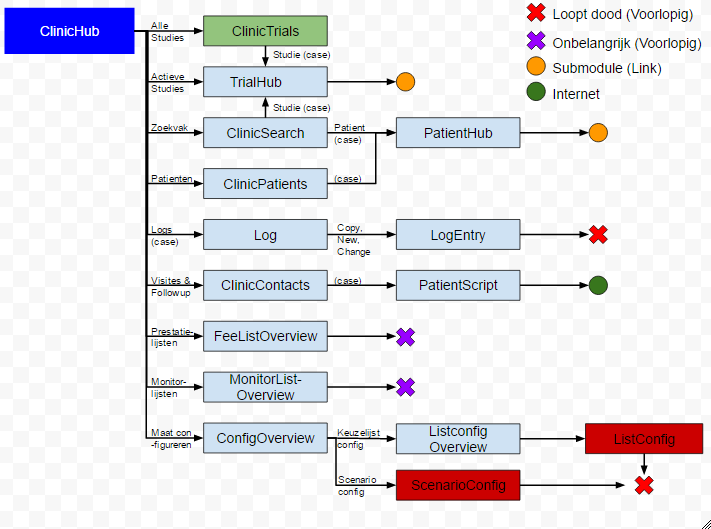
Guidelines

Testing

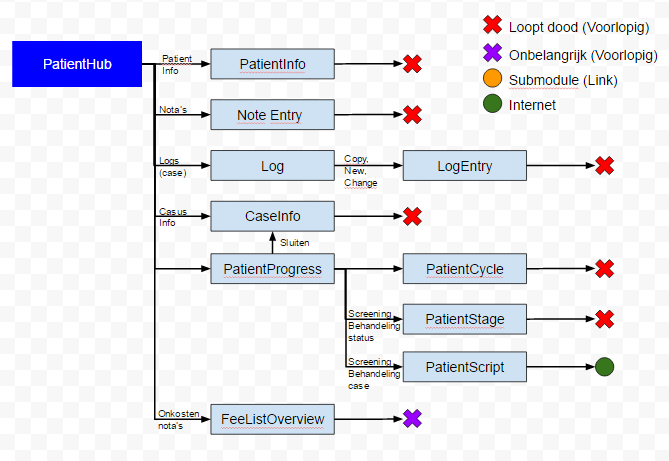
# Hierarchical navigation design

In this section, you can see a hierarchical design of all possible navigations inside the Clinical Trials application, ordered by page. For every page, we have one UI-map and per page, we have a test project for every category on the checklist

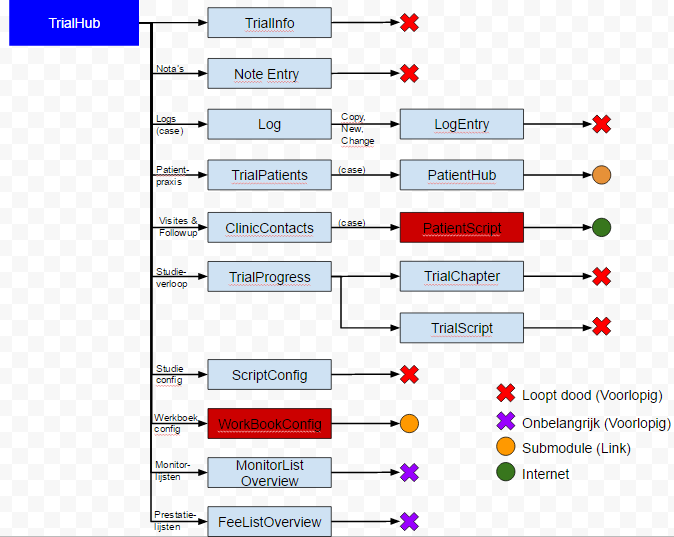
## ClinicHub



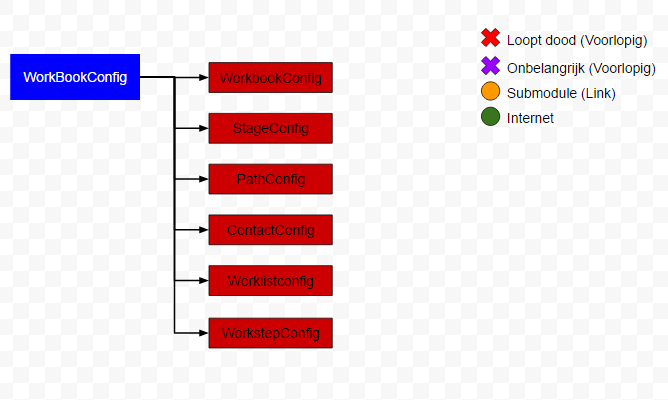
## PatientHub



## TrialHub



## WorkBookConfig



# Method Naming

To maintain the readability of the test project, here are some guidelines on how to name your testmethods

* Simple TestMethod
  + Action + Xaml Name Property
    - Ex: ClickContactButton()
    - Actions:
      * Push a button with mouse Click
      * Push a button with keyboard Enter
      * Edit a field Edit
      * Change combobox selection Select
      * Check a checkbox / radiobutton Check
* Complex TestMethod
  + Desciptive action
    - Ex: TestHomeButton()
      * Runs multiple tests that affect testing of ‘Home’ button
      * In this case
        + ClickHomeButton()
        + EnterHomeButton()

# File Structure

The Clinical Trials application follows a specific map structure for all it’s elements. To keep the file structure readable, we follow the same structure inside our testproject.

Calidos.Maat.CodedUITests

* Screens
  + General UIMap
  + Partial parent .cs file
  + .csv global scenarios
  + Category (clinic,…)
    - .csv categorical scenarios
    - Page (contactspage,….)
      * UImap
      * .csv local scenarios
      * .cs files per category
        + Menu User input at the menubar (shared)
        + Navigation User input that navigates to other pages
        + Functionality User input that affects the page itself
        + Data User input that affects the full application
        + Custom Page specific functionality (different for

every page

# Checklist: Testing

## Content

### Crud

#### Create

#### Read

#### Update

#### Delete

### Custom

## Navigations

### Forward

### Backward

## states

### Multiselect

### Normal

### Semantic zoom

## functionality

### Scrolling

### Menubar

### Controls

### Custom

## 

## 

This is a detailled description of every aspect that has to be tested in the project. We also have an excel file in which this checklist implemented in a matrix together with all the pages we have to test. Like this, we can very easily add new found controls/actions & re-check all pages for the implementation of those tests.

## Menubar

### Home toggle button

Check if using the home-button triggers a navigation to the ClinicHubPage.

Do this for:

* Click action
* Tab select action
  + Space to execute
  + Enter to execute → not (yet?) implemented

### 

### Expand toggle button

Check if using the expand toggle button (in the menubar) makes the menubar expand/contract. To do this, a treshold variable is used. The test will have to be able to see if the bar is already expanded or not, to then assert the correct functionality.

Do this for:

* Click action
* Tab select action
  + Space to execute
  + Enter to execute → not (yet?) implemented

### 

### Setting toggle button

Check if using the settings toggle button triggers a navigation to the *??*Page (not yet implemented).

Do this for:

*possible actions (not yet implemented)*

## Navigation

### listItem navigations

#### Case listItem navigations

One possible way to navigate from one page to another is through clickable listItems, which represent cases of for example patients, studies, logs,.... These are variable and depend on the data loaded into the application. The amount is also variable. These listItems are embedded in a list, which is embedded in a hubsection.

Each listItem in one list will navigate to the same page, but the data on that page will depend on the clicked Item (ex. titles, names,...). The basic structure of the page navigated to will stay the same.

As a result, for each list, only one navigation (one listItem) needs to be tested.

check navigation for:

* Click action
* Tab select action
  + Enter to execute
  + Space to execute → not (yet?) implemented

### 

#### Other listItem Navigations

Other listItems are also embedded in hubsections, but the amount is not variable and each listItem triggers slightly different navigation. Every listItem in the hub navigates to the same page but if a hub has multible listItems, they will all trigger a different highlighted tab/ordering/…

##### multible tab pages

For each listItem in a hubsection, check if navigation to the correct tab of the correct page is triggered.

Do this for:

* Click action
* Tab select action (not (yet?) implemented)
  + Enter to execute
  + Space to execute

##### pages ordered by comboboxes

For each listItem in a hubsection, check if navigation to the correct page, and correct ordering of the page is triggered.

Do this for:

* Click action
* Tab select action (not (yet?) implemented)
  + Enter to execute

Space to execute

### Button navigation

Each button embedded in a hubsection, will navigate to a specific page (which is indicated by the text in the button).

For each button, check if the correct navigation is triggered.

Do this for:

* Click action
* Tab select action
  + Enter to execute
  + Space to execute

### Back navigations

At the end of each navigation test, test the back navigation (like this, the back navigation will automatically be tested for every possible page from which you can navigate back to the page you are testing.)

#### Back button

Check if the backbutton navigates back to the page you are testing.

Do this for:

* Click action
* Tab select action
  + Enter to execute
  + Space to execute

### Hyperlink navigations

***//TO ANALYSE & FINISH***

Do this for:

* Click action
* Tab select action
  + Enter to execute
  + Space to execute

## Page Functionality

### Semantic zoom functionality

#### Zoom out

##### Ctrl-

For each page, check if the semantic zoom functionality of the page zooms out when using the “Ctrl” key in combination with the “-” key.

##### Ctrl and mousescroll

For each page, check if the semantic zoom functionality of the page zooms out when using the “Ctrl” key in combination with the “mousewheel”.

##### Mouseclick on hubsection-titles

For every page, check if the semantic zoom functionality of the page zooms out when clicking on the titles of different hubsections.

#### Zoom in

##### Ctrl+

For each page, check if the semantic zoom functionality of the page zooms in when using the “Ctrl” key in combination with the “+” key. Check if the selected hubsection is positioned correctly (on the left side of the screen). To select a different hubsection, use the tab key.

##### Ctrl and mousescroll

For each page, check if the semantic zoom functionality of the page zooms in when using the “Ctrl” key in combination with the “mousewheel”. Check if the selected hubsection is positioned correctly (on the left side of the screen). To select a different hubsection, use the tab key.

##### Mouseclick on hubsection-titles

For every page, check if the semantic zoom functionality of the page zooms in when clicking on the titles of different hubsections. Check if the clicked hubsection is positioned correctly (on the left side of the screen).

### Scrolling functionality

#### “PgUp” / “PgDn”

For every page, check if the control’s inside a section change position when using the “PgUp” and “PgDn” keys.

#### “Mousewheel”

For every page, check if the control’s inside a section change position when using the “mousewheel”.

#### “Scrollbar”

For every page, check if the control’s inside the page change position when clicking and dragging the scrollbar.

## Titlebar actions

### Open overlay

For each page, check if using the “+” button (ex. “nieuwe studie toevoegen”) opens the pages overlay.

Do this for:

* Click action
* Tab select action
  + Enter to execute
  + Space to execute

### moreButton

For each page, check if using the “...”-button (= more info) opens/closes a bar under the titlebar-buttons in which more info about each button is displayed. Make sure your test can recognize if the “more info”-bar is already displayed or not, and it can execute the test depending on that state.

Do this for:

* Click action
* Tab select action
  + Enter to execute
  + Space to execute

## Page overlay

### Close overlay

For each overlay, check if clicking the cross-button closes the page-overlay.

### Edit datafields

For each page overlay, edit the content of the datafields with:

* mouseclick to select + type text on keyboard
* Tab key to select + type text on keyboard

and check if correct data was added to correct datafield.

To do this, you can use hardcoded strings (because these tests are concentrated on the functionality and not the data)

### Full datadriven test

For each page overlay, edit all datafields with a datadriven test and click on the “add” button (different for each overlay). Then check if the correct data was added, by navigating to the page which will display a new case (ex. new trial, new patient,...) and check if the case you just added exists.

# Checklist: Testing (By control)

## Buttons

**Toggle Button**

1. Check Toggle Button initial toggle state
2. Check if Toggle Button toggle-state is changed when clicked
3. Check if Toggle Button action/navigation is done properly and correct
4. Check if Toggle Button is highlighted when hovering over the element
5. Check if Toggle Button responds to ‘mouseclick’ event
6. Check accessibility of Toggle Button with ‘tab’ event
7. Check if Toggle Button responds to ‘space event’

**Button**

1. Check if Button action/navigation is done properly and correct
2. Check if Button is highlighted when hovering over the element
3. Check if Button responds to ‘mouseclick’ event
4. Check accessibility of Button with ‘tab’ event
5. Check if Button responds to ‘enter’ event
6. Check if Button responds to ‘space’ event

**GridViewItem as button (Fixed)**

1. Check if GridViewItem action/navigation is done properly and correct
2. Check if GridViewItem is highlighted when hovering over the element
3. Check if GridViewItem responds to ‘mouseclick’ event
4. Check accessibility of list with ‘tab’ event
5. Check if GridViewItem responds to ‘enter’ event
6. Check if GridViewItem responds to ‘space’ event

**GridViewItem as button (Variable)**

1. Check if GridViewItem action/navigation is done properly and correct
2. Check if GridViewItem is highlighted when hovering over the element
3. Check if GridViewItem responds to ‘mouseclick’ event
4. Check accessibility of List with ‘tab’ event
5. Check accessibility of ListItem with ‘arrow keys’ event
6. Check if GridViewItem responds to ‘enter’ event
7. Check if GridViewItem responds to ‘space’ event

**AppBarButton (Custom button)**

1. Check if AppBarButton action/navigation is done properly and correct
2. Check if AppBarButton is highlighted when hovering over the element
3. Check if AppBarButton responds to ‘mouseclick’ event
4. Check accessibility of AppBarButton with ‘tab’ event
5. Check if AppBarButton responds to ‘enter’ event

Check if AppBarButton responds to ‘space’ event

## Textboxes

**TextBox**

1. Check if TextBox is enabled/disabled
2. Check if TextBox can receive and show input
3. Check if TextBox action is done properly and correct
4. Check if Textbox is highlighted when hovering over the element
5. Check if TextBox responds to ‘mouseclick’ event
6. Check accessibility of TextBox with ‘tab’ event
7. Check if TextBox responds to ‘enter’ event

**SearchBox**

1. Check if SearchBox action/navigation is done properly and correct
2. Check if TextBox is highlighted when hovering over the element
3. Check if SearchBox button responds to ‘mouseclick’ event
4. Check accessibility of TextBox with ‘tab’ event
5. Check if keyboard input goes directly into the SearchBox TextBox
6. Check if TextBox responds to ‘enter’ event

## Comboboxes

**ComboBox**

1. Check if ComboBox action/navigation is done properly and correct
2. Check if ComboBox is highlighted when hovering over the element
3. Check if ComboBox responds to ‘mouseclick’ event
4. Check if:
   1. ComboBox changes the selected item into the item that is clicked
   2. This change responds with the proper action
5. Check accessibility of ComboBox with ‘tab’ event
6. Check if ComboBox responds to ‘arrow keys’ event
7. Check if ComboBox responds to ‘tab’ event
8. Check if TextBox responds to ‘enter’ event

## Semantic

**SemanticZoom (Zoomed In)**

1. Check if SemanticZoom zooms out correctly with Ctrl & scroll
2. Check if SemanticZoom zooms out correctly with Ctrl+ & Ctrl-
3. Check if SemanticZoom Text (Hubtitle) responds to ‘mouseclick’ event
4. Check accessibility of SemanticZoom with ‘tab’ event
5. Check if SemanticZoom Text (Hubtitle) responds to ‘enter’ event
6. Check if SemanticZoom Text (Hubtitle) responds to ‘space’ event

**SemanticZoom (Zoomed Out)**

1. Check if SemanticZoom zooms in correctly with Ctrl & scroll
2. Check if SemanticZoom zooms in correctly with Ctrl+ & Ctrl-
3. Check if SemanticZoom ListItems responds to ‘mouseclick’ event
4. Check accessibility of SemanticZoom with ‘tab’ event
5. Check if SemanticZoom ListItems responds to ‘enter’ event
6. Check if SemanticZoom ListItems responds to ‘space’ event

## Tabs

1. Check if
2. Check if

Check if

# How to test

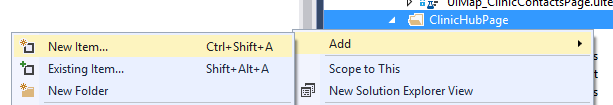
In this section, we will describe how to write tests for every item on the checklist. This is the actual implementation of the tests. In here, we will also describe some guidelines in the test code to keep the code readable and easily adaptable.

## Basics

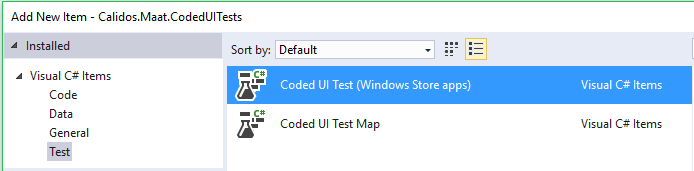
### Coded UI Test Builder and UI Map

To add a Coded UI Test project:

* Right click on the map you want to add the project
* Select ‘Add’
* Select ‘New Item’



* Select ‘Test’
* Select ‘Coded UI Test (Windows Store apps)’



To add a UI map:

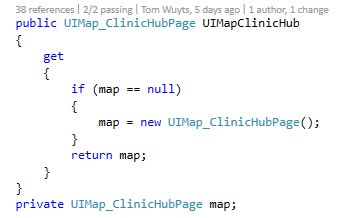
* Do the same
* Select ‘Coded UI Test Map’ instead of ‘Coded UI Test (Windows Store apps)’

When you added a Coded UI Test project, the first thing you need to do is add the right UI Map as a variable in your test project:

* At the top of your project, add a “using” statement for the UI Map you created. If I named my UI Map “UIMap\_ClinicHubPage”, this is what I would have to add:

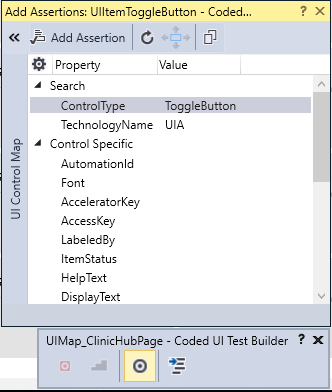


* At the bottom of your project, change the UI Map property to something like this:



After doing this you can start writing tests. To write a test you first have to add the controls you want to use to the UI Map you created. To do this, right click on your ‘.uitest’ file and select “Edit with Coded UI Test Builder”.

The Coded UI Test Builder will launch itself. Dont worry if it minimizes your Visual Studio, the Builder just wants to indicate that you can start up your application you want to be mapping. By dragging the circular marker (‘Add Assertions’) onto the control you want to map you can add it to the UI Map. When you release the marker it will open another window, seen below.



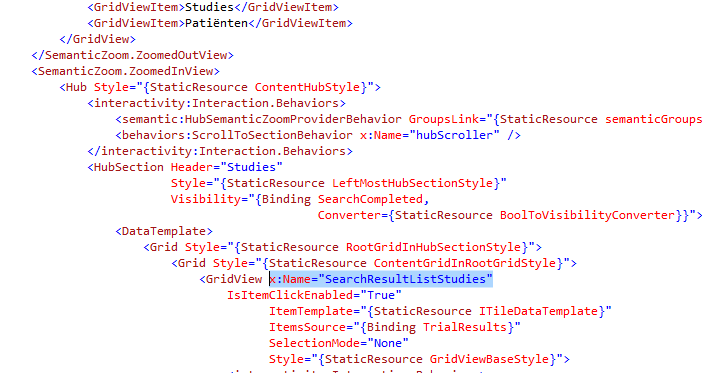
The control you selected will be higlighted within the application with a blue borded and on the right side of the new window you can see more detailed information about the currently selected control. If you click the arrow button in the top left hand corner you can see the hiërarchy in which the control is embedded as well.

Note:

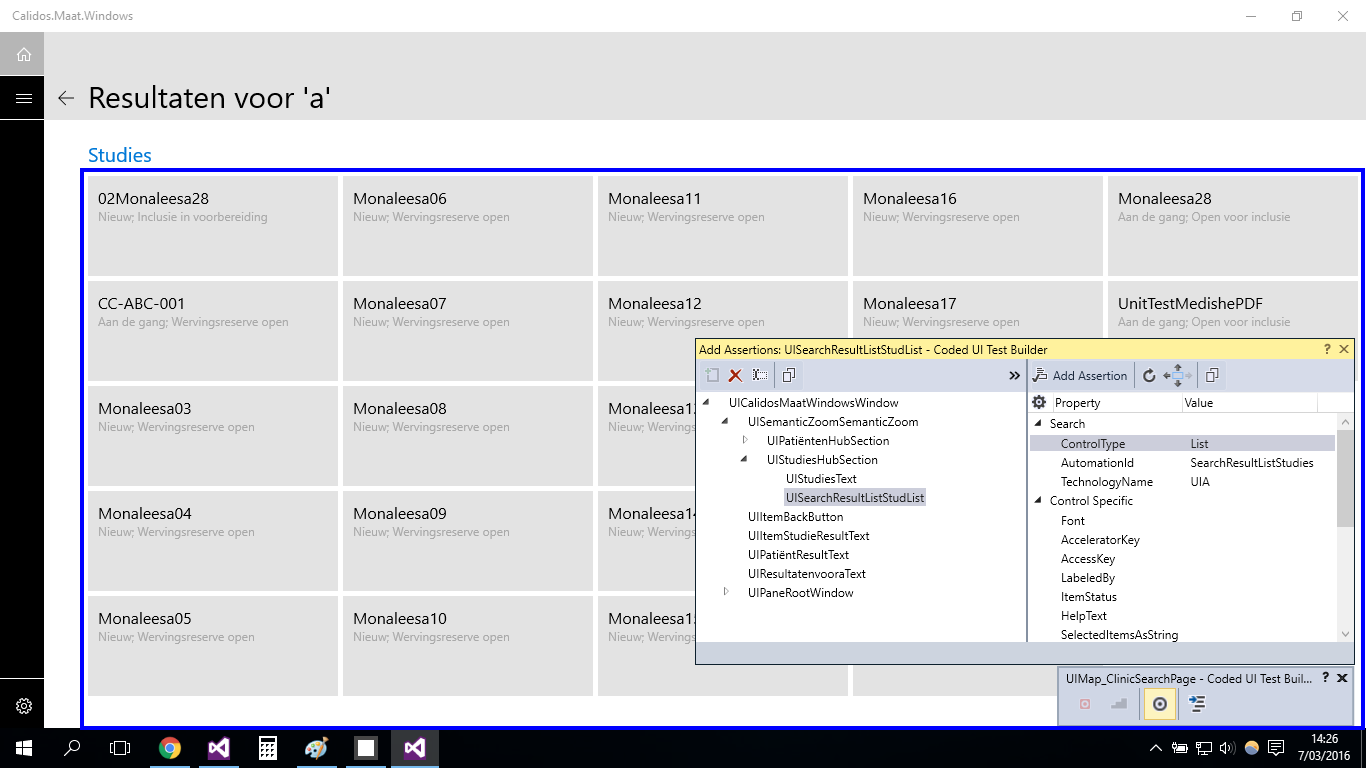
*Because the Clinical Trials application ‘Maät’ is created as a Windows 10 Metro App, the Coded UI technology, mainly the Coded UI Test Builder, is not yet fully adapted for optimal hiërarchy detection. To properly recognize the correct hiërarchy, every control has to have a unique AutomationId. However, almost no control in the application has this Id. For example, to add ListItems we needed to figure out special techniques and workarounds in the test methods, which we will discuss later.*

*To make sure control’s or lists are properly mapped and easy to find by the testprogram itself, we sometimes gave AutomationId’s to the control’s ourselves. To do this, open the XAML file of the page you want to test, and then search for an indication out of which you can derive this section is the section you want to give an AutomationId.*

*Example: I gave a unique Id to a list, so I can later easily access the childs of that list*



*I wanted to give a unique Id to the list of Studies-searchresults, so I searched the XAML file for a while, tried naming some different grid’s and gridviews, untill I named the right one. Now if I select that list with the Coded UI Test Builder, the name I gave to it will appear as AutomationId, as seen below.*



### Testmethods

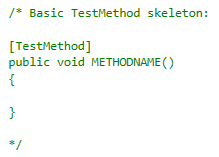
#### Test Classes

When you create a Coded UI Test class, the class must be marked as being for testing purposes. This is done by Visual Studio itself by adding the “[CodedUITest]” tag in front of the namespace declaration. More sepcific, for the application we are working on at the moment, is the “[CodedUITest(CodedUITestType.WindowsStore)]” tag that was auto-generated for us.

#### The ‘[TestMethod]’ tag

Every Coded UI Test needs to be preceded by a “[TestMethod]” tag, else the tests will not be picked up by Visual Studio’s test framework.

A basic testmethod skeleton is depicted below.



#### The ‘[TestInitialize]’ tag

This method will be executed **before every** ‘[TestMethod]’. Its important to note the ‘every’ in bold. Whenever a testmethod is ran, The testing framework loops over the whole test project to check for initialization methods. This method wil always be ran first. Which could make it a usefull method at certain points.

In our project, we will use this tag to navigate to the desired page in the application before every TestMethod using Process() and a URI string that every page in the application posesses

#### The ‘[TestCleanup]’ tag

This will be executed **after** **every** ‘[TestMethod]’. The same as with the TestInitialize, only after each and every testmethod. Usually this method contains a shutdown or close command to close the application after every test.

In our project this tag has not (yet) been used, bacause we dont want to reload all the data every few seconds for the UI tests.

#### The ‘[DataSource()]’-tag

When you want to create a datadriven test (which is a test using external data for execution/assertion/…), this tag can be added on top of the testmethod, together with the [TestMethod]-tag.

ex. [DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", @"C:\dev\CTO\Src\Dev\D.Peter.0\Client\Calidos.Maat\Calidos.Maat.CodedUITests\Screens\Clinic\ClinicTrailsPage\NieuweStudieToevoegenData.csv", "NieuweStudieToevoegenData#csv", DataAccessMethod.Sequential), TestMethod]

Datasource uses 4 variables to connect to a datasource:

* type of dataconnection (ex. CSV, XML,...)
* path to datafile
* table name
* access method (sequential/random)

If you add a datatable to a testmethod, the method will be executed for every row in the table.

The first row will be ignored and can be used to asign variable names to your data for easy accesability.

#### Testcontext

“TestContext”, which is a property declared at the bottom of each testproject, contains information about each test.

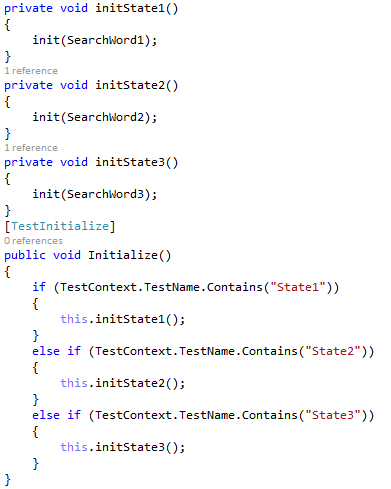
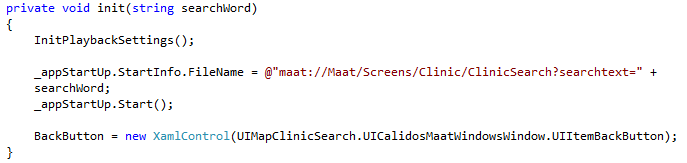
##### Data

when the test contains a datasource, data can be read through TestContext. Do do this, you can use the property “TestContext.DataRow[*row*]” in which row can be an index number or a title of the row you gave in your datatable (in string format)

##### TestName

In some testproject, you will want to be able to use different TestInitialize methods. However the TestInitialize tag can only be used once. To solve this problem, you can write private functions with the different initialization methods, and in the TestInitialize method you can write an if-statement using the TestContext.TestName property. This is a string representation of the name you gave to your testmethod.

full example:



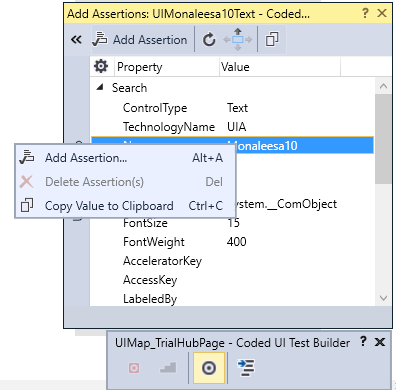
### Generating controls

There are a few phases in order to generate a control in your testmethod:

* Identify the type of control you want to generate (XamlButton, XamlEdit,...).
  + XamlControl is a general name you can use for every type of Xaml control. The downside of doing so, is that you might not be able to use some control-specific properties.
* After you identified the control, give it a unique, yet obvious name.
* Finally, assign a path to your control referring to the control you added to the UIMap.
* After generating a control, you can use its properties to create different kinds of tests. Some example of control properties often used in tests:
  + Name The name of the control
  + Font Textfont (handy when asserting if a tab is selected or not (Bold text)
  + Exists Bool that indicates if the control exist on the current page or not
  + ...

Sometimes you want to use control-specific proporties (like for example with togglebuttons,...). Then you have to give a specific Xaml-type to the control (ex. XamlText, XamlEdit,...)

You will end up getting something like below image.

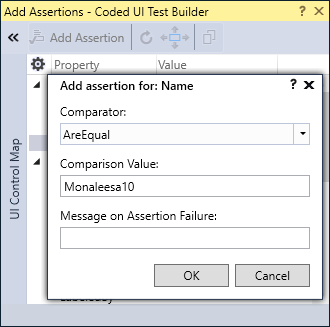


### Writing assertions

#### (Semi) Automatic asserts

The Coded UI Test Builder enables you to write semi-automatic asserts.

After selecting a control with the marker, you can make an assertion for a specific property, by selecting it and clicking “Add Assertion” in the top left corner of the window, or by simply right clicking on the property you wish to assert and choosing “Add Assertion” as illustrated in the snapshot on the right.



A new window will pop up. Here you can choose the comparator, comparison value and message you will receive when the assertion fails.

In the example we go for the ‘AreEqual’ comparator, but there are some more things you could choose from, like ‘Contains’ or ‘StartsWith’.

The comparison value is the value you want the actual value to be, in order to let the assertion pass. In other words, the test framework will get the actual value and check it against the comparison value

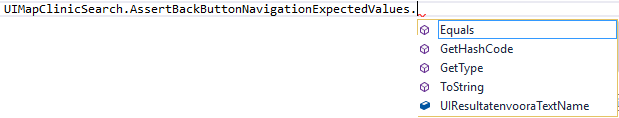
Additionally you can add a Message on assertion failure to get a better understanding what went wrong if an assertion failed. When done, click the ‘OK’ button.

Click the “Generate” button at the Coded UI Test Builder and your assertion a name. You can also add a description to make sure everyone knows what the assertion is all about.

Your assertion is now ready. In order to use it, you have to call your UI Map first. Remember that everything that is generated with the Coded UI Test Builder is added to this map. After calling the UI Map you will see that your assertion is in the list of suggested code



The UI Map in the example is the UI Map of the testproject, and the highlighted area is the name you gave to your assertion method. The assertion contains the expected value you assigned on creation. However, on certain moments we will want to have a different expected value. The value can still be changed within your testmethods or within the partial UI Map class (The part that doesnt contains auto-generated code). Changing the expected value can be done like below:



#### Manual assertions

The advantage of manual asserts is that you have much more control over the control you want to assert to as well as the expected values to make the test as autonomous as possible.

A manual assertion always starts with the ‘Assert’ keyword followed by the comparator. Note that there are different comparators availbable when making a manual assertion (‘Contains’ for example is not available when using manual assertions). The comparator requires parameters to create the full assertion.



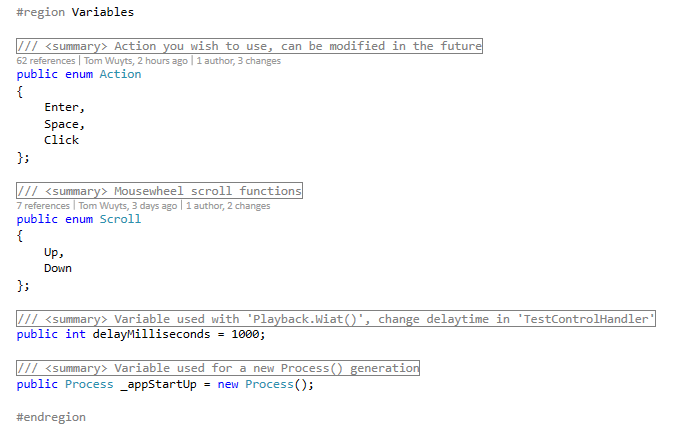
In the example, where we make an ‘AreEqual’ assertion, the first parameter is the expected value. The second parameter is the actual value. Where these 2 value’s come from, is completely under your control. This way of writing assertions is way more intensive then the auto-assertions, but makes room for some more flexibility. Manual assertions are written inside your testmethods.

## BaseClassCodedUI

This is a baseclass we wrote ourselves, in which we write functions for actions we have to use very often. Like this, we can very easily perform the action multiple times in just one line of code. We will also add commonly used variables and global testscenario’s (that don’t require page-specific control’s but overal control’s, available on every page) to this class.

Every testproject that is created, has to inherit from this baseclass.

### Variables



* enum Action
  + Action to be used
* enum Scroll
  + Mousewheel scroll functions
  + Created purely for readability reasons
    - More readable then bool although only used for 2 states, like bool itself
* public int delayMilliseconds
  + The Playback.Wait() delaytime between actions
  + Adjustable variable in 1 place
* Process \_appStartUp
  + Variable that is used to start the application on the right page using a URI

### Functions



* InitPlaybackSettings
  + Initialize the settings used during testing
  + Changed some default value’s to more appropriate value’s for our tests
* UseControl
  + Control that will be used as a clickable object through click or key interaction
    - control XamlControl that will be used
    - action enum Action that is needed
* ComboBoxSelectItem
  + Changing the ComboBox control’s SelectedIndex
    - box XamlComboBox that will be used
    - number Index number of the wished SelectedIndex
* ListItemEnter
  + Use the wanted instance inside a list by using tab & enter
    - list List inside which your control exists
    - instance Instance of the control inside the list
    - action action used (may change in future)
* GetListChild
  + Will return a XamlControl, which is the control you want to click
    - list List you specified
    - instance The instance number of the control you want to click

inside the list

* TabToControl
  + Tabbing towards a certain XamlControl that may or may not yet be visible
    - control XamlControl that will be used
* ScrollToControl
  + Scrolling towards a certain XamlControl that may or may not yet be visible
    - control XamlControl that will be used
    - scrollControl XamlControl used to hover over, so we can scroll
    - scroll enum Scroll to indicate the direction to scroll towards
* WriteAsUser
  + Converts string to char’s and simulates as if a user enters the key’s as input
    - msg string that is converted to char’s as user input
* UseSearchbox
  + Method for using the searchbox
    - searchbox Xamlcontrol used to refference where to click
      * Button of searchbox is untargetable by UIMap
    - msg string that is converted to char’s as user input
      * WriteAsUser is used within this method

scroll enum Scroll to indicate the direction to scroll towards

## Navigation

### Click

#### Tile list navigation

Find a control by adding a list to the UIMap with the Coded UI Test Builder

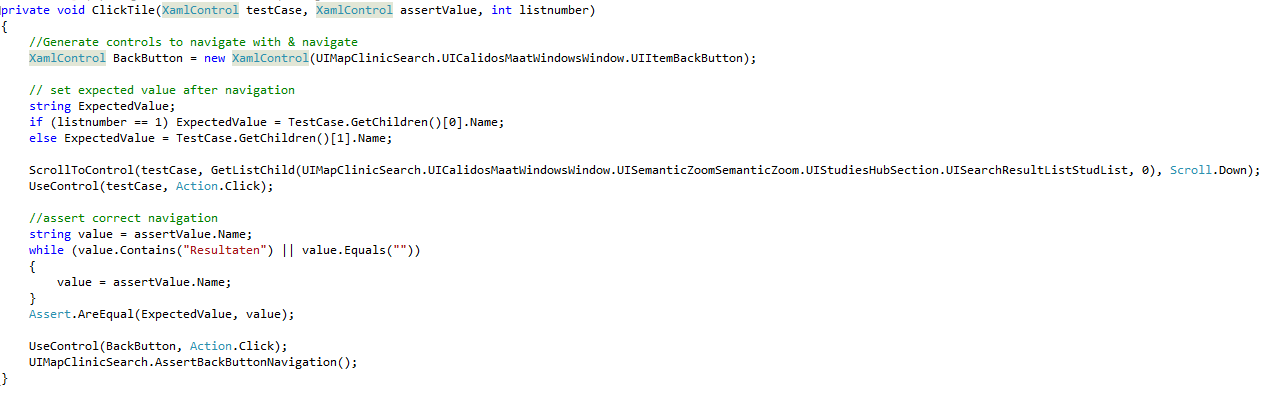
Because most control’s don’t have an AutomationId in the clincical trials application, the Coded UI Test Builder will probably wrongly interpret the hierarchie of controls. Lists of tiles give back wrong hierarchies almost every time. We came up with a solution for this. In the XAML page of the app, there are regions specified for the list in which you want to click a control. You can add a name to it by adding < *x:name=”listname” >*. Be sure to check if you added the correct name by using the Coded UI Test Builder. If correctly implemented, you will be able to always find the correct list you want to use. Because you want to click an element inside this list, you can’t just hardcode the control to your UIMap, even if AutomationId’s would be present. This is because you have to be able to change the instance of the control you want to click later, so te generate a control out of the list, do the following:

* Generate Control for the tile out of one of the list’s elements
  + After adding the list to the UImap, you can use the function GetListChild (out of BaseClassCodedUI) to access the list and get it’s items.

Add the control you want to assert if the click triggered the correct navigation to the UImap (Usually this will be the title of the navigated page)

* + Generate this control in your testmethod
* Generate an expected value by getting the children of your control and selecting the instance in which the desired value is embedded (mostly this will be a name). Later you can assert to the name of this expected value
* Click control
  + In the BaseClassCodedUI, there is a function called ClickControl
* Assert correct navigation
  + Write a while loop in which you make sure the correct title is already loaded before you execute your assert (manual delay function)
  + Assert expected value to the name of your previously added value to assert to

***Full example:***



*This is the basic function used by all click-tests in the ClinicSearchPage.*

* *The back button is generated the same for every test.*
* *testCase is the instance we want to have clicked (found by applying the GetListChild function on the XamlList in which the control is embedded).*
* *assertValue is the title we want to assert to*
* *listnumber is a number to indicate some page specific properties, regarding the expected assert value*

***// to finish***

#### Back button navigation

* Add back button to UImap
* Add back button to every testmethod in your navigation-testproject
* Add title of your page to UI-map and make assert for name of it
* Write click-action for back button at the very end of every testmethod
* Write assert below click action
* example: as above

### Enter

#### Tile list navigation

* Same as click, but use ListItemEnter function from UIBaseClass instead of ControlClick → give list to function (instead of control), the instance of the tile you want to click, and the action you want to execute (enter,...) :

ListItemEnter(list, Instance, action);

#### Back button navigation

* Same principle but instead of ControlClick(), use ControlEnter() function of baseclass

# Adding Coded UI Tests to Build process

Some information considering this part can be found on below URL. This MSDN link has not yet been updated to Visual Studio 2015, but will generally be the same process to set up.

URL: <https://msdn.microsoft.com/en-us/library/ms182465%28v=vs.110%29.aspx#Anchor_3>

# Problems / Functionality issues

* Some navigation controlls cannot be activated using ‘Enter’ or ‘Space’
  + ClinicHub
    - VisitesEnFollowUpsHub: all tiles
  + TrialHub
    - StudieInfoHub: all tiles
    - StudieVerloopHub: all tiles
    - PatiëntPraxisHub: all tiles
    - VisitesEnFollowUpsHub: all tiles
    - NotasHub: tile
* Pop-up error window interupts tests, causing them to fail even when there is no problem with the UI or test itself
* Login window pop-up window interupts test startup, causing them to fail even when there is no problem with the UI or test itself
* SemanticZoom tiles spaced wrong / not scrollable when in zoomed out mode
  + ClinicHub
* Settings Button on menubar does not have navigation (yet)
* Clicking a trial and then clicking the backbutton sometimes makes a hyperlink appear on top of the page, of the trial you just navigated to, but sometimes it doesn’t
  + ClinicTrials

# Info

* Home button toggle isnt working as intended
  + Still toggles if user is already at homepage
* ClinicHub Hubs that contain lists get a different interaction animation with their respective hub titles
  + Probably because they make the width of the button bigger and hereby cause the different interaction
* Buttons that are actual XAML buttons (black) have bad contrast when highlighted
  + Text turns black on black background when mouse hovers the element
* ClinicContacts “Visites” Combobox items sometimes contain small logos. These heighten the combobox when selected.
  + Not sure if this is intended
* Visites & Follow-ups is referenced differently on different pages
  + Sometimes ‘Visites **& F**ollow-ups’ (ex. in ClinicContacts as title of page)
  + Sometimes ‘Visites **en f**ollow-ups’ (ex. in TrailHub as title of HubSection)
  + Uniform naming could improve the testing possibilities
* When using back button after pressing a HubSection button we end up on the right page, but we end up at the beginning of the page, not at the HubSection the user clicked on first
  + Can be confusing
  + Tested at TrialHub by pressing ‘Nota’s’ and backing