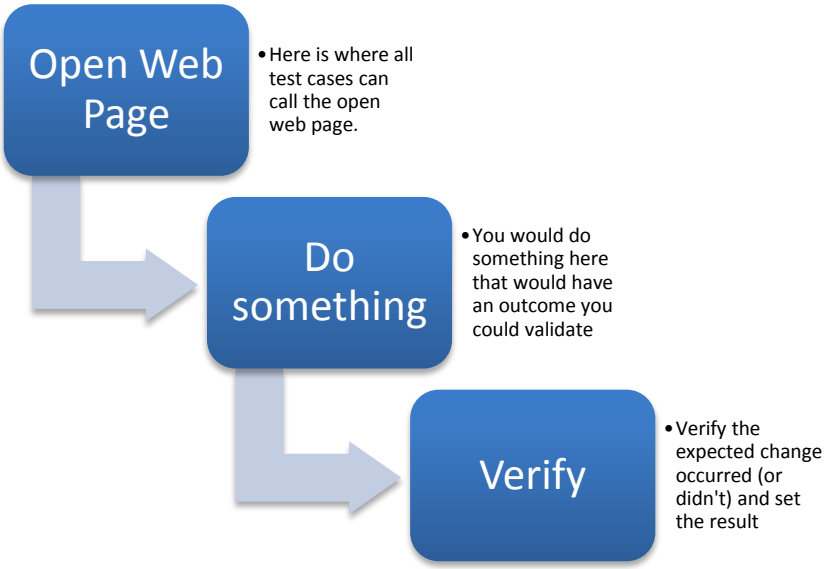


Figure 2 - Generic Test Case Example

In Visual Studio that would look like the following test case example:

```
public class CodedUITest1
{
    public CodedUITest1()
    {
    }

    [TestMethod]
    public void CodedUITestMethod1()
    {

        this.UIMap.OpenExplorer();
        this.UIMap.OpenFabrikamFiber();
        this.UIMap.CreateNewTicket();
        this.UIMap.ValidateSomething();
        this.UIMap.CloseExplorer();
        // To generate code for this test, select "Generate Code for Coded
        // For more information on generated code, see http://go.microsoft
    }
```

Figure 3 - Generic Coded UI Test Case

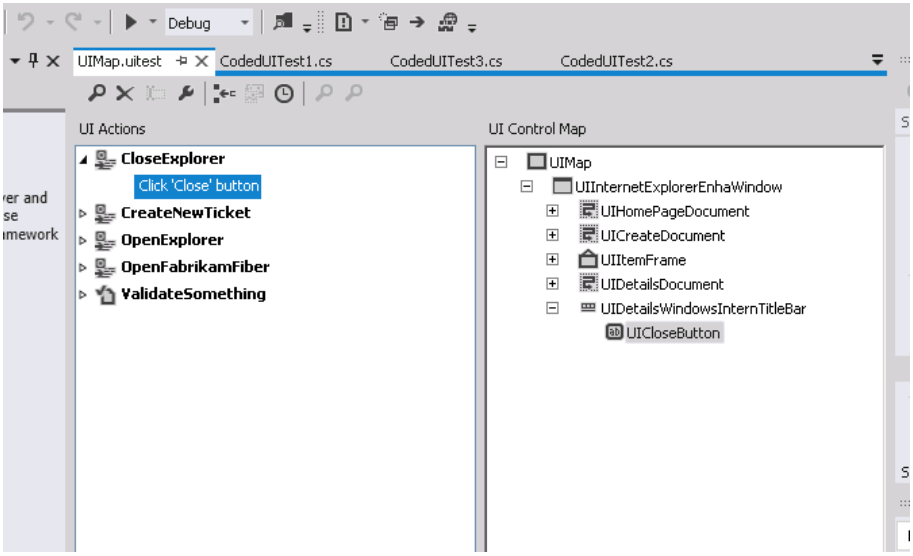


Figure 4 - UIMap.uitest Example

By keeping the functions that you would call repeatedly “shared” you can effectively change one, and it would change all the test cases which call that function.

Summary

You will never be able to stop change from happening, but when recording your test automation and planning you can reduce the impact to your automation significantly by compartmentalizing and making it where the only unique items are single pieces of code in a test case. If it’s used more than twice try to roll that up into something you can easily change.

Expect change to occur. It’s how you adapt to that change, and can quickly react. Updating every test case for every change won’t let you do this effectively, but updating a few methods or functions will. Using UIMaps and UIActions effectively will greatly reduce this effort.

Sharing Coded UI Tests for shared UI components

Coded UI Map Changes

Add UI Object to your UI Map

As you add new UI actions to your UIMap, you will add more objects effectively to your UI Control map as seen below:

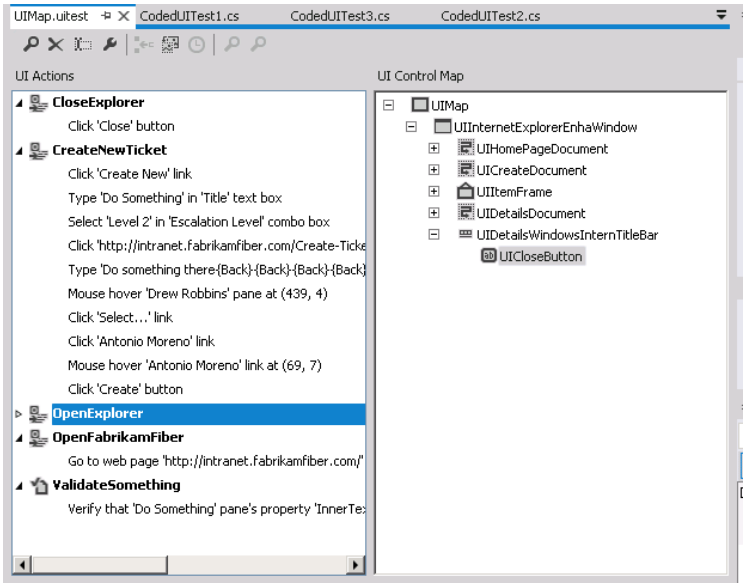


Figure 5 - UI Control Map

Sometimes you may need multiple maps and will want to keep pieces specific to an area in those UI Maps to avoid one large map. Be very wary of adding too many, making maintainability a problem, and potentially duplicating objects/actions. You will want to have a clear plan or "map" of what your project looks like, and how those go into your map.

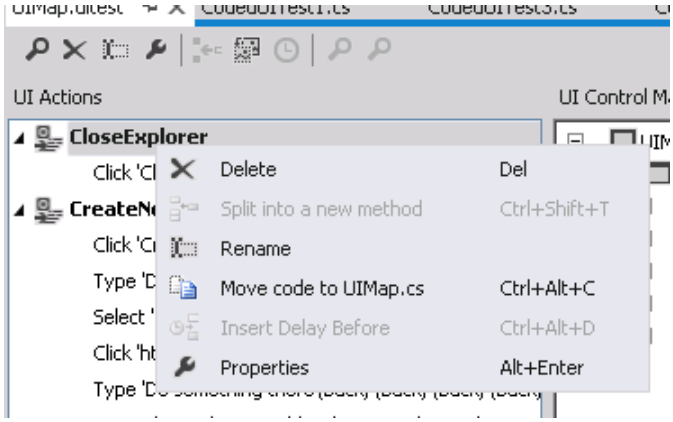


Figure 6 - Right Click UI Action options

Create, Update, Delete

To do this you will want to add a new map to your project by doing the following:

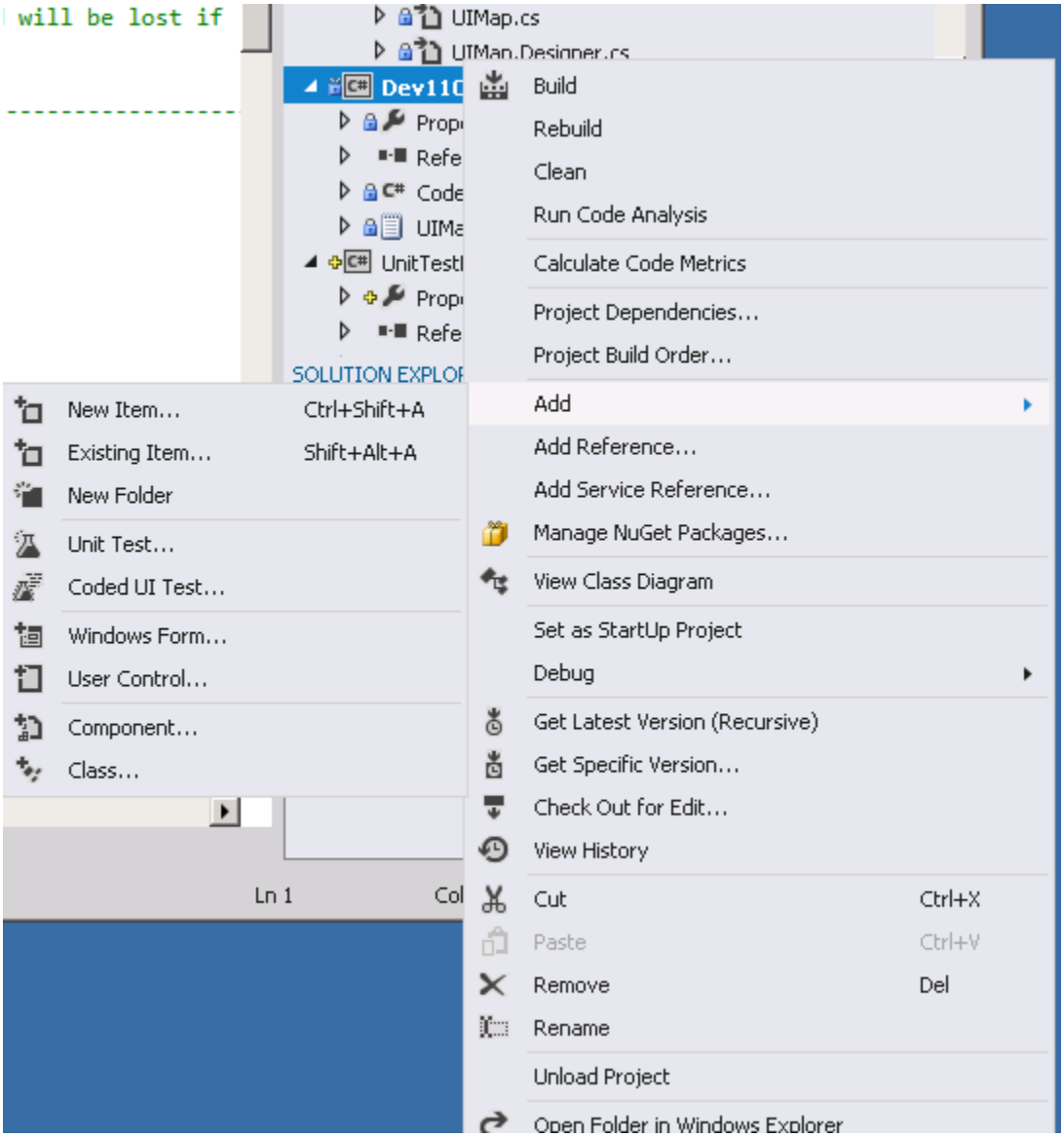


Figure 7 - Adding a UI Map

You will want to select your test project -> right click and add a new item.

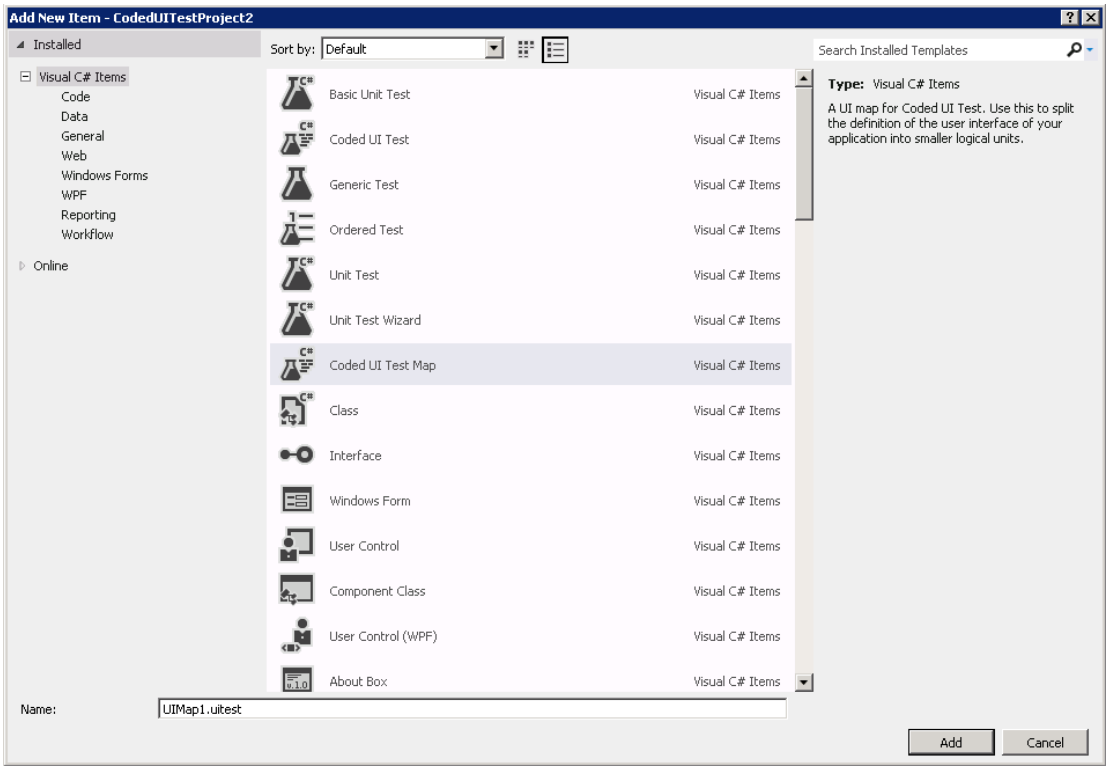


Figure 8 - New UIMap Option in Add Item

Under that item add a new Coded UI Test Map into your project.

You have the ability to locate the controls in your UI Map and find those that aren't currently available. You can do this by right clicking and selecting this object, or locate all objects as seen below.

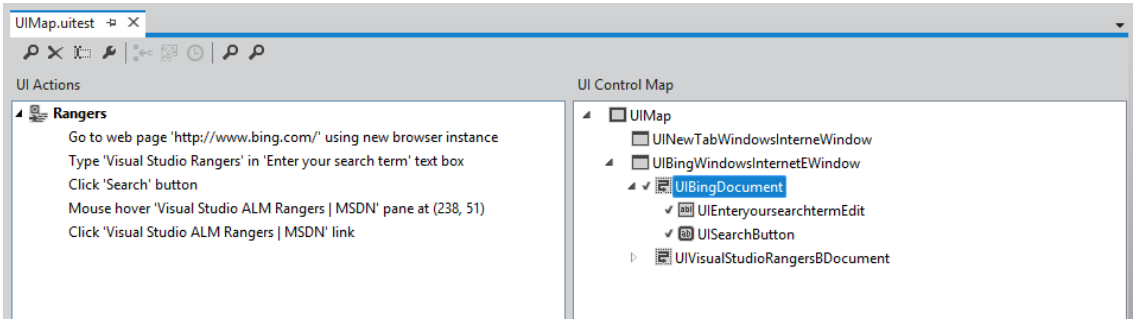


Figure 9 - Locating objects

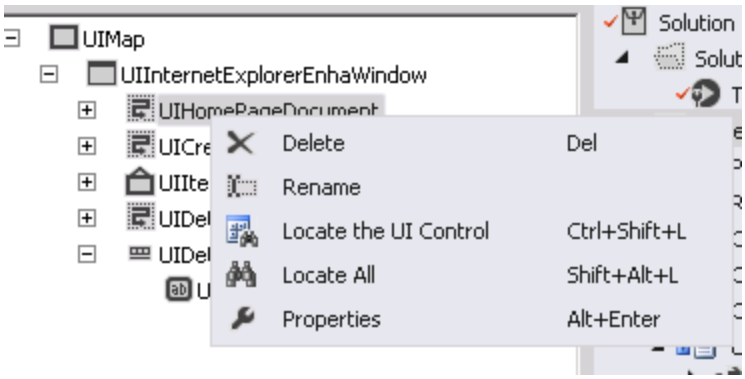


Figure 10 - Right Click Control Map in UIMap

By keeping redundant objects in the UI Map you are leading to object / code bloat and this will become a maintenance issue. Prune the tree regularly to ensure the UI Map is clean. Make sure you update your actions if they were pointed at those old objects.

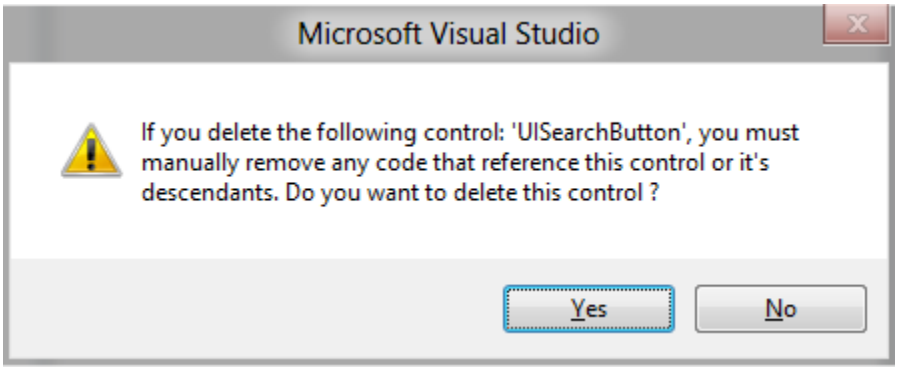


Figure 11 - Visual Studio Warning

ALERT | WARNING

There are many dangers involved in deleting maps. Make sure you don't have any test cases that depend on the code being removed.

Objects

Objects are kept in the right side of the UI Map designer as seen below.

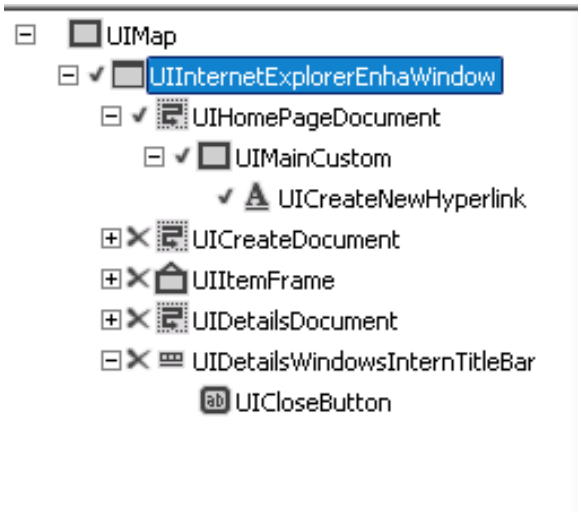


Figure 12 - Locate all controls

With all of these objects are some really good properties that help tell you more about this object.

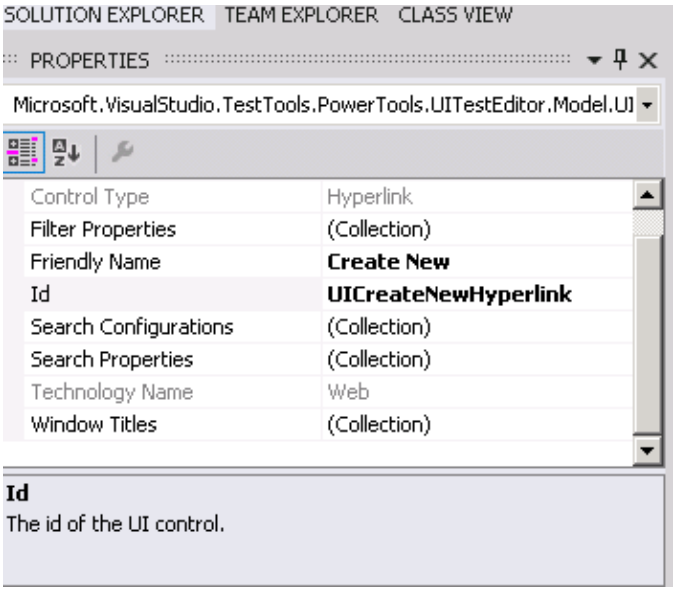


Figure 13 - Object Properties

As you can see from the highlighted area for search properties this will tell you a lot about how Coded UI Test is finding your object, and what it's currently using.

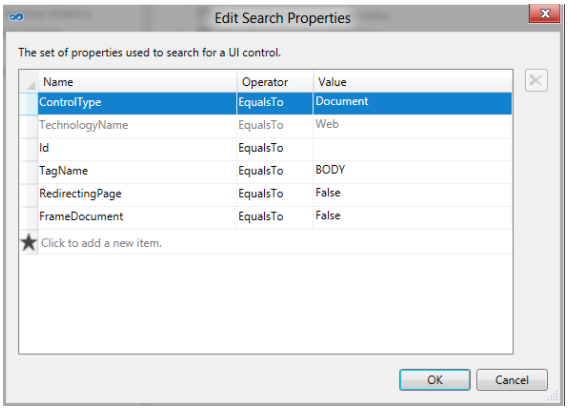


Figure 14 - Search Properties

In this case it's a tagname equal to the body. It also contains a friendly name of Bing and an Id of UIBingDocument. When looking at 2 objects or replacing the reference of one with another – make sure you have the right object, and that in the properties it is that one you're looking at.

UI Actions

The UI actions are probably the easiest to maintain, however you have to have a unique name for each of them, just like a test case method. You will want to again plan these out for use in your test cases. Later in this guidance we'll talk more about cohesion vs. coupling and how you will want to segment these out for maintenance, and the ability to easily update these.

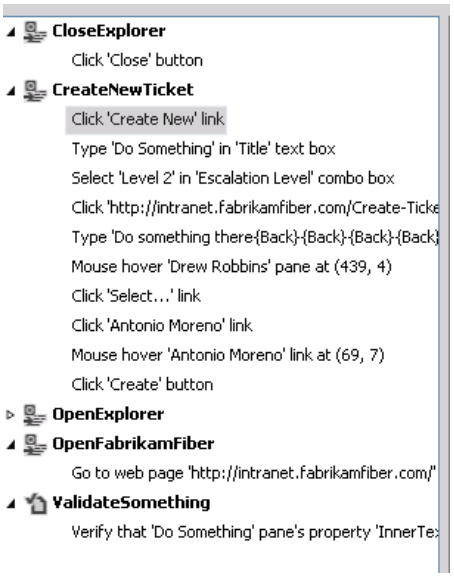


Figure 15 - UI Actions

Validations

Validations as seen above will quickly assert on objects that you will want to make sure contain the expected result you wanted. Validations are marked with a checkmark versus a record button in the UI Actions pane of UI Map.

You can add another validation by right clicking in your code where you generated your code for UI test and selecting the Coded UI Test builder again. You can go right to that object and drag the crosshairs to what you want to validate and add that to your code and UI Map.

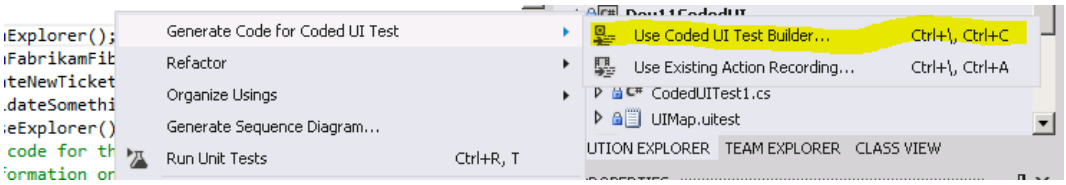


Figure 16 - Coded UI Test Builder

Data Driven coded UI test updates

Data Driven test cases are great because they can remove the hardcoding and will allow you to use XML, CSV, or a Database to drive your automation.

NOTE

There is a great article on MSDN article from Mathew Aniyan around how to do this:

http://blogs.msdn.com/b/mathew_aniyan/archive/2009/03/17/data-driving-coded-ui-tests.aspx

http://blogs.msdn.com/b/mathew_aniyan/archive/2009/04/16/more-on-data-driving-coded-ui-tests.aspx

We can't say it any better except to add that removing the hard coding, and getting to a data driven model will help drive more value out of the test cases, and allow the team to do more automated testing around negative tests, boundary test cases, and min/max testing among other types with the same code, and just changing data.

Friendly naming of objects

If you choose to start renaming methods, or otherwise updating objects be aware that you will have to manually modify that references this. This means that you will have to go back and backtrack these changes.

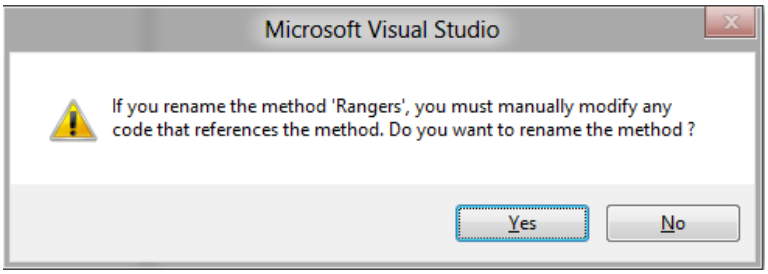


Figure 17 - Rename Object

In some cases a quick control + F will help you find all the instances in your project and you can quickly update these with quick replace. There are also some fine tuning pieces under Find options if you want to match case, or use regular expressions or wildcards.

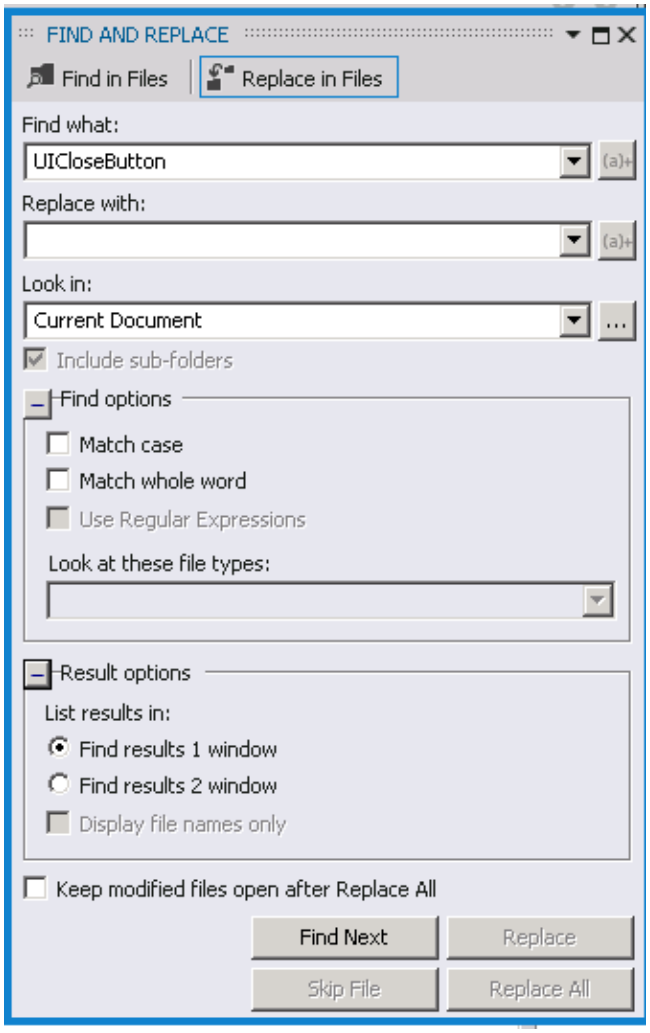


Figure 18 - Find and Replace (Quick Replace)

Also keep in mind the same thing applies to UI Control Map as well as seen below:

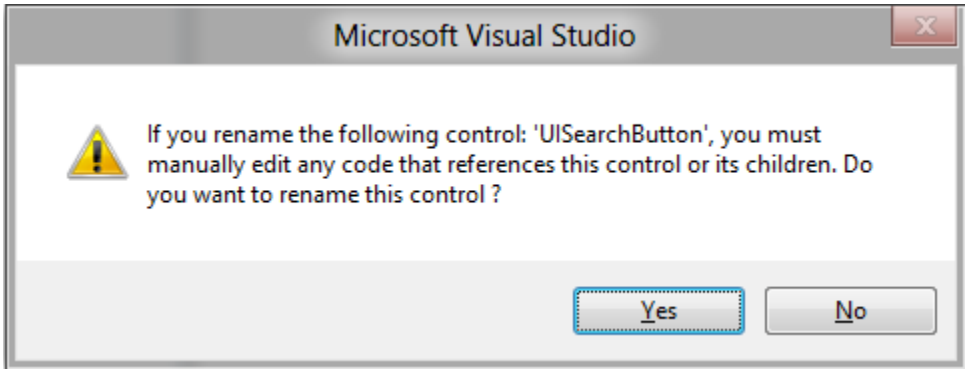


Figure 19 - Rename Control

Splitting your current test methods

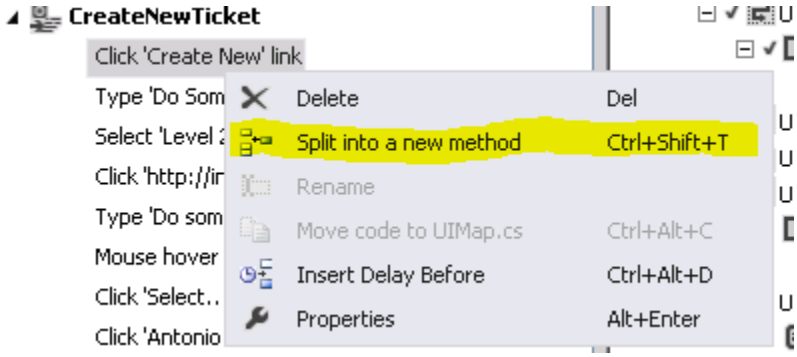


Figure 20 - Splitting Test Methods

If you do have multiple things in your actions, and you need to go back and separate these out it's very simple to do. You can use the split into a new method option. Be aware however that if you do this it's under your control again, and you'll have to update these. One thing you could do would be to split all of them out under a UI Action and then Quick replace that with the new UI Actions you split it out into. Coded UI Test will not manage this for you. These options require you as the owner to update and maintain these.

Writing your test cases

Modular Design

Here is a very small example of a web site that has 2 pages. On page 1 there are 2 items you're going to fill in and verify. On the second page there will be one thing to fill in and a single verification.

In modular design we'd break this down into the following:

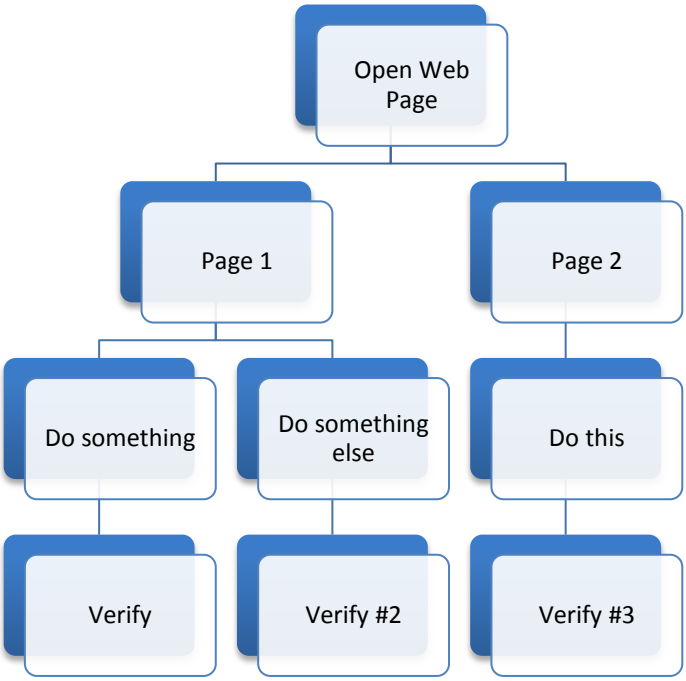


Figure 21 - Test Case Model

Note that we only open the web page with one function. Page 1 and Page 2 navigation were called separately, and then doing something on those were separated for now. Later we could put these together in a base class and reference that, but for now let's keep it simple.

Your test cases would look like this:

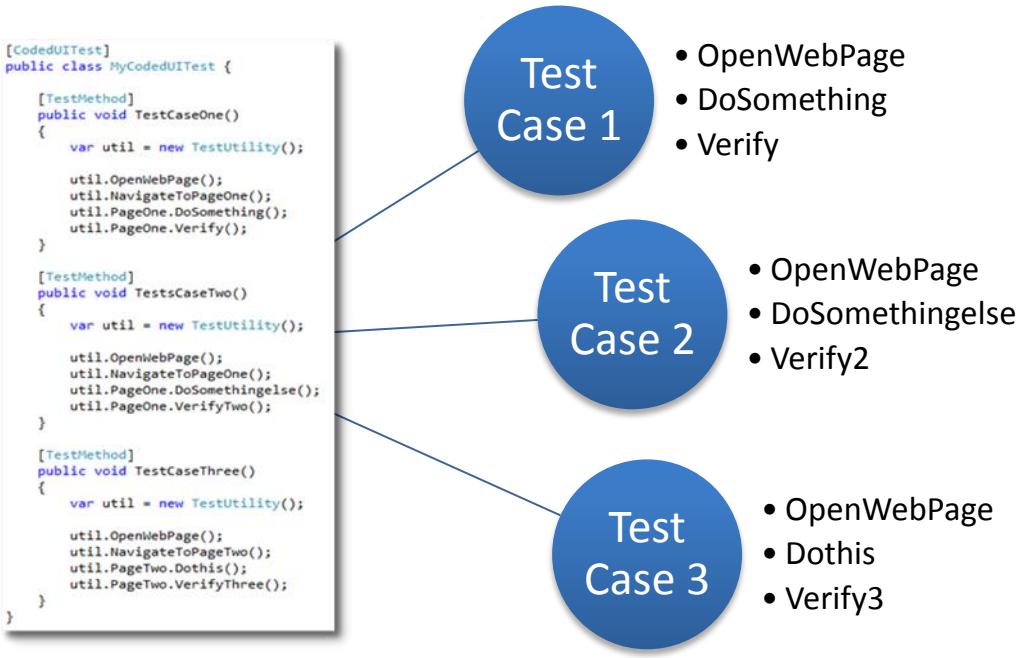


Figure 22 - Modular design

Note that you'd only have to change OpenWebPage in one place to affect all of your test cases. Also your pages are called out so you can navigate easily, then if you wanted to data drive your DoSomething you could easily do this. Leaf nodes would be the only ones that would need to be updated in this model.

With this modular design you could easily change each element if you needed, or insert before or after that element in your test cases. You could also control features or areas of your application under test to create base classes that inherit the items you use.

Cohesion vs. Coupling

It would be really easy for us at this point to start saying "This would be easy to add with this". We could combine these and make it faster. This starts to tie things together that puts you right back into the state you were trying to avoid which is easy maintenance, and the ability to plug items together to form your test case. While the natural tendency is to go this way, try to avoid it!

Cohesion is a much better way to go. Think back to chemistry where items "stick" together to combine different compounds. You're literally looking to do the same. Build your test cases out of these blocks, and stick and change ones as you need. They still form a tight bond, and will produce the output when put together.

The flipside of this coin is around Coupling where it's just like it sounds, and you take 2 pieces of test automation that were separated and combine them. If you've ever seen the coupling on a pipe it's the same concept where you bolt them together forming a rigid pipeline. While at first this seems like a great thing "Hey I would love it my test cases were like pipes!" Its appeal I must admit sounds great at first but would quickly lose its luster when I tell you that you have to dig up your entire front yard to replace a broken leaky pipe. What's more you'd have to do it for every pipe in the neighborhood where you combined them like this. Not so great in hindsight.

So keep them lose – stick them together to form what you need, and allow yourself the ability to stick them together, and quickly replace them, or add more.

Minimize Duplication

This is always something that seems like common sense when you start out, but as you have a bigger team, located in different regions of the world, states, buildings, even floors you quickly find out that duplication happens.

So how can you minimize this? Again going with our example from previously in this chapter we want to try and keep our common items common. You could start out by fielding the pages and splitting these out, mapping your application in a Page Class Library, or even using UML or another solution to capture these pages and objects. UI Map does a great job of this, but as teams get larger and more spread out you'll need to have multiple UI Maps.

See the documentation around multiple UI Maps.

The main point is to map this out so that you'll only have 1 function for each of the objects you want. This will help you avoid many headaches, and having to update, and maintain multiple UI Actions that do the same thing.

Building Coded UI Tests that can be shared across the test team

Project Organization

To successfully build and maintain coded UI tests for a complex application maintained by a team of testers you should establish a well-known project structure and file naming convention. Using the exact naming convention below is not important but establishing and following a naming convention makes it easier for the rest of the team to locate the files they need to work on. Using Pascal casing also makes understanding a files purpose easier. Following is a project structure that you can use:

Project Name: <Application or control under test>.Coded UITests

UI Maps Folder: Put all of your UI Maps in this folder or a tree structure beneath this folder. Name your UI Maps <page/control name>Map

Utility Folder: Put all utility code in this folder

Coded UI Tests: Put your coded UI Tests in the **project root**. Name your coded UI tests something like <area>UITest.

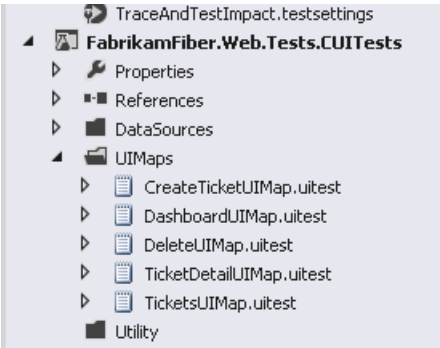


Figure 23 - Project Structure

Multiple Projects

If you are testing a complex application where controls, pages, or other UI assets are shared between different applications, consider creating an entirely different Visual Studio test project for those components. UI Maps, Coded UI Tests, and other Coded UI Test files ***do not*** need to reside in a single project.

UI Maps

Each UI Map is a combination of 3 files:

- **FileName.uitest** – this is an xml file that should only be edited by double clicking on this file and using the UI Map Editor
- **FileName.cs** – This file contains custom code and may be hand edited. It will be blank when it is initially created.
- **FileName.Designer.cs** – This file contains generated code. Do not edit this file as you changes will be overwritten any time visual studio regenerates this file.

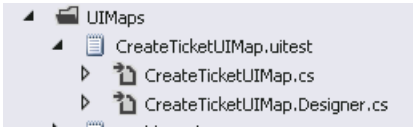


Figure 24 - UI Map Structure

ALERT | WARNING

Do NOT modify code in the *UIMapName*.designer.cs file. Your changes may be lost the next time the designer file is generated. The designer file is regenerated whenever you save the UITest file and when you click on Generate Code in Coded UI Test Builder.

Why Multiple UI Maps

Create Multiple UI Maps to better support a more modular approach to testing. By creating a UI map per complex control, page, or screen the testing team can be working on different UI maps at the same time without worrying about conflicts with other team members. Multiple UI maps will make your test project easier to fix in the case of a breaking change to the UI because you do not have all the UI code in one giant monolithic UI Map. Even if you are using multiple UI Maps, you can still add new controls to the map by using the Coded UI Test builder.

Generating UI Map Code

Just because we are taking more control of our test project away from the code generator does not mean we will not leverage the code generation capabilities. In fact, the vast majority of the code in our UI maps will begin or remain as generated code.

Generating methods and assertions

Consider using Pascal casing and adding comments for each method that is generated. The comment will be inserted into the source code and makes maintaining the test easier.

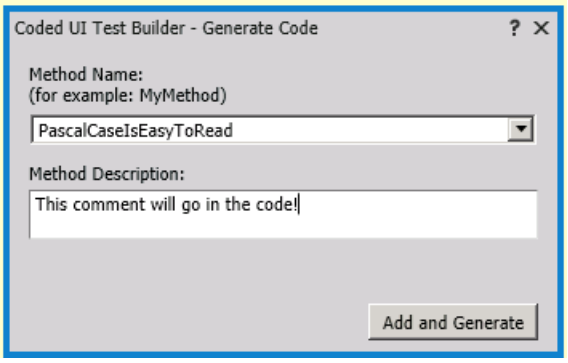


Figure 25 - Generating Code

Add a failure message when adding assertions; this makes trouble shooting test failures easier as the failure message is meaningful:

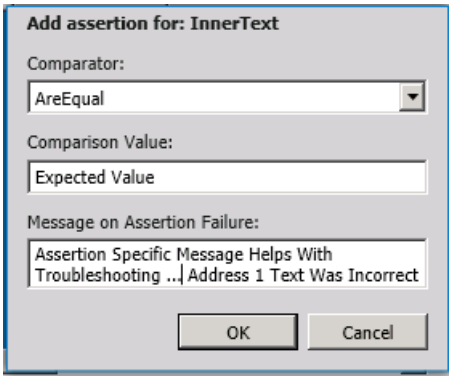


Figure 26 - Generating Assertion

NOTE

Using Pascal case and verbose method names, adding comments and custom assertion failure methods makes developing, debugging and maintaining Coded UI Tests easier.

Adding UI elements without assertions

You may add a control to the UI map without adding an assertion by expanding the Coded UI Test Builder and clicking the Add control (Alt+C) to UI Control Map button

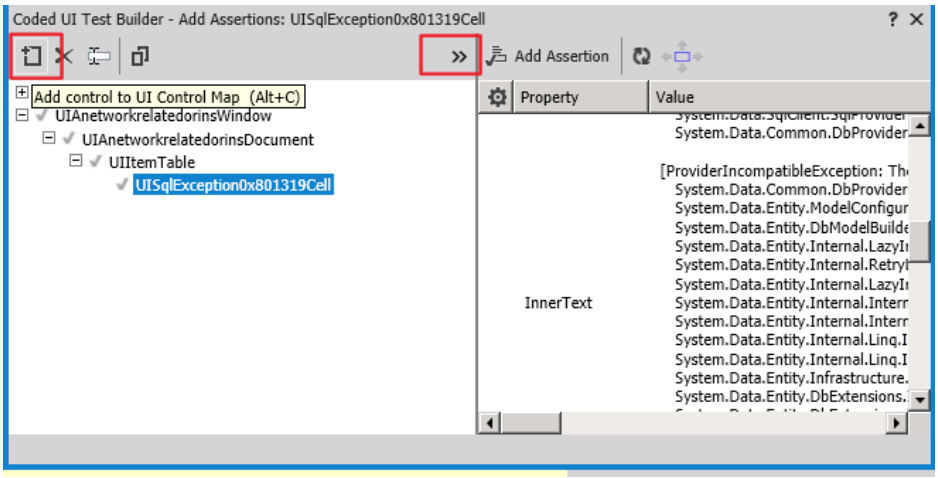


Figure 27 - Adding controls to UI Map

Tying the UI Maps Together

We could reference and call each of these maps individually and successfully test all of our scenarios without a problem. The weakness in doing this is that we have quite a bit of duplicate code in our generated UI maps. The search criterion for each control begins with the top level window. This search criteria are used by the test execution engine at runtime to search for the correct window that each of the controls reside in (top level ancestor). A single change to a window could break a bunch of things in a UI map and force us to regenerate all of the code. Rather than doing that we are going to create a new Class called TestRunUtility. The role of this utility is

going to be to tie all of the UI maps together and force them all to use one single top-level ancestor search algorithm. This can be accomplished by using the UITestControl.CopyFrom or BrowserWindow.CopyFrom method.

NOTE

Consider using the **UITestControl.CopyFrom** or **BrowserWindow.CopyFrom** method to force all of your complex controls or browser pages to leverage a single top level ancestor search criteria.

Coded UI Test

Refactoring with the UI Map Editor

The UI Map editor is the main tool you will use to perform the following tasks; deleting generated code, renaming methods or assertions, moving generated code to the editable code location, Inserting delays, splitting methods, and locates UI controls.

Rename

If you are unhappy with the name of your assertions or methods, you can double click the **uitest** file to open the UI Map and highlight the method you want to change. Press the F2 key or click the rename toolbar button and enter a new name.

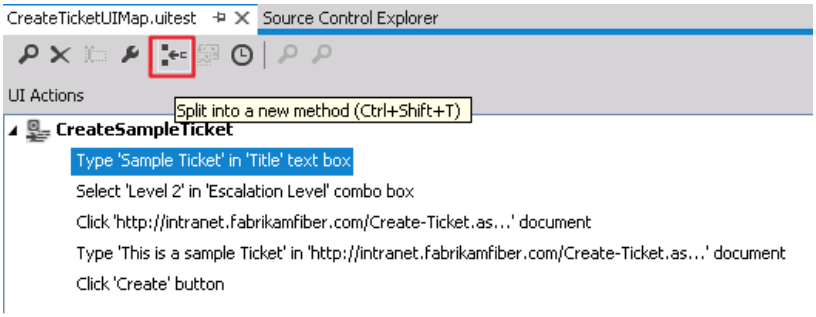


Figure 28 - Rename a method

SplitMethods

The Coded UI Map Editor supports splitting a method into two. Having many small generic methods are easier to reuse and maintain then a few large very specific methods.

NOTE

Having many small generic methods with only a few actions in each method promotes code reuse.

Customize Code

Use the UI Map editor to move assertions and other code you want to modify out of the designer and into the code file so changes are not lost the next time the UI map is generated. Double click on the uitest file that contains the code you want to modify. In the UI map editor highlight the method you want to change -> click the Move Code button (Ctrl+Alt+C) -> click the save all button or ctrl + shift + S.

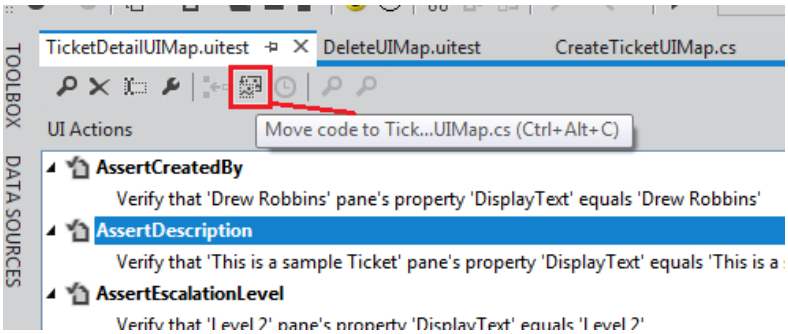


Figure 29 - Moving Code

Now double click the corresponding cs or vb file and notice we have code. We can safely edit the code now without fear that the code generator will overwrite our changes.

ALERT | WARNING

Do NOT modify code in the *UIMapName*.designer.cs file.

Customize Assertions

Override Expected Values to test for results other than those recorded. As an alternative, you can customize an assertion to accept parameters. Writing a completely custom assertion method can also simplify maintenance. Say, for example, we have an address control that is used in many places and therefore many Coded UI tests. Rather than performing an assert for address, state, and zip code in many places, consider writing a single address validator that accepts a parameter for each of the address fields and within the custom assert method, call the individual field assertions.

Best Practice

Test for Errors

Although throwing an error as a part of normal programming flow is generally frowned upon, there are times when it is appropriate. A coded UI test method can and should test that the error is thrown correctly. To do this either use a try-catch block and do an assertion on the error or use the ExpectedException attribute on the test method.

Use Error Handling

Use try-catch-finally blocks in your test method just as you would in your program to make the test method as robust as possible.

Use Databinding

Consider using databinding in a test method to test several different values rather than duplicating a test method and hard coding multiple different values.

Manage the process under test

As a best practice you want to open and close the application under test as a part of the coded UI test. Imagine you are automating dozens of Coded UI tests as a part of an automated build process and that each test launches the

application under test so that it is guaranteed to be in a known good state. If the tests fail to close the application upon completion of each test the server will eventually run out of memory as dozens of instances of the application are left open on the build agent. Another problem with leaving a test open is it may establish state that another instance of the test application will use (like session in a browser application) which can be problematic in navigation. For example in a web application the test script may assume the browser is launched and the user is required to authenticate but if there is already a session established in another browser, the application may assume the user is already authenticated and redirect them to say a home page. To avoid this scenario, each test should launch and close the application under test. Furthermore this should be done in a robust manner so even if the test case fails unexpectedly the application is still closed. To achieve this wrap your test codes in a try/catch/finally block or use the "using" pattern.

```
[TestMethod]
public void CUITMultUsingLAunch()
{
    TestRunUtility utility = new TestRunUtility();

    using (ApplicationUnderTest.Launch(
                @"C:\Program Files\Internet Explorer\iexplore.exe"))
    {
       utility.HomePage.NavigateToTailSpinToys();
       utility.HomePage.ClickModelAirplanes();
       utility.ModelAirplanePAgeObject.ViewTreyResearch();
       utility.TreyRocketPageObject.ValidateHeight();
       utility.TreyRocketPageObject.ValidateWeight();
       utility.TreyRocketPageObject.AddTreyToCart();
    }
}
```

Internet Explorer 10 and HTML 5 support

Beginning with Visual Studio 2012, support will be added for Coded UI testing against web applications hosted in Internet Explorer 10. Internet Explorer 10 will have robust support for the maturing HTML5 and CSS3 standard. The coded UI testing framework will also support these new features. See this MSDN Article for a full list of new features: http://msdn.microsoft.com/en-us/library/bb385901(v=vs.110).aspx

When should I prefer Coded UI testing over other Visual Studio .NET test types

Introduction

In this guidance we will consider Coded UI tests, Unit tests, Web Performance tests, Load tests, and Manual tests using Microsoft Test Manager. We will examine the scenarios when we would prefer one test type over another and under what scenarios to avoid a given test type. Before choosing which approach to use for your coded UI tests, we strongly recommend doing a proof of concept and evaluating all of the following techniques

Unit Tests

The Unit Test is best for testing the smallest unit of source code and includes features in support of this goal such as the ability to integrate Mocks/Stubs/Fakes. In this scenario the developer must have intimate knowledge of the code under test and strong coding skills. The Unit Test framework also work well for integration testing in which case the author may have a lower understanding of the code under test but strong coding skills are still beneficial. Additionally, Unit tests are the preferred method for testing Windows Communication Foundation services. These Windows Communication Foundation service unit tests can and should be used as a part of load testing those services. While it is possible to test web pages with unit tests by including the **HostType**, **UrlToTest**, **AspNetDevelopmentServerHost**, it is not ideal and not the best way of inspecting page responses or testing the user interface.

Web Performance Test

Web performance tests work at the protocol layer. They are designed to send a request via HTTP and receive responses and therefore this test type is only appropriate for applications that use HTTP to communicate between client and server. The HTTP response is what we are inspecting with our validations; we are not actually inspecting a rendered page. This is important to understand as today more and more pages are built with both server-side HTML generation and client-side modification of the HTML. In the Web Performance Test we use extraction rules to dynamically generate the requests and use validations to inspect the response at both the header and content level. Validating UI generated via JavaScript, ActiveX or a plugin is not possible using the off the shelf functionality. The web performance test is exceptionally well suited for recording scenarios for use in load tests. Additionally they are good for; validating request response pairs, testing server response time, validating server generated HTML, validating HTTP response codes, and testing ASMX web services. The web performance test is generally not good for testing user interaction, WCF services, and dynamically generated HTML. It is possible to build very powerful Web Performance tests with little or no coding skills at all though coding skills are beneficial for some of the most advanced scenarios and a solid understanding of Regular Expressions is desirable.

Coded UI Tests

We are going to consider four different styles of Coded UI tests. The first will be referred to as a “standard” coded UI tests. In this method will be using a single UI map. The second method will be referred to as a Coded UI test with multiple UI Maps. In this method we will deviate from the standard test in that the team will be modularizing the application and building a UI map for each module. The final two methods will be referred to as “Code First”[2] Coded UI and “Coded UI Enhanced” (CUITe) Framework[3].

Standard Coded UI Tests

In this method the author allows Visual studio to completely manage the UI Map. With this technique the author prefers the recording and code generation capabilities of Visual Studio to manual coding and code manipulation. Because a single UI map is used, this map will become large and difficult to maintain and is therefore really only appropriate for relatively simple applications with a mature and stable user interface. Additionally, the single UI map makes it difficult for multiple people to checkout this file at the same time and therefore this technique is best suited for a really small team or better yet a team of one. This method may be used to test any application outlined in the platform support matrix[4] and requires the least amount of coding skills though practically speaking

basic programming skills will be required to perform fundamental maintenance of the tests. While it is possible to use any style Coded UI test in a load test it is not practical as each virtual user requires a dedicated test agent[5].

Coded UI tests using multiple UI Maps

In this method the author modularizes the management of the user interface by creating a UI map per "module" where a "module" may be a page, form, custom user control, or any other logical breakdown of the user interface. Even though the author has taken control of the UI map, they may choose to allow Visual studio to manage most of the coding via code generation. Multiple UI maps promote code reuse and are conducive to a team environment in that members of the team may be working in different areas of the application without having to checkout and modify the same UI map. Using multiple UI maps makes this technique better suited for testing more complex user interfaces but this technique is still time consuming and is still most appropriate for mature user interfaces. This method may be used to test any application outlined in the platform support matrix and requires a modest amount of coding skills to assemble the generated code and perform fundamental maintenance of the tests. While it is possible to use any style Coded UI test in a load test it is not practical as each virtual user requires a dedicated test agent.

Code First Coded UI Tests

In this method, the author does not use a UI map and its corresponding control search capability rather a search of the DOM is used to dynamically locate a control. This technique will only work with browser based applications. Performing a DOM level search per control is going to be more resilient to change than using the generated control search using UI maps therefore this technique can be implemented earlier in the life of an application and is also better suited if the UI changes often. The tradeoff is that searching for every control beginning at the DOM will not perform as well as using the first 2 techniques therefore if test performance is a major concern you may not want to use this technique. This technique is appropriate for complex web applications and works well in large teams as there is no need to share the UI maps. This technique does not have the notion of a UI Map Editor and requires expert coding skills to effectively build and maintain robust tests. While it is possible to use any style Coded UI test in a load test it is not practical as each virtual user requires a dedicated test agent.

CUITe (Coded UI Test enhanced) Framework

In this method, the author does not use a UI map rather a completely different repository is used to assist in the location for controls. The object repository may be maintained using a recorder but using the recorder or even a repository is not strictly necessary. This technique will only work with browser based applications and includes some Silverlight support. The amount of code required with this technique is drastically reduced and therefore maintainability should be increased substantially. This technique is appropriate for more complex web applications than technique one and two because there are improved search capabilities especially around testing an html table but it will not be as resilient as the Code First method unless you use the ability to interact with controls without creating an ObjectRepository class which is not recommended by the authors of the framework. This technique will work well in large teams as there is no need to share a single UI map or repository. This technique requires expert coding skills to effectively build robust tests. This technique is dependent on the Coded UI test engine so some lag time between the release of new versions of Visual Studio and CUITe support may be expected though this technique is used by some teams internally at Microsoft it is still an out-of-band solution. While it is possible to use any style Coded UI test in a load test it is not practical as each virtual user requires a dedicated test agent.

Manual Tests with Microsoft Test Manager

Most folks understand that recording manual tests against applications outlined in the platform support matrix and using the fast forward feature is the sweet spot for Microsoft Test Manager but it is useful for testing far more than that. Microsoft Test Manager is excellent for performing exploratory (unscripted) testing, then creating test scripts from the action recording collected during the exploration. Anything that is scripted can be manually executed using Microsoft Tests Manager. This includes executing tests that require physically manipulating a device such as a power supply or testing applications to complicated to script. Favor using Manual tests in

Microsoft Tests Manager until the user interface is mature enough to make it worthwhile to move over to a Coded UI tests. Favor using manual tests for any test that does not contain a user interface.

Load Test

The primary goal of a load test is to simulate many users accessing a server or server farm at the same time. Use Web Performance tests to a load test to simulate multiple users opening simultaneous connections to a server and making multiple HTTP requests. Add unit tests to a load test to simulate multiple users or agents hitting a Windows Communication Foundation service or to load a non-Web based server.

Improving the performance of Coded UI tests

Introduction

Out of the box, the Coded UI test record and playback engine is configured to be as resilient as possible. This configuration comes at the cost of performance. If you find your coded UI tests are taking too long to run there are some simple things that may be done to improve performance and there are also several "not so simple" things that may be done. None of these steps may be blindly taken and guaranteed to speed up test execution without breaking a test. These steps should be executed as more of a tuning exercise.

Programming Best Practices

Set the name Name/ID for all controls that will be used in coded UI test. This is especially important for forms or pages with many of the same control type. Finding a control with an ID is much faster than finding a control based on inner text or some other attribute. For example searching for a link on a web page with a couple thousand links without using an ID can take up to 50 seconds. The same search for a link with an id is sub-second.

Search Tuning – Match Exact Hierarchy

Set **Playback.PlaybackSettings.MatchExactHierarchy = true;** this setting will not immediately do anything to improve performance of the tests that are still passing once the switch is thrown. This setting also makes the test less resilient to change because the playback engine will only look in the exact hierarchy location as it was found before reporting a failure. The idea here is that we throw the switch so that the playback engine only matches exact hierarchy and we should start to see some tests fail (presumably the slower tests will fail). The tests that are failing are going through the longer search paths and we should refactor these wherever possible to use the exact match criteria thus improving the total test. It is possible to turn this setting to true for most tests but leave as false for tests that execute against a more dynamic UI that cannot be refactored to use the exact match.

Search Tuning – Wait For Ready

Consider the impact of **Playback.PlaybackSettings.WaitForReadyLevel.** Using **Playback.PlaybackSettings.WaitForReadyLevel.WaitForReadyLevel.Disabled** will disable the wait for ready feature. Consider disabling this feature across the board then using a timer or selectively enabling the wait for ready feature to tune the test cases that no longer pass.

Playback settings that affect execution time

The following timer based settings may impact test execution time. Consider adjusting these based on your application characteristics and environment. All times are in milliseconds.

The amount of time the playback engine "pauses" between actions. 100ms is default and the minimum value.

```
Playback.PlaybackSettings.DelayBetweenActions = 200; //200 milliseconds
```

The amount of time the playback engine attempts to locate a control, default is 120 seconds.

```
Playback.PlaybackSettings.SearchTimeout = 20000; //20 seconds
```


The amount of time the playback engine will wait for a control to be ready. By Default, 60 seconds.

```
Playback.PlaybackSettings.WaitForReadyTimeout = 30000; //30 seconds
```

Think Time multiplier may be used for slow machines. Assign a value greater than 1 so that the recorded think time is uniformly increased to handle slower application responses.

```
Playback.PlaybackSettings.ThinkTimeMultiplier = 2;
```

Identifying slow searches

Use the HTML logger to troubleshoot performance issues related to:

- Smart match
- Unnecessary mouse hovers with and without 'Continue on error'.
- Wait For Ready related delays
- Skip Intermediate elements activation

To Enable the HTML logger set EqtTraceLevel > 0 in the QtAgent32.exe.config file.

- For EqtTraceLevel value >= 3, screenshots are taken for each action.
- For EqtTraceLevel value 1 and 2, screenshots are taken only for error actions.

```
<system.diagnostics>
  <switches>
    <!-- You must use integral values for "value".
         Use 0 for off, 1 for error, 2 for warn, 3 for info, and 4 for verbose. -->
    <add name="EqtTraceLevel" value="4" />
  </switches>
</system.diagnostics>
```

If you want to disable the screenshot creation irrespective of the EqtTraceLevel level, add the following key entry in the configuration file

```
<appSettings>
  <add key="EnableSnapshotInfo" value="false"/>
  <add key="StopTestRunCallTimeoutInSeconds" value="5"/>
  <add key="LogSizeLimitInMegs" value="20"/>
  <add key="CreateTraceListener" value="no"/>
  <add key="GetCollectorDataTimeout" value="300"/>
</appSettings>
```


ALERT | WARNING

The way that the HTML Logger is enabled changed between the RC and the RTM of Visual Studio 2012. In the RC an EnableHtmlLogger key also had to be set. The process for enabling the feature in the RC is as follows:

To Enable the HTML logger in the release candidate (RC), set EqtTraceLevel > 0 in the QtAgent32.exe.config file. Set EqtTraceLevel > 3 to generate screenshots for each action.

```
<system.diagnostics>
  <switches>
    <!-- You must use integral values for "value".
         Use 0 for off, 1 for error, 2 for warn, 3 for info, and 4 for verbose. -->
    <add name="EqtTraceLevel" value="4" />
  </switches>
</system.diagnostics>
```

We must also set EnableHtmlLogger=true to enable the html logging feature.

```
<appSettings>
  <add key="EnableHtmlLogger" value="true"/>
  <add key="EnableSnapshotInfo" value="true"/>
  <add key="StopTestRunCallTimeoutInSeconds" value="5"/>
  <add key="LogSizeLimitInMegs" value="20"/>
  <add key="CreateTraceListener" value="no"/>
  <add key="GetCollectorDataTimeout" value="300"/>
</appSettings>
```

Here is a screen shot of the HTML Logger and I can immediately see that mouse hovers are being executed and I can see an example of Wait For Ready delays by looking at the timing in the left hand column:

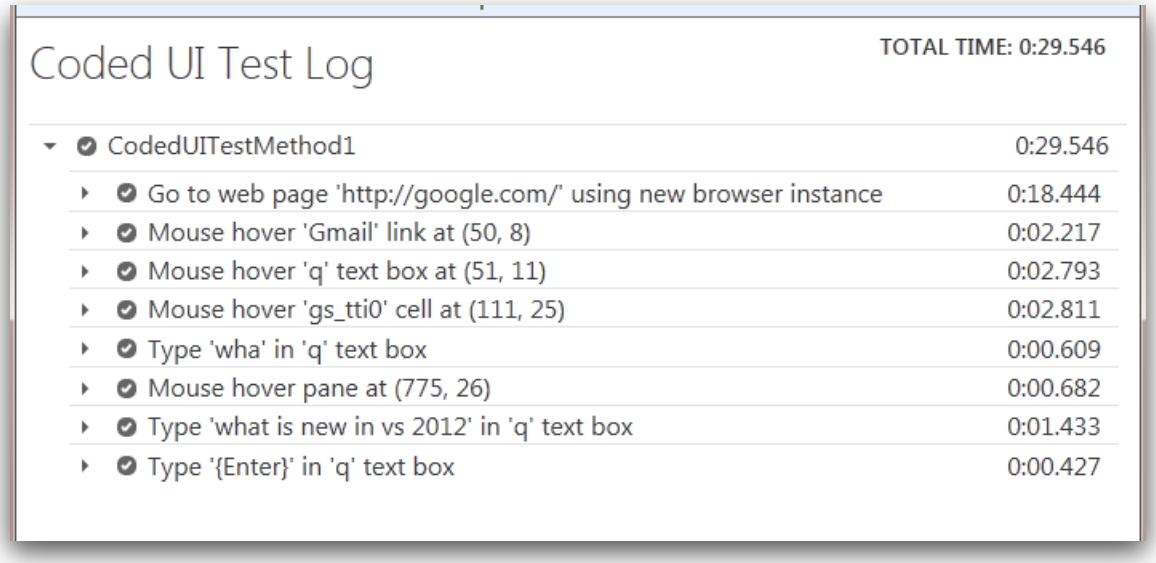


Figure 30 HTML Logger - Mouse Activity

Here is an example of a control not being found but continue on error is set so the playback engine moves on.

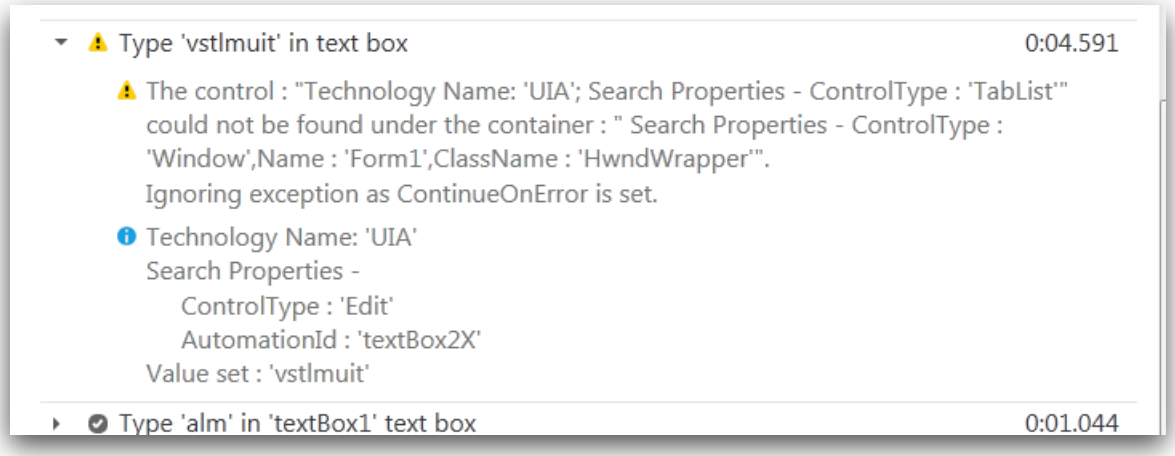


Figure 31 HTML Logger – Continue on Error

Here is an example of Smart Match being used. The playback engine is essentially telling us that it didn't find exactly what it was looking for but something close. Also Notice this step took over 19 seconds to complete.

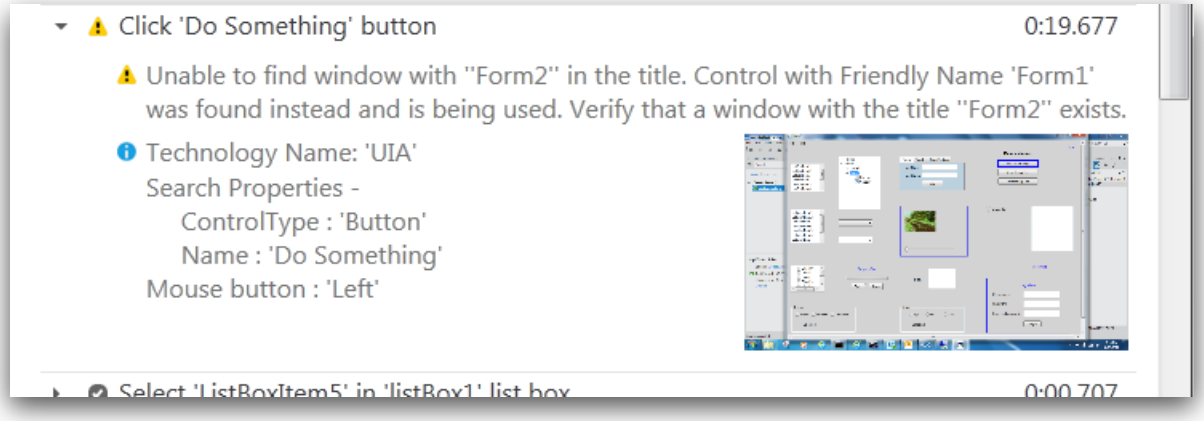


Figure 32 HTML Logger - Smart Match

This image demonstrates the output when MatchExactHierarchy is enabled.

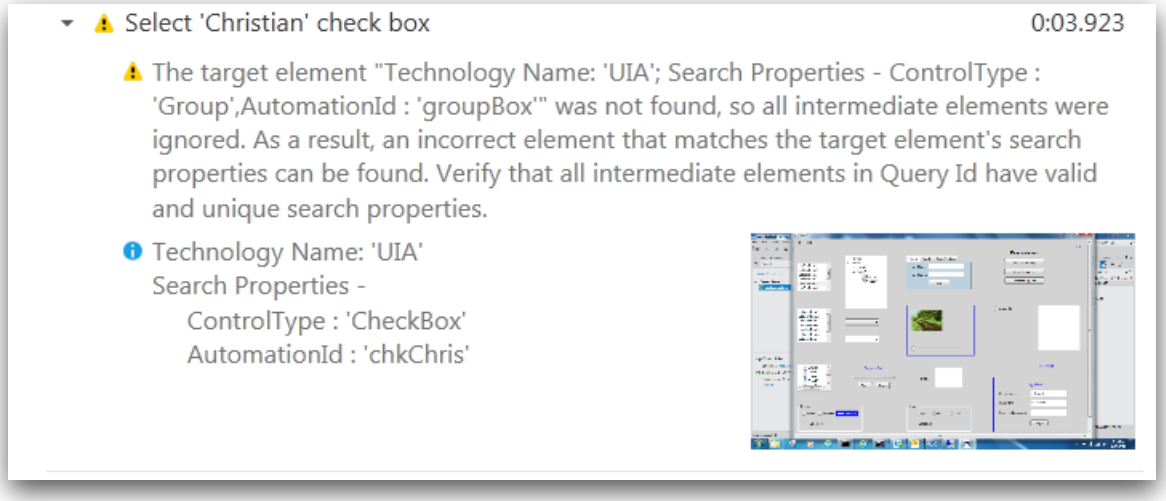


Figure 33 HTML Logger - Match Exact Hierarchy

Avoid recording unneeded actions:

When testing a web page for tooltips or menu changes, the recording engine looks for property changes during mouse moves to figure out if there's a need to record a hover. The default recording engine behavior may result in a lot of hovers being recorded. If you do not wish to test hover behaviors, modify the CodedUITestBuilder.exe.config file and add the following

<add key="ImplicitHoverLevel" value="1">

If you do want to test hovering behaviors, consider including the following restrictions

IgnoreClassNameChanges = 2 - CSS Class Name changes will be ignored

IgnoreMouseMoveChanges = 4 - Property changes in Mouse move will be ignored

IgnorePostHoverChanges = 8 - Property changes post mouse hover (like using timer to change the property) will be ignored.

These items may be OR'd together as well for example

IgnorePostHoverChanges = 6 - Property changes in Mouse move and CSS Class Name changes will be ignored

http://blogs.msdn.com/b/shivash/archive/2011/01/24/hover-recording-in-coded-uitest-builder-and-microsoft-test-manager.aspx

MaxDepth

If you are hand coding searches and using the smart match capabilities you can and should consider using the MaxDepth property to limit the depth in the control tree that the search engine looks for a given control. If a control cannot reasonably be say 3 levels deep, then set the MaxDepth to 2 and save the playback engine from wasting cycles by searching the control tree deeper than the control can reasonably be.

myCell.SearchProperties.Add(**Other search properties**)

myText.SearchProperties.Add(WpfText.PropertyNames.MaxDepth, 1);

http://blogs.msdn.com/b/tapas_sahoos_blog/archive/2011/05/10/test-automation-for-silverlight-datagrid-in-coded-ui-test.aspx

WebWaitForReadyLevel

The default WebWaitForReadyLevel is 0 which is the most robust setting as timer and AJAX behavior is tracked. The tracking is accomplished by injecting script into the page. Setting the WebWaitForReadyLevel to 1 omits the injection of the timer script tracker. Setting the WebWaitForReadyLevel to 2 omits the injection of the AJAX script tracker. These values may be OR'd together so setting WebWaitForReadyLevel to 1, 2, or 3 will improve performance but the test will be less robust. If the addition of these injected scripts causes any behavior changes to the application try setting WebWaitForReadyLevel to 4 for timer problems or 8 for ajax problems or 12 for both. This won't have much of an impact on performance.

Modify the QtAgent32.exe.config file to change this behavior:

```
<appSettings>
 <add key="EnableHtmlLogger" value="true"/>
 <add key="EnableSnapshotInfo" value="true"/>
 <add key="WebWaitForReadyLevel" value="3" />
 <add key="StopTestRunCallTimeoutInSeconds" value="5"/>
 <add key="LogSizeLimitInMegs" value="20"/>
 <add key="CreateTraceListener" value="no"/>
 <add key="GetCollectorDataTimeout" value="300"/>
</appSettings>
```

More details on Coded UI test execution performance Improvements

http://blogs.msdn.com/b/vstsqualitytools/archive/2009/08/10/configuring-playback-in-vstt-2010.aspx
http://blogs.msdn.com/b/visualstudioalm/archive/2012/02/01/guidelines-on-improving-performance-of-coded-ui-test-playback.aspx
http://blogs.msdn.com/b/vstsqualitytools/archive/2011/07/06/improving-the-performance-of-your-coded-ui-tests.aspx

Adding Coded UI support to Custom Controls

Coded UI Test allows users to do functional UI testing. Coded UI replicates the actual user action by sending the mouse/keyboard inputs to the control within the screen. One of the primary objectives of a Recording during automation is to generate Robust Search Logic for identifying the UI Control during Playbacks. There are certain accessibility requirements for the controls and Coded UI Test makes some strong assumptions to identify a control. The Coded UI Test tool is dependent on the UI Technology that is being used in the application and based on the UI Technology it uses the associated technology to retrieve properties and controls of the forms.

Coded UI Supports the below UI Technologies for carrying out the Search of the controls.

1. Internet Explorer Testing: Uses MSHTML, DOM to retrieve properties and identify controls hosted within the Internet Explorer.
2. UIA: UI Automation is the new accessibility framework for Microsoft Windows, available on all operating systems that support Windows Presentation Foundation (WPF).
3. MSAA: This is picked for Winforms Controls, Win32 controls, MFC applications. All the controls that are not picked up by the above two are picked by MSAA.

Coded UI Primarily relies on the Automation Properties of the Control to see, if the control is supported for Automation or not.

For a control to be supported by Coded UI, the control should possess one of the below properties

1. AutomationId,
2. Name,
3. LabeledBy,
4. HelpText,
5. AccessKey,
6. AcceleratorKey
7. DisplayText
8. Source – In Case of an Image Control
9. Column Index -In Case of a Data Grid

Adding Support to the Controls not Identified by Coded UI

The testing framework for coded UI tests and action recordings does not support every possible user interface. It might not support the specific UI that you want to test. For example, you cannot immediately create a coded UI test or an action recording for a Microsoft Excel spreadsheet. However, you can create your own extension to the coded UI test framework that will support your specific UI by taking advantage of the extensibility of the coded UI test framework.

UI Testing of any control can be enabled by implementing the extension points available in the UI Test Framework. The user has the option of reusing the Visual Studio UI Test plug-in for extending supportability of UI automation, by implementing the appropriate accessibility for

the custom control. If you are building support for a new UI technology, you have the choice to determine the level of UI Testing support to enable for your technology. Based on the level of required support, you will have to implement a set of extension points in the UI Test framework. The UI Test Framework is heavily dependent on the UI Technology that is being used in the application.

There are a large number of UI Technologies present in the market today and new ones are coming up all the time. On top of that, external vendors create additional set of UI Controls which provide richer features for each one of these technologies. The following are the guiding principles Microsoft will use to build UI Testing support for these multitudes of UI Technologies:

Microsoft will build and support UI Testing for

- Microsoft platforms [Windows, Internet Explorer, Windows Phone, SharePoint, Office].
- Core controls on Application development technologies [Windows Forms, Windows Presentation Foundation, Silverlight]

For implementing UI Testing support for any new UI Framework, you can use the below table to decide on the UI Test Framework.

Technology	UI Test Implementation Model
Windows Forms	Microsoft Active Accessibility (MSAA)
Windows Presentation Foundation	UI Automation (UIA)
Internet Explorer	MSHTML
Firefox	JavaScript and Firefox DOM
Silverlight	Code Injection and reflection

Table 2 – UI Test Framework

The existing custom UI Technology frameworks do not implement sufficient accessibility for Visual Studio UI Test framework to support them seamlessly. The Coded UI tool has various extensibility points that let users and partners build support for technologies not support by the base product.

Extensibility Point	Description and use
Extension Package	The entry point to any UITest extension.
Technology Adapter	Used to add support to technologies not supported by the tool in the box. For example, use this to add support for Java AWT classes.
Filter/ Aggregation Rule	Used to add new filter or aggregation rule to the recorder. For example, add an aggregation rule for custom DataPicker control to record richer SetValue of Date action instead of individual clicks.
Property Provider	Used to supply information about various properties supported by the control and how to use these properties. For example, use this to add more properties to an existing control (say Today to WPF DatePicker).
Browser Service	In addition to technology adapter extension point, this is needed to support new browser.
Add\Modify API	Customize how mouse or keyboard actions behave.
UITest Object Model	Listen to various events from UITest to do some custom action or, do something completely different with the recording.

Table 3 - Extensibility Points

Improved Validations with Visual Studio 2012

Several enhancements have been made to the Coded UI Test API to simplify validating complex controls such as trees, grids, lists, and groups of controls.

Validating groups of controls

A single call to GetNamesOfControls or GetValuesOfControls will return an array list of the names and values of the controls respectively allowing us to validate the names and values of an entire group of controls with a few lines of code.

GetNamesOfControls

```
//Validate all of the names of the edit controls on a page
BrowserWindow win = BrowserWindow.Launch(new Uri("http://www.SiteUnderTest.com"));
…
UITestControl doc = win.CurrentDocumentWindow;
HtmlControl control = new HtmlControl(doc);
control.SearchProperties.Add(HtmlControl.PropertyNames.ClassName, "HtmlEdit");
UITestControlCollection controlcollection = control.FindMatchingControls();
string[] expectedNames = { "txtUserId", "txtPassword", "btnOk"  };
string[] actualNames = testCollection.GetNamesOfControls();
for (int index = 0; index < expectedNames.Length; index++)
{
    Assert.AreEqual(expectedNames[index], actualNames[index]);
}
```

GetValuesOfControls

```
//Validate all of the values of the edit controls on a page
BrowserWindow win = BrowserWindow.Launch(new Uri("http://www.SiteUnderTest.com"));
…
UITestControl doc = win.CurrentDocumentWindow;
HtmlControl control = new HtmlControl(doc);
control.SearchProperties.Add(HtmlControl.PropertyNames.ClassName, "HtmlEdit");
string[] expectedValues = { "Some User", "Secret Password", "Ok"  };
string[] actualValues = testCollection. GetValuesOfControls ();
for (int index = 0; index < expectedValues.Length; index++)
{
    Assert.AreEqual(expectedValues[index], actualValues[index]);
}
```

Similarly, if you have a UITestControllCollection and you want to evaluate a specific property on all of these controls you may use the GetPropertyValueOfControls method.

GetPropertyValuesOfControls

```
//Validate the checked property of all of the checkbox controls on a page
BrowserWindow win = BrowserWindow.Launch(new Uri("http://www.SiteUnderTest.com"));
```

```
…
UITestControl doc = win.CurrentDocumentWindow;
HtmlControl control = new HtmlControl(doc);
control.SearchProperties.Add(HtmlControl.PropertyNames.ClassName, "HtmlCheckBox");
bool[] expectedCheckedValues = { true, true, false };
bool[] actualCheckedValues =
testCollection.GetPropertyValuesOfControls<bool>(HtmlCheckbox.PropertyNames.Checked);
for (int index = 0; index < actualCheckedValues.Length; index++)
{
     Assert.AreEqual(expectedCheckedValues [index], actualCheckedValues [index]);
}
```

Validating Grids / Tables (WPF, Win Forms, HTML)

Validating grids or tables in the past could be a challenge as well, GetColumnValues, GetCell, GetRow, GetContent, GetColumnNames, FindFirstCellWithValue have been added to help address this challenge.

Assume we are testing a Windows Forms Application, and testing a data grid:

GetCell, GetRow, FindFirstCellWithValue

Assume we are testing a Windows Forms Application, and doing validations on a data grid:

```
WinTable dataGrid = new WinTable(appUnderTest);
dataGrid.SearchProperties.Add("Name", "dgResultsGrid");

//Test a single cell
WinCell cellActual = dataGrid.GetCell(0, 1);
Assert.AreEqual("Cell 1 Value", cellActual.Value);

//Validate an entire row
WinRow dataGridrow = dataGrid.GetRow(2);
int index = 0;
string[] actualRow = { "Order ID", "Order Name", "Order Status" };
foreach (WinCell cell in dataGridrow.Cells)
{
    Assert.AreEqual(actualRow[index], cell.Value);
    index++;
}

//FindFirstCellWithValue Test
```



```
WinCell dataGridCell = dataGrid.FindFirstCellWithValue("12345");
Assert.AreEqual("12345", dataGridCell.Value);
```

Validating List Views (Win Form)

Performing validations on a listview is also simplified with the following methods:

GetColumnValues, GetColumnNames, GetContent

```
WinWindow listViewWindow = new WinWindow(AppUnderTest);
listViewWindow.SearchProperties.Add("ControlName", "lvResultsList");
WinList listView = new WinList(listViewWindow);
string[] actualColumnNames = listView.GetColumnNames();
string[] expectedColumnNames = new string[] { "Order ID", "Order Name", "Status" };
CollectionAssert.AreEqual(expectedColumnNames, actualColumnNames);
string[] actualContent = listView.GetContent();
string[] expectedContent = new string[] { "111", "Order 1", "Pending", "112", "Order 2",
"Shipped", "113", "Order 3", "Closed" };
CollectionAssert.AreEqual(expectedContent, actualContent);
WinListItem listViewItem = new WinListItem(listView);
listViewItem.SearchProperties.Add("Name", "111");

//Validate the entire list item
string[] actualColumnValues = listViewItem.GetColumnValues();
string[] expectedColumnValues = new string[]{"111", "Order 1", "Pending" };
CollectionAssert.AreEqual(expectedColumnValues, actualColumnValues);
```

Validating Accessible Description (Win Form)

Validating the Accessible Description is now possible using the new API available in Visual Studio 2012

```
WinWindow win = new WinWindow(allWinControls);
win.SearchProperties.Add("ControlName", "lvReport");
WinList lvReport = new WinList(win);
Assert.AreEqual("This is a list view with important names and values.",
lvReport.AccessibleDescription);
```

Validating ToolTipText (Win Form, WPF)

Validating the Tool Tip Text is now possible using the new API available in Visual Studio 2012

```
WpfWindow win = new WpfWindow();
win.SearchProperties.Add("Name", "All WPF Controls");
WpfComboBox comboBox = new WpfComboBox(allWpfControlsWindow);
comboBox.SearchProperties.Add("AutomationId", "cbNames");
Assert.AreEqual("The cbNames tool tip comboBox.ToolTipText);
```


Validating Item Status (WPF)

Validating Item Status is also possible using the new API available in Visual Studio 2012. For this example, assume the implementation of an extended WpfButton control has color mapped to the ItemStatus of the custom control. For a full working example, follow this blog post.

```
WpfWindow win = new WpfWindow();
win.SearchProperties.Add("Name", "All WPF Controls");
WpfButton itemStatusButton = new WpfButton(win);
itemStatusButton.SearchProperties.Add("AutomationId", "ItemStatusButton");
Assert.AreEqual(Color.Red, ColorTranslator.FromHtml(itemStatusButton.ItemStatus));
```

Select and validate an item in a list control (WPF, Win Forms, HTML)

Adding the ability to select an item from a list also simplifies the validation process

.Select

```
WinList listControl = new WinList(appUnderTestWithList);
listControl.SearchProperties.Add("Name", "lNames");
WinListItem listItem = new WinListItem(listControl);
listItem.SearchProperties.Add("Name", "thirdItemInList");
listItem.Select();
Assert.AreEqual("I am Number 3", list.SelectedItemsAsString);
```

Best Practices for Handling Dynamic Content:

*Use **Window Search Properties and Smart Match Property** effectively in case the control properties change frequently:*

Normal Controls and Windows

The window title of the application might change based on the version number or the environment in that they are testing.

For e.g. the application that is running on the local machine might have the title window name such as “App Test Version 1.0” and recording will be done with the name as “App Test Version 1.0 “ and the one which is running in the Test environment can be “App Test Version 1.5”.

The Record and Playback Engine identifies controls on the user interface by its search properties (like window name, class name etc.)

In the above scenario where the properties of the control changes frequently , the Record and Playback engine uses a Smart Match algorithm to identify controls if they cannot be located using the exact properties. The smart match algorithm uses heuristics and attempts to locate the control using variations on the search properties.

You can control when Smart Match should be applied. By default Smart Match is applied for Top Level Windows and all controls. You can turn off Smart match using the following code snippet.

Playback.PlaybackSettings.SmartMatchOptions = SmartMatchOptions.None;

One disadvantage of using SmartMatch Option is that it will slower the execution of the test run, this can be handled by modifiying the Window Search handle properties , please check the below example:

public UIMap()


```
{
   this.UIApplicationv1013Window.SearchProperties.Add(WinWindow.PropertyNames.Name, " App Test Version ",
PropertyExpressionOperator.Contains);
}
```

Window search is always faster when compared to using Accessibility Tree, so it is better to use Window Search Properties then using Accessibility Tree.[2]

By Default smart match is enabled for finding top level windows and controls within a top level window. There might be cases especially when you are stilling in the development life cycle and the UI is subject to constant changes , where we can find wrong control due to smart match. You can control when Smart Match should be applied. You can turn off Smart match using the following code snippet.

```
Playback.PlaybackSettings.SmartMatchOptions = SmartMatchOptions.None.
```

Once the UI is stabilized ,the setting can be enabled again.

DataGrid

The search hierarchy for a Datagrid is Datagrid→Table→Cell. The search properties of Cell and Row are of more interest since that's where the performance and resiliency of the playback comes into picture.

The Cell is primarily identified by its ColumnHeader property value obtained from the Cell's ITableItemProvider. If the ColumnHeader is not defined or not unique, the ColumnIndex is generated as the Cell's identifiable property. The parent Row's search properties is either the Automation Id or the Name (in case Automation Id is not defined). If the Row has data bound issues i.e. the Datagrid Rows do not have a proper identifiable Name, it would throw a NotSupportedException during recording mentioning that the Row control does not have any good identification property.

You could safely hand code the search properties of a Cell to include Instance property (for example, in Cell where none of the ColumnHeader or ColumnIndex are defined properly). However, you need to ensure your test automation script is updated accordingly in case of future additions/deletions of columns.

In case of Rows, it is not advisable to use the Instance property. Additions/Deletions are very much common in Rows and your automation script is bound to fail at some point of time if you rely on the Instance property. Moreover, during playback there is always a bit of search performance overhead while using Instance property.

*Using **AlwaysSearch** Option:*

Your testing a scenario in which you are trying to test a context menu item and from the context menu, if the menu item is not disposed correctly,in the next iteration , the test would refer to an invalid reference to the context menu control.

Using **AlwaysSearch** option to make sure that when you search for a control it does not return the control from cache and it always returns a valid control.

A drawback of using AlwaysSearch is that it will have a performance hit in normal scenarios. It is recommended not to include configuration by default in your control search.

*Use **Match Exact hierarchy** in case of a change in the UI of window.*

The Record and Playback Engine uses a hierarchical structure to locate controls.

[2] Note that some technologies (WPF, Silverlight, Html) have non-windowed controls. This statement is only applicable for top level windows and windowed controls

When the application changes and if the hierarchy gets modified. Record and Playback engine attempts to locate the control, in such cases, by skipping the intermediate elements in the hierarchy. Thus in the example, if it is unable to locate a control, it will search for control under parent window and all its children.

This behavior can be controlled by the "Match Exact Hierarchy" setting. This is done by the following code snippet.

```
Playback.PlaybackSettings.MatchExactHierarchy = false;
```

*Use **WaitForControlEnabled** method of the control:*

In many cases the application has to wait for control before it is visible this could be due to a business logic which is building the controls in a dynamic or an event complete (for e.g clicking of "Ok" button, or completion of a background process) or slowness in the performance of the application. In such scenarios use **WaitForControlEnabled** method of the control.

```
Control. WaitForControlEnabled()
```

Use **MaxDepth** to restrict your search sub-tree.

By default, the playback searches within the entire sub-tree of the container specified. Restricting the depth of the search (BreadthFirstSearch) improves the search performance to a good extend especially in scenarios such as Datagrid.

As a best practice it is advised to leave the default settings to the MaxDepth property in the early stages of the Application development (when the UI is not stable) and later the value can be reduced to a smaller value.

References

[1] Aniyan, Mathew, 03-17-2009, “Data Driving Coded UI Tests”
http://blogs.msdn.com/b/mathew_aniyan/archive/2009/03/17/data-driving-coded-ui-tests.aspx
[2] http://codeduicodefirst.codeplex.com/
[3] http://cuite.codeplex.com/
[4] http://msdn.microsoft.com/en-us/library/dd380742(v=vs.110).aspx
[5] http://msdn.microsoft.com/en-us/library/ff468125(v=vs.110).aspx

