Notes on Mobile Application Testing

And Its Automation

Industry at a Glance

Current Situation:

People are accessing the Internet and digital content in general more and more frequently from mobile rather than fixed platforms. It's only in the past two years that the players in these businesses have caught up to their customers in this trend. As is usual in the application development business, the quality assurance component of the development pipeline is about a year behind that.

The application development business has become used to open standards over the past ten years, with open technologies such as MySQL, Java, and the Internet itself. However, mobile application development is a throwback to the platforms wars of old, where one platform's application will not work on another, and cross-portability is lacking, not just in the code, but in the development pipeline making that code.

In response to these dilemmas, two things will shape up the mobile platform business over next several years. First, mobile platforms will consolidate to two or three distinct proprietary standards. Second, HTML5 is going to be big, as it gives access to the hardware-accelerated look and feel of a native application but is delivered as a platform-independent web page.

Current Mobile QA Situation:

In the past year the importance of automation in QA testing became apparent to the mobile development community. Up until this point, increasingly futile attempts at black-box testing of mobile devices were deemed sufficient. With virtually all mobile applications dependent on network data and off-device ('cloud') storage, QA testing of mobile applications extends *beyond* the local application to the network it's plugged into and must interact with. This point still escapes the broader industry in their approach to mobile QA.

With the advent of social media, the mobile application must not only be tested with the networks and systems it is plugged into, but also with third-party analogs over which the developer and tester have no control (such as Facebook).

Apple iOS at a Glance

Development Environment Advantages and Disadvantages

For Apple iOS, apps are developed entirely in Objective C on Apple's own IDE (Xcode). Despite the paucity of programming skills in the language vis-à-vis Java, Xcode provides the most thorough end-to-end development environment for a mobile platform. Lots of out-of-the-box graphical routines, debugging, and QA tools are provided via the integrated Instruments application.

Xcode compiles test applications to x86 native code and loads them in desktop-hosted handset Simulators that far-and-away are the best handset analogs in the mobile development business. Xcode's iPad and iPhone Simulators have the look, feel, speed, and general robustness as the same code compiled in ARM native for real handsets. A developer can achieve a very good approximation of an iOS application's performance and stability in the Simulator before ever deploying to a real device.

The iOS device family at present consists of two tablets (iPad 1 and 2), five handsets (iPhone series), and four PDAs (iPod Touch). Given that the PDAs are essentially iPhones minus cellular radios, there are basically seven devices spread over three generations for the developer and QA engineer to account for in the software pipeline. What's more, for both form factors display aspect ratios are identical, and there are no extraneous features such as physical keyboards to account for, making for a much simpler development and QA process.

Apple maintains tight control on their developers and developer environments. Real devices are limited to a maximum of one hundred units per company development account. Development machines that can compile code into Simulators must be provisioned via Macintosh Security Keychains and must be registered to owners of Apple Developer Accounts. Provisioning of builds is controlled in four tight schemas - Development, Test, Beta, and Release - each with its own different set of rights, permissions, and restrictions.

Distribution of applications is through one portal: The Apple App Store. All applications must be approved by Apple - facilitated through an Apple contact assigned to the account - before being posted to the App Store. Typical time-to-App Store market from upload to Apple for review, assuming there are no problems, is anywhere from a week to ten days. Smaller applications from smaller companies naturally receive less attention, hence greater delays.

The App Store bottleneck precludes currently popular models of software development, such as Continuous Integration and Agile Scrum methodologies, from being particularly effective. Defects that make it into a release are going to be stuck in that release for weeks before any fix or patch can be rolled to the application. Hence, iOS application QA by nature has to

be much more thorough than current standards for other platforms or web services that can be fixed and updated in near real-time by their respective owners.

Google Android at a Glance

Development Environment Advantages and Disadvantages

Android OS is a stack built on a Linux kernel for a file system, standard C middleware for hardware control, and applications developed in open-source Java classes (Apache Harmony) with numerous special classes for Android alone (standard dialogs, multi-touch gestures, etc.). The proprietary Davlik virtual machine runs Java applications. Android applications can be developed by anyone with Java programming knowledge.

Android development uses the Eclipse IDE and the Google-supported Android Development Kit on any major computer platform (OSX, Windows, or Linux), making Android development truly open-source and open-platform.

Distribution of applications is through the Android Marketplace and several derivative subsets of the Android Marketplace (Amazon.com maintains their own Android Market, for instance). Android applications can be signed, updated, and released to market instantly, enabling popular methodologies such as Agile Scrum and Continuous Integration to be employed smoothly in Android software development on a near real-time basis.

Android code is not simulated as is the Apple Xcode iOS environment, but *emulated* directly through an Android Virtual Machine on the host development PC. The Android Emulator is very slow, very buggy, and is not a good analog of any real Android device – making it useless for QA purposes. Android quality assurance beyond bare code functionality has to be done on real devices.

The hardware universe of Android is vast, with several dozen devices released just in past six months, from four major manufacturers (Motorola, HTC, Samsung, and LG). These devices cover a wide spectrum of screen resolutions and screen aspect-ratios, and incorporate features such as hardware keyboards that both developer and QA must cover.

Android manufacturers can manipulate Android as they wish, leading to oddities such as HTC, Motorola, and Samsung devices running the same core version of Android but with user interfaces completely different among them. In addition, each cellular data provider has its own version of otherwise identical devices released for their network. So an 'HTC Inspire' on AT&T is also an 'HTC EVO' on Sprint and a 'Droid Incredible' on Verizon, each with its own UI tweaks and differences, complicating a testing strategy for coverage all the more.

Sikuli IDE: An Ideal Solution for iOS Automation

Brief Introduction to Sikuli:

Sikuli IDE is a tool maintained under license from MIT and is freely available. Sikuli is based on the Python language and is similar in operation to automation test harnesses that are industry standards for web application testing, such as Selenium.

Sikuli, however, has a critical difference compared to harnesses such as Selenium in that it is not built to test run particular APIs or technologies in defined tests. Instead, Sikuli has one embedded API based on the emerging technology of *machine vision*; Sikuli 'looks' at the screen the way a real user does.

Advantages of Sikuli:

When scripting a Sikuli test, an engineer tells Sikuli to find an element on the screen using screen captures of the elements themselves embedded into the script. Then using Python, the engineer specifies ways Sikuli is to interact with that element using mouse clicks and keyboard inputs.

As a result, a breakthrough of sorts is achieved: Sikuli testing scripts look for and interact with elements on a screen *exactly the same way a human does*. What's more, complete API and platform independence is achieved this way. No matter the application, API, whatever, Sikuli can interact with it. *If it's on a screen*, *you can test it with Sikuli.*

There is also an ever-increasing level of sophistication to the Sikuli scripting environment. On one hand, a black-box tester can put together - in fairly short order - a sophisticated automatic application tour script. On a more advanced level, Sikuli scripts can be configured by an advanced SDET-level engineer (Software Design Engineer in Test [SDET] - a developer who writes software for the purpose of testing other software) using Python logic to run as defined pyUnit test scripts (Python version of the JUnit test wrapper) directly from a command line, importing specific classes, image directories, and logic dependencies to react to results of the test (IF, RANGE, etc.) for Sikuli to run.

Disadvantages of Sikuli:

Sikuli IDE's advantage of machine vision is also, in a way, its ultimate limitation. For instance, Sikuli can be set up to automate a web-application through a browser. However, if the browser is changed (Firefox instead of Chrome), or if even the UI 'skin' of the same browser is changed, Sikuli will not 'see' what it needs to see. This also occurs if you change the resolution or color-depth of the native display you wish to run your tests on. These are basic limitations and are obviously far-reaching in their scope.

Sikuli as the Ideal Automation Solution for iOS Development:

As mentioned before, the Xcode IDE/iOS SDK Simulator environment provided by Apple is truly a WYSIWYG version of the iOS experience in real hardware. The only limitations to testing on the Simulator are Alert events (real hardware ID is part of the event JSON payload from a network) and financial transactions through Apple's App Store Sandbox (real device synced with real iTunes account required). Everything else an iOS device does can be performed on the host Simulator.

The 'walled garden' of the iOS Simulator unintentionally plays to all of Sikuli's strengths and obviates the majority of its weaknesses. The Simulators are an identical native resolution irrespective of the host's native monitor resolution. There are only a few devices in the iOS universe, with identical UI elements, permitting the same Sikuli script to run on any Simulated iPhone. That fact, when combined with a fixed native host resolution and browser, allows Sikuli to use its strengths in profound ways.

For instance, let's say the iPhone application under test is a networked game, and the test is to load the application from source, compile and build the application, then launch it on the Simulator and interact with the game. Interacting with the software will generate an event that posts to a Facebook user's wall as part of the features of the application. Sikuli provides the following capabilities:

1. With Sikuli automation, you can have Sikuli launch Xcode, access the repository of the build using Xcode's Subversion utility, identify the master branch, and compile the application to a specific iOS Simulator.
2. Sikuli can visually verify build success by literally reading 'Build Successful' in Xcode. (Sikuli Machine Vision uses OCR in its machine vision. Sikuli can read!)
3. Sikuli then launches the application, logs in to a pre-defined game-user's account. By looking for visual elements and clicking ('touching') the Simulator, Sikuli will interact with the game.
4. After completing interaction with the game, Sikuli can launch the host Macintosh's Safari browser, go to Facebook, login with a pre-defined Facebook account, and confirm the Facebook wall event appeared and is in good order.
5. It can then launch an Oracle SQL Thin Client (as an example) and run a pre-defined SQL script on the application developer's own database and check their side of the Facebook-generated event.

In that test series, the automation script interacted with a development environment, compiled a build, interacted with the application, interacted with a browser on a Facebook account, and then checked the backend of the game developer's environment with a database query. Sikuli can run a linear end-to-end test of a mobile application, *and* the entire networked environment and social-media footprint it interacts with. There is *no other* automation tool that can remotely approach that flexibility and power of interaction across so many applications and technologies in one.

What's more, using the simple auto-tour implementation of the IDE, a good scripter can put that whole test together as a long linear 'tour' in about six hours. It would take at least twice that amount of manpower and debugging to automate the Simulator interaction alone in the Xcode Instruments Java tool.

At the SDET level, as the tours become more and more detailed, the tour scripts can be readily converted into defined unit tests using setup and teardown methods common to the Junit framework as implemented in Python ('pyUnit') and adapted to command-line precision. This affords a gradient of input and detail of tests where the whole team - from black-box tester to high-level SDET Engineer - can contribute to creating an end-to-end automation solution over the development lifecycle.

Between the nature of the iOS development environment and power and flexibility of Sikuli's machine vision, this is the ideal and easiest solution for iOS application automation today.

Android Automation: No Easy Answers (Yet)

Android Emulator Introduction:

The Android Emulator is a user-configurable 'Skin' built by the developer as a testing environment. The Emulator can be configured to have all the elements - such as hardware keyboards and custom display resolutions - seen in real Android devices emulated as a true stand-alone virtual device on the host.

This emulation makes Android Virtual Devices (AVDs) notoriously slow on the host, with large lag times between user input and resulting outputs on the screen. The display itself must be carefully calculated between the host's resolution and the AVD's resolution so as to always be a perfect integer; otherwise the AVD will alias its display to the host. This makes the emulator's graphics appear 'fuzzy' and low quality, not to mention slowing the emulator's response down (and increasing the host's workload) even more.

This peculiar requirement - combined with ever-growing display resolutions of real Android devices - leaves tough choices for a developer as the integer solutions line up between the host and the desired AVD screen. Some resolutions are very large on the host, others very small: each is different depending on the host's native resolution and DPI. In addition, the screen elements of a stock Android AVD are not accurate representations of real-world devices with their custom UI's and built-in elements: each differs from manufacturer to manufacturer and Android release to Android release (currently ten-plus in the SDK).

Sikuli and Android in General:

Inverse to the iOS experience, the Android AVD environment amplifies Sikuli Machine Vision's weaknesses and undercuts its strengths. Only with the most tightly-controlled host

and AVD environment will Sikuli consistently 'see' what it needs to test on an AVD screen, and although it is possible to fully automate the Android Emulator with Sikuli (and all the benefits it entails), there is one more problem: Emulators are a poor analog of real Android devices.

While good for a developer checking code functionality, running QA on an application in the Emulator will lead to an application that will run well on the *instance of the Emulator under test* and little else (not even on other Emulator instances, given all the variables that go into them). The only truly valid testing - manual or otherwise - for release-quality Android application sign-off is on real Android devices.

Robotium: 'It's like Selenium, but for Android'

Given the limitations of the Android SDK Emulator, an open-source solution has appeared - endorsed and nurtured by Google itself - over the past year that can automate Android applications in the Emulator but, more importantly, compile with the application and run on real devices. It is called 'Robotium.'

Robotium is as its slogan suggests: 'Selenium for Android.' Although it runs in Eclipse rather than its own IDE, like Selenium Robotium is a special API using custom methods to invoke Android OS functions, such as typing text, clicking Android UI buttons, etc. Through writing tests in the Eclipse framework and compiling the application under test with the Robotium 'TestRunner,' the application can be automated, and that compile can be installed and run on a real device. This eliminates the spurious results and slowness of the Android AVD environment and enables a developer to deploy the application to multiple devices to test concurrently, giving a semblance of the device coverage Android QA demands.

There is a limitation with Robotium and real devices, and that is debugging with automation. If you run the test and there is a problem, you must go back to the IDE, figure out where the test 'broke,' fix it, recompile the build, redeploy it, and run the tests again. For complex, interactive applications, this will involve a considerable QA effort to debug the application if the intent is confidence in stability across, say, three devices representing three Android SDK targets – a pittance of what's out there in the wild.

TestDroid: An Emerging, Proprietary Solution to Android Robotium Automation.

With the problems outlined above, it is only natural that a solution would appear given the burgeoning size (and money) in the Android development world currently. The most viable to emerge so far is a product known as TestDroid Recorder offered through a start-up company called BitBar.

TestDroid Recorder is the missing piece from the 'Selenium' analog Robotium claims to be by itself. It allows user inputs - directly from a real device - to be recorded as tests using Robotium methods. This allows a black-box tester (after appropriate setup of the Robotium TestRunner) to execute a manual script through an application, and TestDroid will record all

the inputs into one single Test class, which can then work as an automation script. A more sophisticated user, by inserting setup and teardowns in the test runner, can build a real Junit script of unit-tests. Since it runs as a recorder instead of manually writing the tests, TestDroid saves a lot of time both writing and de-bugging the tests, and lets the whole QA team contribute to the automation process.

What Android Devices to Test?

As mentioned before, there are a vast number of Android devices on the market. As an example, AT&T sells three iPhones today: the 3GS, the iPhone 4, and the new iPhone 4S. AT&T also sells approximately 30 Android devices from at least six manufacturers and for development using different SDK 'targets.'

An emerging solution to this intractable (and growing) problem is a company called DeviceAnywhere. DeviceAnywhere is an end-to-end QA service employing both a custom automation solution and 'banks' of devices remotely accessible to the developer, to run those custom automation tests on. It is a vast service for developers with vast budgets.

Outside of cost, the service is consistently challenged by the sheer number of both Android devices constantly released and guessing which ones developers will want to use. Lead times between a new device launching and its availability on DeviceAnywhere's service can be several weeks and involve modifications of contracts between the developer and DeviceAnywhere.

For a developer with limited (not Facebook-size) resources, there are two good strategies to use to cope with such a universe of devices.

The first strategy is to develop to the most popular devices. At any given time, only four or five models comprise the bulk (%70+) of Android's constantly evolving and growing installed base.

The other option is to develop to Google's own standard reference devices, the Nexus series. There are currently three such units, the Nexus One, the Nexus S, and the very new (as of this writing) Nexus Galaxy. Each represents a milestone in the Android SDK release schedule, and Nexus devices are pure Android; there are no custom elements and tweaks from manufacturers. Nexus devices are also 'unlocked' from any carrier or manufacturer, giving them a range of access to developers and testers alike not available in normal-release Android devices.

A potential solution to Android automation and development is outlined below:

Robotium + TestDroid + Top Five Devices:

The strategy is as the title describes. A small application company has completed the application to where at least some of its functionality is approaching code-complete status, and is ready to move from Emulator in dev to a compiled build running on a device. Assuming one developer and one black-box QA resource, the process would be as follows:

1. The developer sets up the Test Runner for the QA tester to execute a manual script.
2. The QA tester executes the manual script using a pilot test device.
3. Assuming success, the developer compiles the build with the Robotium test-runner class, and instructs the QA resource to deploy it to the other four devices, launch the tests, and return the results – noting any differences among the four devices in their behavior with the application.
4. As the software evolves in functionality towards function-complete, the developer reviews and modifies the automation tour into discernible unit-tests with hard-assert properties. This achieves a steady increase in granularity and tighter Pass conditions towards a release candidate as run by the QA resource, who consistently reports issues found in the tests as they are run.
5. The release candidate is function-complete and passes the tests on all five devices. Now with a tightly-defined and debugged test-runner, the developer and QA resource can try running the application on a multitude of different devices available to them from friends, family, anybody with an Android handset the developer can 'borrow' for a final confidence run.

For small to mid-size development efforts, this achieves thorough code coverage on a wide variety of devices. (Two developers and two QAs can run the tests against ten or more devices in a slightly scaled-up version of the same methodology.) Such an approach can approximate 90% of an Android application's potential installed user-base at a relative fraction of the cost of manually testing all those devices alone.

A Future of Sikuli Automation for Real Android Devices.

The Testdroid Recorder method outlined above is perhaps the best compromise between automation, device coverage, and manpower efforts for the small to mid-size Android development house. However it *is* a compromise and, like every other automation solution, cannot automate an end-to-end experience for an application's interaction with networks and storage beyond the Android device.

Both Development business and QA must become aware that applications are more and more dependent on these outside environments to work properly. Other than some notable stand-alone games (Angry Birds!), virtually every significant application for mobile today is dependent on inputs and outputs beyond the local device; and, given they must work together to function properly, ideally they should be tested together in the same manner.

The only automation tool flexible enough to work between so many environments at once is one that is API independent. The only automation tool that can do such a thing is Sikuli and its machine vision.

Controlling Real Android Devices with Sikuli:

A small, unsupported open-source project known as Android Screencast has emerged in the unofficial Android community. It is a piece of Java code you 'drop' into a specific folder of the Eclipse Android SDK development environment. The application works through a command-line interface of the Android Developer Bridge and enables an Android device plugged into the host machine to be displayed on its monitor, and to be controlled by the host with mouse clicks and keyboard events analogous to an iPhone Simulator.

There are significant limitations with Android Screencast. The first is that Screencast is being designed to display the broadcast device's display in any sized window on the host. This entails a layer of aliasing and resizing the input signal that makes the signal lose many frames and lag the actual device's display by several seconds between device input and output.

The second limitation is that Android devices do not output their display through the USB port by default, nor accept user-inputs coming back the same way. The only devices that do allow such are 'unlocked' devices. Unlocked devices from carriers - while popular - are far less stable than an identical device running the carrier's distribution of the Android operating system, even the same Android release.

A solution to the lock–problem: Google Nexus Reference Devices

Google Nexus devices are a series of three devices sold by Google specifically to enthusiasts and as a development reference. The primary advantage to these devices is they have reference releases of the Open Handset Alliance's 'official' distributions of Android SDK targets, and they are unlocked by design. Android Screencast will work with a Google Nexus device directly, without any unsupported hacks or Android OS distributions.

An opportunity emerges not yet extant:

While Android Screencast is a powerful tool, an ideal *automation* tool for Sikuli to access an Android device would be one designed to express the broadcast signal on a directly-addressed-per-pixel buffer to the host display. In other words, if the device display is WVGA (800 x 480 pixels) the output to the host is directly addressed at 800 x 480 pixels, no resizing. In an ideal implementation, this would involve using a very high-definition widescreen monitor oriented in Portrait.

With the pixels in the signal directly addressed in such an application, the output signal from the device would appear consistently and correctly on the host, and would lose no frames in time wasted on resizing the buffer. The Sikuli script would be written using the actual image assets of the application under test *before the application was ever compiled* and then run

directly on the Android device. The developer at that point could build the application to how it was envisioned to work and scoped in the test. The first build would be ready for a complete automation run, on a real device, after the very first compile in such a development cycle.

What about the issues of device testing outside of the actual fixes as bugs were found, such as the uninstall, recompile, re-install, and re-launch of the tests? This is where Sikuli's flexibility becomes a serious advantage: all those functions involve interacting with the Eclipse environment – and can be automated with Sikuli, as well. The developer would literally identify the fix, implement it, and then hit 'Run' on the Sikuli script again and the whole process - not just the unit tests themselves - would be completely executed. It's like having a black-box tester built into the development environment.

With a sufficiently developed piece of software, the same end-to-end test script as previously described for iOS emerges for Android: Testing the application and all things unique to the device that it touches in one complete test becomes possible.

And in an interesting potential trick, once such a Sikuli test passes to satisfaction you could set up the TestDroid Robotium Recorder in Eclipse, run the Sikuli script, and literally *automate your automation script* for other devices beyond the Google Nexus reference units needed for release candidate testing. Now that would be some automation!

Nobody has developed such a successor to Android Screencast, but such an application combined with Sikuli IDE has a potential to be 'the' solution that everyone is looking for today in the Android application development business.

Links:

http://sikuli.org/ - Project Sikuli

http://developer.android.com/index.html - Android Developer's Portal

http://code.google.com/p/robotium/ - Android Robotium Portal (Note Emulator video is just Settings panels, speed in Emulator with real application is incredibly slow).

http://bitbar.com/ The makers of TestDroid, and note the intriguing not-yet-released Test Server.

http://www.google.com/nexus/ The latest Nexus reference device from Google.

http://www.deviceanywhere.com/mobile-application-enterprise-testing.html DeviceAnywhere

http://code.google.com/p/androidscreencast/ Android Screencast

http://developer.apple.com/devcenter/ios/index.action iOS Developer Portal (requires Developer Connect account to access downloads and anything other than Featured Content and Libraries)