Contents

[Disclaimer 1](#_Toc318835035)

[Introduction 1](#_Toc318835036)

[Getting started with UIMap Toolbox 2](#_Toc318835037)

[Downloading and installing the tool 2](#_Toc318835038)

[Uninstalling the Add-in from Visual Studio 2](#_Toc318835039)

[Making a simple UIMap for Notepad 3](#_Toc318835040)

[Restructuring an UIMap 4](#_Toc318835041)

[Backup files saved by the tool 7](#_Toc318835042)

[Moving elements to other UIMap 7](#_Toc318835043)

[Merging UIMaps 9](#_Toc318835044)

[Developing robust and maintainable CodedUI based tests 9](#_Toc318835045)

[Encapsulating System under Test using UIMaps 10](#_Toc318835046)

[System Under Test layer 10](#_Toc318835047)

[UIMap(s) layer 10](#_Toc318835048)

[Actions layer 10](#_Toc318835049)

[Test Cases layer 10](#_Toc318835050)

[Strategies for multiple UIMaps 11](#_Toc318835051)

[UIMap-per-test-case strategy 11](#_Toc318835052)

[UIMap-per-window strategy 11](#_Toc318835053)

# Disclaimer

Use this tool at your own risk. Similar to other refactoring tasks, it’s often a good practice to keep restructuring of UIMaps to a separate source control check-in. The tool creates a backup of the UIMap file before saving a UIMap, but still I often find it helpful to keep the restructure operation distinct.

# Introduction

This guide describes how to use the UIMap Toolbox tool for restructuring UIMaps. The guide assumes some knowledge of UIMaps, the Microsoft CodedUI Framework and general development of automated testing.

UIMap Toolbox is a tool that extends the features available in the UIMap editor of Visual Studio 2010, namely:

* Restructuring a UIMap, by dragging nodes internally in a UIMap
* Splitting a complex UIMap in two, by dragging elements to another UIMap
* Merging two UIMaps into one

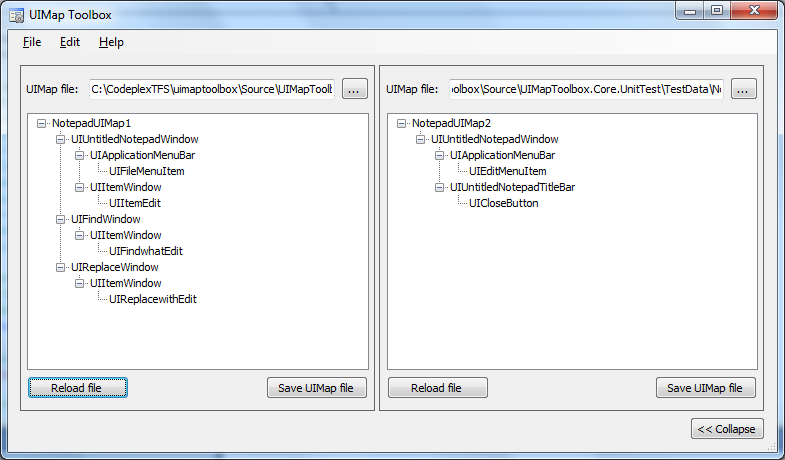


Figure 1 - UIMap Toolbox with two UIMaps opened

Note that the tool currently only has limited support for actions inside the UIMaps, e.g. it’s not possible to move elements, which are being referenced by actions in the UIMap itself. Later we’ll go through a suggestion for an architecture where actions are located in a separate layer from the UIMaps.

The first part of this guide shows a short example of using the tool on a simple UIMap for Notepad. Second part contains guidelines for developing robust and maintainable UI based tests, e.g. by having multiple UIMaps.

# Getting started with UIMap Toolbox

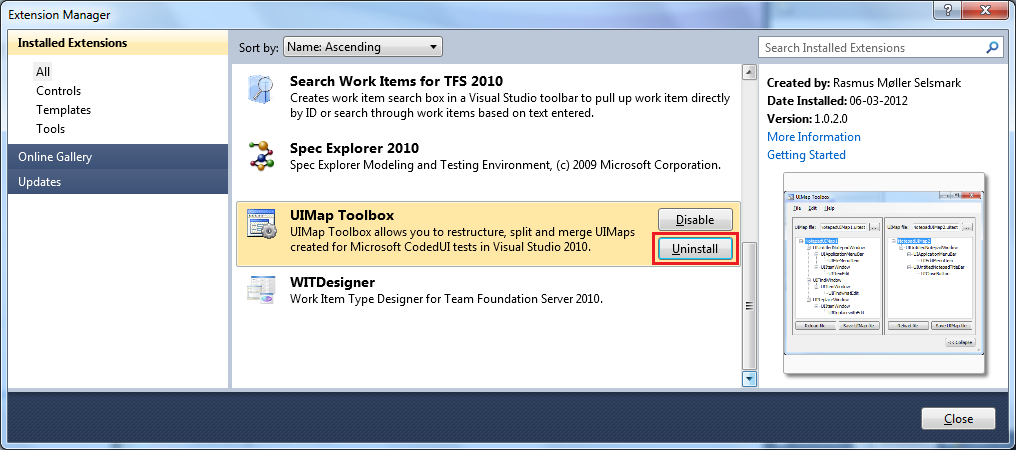
## Downloading and installing the tool

The tool can be downloaded from <http://uimaptoolbox.codeplex.com/>. It’s possible to download either as a Visual Studio Add-in (VSIX) or just the binaries. It’s recommended to use the Visual Studio Add-in, since this enables opening of UIMaps directly from Solution Explorer window in Visual Studio.

This guide primarily describes how to use the tool as a Visual Studio Add-in.

## Uninstalling the Add-in from Visual Studio

If you need to uninstall the add-in from Visual Studio, open Extension Manager from the Tools menu, find the UIMap Toolbox entry and click the Uninstall button.



## Making a simple UIMap for Notepad

In the following example, we will create a simple UIMap for Notepad.

Start by creating a new Test Project in Visual Studio 2010:

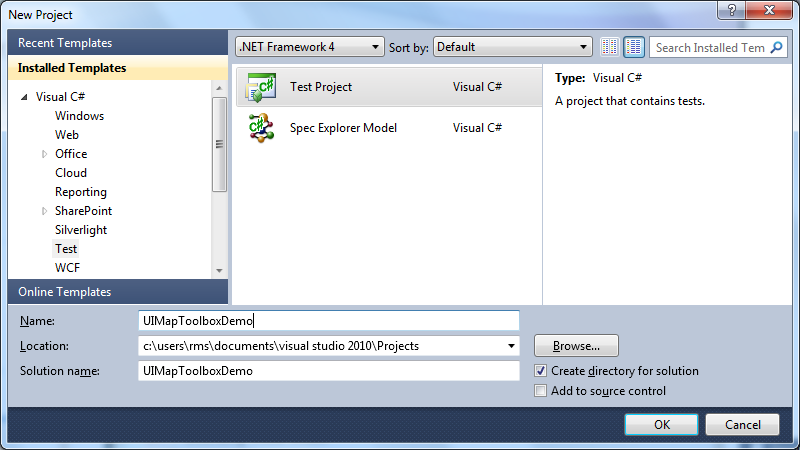


Figure 2 - Create new Test Project

Add a new UIMap named “NotepadUIMap”, by right-clicking the project and select Add->New Item…

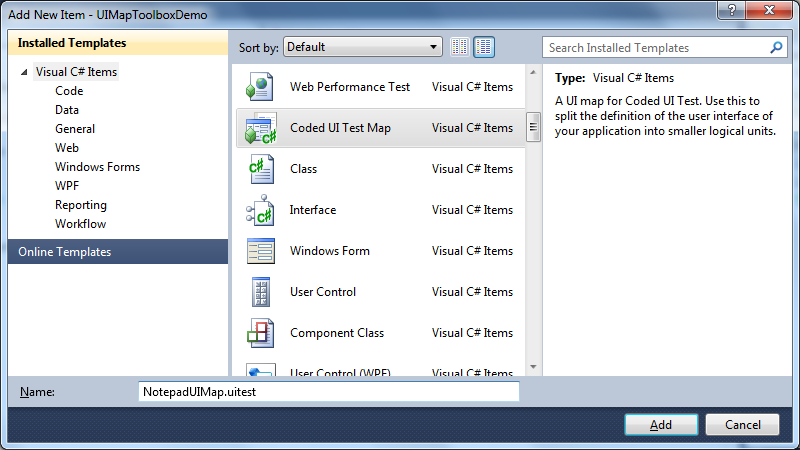


Figure 3 - Add a UIMap to the project

Now we’ll record some actions using the CodedUI Test Builder. Note that we have saved the file during the test, so we have two top-level windows for Notepad, “UIUntitledNotepadWindow” and “UINotepadFiletxtNotepaWindow” (since the file was saved as “NotepadFile.txt”)

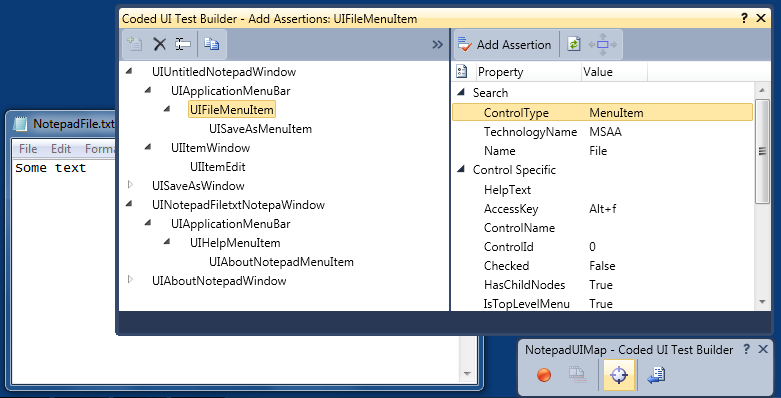


Figure 4 - Editing UIMap using Coded UI Test Builder

## Restructuring an UIMap

In order to move the elements from “UINotepadFiletxtNotepaWindow” to the “UIUntitledNotepadWindow” element, we can move them using UIMap Toolbox. Right-click on the UIMap file and open it by selecting the “Open with UIMap Toolbox…” option:

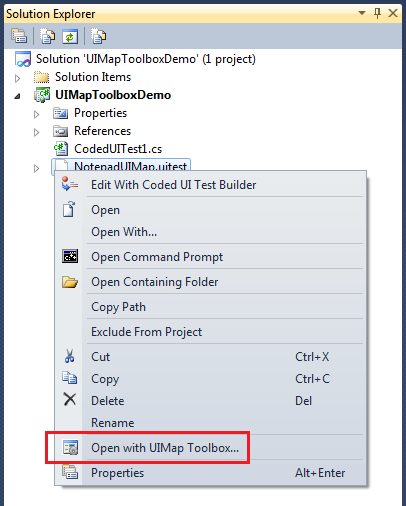


Figure 5 - Open UIMap from Solution Explorer window

Open the UIMap file in UIMap Toolbox, and drag the UIApplicationMenuBar element to UIUntitledNotepadWindow

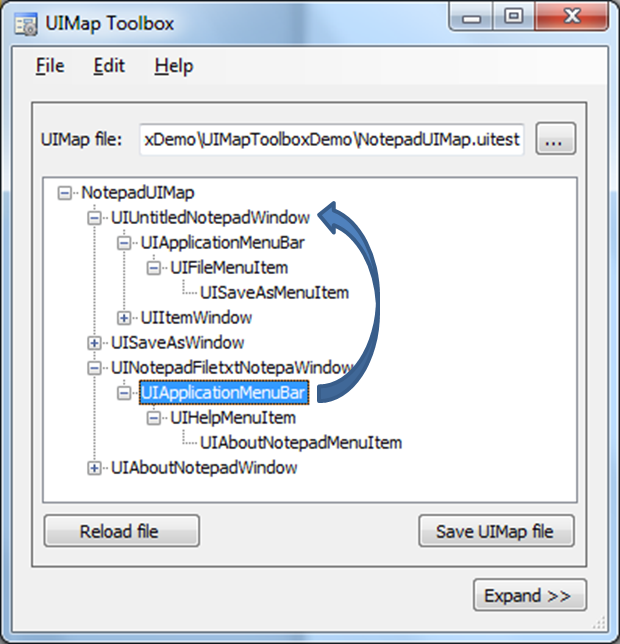


Figure 6 - Restructure UIMap

Since the UIApplicationMenuBar element already exists, the children is being merged, and the result is that all menu items now are available under the UIUntitledNotepadWindow element (after deleting the now empty UINotepadFiletxtNotepaWindow element):

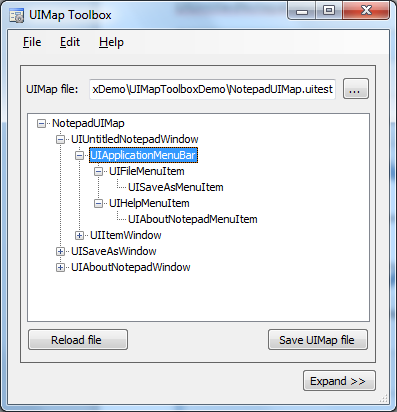


Figure 7 - UIMap after restructure

Click the “Save UIMap file” button to save the UIMap. Note that you must open the UIMap in Visual Studio and save it (by clicking Ctrl+S) to regenerate the C# code.

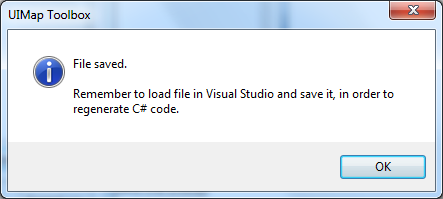


Figure 8 - Saving file in UIMap Toolbox

An import note here is that the Search Properties for UIUntitledNotepadWindow should be modified to “contains Notepad” in order to match any Notepad window (regardless of title bar text):

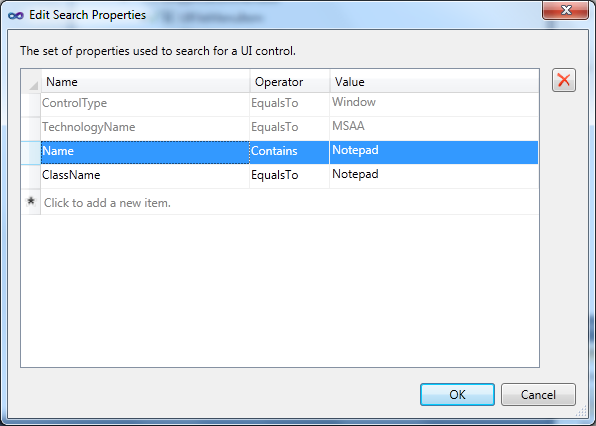


Figure 9 - Change Search Properties for top level window

## Backup files saved by the tool

When you save the UIMap, the tool automatically creates a backup of the file in a folder named “UIMapEditorBackups” in the same folder where the UIMap is stored. This way you can always get back to the original UIMap file in case something went wrong. As suggested earlier, it can often be a good idea to keep “refactoring” of UIMaps in a separate checkin, so you can more easily revert your changes if something went wrong.

## Moving elements to other UIMap

**Important:** In order to move elements between UIMaps, you should remove all actions in the UIMap file, since UIMap Toolbox doesn’t support moving elements that are referenced by actions in the same UIMap. This can be done by right-clicking any action and selecting “Move Code”. In this case the code for the action will be moved from NotepadUIMap.designer.cs to the NotepadUIMap.cs file, which means that it’s now your responsibility to maintain the code.

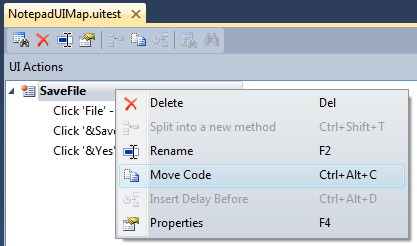


Figure 10 - Move action code from Designer.cs file

With our example from above, we would like to split the UIMap into two. This is useful when a UIMap becomes too large, or you simply want to make your UIMaps simpler. The rule of cohesion also applies to UIMaps, i.e. the a more narrow/focused responsibility is preferred for a UIMap. In this case we are going to move the UISaveAsWindow element to another UIMap.

Start by creating a new empty UIMap named SaveAsWindowUIMap:

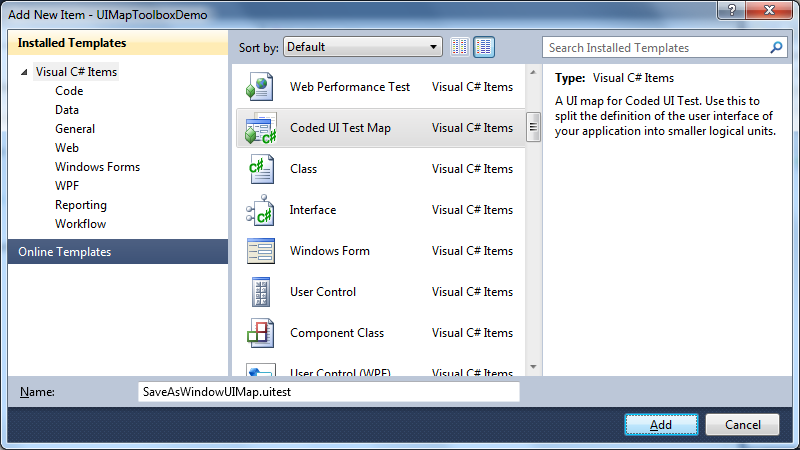


Figure 11 - Add a new UIMap

When the Coded UI Test Builder window appears, click the Generate Code button to save an empty UIMap file. Click the Generate button even though it says there are no pending changes. Don’t just close the window, since this will result in an UIMap file that cannot be loaded.

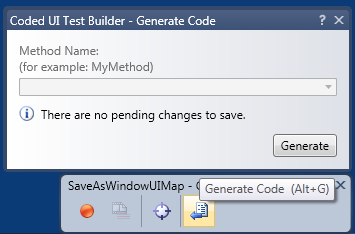


Figure 12 - Generate empty UIMap

After the new UIMap has been saved, open it in UIMap Toolbox, next to our original UIMap (after clicking the “Expand >>” button in bottom of window).

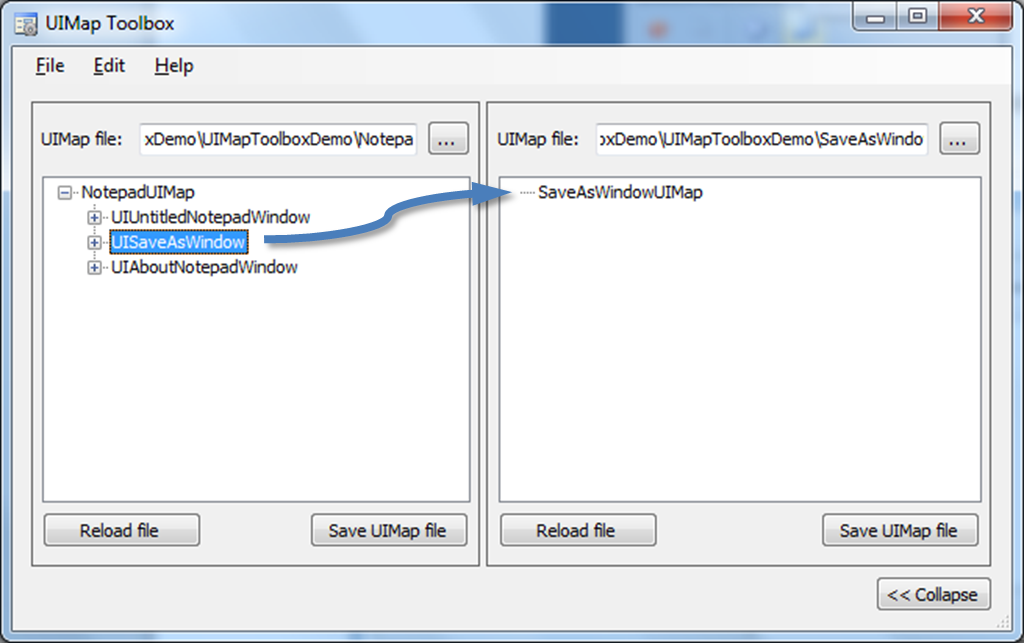


Figure 13 - Move element to other UIMap

The result of this operation is that the UISaveAsWindow element is moved to our new UIMap file:

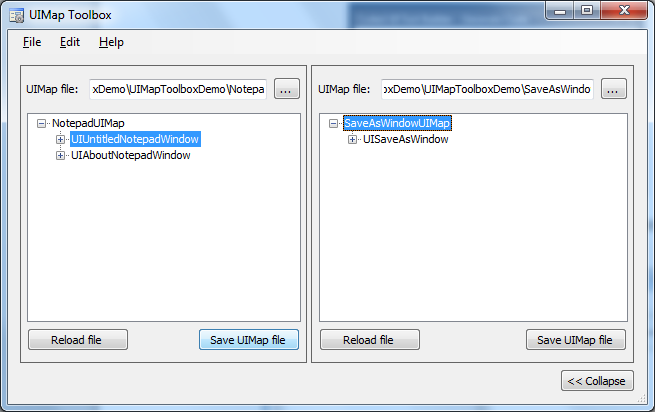


Figure 14 - UIMaps after restructure

Afterwards, save both UIMaps, load and save them in Visual Studio to regenerate the C# code.

## Merging UIMaps

Similar to moving elements from one UIMap to another, it’s possible to merge elements from two UIMaps into one. Just open the two UIMap files, and start dragging the top-level elements between the files.

# Developing robust and maintainable CodedUI based tests

This section covers my thoughts and experiences on developing robust and maintainable UI based tests using the Microsoft CodedUI framework.

The CodedUI framework from Microsoft is yet another framework for developing automated UI based tests, and probably all the features of CodedUI is available in other test automation UI frameworks as well (or could be developed by yourself). The benefits of CodedUI and UIMaps are however, as I see it:

* Built-in design tool support in Visual Studio for authoring UIMaps (especially useful when system under test changes)
* When used correctly, helps you structure and organize the test abstraction for your system under test in a way that makes the tests maintainable by yourself and others

## Encapsulating System under Test using UIMaps

The following architecture simply shows how UIMaps are encapsulating the system under test.

Figure 15 - Test automation architecture

### System Under Test layer

This is the system or application we’re testing. Not as such a layer in the test automation code, but shown here to get the conceptual idea of the architecture.

### UIMap(s) layer

This layer encapsulates the system under test in type-save UIMaps, that we can access from our actions. The UIMaps should be developed in such a way that if e.g. a button changes, it should only be necessary to change one UIMap to have tests running again.

This level of the architecture can also contain other ways of interacting with the system under test, e.g. direct HTTP communication etc.

### Actions layer

The actions layer consists of the “building blocks” used by the tests. The goal is to develop actions in such a way that each action can be used in several test cases.

### Test Cases layer

This layer contains the actual test code. Test cases should only access the actions layer (similar to production code, where e.g. a UI layer only should access the business-logic layer and not going directly to database)

Having defined an architecture is crucial if you want your test automation project to be successful. The automated tests should run as long as the product lives, and therefore should have the same quality (and often better) than the production code.

## Strategies for multiple UIMaps

One very important aspect of developing maintainable CodedUI tests, is to use multiple UIMaps. You can read more about multiple UIMaps at:

* “Testing a Large Application with Multiple UI Maps”: <http://msdn.microsoft.com/en-us/library/ff398056.aspx>
* “Walkthrough: Using multiple Coded UI maps in test automation”: <http://blogs.msdn.com/b/anutthara/archive/2010/02/10/walkthrough-using-multiple-coded-ui-maps-in-test-automation.aspx>

If you don’t use multiple UIMaps, you most likely end up with one very big and unmaintainable UIMap containing all windows and controls used by your tests. The XML file behind this UIMap can easily become several megabytes in size, which makes it very difficult to pinpoint any problems found, especially if the system under test later changes.

Therefore I highly recommend that you look into how to use multiple UIMaps for CodedUI tests. I see the following two strategies for using multiple UIMaps in test (there might be more):

* UIMap-per-test-case
* UIMap-per-window

### UIMap-per-test-case strategy

This approach is the simplest of the two to implement. Whenever you start developing a new test, you add a new UIMap containing the controls used by the test. If the test later breaks, you edit (or re-record) the UIMap.

This is the simplest of the two approaches, and has the advantage that UIMaps are separated, i.e. you don’t risk breaking other tests, when making changes to one UIMap. Disadvantage is that if system under test changes, you might have to edit several UIMaps, to correct e.g. a button id, since the same UI elements can be defined in several UIMaps.

### UIMap-per-window strategy

This is my preferred approach, and also the one that UIMap Toolbox was designed to support. In this strategy you add a new UIMap for each major part of system under test. For a Windows application this is typically one UIMap per window, whereas for a web application it can be one UIMap per page.

Advantage is that this gives a nice encapsulation of the system under test, but is not as easy to implement as the UIMap-per-test-case strategy. Especially you often have to restructure UIMaps, e.g. when a window title has changed and Visual Studio therefore generates a new top-level element in the UIMap.

Earlier I mentioned “cohesion” (see e.g. <http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29>) which is a metric of how “strong” a class is. Cohesion also applies to UIMaps (since they are in fact classes), so we also want UIMaps with well-defined responsibilities, in order to keep our test automation code maintainable. In order to accomplish this, I find the UIMap-per-window strategy to be the best choice when developing CodedUI tests using multiple UIMaps.