Individual Professional Practice in the Company
Hereby I declare that this bachelor thesis was written on my own. I have quoted all references I have drawn upon.

In Ostrava 6th of May 2011
Abstract

The content of this thesis is to describe how to develop automated graphical user interface tests in Microsoft Visual Studio 2010 with C#, NUnit and WatiN. The automated tests are focused on Tieto Elevation CMS. The knowledge of the technologies is received within professional practice in Tieto Czech.

Keywords: automation, GUI, testing, WatiN, C#, Tieto
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.Net</td>
<td>Microsoft’s web application framework for building dynamic web sites, web applications and web services</td>
</tr>
<tr>
<td>C#</td>
<td>C Sharp is Microsoft’s object-oriented programming language</td>
</tr>
<tr>
<td>CruiseControl.Net</td>
<td>CruiseControl.Net is an automated continuous integration server</td>
</tr>
<tr>
<td>CMS</td>
<td>Content Management System</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language for building web sites</td>
</tr>
<tr>
<td>JSSh</td>
<td>JavaScript Shell</td>
</tr>
<tr>
<td>MSBuild</td>
<td>Microsoft Build Engine</td>
</tr>
<tr>
<td>NUnit</td>
<td>Unit testing framework for .Net languages</td>
</tr>
<tr>
<td>TFS</td>
<td>Microsoft Team Foundation Server</td>
</tr>
<tr>
<td>WatiN</td>
<td>Web Application Testing In .Net</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
1. Introduction

This thesis is about my professional practice in Tieto Czech, Digital Transformation & Consulting service unit in Ostrava. I have worked from November 2010 to May 2011 two days a week and in this time I have been in a team which has developed Tieto’s own content management system called Elevation.

Elevation is used by many companies to manage the content of their web sites. Elevation is also used on Tieto’s internal web sites. The system is developed with C#, ASP.Net, JavaScript and AJAX. It consists of two main sites: management site which manages the content and browsing site where the content will be placed.

My assignment during the professional practice was to develop automated graphical user interface tests which would test the content management system and browsing site. The test should demonstrate that the system work with Internet Explorer and Firefox browsers.

Building test cases had been started before I started my training period and there were a dozen of working test cases at that time but they were running only in Firefox. I got a three week pair coding period with the developer who had been working on this matter. I received knowledge how to develop tests and what are good practices in user interface testing.

All the shown code examples are not straight from the professional practice period. The examples are imaginary and only to demonstrate how to develop tests. However they demonstrate real life cases well.

In part 2 is introduced the testing methods used within the practical training. Part 3 determines the needed preparation for tests. Part 4 covers procedure of developing tests and how to run them. In part 5 is described the automation of tests. Part 6 is a conclusion of the professional practice.
2. Software testing

Software testing is an important part of building software and the importance increases when developing significant changes to the software. This is why tests are nowadays automated so that they can be run many times a day and by doing this the developer will quickly know if the newly done code has broken some other part of the program. The more times the tests are run within a day smaller the risk is to have parts of software broken. In the following chapters is described different testing method used in the professional practice.

2.1 Unit testing

Unit testing is a principle of testing smallest testable parts of the software which are not dependent on other parts of the software and so can be tested separately as their own units. This gives the ability to automate testing of units. Unit tests are run in batches and each test determines if the unit work as desired. This gives straight feedback to the developer that where the test have failed and why [1].

2.2 Behavioral testing

Behavioral testing also known as black-box testing is a test method where the test developer doesn’t necessarily know the internal structure of software but only the requirements and the desired outcome of function that is been tested. The tests in professional practice where implemented with the black-box testing method [2].

2.3 GUI testing

GUI testing differs from normal unit testing in various ways. To develop GUI tests the developer don’t need to get access internal code of the application since internal code should be already tested with unit tests. GUI testing works with a tool that can control web browser. In practice WatiN was used to accomplish this. WatiN will simulate user actions in a web page such as clicking the buttons and filling up forms. The GUI testing tool doesn’t know anything about the code done server side it just access the web sites HTML elements as a user would.
3. Preparation of developing environment

In this chapter is described the needed preparations to the working environment before tests can be build. It is assumed that the workstation has already installed Microsoft Visual Studio. Also the workstation and server should have Internet Explorer and Firefox browsers installed. In professional practice the server had Internet Explorer 6 and Firefox 3.6 installed while the workstation had Internet Explorer 8 and Firefox 3.6 installed.

3.1 Team Foundation Server

First when we start to build our test solution we need to connect to TFS. TFS enables many people to easily work with same project and therefore is certainly needed in software development with multiple developers. TFS enables to monitor if someone is modifying a file. The user can check if there are any pending changes on the files or do the user have latest version of the files. TFS is also used to keep track on the development process by work items which will include for example what parts need to be done and what are their priorities.

Connecting to TFS in Visual Studio 2010 is done from the Team menu and selecting the Connect to Team Foundation Server option. From the opening dialog box you can choose the TFS and select the project from a list. When connected files of the project are accessible from the Team explorer [3].

3.2 Building testing sites

It is sensible to isolate testing web sites from CMS development itself so new web sites are built just for testing. The web sites are built from the latest version of development database so new features and bug fixes will come into effect. The database will be regenerated later on when tests are automated.
3.3 Needed frameworks and plug-ins

3.3.1 NUnit

Nowadays there are dozens of different frameworks for automating unit tests in different programming languages and many of them are open source software. For .Net languages a popular choice is NUnit. Installation of NUnit is really straight forward. Just downloading the latest version of NUnit from their web site and installing it in to the workstation. In this professional practice NUnit version 2.4.8 was used which was later on updated to version 2.5.10.

3.3.2 WatiN

There’s also couple different open source software to emulate user actions in browsers for .Net languages. In this professional practice WatiN version 2.0 RC was used which was later on updated to 2.0 Final. WatiN can be downloaded as a compressed package where you can find different versions of WatiN.Core.dll to match your version of .Net framework used in the test solution. In this project the .Net framework was 4.0.

3.3.3 Controlling browsers

To control Firefox JSSh plug-in needs to be installed. The plug-in comes within the WatiN installation folder and it has different versions corresponding to Firefox version. The plug-in supports Firefox versions from 2.0 up till version 4.

For controlling Internet Explorer Microsoft.mshtml.dll and Interop.SHDocVw.dll needs to be added into the references of the test application. These files are also included in the WatiN installation folder with versions corresponding to .Net framework used.

3.4 Additional software

There is lots of additional software that can help developer to build GUI tests. In the next few paragraphs is introduced a couple of software which are not mandatory for the test environment but increase the productivity of building test cases and make developers work easier.
3.4.1 Firebug / Internet Explorer developer tools

Firebug or Internet Explorer developer tools are plug-ins for the browser. These plug-ins are used to determine HTML elements unique attributes so that the elements can be separated from each other. This is crucial for the tests because if the attribute found is not unique WatiN will control the first occurrence of element with same attribute. Using unique attributes also increases the reliability of the test because right element is found even if it has changed its place in the web site. Both plug-ins work by using an element selection tool clicking on desired element to be controlled and the plug-in will show HTML code of the selected element. Firebugs function is demonstrated below (Picture 1).

![Firebug plug-in determining name attribute of Google’s search button.](image)

Picture 1: Firebug plug-in determining name attribute of Google’s search button.

3.4.2 TestDriven.Net

TestDriven.Net is a small plug-in for Visual Studio which enables you to run tests or test fixtures with a single click of your mouse. TestDriven.Net supports all Visual Studio versions and a dozen unit testing frameworks so it can be used in similar projects even if the frameworks are different.
3.5 Configuration

Default settings for tests should be in a different config file to store common data that all test cases use. This helps for example when the site URL, login information or site id changes so updating these values only to the configuration file is needed. This increases the dynamicity of the tests. To access these settings the user can do as in the following example (Code snippet 1).

```csharp
string address = ConfigurationManager.AppSettings["wwwAddress"];
```

Code snippet 1: Getting values from configuration file

The configuration file should also include a section for NUnit which will setup the apartment state to single threaded apartment shown below (Code snippet 2). Other way to setup the apartment is to use [RequiresSTA] statement shown later on (Code snippet 3). The single threaded apartment is needed because Internet Explorer is not thread safe and running tests in Multi Threaded Apartment will cause an error.

```xml
<?xml version="1.0"?>
<configuration>
    <configSections>
        <sectionGroup name="NUnit">
            <section name="TestRunner" type="System.Configuration.NameValueSectionHandler"/>
        </sectionGroup>
    </configSections>
    <NUnit>
        <TestRunner>
            <add key="ApartmentState" value="STA"/>
        </TestRunner>
    </NUnit>
    <appSettings>
        <add key="wwwAddress" value="http://www.address.com"/>
    </appSettings>
</configuration>
```

Code snippet 2: App.config
4. Developing tests

Developing of tests is done with C# with the help of NUnit and WatiN frameworks. The test are divided to different classes by their subject so for example all tests that are testing calendar are in one class and ones that are testing poll are in another class. All test classes are using a base class where they inherit commonly used methods and variables. This is done to decrease duplication and increase the dynamicity of the tests.

4.1 Techniques and programming languages

C# is used to build classes, test methods and any needed variables. The code is written in Microsoft Visual Studio which is connected to Team Foundation Server. WatiN is used to control browsers. NUnit enables to automate the running of test fixtures and test cases. It is also used to make assertions that for example: given value is true or two given values are equal.

4.2 Building test cases

In this chapter is described the workflow of developing test cases mainly focusing on how the tests are build from the beginning without test recorder software.

4.2.1 Test descriptions

Building test cases starts with reading descriptions from TestDirector. The description includes where in the web site the operations take place, what should be done and what is the expected outcome of the operations. The test should then be manually verified that it can be done.

4.2.2 WatiN Test Recorder

WatiN Test Recorder is an independent open source project which main goal is to automatically generate WatiN code from users’ actions. This software is still on beta stage and doesn’t function so well yet that you wouldn’t have to modify the code it generates. Also the recorder sometimes writes unnecessary code or doesn’t always use unique attributes when finding elements. For these reasons I didn’t use this software during the professional practice.
4.2.3 Hand written tests

WatiN elements have been mapped from HTML elements to WatiN classes so that the elements have logical names and the user doesn’t have to have comprehensive knowledge of HTML language. This is one reason why WatiN is really easy to assimilate [4].

With WatiN the user can control any website with HTML elements. Controlling an element is done by typing where to look for the element, what kind of element and with what conditions. The conditions are determined by looking up a unique attribute of the element as seen below (Picture 1). In the example is demonstrated a simple test case where browser navigates to www.google.com, finds a text field and writes WatiN into it, presses search button and verifies that the word “watin” is found from the result (Code snippet 3).

```csharp
namespace WatinTests
{
    [TestFixture]
    [RequiresSTA]
    public class ExampleTests
    {

        [SetUp]
        public void SetUp()
        {
            // SetUp is called before every test case
        }

        [TearDown]
        public void TearDown()
        {
            // Method is called on failure or after each test
            // If anything needs to be cleaned before next test
            // this is the place for it.
        }

        [Test]
        public void GoToGoogleAndTryToFindWatiN()
        {
            // Go to google with IE
            using (Browser browser = new IE("www.google.com", true))
            {
                // Type watin and push search
                browser.TextField(Find.ByName("q")).TypeText("WatiN");
                browser.Button(Find.ByName("btnG")).Click();

                // Verify that text "watin" is found from the results
                Assert.IsTrue(browser.ContainsText("watin"));
            }
        }
    }
}
```

Code snippet 3: Going to www.google.com and searching for watin
TestFixture is the opening point of test class and by setting the [RequiresSTA] all tests are run in single threaded apartment. SetUp method is run before every test and TearDown method is run after each test or in case of test fail. It is also possible to use TestFixtureSetUp and TestFixtureTearDown which both are run only ones within one TestFixture, before all the tests and after all the tests. SetUp method could include making instance of the browser and making preparations for example go to a particular page and create some content. TearDown could make clean up operations as in undoing the operations done in SetUp and then closing the browser.

In the example (Code snippet 3) has been used using statement which will automatically make new instance of the browser and close it after the test is done. This method is otherwise good but if the test case should be tested with multiple browsers the using part should be duplicated and replace the Internet Explorer with Firefox. A workaround to avoid making duplicate code is to create a method in the base class which will find out from the configuration file which browser should be started. This method is then called in the SetUp method so every test is run with one browser. After all tests have been run changing the configuration file to another and running the tests again will run the tests with the second browser.

To verify if the result is what it should be Assert is used which is a property of NUnit. The problem is that assertion will sometimes fail even if it shouldn’t if the web site is functioning slowly. This can happen when large amount of data is being processed. To avoid such falsely failing asserts you can add waits. One way to add wait is to use Thread.Sleep. For example System.Threading.Thread.Sleep(1000) will cause the main thread to wait for 1 second. This might eliminate the assert problem or not because the loading time of the web application can vary a lot. Safer way is to use WaitUntilComplete() or WatiUntilContainsText("someText") methods of WatiN. By doing this the test will not be waiting for nothing or it will wait just as long what is needed.
4.3 Running tests

The tests can be run straight from Visual Studio with the TestDriven.Net plug-in or with the test runners that come with NUnit installation. When developing tests the tests are run multiple times on the users’ workstation and after verifying that they work as planned they are uploaded to the server. This chapter demonstrates how the NUnit runners work.

4.3.1 NUnit GUI runner

NUnit GUI runner enables the user to run all tests, test fixtures or specific test cases. The runner reports if test have failed or encountered an error. The outcome of the tests is easily seen from the graphical presentation green meaning passed, yellow meaning skipped or ignored and red meaning fail or error. In the case of fail or error the report will show which tests have failed and why. The GUI runner is demonstrated below (Picture 2).

![Picture 2: NUnit GUI runner showing test results](image-url)
4.3.1 NUnit console runner

NUnit console runner enables same functions as the GUI runner but instead of the graphical presentation of the outcome there will be plain text. To run tests the user has to navigate with command prompt to the directory where the test assembly is in this case “C:\WatinTests”. Next step is to make a command where first is specified the path of the “nunit-console.exe” inside quotation marks. In case of 32-bit operating system “nunit-console-x86.exe” is used. Second part is to define optional parameters, test cases and test assembly. The test cases can be chosen from Namespace, Test Fixture or Test Case. You can have multiple choices by separating them with a comma [5].

To select only some of the tests NUnits /run: option should be used. It takes names of Namespace, TestFixture and TestMethod as parameters for example: /run: WatinTests.ExampleTests.FailingTest which would run only one test case called FailingTest. The picture below demonstrates how to run all tests from that assembly and how does the console runner show the results (Picture 3).

![Picture 3: NUnit console runner showing test results](image-url)
5. Automating test cases

For automating project builds there are many different software on the market. Tieto has been using MSBuild and CruiseControl.Net for this purpose so this chapter will tell how to configure them.

5.1 MSBuild project file

MSBuild is used to build applications and execute third party software. MSBuild is integrated to Visual Studio but it also enables to build software in an environment where Visual Studio is not installed. To do this a XML project file needs to be created. In the case of practice the project file included Project, PropertyGroup and Target elements [6].

The Project element is a root element which has all the other elements inside it. PropertyGroup element includes the path of source files for the CMS, destination path where the solution will be deployed and path for NUnit console runner. Target element includes all the operations that should be done for example copying of the source files, executing database script and execution of the test cases.

To run all the tests with both Internet Explorer and Firefox one workaround was made. The Target element has execution command of NUnit console runner which will run the test assembly resulting that all tests are run with Firefox. After the tests have run a second test solution configuration file is copied to the solution folder. This will overwrite the existing configuration file and when the NUnit console runner is executed second time all the tests are run with Internet Explorer. This is shown below (Code snippet 4).

```xml
<Exec Command="$(NUnit) /nologo /noshadow Tieto.Elevation.Tests.GUI.dll" WorkingDirectory="$(Destination)" ContinueOnError="true"/>

<Copy SourceFiles="$(SourceDirectory)\Source\Tieto.Elevation.Tests.GUI\AppIE.config" DestinationFiles="$(Destination)\Tieto.Elevation.Tests.GUI.dll.config"/>

<Exec Command="$(NUnit) /nologo /noshadow Tieto.Elevation.Tests.GUI.dll" WorkingDirectory="$(Destination)" ContinueOnError="true"/>
```

Code snippet 4: MSBuild project file execute commands for running tests
5.2 Automating test runs with CruiseControl.NET

CruiseControl.Net is used for automating project builds by monitoring the solution directory and triggering build when modifications have been done. After the build results are shown to the developer. CruiseControl.Net has two applications to view the status of projects: Web dashboard and CCTray application. Web dashboard is a web site where you can see the status of builds, trigger a build and watch build history of each project. CCTray is a small application installed to your workstation which informs the build results and you can trigger builds [7].

In the case of GUI test building the project every time when modifications happen wouldn’t work because running all tests takes over 4h. Best solution is to set the tests to run every night. At this point the tests are run by force build which means that they are run only when the build is triggered from the Web dashboard or CCTray.

When a build is triggered CruiseControl.Net will run tasks that are in the configuration file. In the professional practice the configuration file included two tasks one for building the test solution and one for running the MSBuild project file. CruiseControl.Net just executes the MSBuild.exe with parameters what to build and then makes a report of it. Below is an example of the MSBuild report (Picture 4).

![MSBuild report](Picture 4: WatiN test results in MSBuild report)
6. Conclusion

In this chapter will be some of the problems headed in the professional practice, overall success and improvements for the future.

6.1 Problematic areas

During the professional practice I’ve encountered many difficulties which nearly all have been solved. Main difficulties have been the behavioral differences between different browsers. The documentation isn’t so large with WatiN which

6.1.1 Consistency with browsers

Throughout the professional practice I have faced lots of challenges mainly concerning Internet Explorer. Till this date I haven’t found answers why Internet Explorer doesn’t always work according to the documentation. For example Internet Explorer doesn’t seem to find a frame inside a frame as shown below (Code snippet 5). In this case attempt is to find from a web site with three frames a frame which name is “Main” and in the first frame inside it click a link. Both browsers should work with same commands but somehow this workaround seemed to be the only fast solution to overcome the problem.

// With Firefox
Frame main = browser.Frame(Find.ByName("Main"));
main.Frames[0].Link(Find.ByName("Ok")).Click();

// With Internet Explorer
browser.DomContainer.Frames[2].Frames[0].Link(Find.ByName("Ok")).Click();

Code snippet 5: Looking for frame with name “Main” fails in Internet Explorer

Second problem occurred only with Internet Explorer 6 on the server side. The CMS has an auto save function that when a user starts to write new content the page is saved after one minute. The test was to verify that the auto save happens on time. Test would start by creating a new content page and typing a title “Auto save” for it. The test would then wait until the auto save was triggered close the page without saving and then verifying that the incomplete page would be found with the “Auto save” title. Test was running fine with Firefox but with Internet Explorer 6 when the content page was created and the browser started to type in the title field it still waited for some items to be loaded which caused auto save to happen before the title was written. For this problem I haven’t found a solution yet.
6.1.2 Auto generated id attributes

Another problem occurred with a feature of the ASP.Net where it generates automatically id attributes for elements. For example if an element is placed in ContentPlaceHolder and you are trying to find an element inside it by its id attribute it will work fine when searching as shown below in row one (Code snippet 6). But if the element is moved out of the ContentPlaceHolder test will cause an error with ElementNotFoundException. Although this doesn’t happen so often it still might break some test cases. To overcome this problem finding of the element could be done with some other attribute. If this isn’t possible you can use regular expression as shown below on row four and five (Code snippet 6). The regular expression ignores the beginning of the text so only text “Label1” is looked up from the id attribute. This will remove the problem unless the name of the label is changed or there will be added more labels with same name.

1. string lblText = browser.Label(FindById(“ctl00_ContentPlaceHolder1_Label1”)).Text;
2. // With regular expression
3. Regex r = new Regex([a-zA-Z0-9\$]*Label1);
4. string lblText = browser.Label(r).Text;

Code snippet 6: Finding a label by ASP.Net auto generated id attribute

6.1.3 CruiseControl.Net and build time out

One problem occurred when starting to automate tests with CruiseControl.Net. The problem was that build failed but the report didn’t include any error message neither it contained any information about the test cases. At the same time all the tests were running fine in workstation and in server with NUnit console runner. After further inspection I found out that some of the tests were run because the CMS sites recycle bin had content pages in it created by WatiN. The solution was found when taking a second look at the CruiseControl.Net configuration file. The file contains a timeout element which was set to 2700 seconds and by increasing this value all tests run without a problem. This though isn’t a solid solution because if we increase the number of test cases it will take a longer time to run them and so result in a timeout at some point if the timeout value isn’t increased as well.

6.2 Success in overall

The professional practice in Tieto has gone better than expected to take in consideration that most of the techniques were unknown for me beforehand. There is approximately 70 test cases working at this time. Although lots of test cases have been done a lot of time has been wasted on solving problems.
6.3 Improvements for the future

Definitely a big improvement for the GUI tests would be a proper report which would be separated from the MSBuild report. The MSBuild report is so massive that it takes several minutes to load and because it’s so large it really isn’t so convenient to search for the test results.
References

[1] Unit Testing

[2] What is black box/white box testing?

[3] Connect to and Access Team Projects in Team Foundation Server
   http://msdn.microsoft.com/fi-fi/library/ms181475.aspx (3.5.2011)

[4] Element Class Mapping Table
   http://watin.org/documentation/element-class-mapping-table/ (1.5.2011)

[5] NUnit-Console Command Line Options

[6] MSBuild

   http://confluence.public.thoughtworks.org/display/CCNET/What+is+CruiseControl.NET
   (4.5.2011)