Effective Early Stage Model-Based Testing for an IT UI Application

Xin Bai
Alexander Ivaniukovich

The Microsoft Corporation
Redmond, WA 98052
xinbai@microsoft.com
aivan@microsoft.com

Abstract

Model-based testing is a technique which has been practiced by many software test teams. A prototype has been developed as a proof of concept (POC) for a User Interface (UI) application by many of the development teams. A combination of the two gives the project teams a capability to virtually work on an IT UI application at the early stage of design purely based on the function specification and come up with a solution with high confidence from both sides of development and quality assurance.

Keywords: Model-Based Testing, software testing, Spec Explorer, Visual Studio, UI, Sketchflow.

I. Introduction

In the world of software testing, the UI automation is always a challenge. Not only because it is hard to maintain the automated codes to accommodate a frequent UI change on its nature; but also, it is difficult to obtain the development codes, which contain the required components for UI automation at the early stage of the project cycle. As a result, we often observed that the UI automation work cannot be completed until the end of the project cycle and thus it provided a little value to the current project release (it may add more value to the next release as part of regression).

Although many project teams have tried to solve the issues by a team effort, model-based testing is an effective and proven strategy to tackle on the problem.

By practicing model-based testing, in the design phase of the software development, developers focus on creating a UI prototype to mimic the real UI and navigation, in the meanwhile, testers just focus on authoring a machine readable model based on requirements by using a tool to generate a complete and maintainable test case suite [1].

The strategy and solution presented in this article facilitates the project teams with a method to test a UI application with automation at the early stage of a product development cycle. The paper presents a case study of model-based testing, using Microsoft Visual Studio add-on Spec Explorer [2] as a tool to create a machine readable model based on requirements. The combination of the two efforts provides the project teams with the capability to create an UI application prototype at the early stage even without any production codes, to discover inconsistent and missing requirements in building a model, and to ensure the quality by finding the issues in the design phase.

II. Today's Automation

In today's software testing against an UI application, automation is not possible to accomplish at the early stage of design phase of the project cycle. It has always been a challenging task for testers to work on UI test automation while developers are working on a prototype in the software design phase. Even during the build phase, if the needed UI properties, for example, automation id, name, and class name for UI controls, are missing, it will block testers from continuing any UI automation work.

Traditionally, testers create test cases and automation test scripts per user behaviors based on the existing requirements. As a result, a static set of test cases with scripting are created during the build and stabilization phases. If UI design is changed, which is a normal practice during the development of an UI application, the test scripts need to be updated frequently and manually. It results in much maintenance work and is the number one complaint during UI automation. The same issue occurs when a new release comes into the picture.

Furthermore, testers manually design test cases and write test scripts based on requirements. In fact, each tester may create a different set of test cases based on one's experience. As a result, some use cases may be missed during this manual test case design and creation process.

In Figure 1, it displays the traditional software testing activities at different phases of Requirement, Design, Build, and Stabilization. It shows that testers normally start writing test automation scripts at the Build phase and they have to continue updating the scripts through the Stabilization phase, which is the later stage in the project cycle.

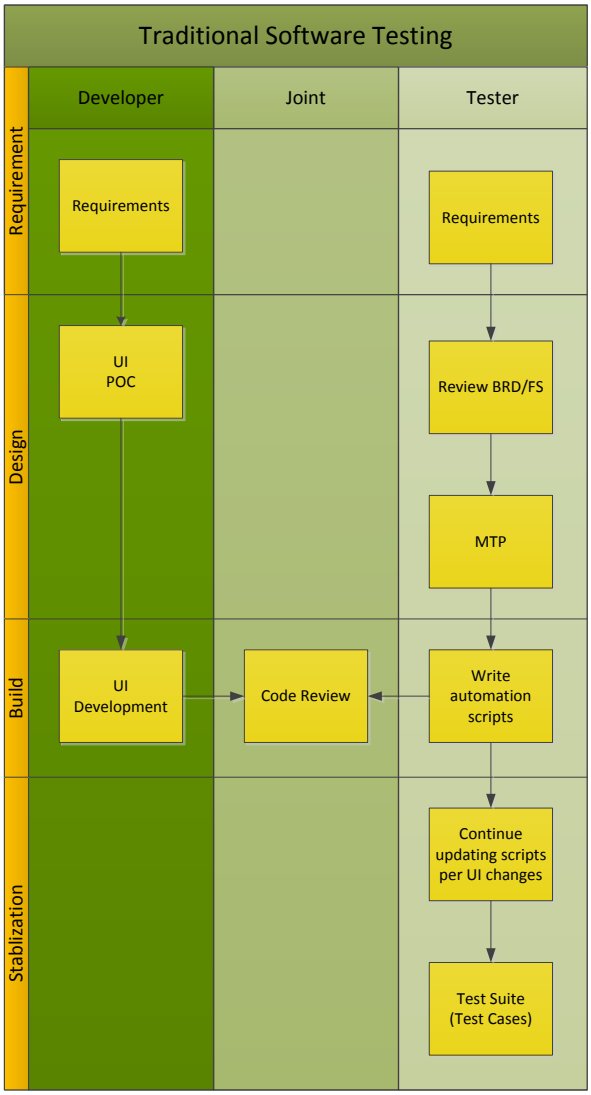

Figure 1 Traditional Software Testing

Basically, there are three major problems by using this traditional software testing process:

- Limited test automation against System Under Test (SUT) at the early stage of the software development cycle.
- High maintenance cost of UI test automation.
- No complete and automatically generated suite of test cases, which cover user behaviors hundred percent.

The vision and recommendation presented in the paper is to use a Model-Based testing technique as an alternative test method to fill the gaps above. Model-based testing is a test technique that system behavior is checked against a model. And the model can be built against the requirements as early as at the design phase.

Since the model is a simpler description of the system under test, it can help us to understand and predict the system's behaviors at the early stage.

It does not need to be an all or nothing approach while testers try to leverage model-based testing. At least, it can help them to fill the existing gaps in the traditional software testing. Particularly, it is feasible to be applied to a complex, state rich, and UI based application due to its internal nature.

III. Model-Based Testing

Model-based testing is a new level of testing, although it has been around for many years. It simulates the user behaviors based on a well-built model by testers. The model is an abstraction of the system under test from a particular perspective.

In Figure 2, it displays a high level conceptual architecture of model-based testing and its related activity components. It shows that testers can start building a system model at the design phase when developers are working on a POC solution, for example, an UI prototype using MS Sketchflow [3].

Testers may have already found some inconsistent, unclear, even missing requirements while building the model. The testers would feedback the findings to the project teams, which should have tremendously improved the requirement inspection effectiveness since it is at the early stage of bug detection process. Of course, it requires the testers to put some up-front effort in the project cycle.

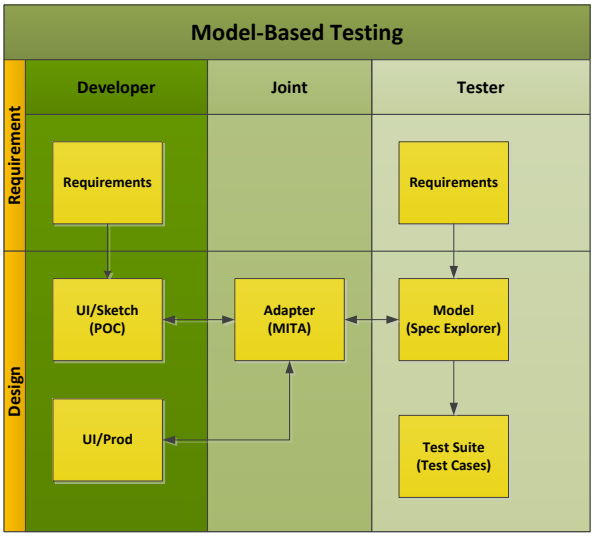

Figure 2 Model-Based Testing

IV. Benefits

A. Benefits

There are a couple of immediate benefits for the project teams by using model-based testing:

- Early stage testing to reduce costs
- Early automaton to detect bug
- Automatic generation of test cases
- 100% coverage of user behaviors
- Easy test script maintenance

In model-based testing, the suite of test cases is automatically generated out of the model by a tool. For example, MS Spec Explorer, it has been integrated with Visual Studio as an extension. There are some advantages by using this tool:

- The test cases are automatically generated.
- The suite of generated test cases covers the most complete paths, and thus it has a better coverage.
- It is easier to maintain the test cases. Each time when a new feature is added, we just need to update the model and re-generate test cases.

B. Challenges

There are some obstacles, especially when it is the first time testers use the model-based testing methodology. Basically, an adapter needs to be developed as a bridge between the UI Prototype and the MS Spec Explorer model.

- Testers are not comfortable to use it at beginning since they are not familiar with the technique.
- The initial effort to build a model is high.
- Need dedicated resources to work on it. A limited tester resource may have assigned to work on requirement inspection as well as the test plan.
- Specific technical skills and tools are required. Testers need to learn those technical skills.
- Testers are capable of making a design suggestion based on the practice of building a model.

In the next section, a case study will be presented on how to create a model, to generate test cases, to bind the model with a system adapter to generate some real test cases against the prototype, and finally to achieve the goal of performing an early stage automation and testing for an UI application.

Although there are some other tools in the market to help with the model-based testing. Here, we use Microsoft Spec Explorer 2010 to demonstrate a case study since it is integrated with Visual Studio well and a VS solution is presented in this case study.

VI. A Case Study

A Silverlight UI application is used as an example to elaborate the major steps during a model-based testing.

As a case study, all solution details assume that Microsoft Visual Studio 2010 or greater and .Net are in use. Also, it assumes that Microsoft Spec Explorer 2010 Visual Studio Power Tool and MS Sketchflow are in use.

A. Prototype

In the design phase, developers usually develop a prototype for the purpose of POC based on the existing requirements. By using MS Sketchflow, developers can quickly create a UI prototype which closely mimics the behaviors of a real UI application. It is not required to develop a middle tier or/and backend code.

In Figure 3, it shows a prototype solution named ‘Solution MitaApplication’ in Visual Studio. For simplicity, it includes a project named

'MitaApplcation', which has only one static class – 'MitaClass' with one static method – 'MainMethod ()'.

MITA (Microsoft Internal Test Automation) is a UI automation framework. In the case study, MITA is leveraged for creating a sample application adapter code. Actually, in the real world, testers can use any other UI automation framework, for example, they can use the Coded UI feature in Visual Studio [4]

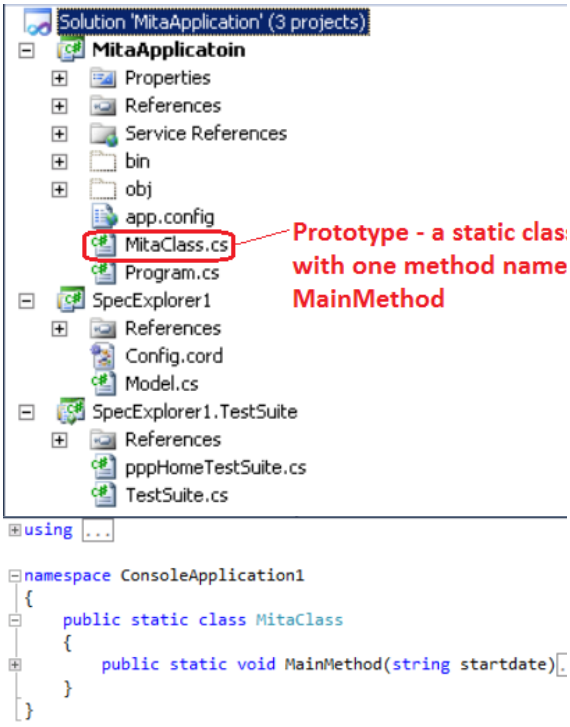

Figure 3 Sample of Prototype Solution

The method MainMethod () represents a test scenario for a UI dashboard prototype in Figure 4, which basically performs the following actions:

1. Launch a Silverlight UI application.
2. Create a new project.
3. Enter the project name and other required items for the project.
4. Enter Start Date for the project.
5. Save the launch project onto the dashboard.

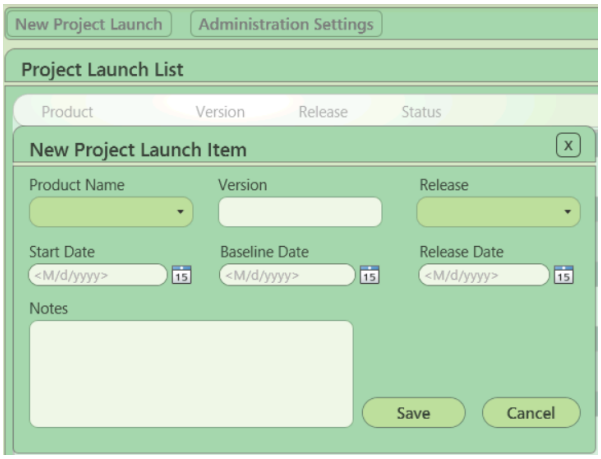

Figure 4 Prototype UI

In practice, there are some tools which help the developers to develop the POC of an UI application, for example, Microsoft Expression Blend [5], which is a design tool to help with creating an UI sketch at the early stage of the project. The tool is easy to use and it has rich feature to create an UI sketch without any middle tier and database associated with it. It enables a developer to deliver on the ideas faster.

B. Model

At the same period of time when developers work on a POC of the application, testers can work on a model per existing requirements, which simulates the system and incorporates all user behaviors. The task is beyond the traditional activities a tester normally works on at the design phase, which is only to inspect the requirements and write a master test plan.

Microsoft Spec Explorer 2010 Visual Studio Power Tool is a tool that extends Visual Studio for modeling software behavior, analyzing that behavior by graphical visualization, model checking, and generating standalone test code from models [2]. Although some initial effort need to be put in building a model, the tool itself is relatively easy to use and the test cases can be automatically generated. Besides, the suite of test cases is more complete than those manually created, and it is designed to cover user behaviors 100%. As a result, it enables testers to detect the bugs at the early stage, in the meanwhile, to maintain the test cases with a better flexibility.

Since Spec Explorer has been integrated with Visual Studio, after its installation, a 'Spec Explorer' menu is created within Visual Studio client as in Figure 5.

Figure 5 Spec Explorer Menu in VS Client

Now, testers can also add a new project of 'Spec Explorer Model' type as shown in Figure 6, which holds a C# model file as well as a configuration coordination file.

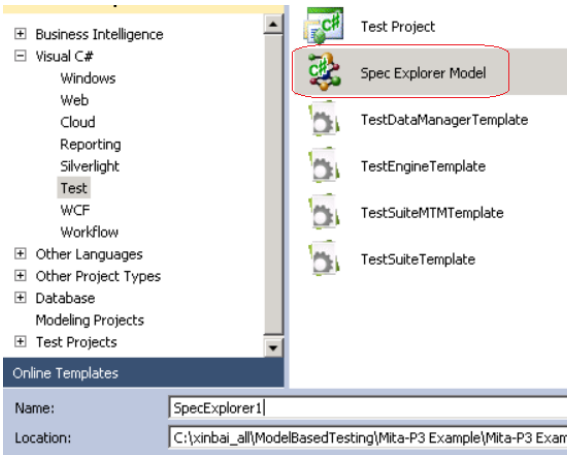

Figure 6 Spec Explorer Model project

While the project is created, there is an option to create a separate test suite project, which holds all of the auto-generated test cases out of the model by Spec Explorer.

In Figure 7, it shows the two projects described above in Visual Studio: SpecExplorer1 and SpecExplorer1.TestSuite.

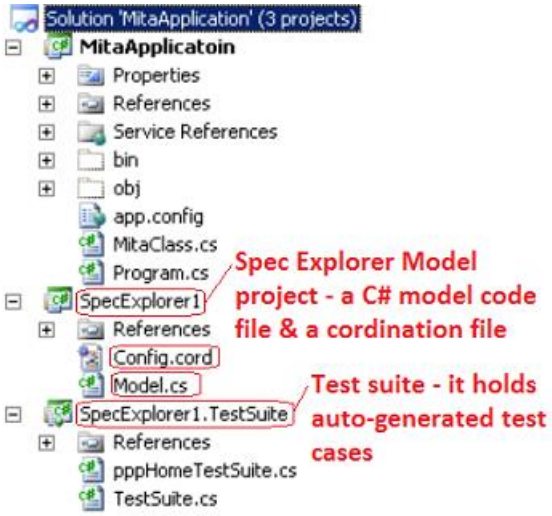

Figure 7 Spec Explorer Model project and Test Suite project

In the project SpecExplorer1, two files are created by defaults: one is 'Model.cs', which holds the C# model codes representing different rules; the other is 'Config.cord', which holds all actions, bounds, switches, and the state machine definitions.

In Figure 8, it displays the content of the model program 'Model.cs' for the case study. The rule in the model program is represented by the test method, here, it is 'MainMethod ()'.

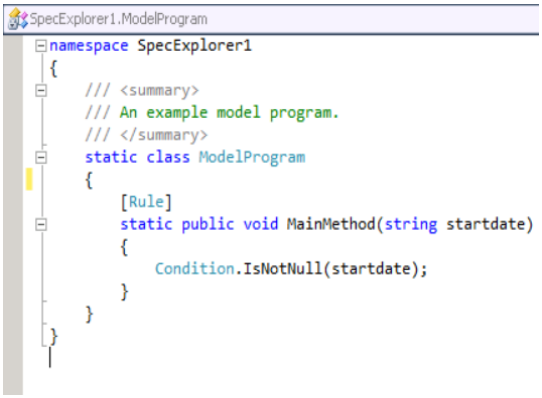

Figure 8 Sample of Model Program

In Figure 9, it displays the content of coordination file 'Config.cord'. It contains actions of the model which is to bind to either a model program or adapter functions. It also defines all of the switches, configurations, main state machine, sliced machines for specific scenarios.

```
// This is a Spec Explorer coordination script (Cord version 1.0).
// Here you define configurations and machines describing the
// exploration task you want to perform.

/// Contains actions of the model, bounds, and switches.
config Main
{
    action all ConsoleApplication1.MitaClass;
    switch StepBound = 10;
    switch PathDepthBound = 10;
    switch GeneratedTestPath = "..\\SpecExplorer1.TestSuite";
    switch GeneratedTestNamespace = "SpecExplorer1.TestSuite";
    switch TestEnabled = true;
    switch TestClassAttribute = "Microsoft.VisualStudio.TestTools.UITesting.CodedUITest";
}

/// This configuration provides a domain for parameter in the previous one.
config ParameterCombination: Main
{
    action abstract static void ConsoleApplication1.MitaClass.MainMethod(string startdate)
        where startdate in {"12/12/2010", "1/1/2011", "3/3/2011"};
}

machine pppHomeModelProgram() : Main where ForExploration = true
{
    construct model program from ParameterCombination where Scope = "SpecExplorer1.ModelProg
}

machine createLaunchProject() : Main where ForExploration = true
{
    (MainMethod) || pppHomeModelProgram
}

machine slicedModelProgram() : Main where ForExploration = true
{
    createLaunchProject || pppHomeModelProgram
}

machine pppHomeTestSuite() : Main where ForExploration = true, TestEnabled = true
{
    construct test cases where strategy = "ShortTests" for slicedModelProgram()
}
```

Figure 9 Sample Coordination File

C. Test Suite

As soon as a model is built based on requirements, Spec Explorer can generate automated test codes and save them in the Test Suite project, which is created in previous Section B. Here, it is the project of 'SpecExplorer1.TestSuite'.

While a model is being built, an exploration graph can be generated with states as well as transitions between the states. Testers can review the graph that helps them with the model design. In Figure 10, it shows the exploration graph in the case study. There are two states S0 and S4, and three transitions between them. Each transition takes a different parameter value of Start Date.

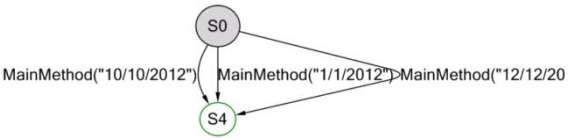

Figure 10 Sample of Exploration Graph

D. Adapter

In previous Section C, a set of test cases are generated by Spec Explorer after a model is developed. Within such a test case, action invocations don't call the system under test directly because they belong to modeling. In order to call a real SUT, an adapter must be developed upon the prototype so that Spec Explorer generates a new set of test cases by binding to the adapter. In Figure 2, it shows how UI/Sketch, Adapter, and Spec Explorer play together.

The adapter codes are part of prototype project 'MitaApplication', which is mentioned in the Section A. For a UI application, the adapter codes hold any functions and user behaviors based on the requirements. Figure 10 shows a sample code of the adapter.

```
namespace ConsoleApplication1
{
    public static class MitaClass
    {
        public static void MainMethod(string startdate)
        {
            WindowOpenedWaiter wmpOpenedWaiter = new WindowOpenedWaiter(UICondition.
            Process.Start("iexplore.exe");
            System.Threading.Thread.Sleep(5000);

            try
            {
                wmpOpenedWaiter.Wait();

                UIObject wmpWindow = wmpOpenedWaiter.Source;

                UIObject addressbar = wmpWindow.Descendants.Find(UICondition.CreateF

                //wmpWindow.Descendants.Find(UICondition.CreateFromName("Address and
                wmpWindow.Descendants.Find(UICondition.CreateFromName("Address and s

                wmpWindow.SendKeys("{ENTER}");
                System.Threading.Thread.Sleep(15000);

                wmpWindow.SetFocus();

                Button newProjectLaunchButton = new Button(wmpWindow.Descendants.Fir

                newProjectLaunchButton.Click();

                ComboBox box = new ComboBox(wmpWindow.Descendants.Find(UICondition.C
                box.SetFocus();
                box.Expand();

                ListBoxItem productitem = new ListBoxItem(box.Descendants.First());
                System.Threading.Thread.Sleep(2000);
                productitem.Select();

                System.Threading.Thread.Sleep(1000);
                box.Collapse();

                UIObject versiontxt = wmpWindow.Descendants.Find(UICondition.CreateF
                versiontxt.SendKeys("1.1");
```

Figure 10 Sample Code of Adapter

Here, in the study case, the adapter codes are in a C# file named 'MitaClass.cs', which has been discussed in the Section A. But for a complex application, it is better to separate the UI element codes. The definition of those UI element objects can be generated by using a tool, for example, the Coded UI feature in Visual Studio.

The Adapter codes for sketch UI can be leveraged to access the production UI controls for automation purpose if the AutomationId and other control properties of the sketch UI are designed to be the same as those of production UI. In Figure 2, it shows how UI/Prod, Adapter, and Spec Explorer play together.

VII. Summary

In software development life cycle (SDLC), early test automation and bug detection are always challenging, but badly desired by the teams since it is going to save many costs and ensure product quality. But, in the early stage, testers may not have access to the development codes, even the prototype codes. Therefore, testers are blocked from starting their test automation. This requires testers to think creatively and work out a different way to do testing. The model-based testing is a strategic methodology to tackle on the challenge.

Traditionally, on one hand, testers create test cases and automation scripts manually based on ones' experience. During the process, some of the important use cases may be missed. On the other hand, the test scripts are piled up as testers try to cover more and more use cases, which make it harder to maintain them, especially for a UI application.

The proposed 'early stage model-based testing for an UI application' can fill these gaps as an alternative way of testing. Testers can start doing some preliminary testing based on the model that they have built and thus find some design bugs at the early stage. Also, by leveraging some tools, such as, Spec Explorer, a complete suite of test cases can be generated automatically and maintained easily.

Finally, testers will achieve a better job satisfaction by doing a model-based testing since they are going to be more involved in creative architecture and design process. Also, they will learn new technology and tools and do more coding by working on the model and adapter with developers.

VIII Acknowledgements

We would like to thank Mrs. Poorvi Shrivastav and Mr. Raghavender Anegouni, the software development engineers in test, and Mrs. Swati Kaul, the team's test manager, in Microsoft MSIT, took time to review the paper or listen to the presentation demo, and provided some valuable feedback.

IX. References

[1] Nico Kicillof, "What is Model-Based Testing?" http://blogs.msdn.com/b/specexplorer/archive/2009/10/27/what-is-model-based-testing.aspx

[2] MSDN, "Spec Explorer 2010 Visual Studio Power Tool." http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745

[3] Expression Team, "SketchFlow: An Overview", http://expression.microsoft.com/en-us/ee215229.aspx

[4] Junfeng Dai, "Use Spec Explorer to do UI automation test", http://blogs.msdn.com/b/junfengdai/archive/2010/08/02/use-spec-explorer-to-do-ui-automation-test.aspx

[5] Wikipedia, "Microsoft Expression Blend", http://en.wikipedia.org/wiki/Microsoft_Expression_Blend