Exploring Cross-Platform Testing Strategies at Microsoft

Jean Hartmann
Principal Test Architect
Office Engineering - Test

Microsoft Corp.
Redmond, WA 98052

Tel: 425 705 9062
Email: jeanhar@microsoft.com

Abstract

The Office Productivity Suite including applications, such as Word, Excel, PowerPoint and Outlook, has now been available for Windows PCs and Apple Macintoshes for many years. During these years, the respective product and test code bases have grown significantly, with increasing numbers of features being added and requiring validation. When validating PC-based products, testers leveraged some of the benefits of the Windows platform including the availability of the .NET framework to implement their tests using managed programming languages. For test execution, they used Windows-supported mechanisms, such as COM/RPC, to communicate in- and out-of-process with the application under test. Thus, when teams needed to deliver related Office products on a different platform, such as the Apple Macintosh, test teams were faced with a dilemma - either attempt to port test cases, together with the required test infrastructure, or create new tests using the platform-preferred development/test environment. Both were time-consuming.

With the advent and rapid evolution of mobile platforms, such strategies are becoming more difficult to justify - implementing test suites from scratch is just too costly and slow. New test strategies, tools and processes are needed that promote the construction of portable tests and test libraries and enable testers to quickly retarget a given test case for different platforms and devices. This approach is particularly valuable when validating the common or 'core' application logic of each Office product for different platforms and devices, resulting in a more consistent level of quality for core functionality. It also gives test teams more time to focus on validating those product features that are unique to a specific platform or device.

This paper chronicles our ongoing exploration of platform-agnostic testing strategies during the current shipping cycle. It highlights the challenges that we have faced so far and attempts to illustrate and emphasize key concepts of our work using examples.

Biography

Jean Hartmann is a Principal Test Architect in Microsoft's Office Division with previous experience as Test Architect for Internet Explorer and Developer Division. His main responsibility includes driving the concept of software quality throughout the product development lifecycle. He spent twelve years at Siemens Corporate Research as Manager for Software Quality, having earned a Ph.D. in Computer Science in 1993, while researching the topic of selective regression test strategies.

1. Introduction

Companies, such as Microsoft, are embracing mobile technologies in the form of tablets and smartphones at a rapid pace. For the Microsoft Office Division and its suite of productivity tools and cloud services, this means complementing traditional desktop client applications for the PC and Apple Macintosh as well as cloud offerings, such as Office365, with native and browser-based mobile applications for the most popular platforms and devices including Apple's iPhone and iPad, Google's Android, Nokia's Symbian and Microsoft's Windows Phone 7.

To address the need for cross-platform testing, the division could either develop required expertise in-house or outsource the effort to established mobile application testing vendors. For manual testing, groups of testing professionals can be hired - they use a range of mobile devices in real-world conditions and attempt to find bugs. Vendors, such as Mob4Hire [1] and uTest [2], offer such services. Alternatively, third-party vendors including DeviceAnywhere [3] and Perfecto Mobile [4], provide automated testing solutions, which have the benefits of speed and control as well as scalability.

The decision taken by the Office organization was to build up the required in-house expertise based on an automated testing solution. As we embarked on developing such a solution, there were a number of aspects that we needed to take into consideration:

a) **Mobile platforms and devices have significant limitations, when compared to our desktop environments.** When testing in our desktop environments, we take certain things for granted. For example, we assume access to fully-fledged runtime environments, such as the .NET framework, to execute our managed test code and products, operating systems that are capable of supporting multiple processes to execute our regression tests out-of-process and sophisticated communications or marshaling mechanisms, such as COM/RPC, to allow us to interact our target applications' interfaces. These issues all need to be re-examined in the context of cross-platform test execution.

b) **Leveraging existing test automation environment is desirable**. We did not want to re-invent the wheel, when it came to defining a suitable, automated test infrastructure. Instead, we wanted to explore ways of seamlessly extending the power of our *existing* PC-centric test environment to encompass *actual* mobile devices rather than interacting with emulators[1]. We investigated mechanisms that would enable PC-based test clients to efficiently and effectively interact with devices applications. As a result, existing test cases could continue to be executed on the PCs with interaction being redirected at the mobile applications.

c) **Leveraging existing, automated test suites is desirable.** Given that the Office organization has built up extensive test suites for its existing desktop applications, the goal is to determine whether it is more cost-effective to refactor those existing tests for portability and share them across new devices and platforms versus implementing and then maintaining new, platform- or device-specific test suites. The decision hinges upon the amount of *product* code and functionality that can be shared across platforms and devices – if a large percentage of the core application logic is common across platforms and devices, then refactoring for portability is justified. In contrast, if products share little or no functionality across platforms and devices, then it would not be worth the refactoring effort; custom test suites would be more appropriate.

The remaining sections of this paper describe our ongoing investigation into cross-platform testing strategies. In Section 2, we explain how we design for test portability. Section 3 outlines our enhanced test infrastructure built upon our existing test tools. For Section 4, we highlight

[1] Based on the fact that the current emulators provided by platform and device manufacturers are not of the required quality and sophistication for large-scale, automated, regression testing of enterprise applications.

some process- and product-related issues related to this test initiative. In Section 5, we summarize our ongoing investigation by outlining an ongoing feasibility study that brings together our findings and concepts as a set of sample test cases and prototyped test tools. Finally, Section 6 presents conclusions and next steps in our effort to deploy this approach throughout the Office division.

2. Designing for Test Portability

Many thousands of automated, PC-based test cases have been written by Office teams over the years. The code manipulates a product's UI (User Interface) or OM (Object Model), making each test useful only in its original PC-based application context or *application-context-specific*.

Portable tests, on the other hand, should **not** be application-context-specific! The test code should not exhibit platform- or device-specific details and instead emphasize 'test intent rather than implementation'. The associated test libraries or *task libraries* as they are referred to within the Office division, need to support portability by exposing an abstract set of interfaces to the test code and encapsulating the nuances of the different application contexts (application/platform combinations). The following section describes the basics of designing such test cases and libraries.

2.1 Task Libraries

A key challenge for test portability lies in the design of well-abstracted task libraries for each product. Whether these task libraries are being implemented for new products or refactored based on existing libraries, appropriate abstract interfaces need to be defined. These interfaces need to reflect the application's common and unique properties for each context, that is, platform and device. Consider the Word word-processing product and some of its properties; we can define three new abstract interface classes:

```
interface IWordApplication
interface IWordDesktopApplication : IWordApplication
interface IWordBrowserApplication : IWordApplication
```

The first interface encapsulates the *common* properties of the application across all target platforms and devices. For example, in the Desktop and Browser versions of the Word application, testers can manipulate the canvas programmatically. So, the `WordCanvas` property should be declared in the `IWordApplication` interface to indicate that all implementers expose a document canvas to the tester. If necessary, a new `IWordCanvas` interface can be defined to abstract the canvas implementations and define public canvas tasks for all application contexts.

The other two interfaces reflect the *unique* properties of the application on specific platforms or devices. For example, only the Desktop version of Word supports COM Add-ins, so a `ComAddIns` property might be exposed in the `IWordDesktopApplication` interface, but not the `IWordApplication` or `IWordBrowserApplication` interfaces. Any changes to the base class `IWordApplication` with its common Word properties are then automatically inherited by the application-context-specific interface classes.

In the case of existing task libraries, once these abstract interfaces have been defined, the existing task library classes need to be modified to implement the new interfaces and the test cases are updated without the test code requiring modification. For example, for an existing Word task library class `DesktopWordApplication`:

```
/// <summary>
/// Class DesktopWordApplication is the base class that defines
/// tasks available for the Desktop Word application.
/// </summary>
public class DesktopWordApplication
{
    /// <summary>
    /// Gets the Word canvas.
    /// </summary>

    public WordCanvas Canvas
    {
        get
        {
            // return canvas...
        }
    }
  // Other Desktop Word Application members.
}
```

We now define a new interface `IWordApplication` containing only or all the public signatures from the existing task library class:

```
/// <summary>
/// Interface IWordApplication is the base interface that declares all
/// tasks available for the Word application.
/// </summary>
public interface IWordApplication
{
    /// <summary>
    /// Gets the Word canvas.
    /// </summary>
    WordCanvas Canvas { get; }
    // Other common Word Application members.
}
```

The existing task library class now implements the new interface:

```
public class DesktopWordApplication : IWordApplication
{
    /// <summary>
    /// Gets the Word canvas.
    /// </summary>
    public WordCanvas Canvas
    {
        get
        {
            // return canvas...
        }
    }
    // Other Desktop Word Application members.
}
```

The refactoring of the existing task libraries also presents opportunities for cleaning up any test code that explicitly references context-specific logic. This logic needs to be moved out of the test case and into the new, abstracted task libraries as an appropriate property, method or class.

2.2 Test Cases

With the task libraries structured and exposing a set of suitably abstracted interfaces, testers are ready to code against those interfaces. In the case of refactored task libraries, an additional step is needed to ensure that the same test cases can be executed against an application running on multiple platforms and devices. The test code needs to be updated by replacing each class Type with the Type of the highest, relevant interface in the inheritance hierarchy.

For example, consider an existing test case *Word Canvas Bold Test*. The test case logic declares a variable, `WordCanvas canvas`, and accesses `WordCanvas` members to automate the appropriate user tasks. Assuming that the `WordCanvas` class is now extending an abstract interface `IWordCanvas` with the same public member signatures, the test case code needs to be modified, so that the `canvas` variable is declared as Type `IWordCanvas`. Then, successive invocations of members on the canvas object will execute as they did before. Also, if a `NewContextWordCanvas` class is added to implement the IWordCanvas interface, the modified test case can be executed in the new context without further modification.

Existing Test Case	Updated Test Case
`WordCanvas canvas = …;` `Log.VerifyTrue(canvas.Selection.Bold, "Comment");`	`IWordCanvas canvas = …;` `Log.VerifyTrue(canvas.Selection.Bold, "Comment");`
`NewContextWordCanvas canvas = …;` `Log.VerifyTrue(canvas.Selection.Bold, "Comment");`	`<Same as above – no modification required.>`

Test cases and their task libraries are now ready to be used. The former will only contain code that references a set of abstract interfaces; they do not contain any platform-specific code. For that, an additional step is needed during test execution in which the test code is compiled and linked against the task libraries and in particular, *task library implementations* – these represent code snippets for exercising product interfaces on specific platforms and devices.

2.3 Task Library Implementations

These code snippets, together with the automation framework and communications mechanism described in Section 3, provide the third piece of the test portability puzzle. The snippets represent sequences of C# calls, which the test infrastructure maps to calls against the native application interfaces on a specific device/platform. The infrastructure then marshals those calls between the PC executing the test case and the target device. Successfully mapping and marshaling these calls is a key aspect as well as risk to the test initiative and will be discussed in more detail in Section 5. At this time, we are prototyping simple test cases, task libraries and task implementations for select Office applications on Android and iOS devices in order to explore the pros and cons of different approaches to mapping/marshaling problem.

2.4 Application Context

Devoid of any application context and without any direct class references, the portable test cases containing the abstracted interfaces now need to be bound to specific task library implementations at runtime. To achieve this, testers need to define a *test profile*, that is, specify additional test case metadata representing the platforms or devices on which they need to run. In Office, such data is captured in a *scenario* file. The test execution harness (discussed in detail in

Section 3) then provides a mechanism to load a value indicating the application context in which the test case should be executed. The mechanism then interprets the application context value and creates a factory object to a global data store. This factory object is retrieved at runtime to instantiate the appropriate task library implementation for the specific platform or device.

3. Supporting Test Infrastructure

3.1 Existing Test Infrastructure

Traditionally, Office test teams have relied upon a sophisticated, in-house set of test tools. A PC-based automation framework known as *Motif* and a distributed test management system called *Oasys/Big Button* are the two key tools for executing tests in batch. *Oasys,* or *Office Automation System*, a distributed test management system, comprises of a 'master' controller that interacts with a number of 'slave' PCs on which tests are executed one at a time. The interaction between master and slaves is maintained via agent software known as *OAClient*, resident on each of the slave machines. Also resident on each slave machine is the Motif automation framework and the Office application under test. The C# test code and associated task libraries are executed with the help of the Motif framework. Figure 1 shows the existing infrastructure with the OASYS server acting as master controller and two slave or OAClient machines.

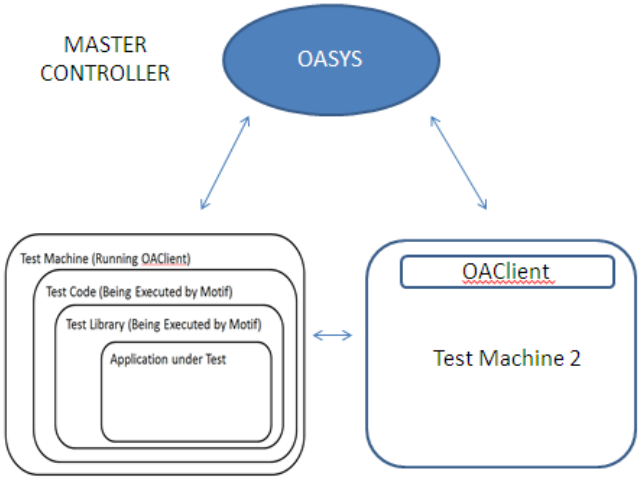

Figure 1: Existing Test Infrastructure

The system makes use of large server farms containing thousands of machines to maximize test throughput – thus the name *Big Button*. Given the scale of past investments in this testing infrastructure and the maturity of these tools, it was desirable to leverage them and not build another dedicated environment from scratch. The challenge, therefore, was how to extend the existing infrastructure to these new platforms and devices and at the same time, avoid porting of infrastructure, such as the OAClients, to the new devices and platforms. The latter would have been difficult due to the restrictive nature of the mobile devices and platforms mentioned in Section 1.

3.2 Extensions

Figure 2 illustrates how we are extending the existing test infrastructure. The extensions comprise of additional hardware and software attached to the slave or test machines. On the hardware

side, we envisage that for each PC-based slave or test machine, there is an attached Windows PC or Apple Macintosh acting as a *device host machine.* Each device host machine is, in turn, attached to one or more devices, for example, smartphones or tablets via USB cable. Device hosts are *required* for flashing and deploying applications onto the various devices. During test execution, whenever test or product failures occur, the machines are needed for collecting and passing back logging data to the test machine to aid debugging.

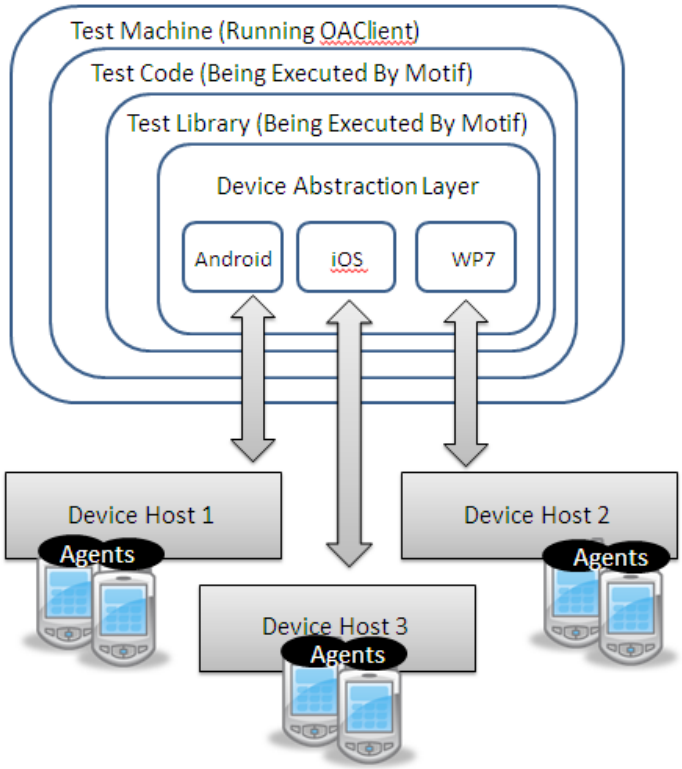

Figure 2: Extensions to Test Infrastructure

As each platform/device has specific installation and deployment requirements, a set of device management APIs, depicted in Figure 2 as the *device abstraction layer,* is required. Teams call into these service interfaces, when defining configuration scenarios prior to their test automation runs. The goal of this layer is to define a common mechanism for tasks, such as setting up and tearing down target applications prior to executing the body of the test case. During the test set-up and teardown phases, the test machine will send instructions to the target device via the attached device host and USB connection to deploy and remove applications.

Another key piece of software relates to the communications between the device and the OAClient machine. During test execution, this communications channel is responsible for 'remotely' stimulating the application under test on the device and receiving responses as well as log data. In essence, it forms a 'channel' between the PC executing the test case and the target device under test. We envisage communicating directly with the target device via wireless (WiFi) rather than via USB and attached device host. Although USB-based communications is faster and more reliable, it would require some brands of mobile devices to be 'jailbroken', which is illegal.

Building such a channel requires supporting infrastructure on both the PC test machine and each target device. For that, we decided to implement '*agents'* on each type of device, shown in Figure 2 as black bars. Corresponding software is available on each test machine to interact with the different agents. Each agent, which represents a minimal amount of code, helps to establish a bi-directional TCP/IP-based, that is, socket-based communications between the target devices and

test machine[2]. This channel enables the native application (APIs) running on the devices to be stimulated from the PC side by passing XML messages containing API and related type information to them via the agent.

No additional test infrastructure resides on the device. It is paramount that these agents are kept as lightweight as possible, because using any significant device memory and processor resources means that the application would not be tested under the same conditions as when it is actually being deployed.

3.3 Supported Test Scenarios

The above approach was developed to support the following usage scenarios:

a) **Single-client scenarios** – tests need to validate features of an individual Office application on a specific mobile platform or device. Scenarios, for example, include the application opening a file on SharePoint, then editing, saving and closing the file, then repeating that operation sequentially across different platforms/devices.

b) **Multi-client scenarios** – these are tests that validate the features of an individual Office application, which supports concurrent interaction between users on multiple platforms/devices. For example, two users of Word's co-authoring feature may be editing and reviewing a common document residing on SharePoint. Another example is the Lync communications application with multiple users around the world, each with their Lync client, interacting via voice/video.

4. Related Issues

Whenever new testing strategies and tools are introduced into an existing software development process, there are process- and product-related issues that are impacted. Once portable test collateral has been created, a process-related issue is how to maintain these portable tests over time. Rather than each platform team maintaining its own independent test suite, the responsibility for test maintenance is now shared across platform teams. Teams need to have an agreement or understanding in place, so that whenever product or test failures occur on a specific platform, the team owning that product/platform combination takes action. This action could result in updates to product, test cases, task libraries or task implementation.

Creating a common test suite across platforms may also lead to a redistribution of available testing resources with one part of the test team maintaining this shared test suite to validate the core functionality of the application against all platforms with the rest of the team conducting user interface testing of the product, specific to each platform. This assumes that common or 'core' product functionality has been identified and can be tested separately[3].

Product-related risks include the correct mapping of application interfaces to their corresponding task implementations and the marshaling mechanisms between the test PC and the 'remote' devices. Given that task implementations must be available in C# and Office mobile applications, are currently implemented in a variety of programming languages including Java and C/C++, 'stub generation' strategies and tools are needed that keep the two worlds in sync. Moreover, some applications expose their test interfaces using Microsoft's COM/RPC. If tests are to be maintained in C# on the test machine, how do we communicate with these COM-based interfaces

[2] At the time of writing, we are investigating different IP-based communications protocols including HTTP.
[3] For example, it may have been identified as part of a product refactoring effort that has led to use of the model-view-controller (MVC) design concept.

on the remote devices? Do we port the COM/RPC stack or redefine the test interfaces, keeping them simple and enabling a simpler marshaling mechanism or protocol?

5. Feasibility Study

In order to prototype and validate our test code and test infrastructure, we are currently conducting a small feasibility study. For validating our test infrastructure improvements, we established a private or miniature test lab containing a scaled-down version of our typical Oasys test environment. This 'minilab' included two Windows-based OAClient machines and a representative set of target platforms and devices including Apple equipment (Macbook with tethered IPod and IPad) as well as Android- and WP7-based mobile smartphones and tablets. At the time of writing, we are investigating the different deployment schemes for installing the operating systems and native applications as well as the custom, device-side agents that we need for device communication. This will enable us to reach a point where sample tests can be executed on the target devices. For validating our test code and task library improvements in the context of the supported scenarios, we are implementing a set of test cases representing: a) single-client, interoperability scenarios focused on file I/O where a set of common Office document types being loaded, saved and compared on each platform and device and b) multi-client scenarios that include the OneNote co-authoring scenario.

6. Conclusions and Future Work

In this paper, we described the latest findings from our ongoing investigation concerning the topic of cross-platform testing. We highlighted our motivation for this effort and outlined the key areas in which we made improvements, namely test code and associated task libraries, test infrastructure and test processes. Our exploration is far from over and we still face a number of technical hurdles including the implementation of logging and debugging support, etc. Future work will also entail pilot studies with select teams that will result in best practices and guidance concerning the design of portable tests and task libraries.

7. Acknowledgements

I would like to thank Curtis Anderson, Principal Test Manager of Office Engineering Test, Tara Roth, Corporate VP of Office Shared Services, Mary Smith and Rob Daly, Principal Test Managers for their ongoing support. I also want to express my gratitude to all cross-platform testing initiative team members for their contributions, support and discussions concerning this work. In particular, I would like to thank Forrest Dillaway, Julio Lins, Richard Wright, Jay Daniels, Ashley Tran and Jeffrey Weston. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

8. References

1. http://www.mob4hire.com
2. http://www.utest.com/what-we-test/mobile-application-testing
3. http://www.deviceanywhere.com/
4. http://www.perfectomobile.com/