SHOWCASING THOUGHT LEADERSHIP AND ADVANCES IN SOFTWARE TESTING

LogiGear MAGAZINE

Mobile Test Automation

Test Automation Interfaces for Mobile Apps

By Julian Harty

Mobile Test Automation: Best Practices
By Abhijit Phatak, mAutomate

Leverage Automation to Make Mobile Testing Manageable
From LogiGear

Extending Test Automation to Mobile
By Edward Hill, HP

December 2012 | Volume VI | Issue 5

LETTER FROM THE EDITOR

Editor in Chief
Michael Hackett

Managing Editor
Brian Letwin

Deputy Editor
Joe Luthy

Worldwide Offices

United States Headquarters
2015 Pioneer Ct., Suite B
San Mateo, CA 94403
Tel +01 650 572 1400
Fax +01 650 572 2822

Viet Nam Headquarters
1A Phan Xich Long, Ward 2
Phu Nhuan District
Ho Chi Minh City
Tel +84 8 3995 4072
Fax +84 8 3995 4076

Viet Nam, Da Nang
7th Floor, Dana Book building
76-78 Bach Dang
Hai Chau District
Tel +84 511 3655 33
Fax +84 511 3655 336

www.logigear.com
www.logigear.vn
www.logigearmagazine.com

Submission guidelines are located at
http://www.logigear.com/magazine/issue/news/editorial-calendar-and-submission-guidelines/

In the November 2011 issue: *Mobile Application Testing*, I began my column with the statement, "Everything is mobile." One year later the statement is even more true. More devices, more platforms, more diversity, more apps.

It boggles the mind how fast the landscape changes. Blackberry has been kicked to the curb by cooler and slicker Apple and Android devices, and Microsoft is making a run with Windows Mobile 8 and the Surface tablet. As of this writing Windows Mobile 8 just launched and the tech press boasts "Nokia Lumia 720 (running Windows Mobile 8) is 'sold out!'" and Surface is "selling well."

Mobile used to mean phones. Now it's netbooks, smart phones, pads, scanners, mobile POS (point of sale) devices… it's data, consumer products, corporate application, financial services and games — the list goes on.

It's also interesting that while a lot of corporate IT staff are just now warming to supporting smart devices for internal users, the corporate market is the largest consumer of mobile computing. From mobile auto insurance data processing to the long existent mobile sales and inventory systems on every Coke, FedEx, FTD and DHL delivery truck around the world, mobile computing makes business run!

Inside software and product development, we know that along with the explosion in devices, in this new millennium, development teams must be leaner, more agile, more distributed and faster! But how? In the case of testing, the only answer is to automate more! Yet this answer has new twists today. We know dev tools on new platforms come first and that test tool development lags. The test tools for mobile are coming fast- yet the overwhelming majority of current tools are single platform tools. So how do we automate for so many platforms? What interface is best to automate? What issues will we miss by using emulators/simulators and not the real devices? Should we do security and performance testing for each release?

Our test strategies are going to have to change as well. A common situation I see is releasing a browser application with the focus on Internet Explorer first. Then add Chrome and Firefox, and maybe partial support for Safari. Now this seems simple. But what happens when you have to add mobile apps for Android, iOS and Windows Mobile? What will have to change in your test strategy?

Some companies have just managed to come to grips with significant cross browser test automation. Adding mobile platforms can put them right back into full manual regression testing mode. Mobile requires us to get better faster. I hope this issue helps you along the way.

In this issue, the LogiGear team shows the importance of understanding the mobile ecosystem; Abhijit Phatak shows us some best practices for mobile test automation; Julian Harty highlights what you need to know in order to have effective and reliable test automation for your mobile apps; HP's Ed Hill discusses the importance of keeping up with mobile testing trends and Motorola's Ittichai Chamavanijakul reviews the book *Android Application Testing Guide* by Diego Torres Milano.

In our next issue, *The Changing Landscape of Software Testing*, we'll examine major shifts in testing that are pushing teams, technologies, methods and tools to change at a rapid pace. How can you keep up? Find out in February! And don't forget, you can check out the editorial calendar on our website.

We wish you a great holiday season and happy new year going into 2013!

Michael Hackett
Senior Vice President
Editor in Chief

IN THIS ISSUE

IN THE NEWS

Video: Jean Ann Harrison on the challenges of mobile testing

For testers who like a good challenge, the mobile arena is the place to be. LogiGear magazine was able to catch up with Jean Ann at CAST and talk about what makes mobile testing so unique. Jean Ann's real-word example of testing a medical application designed for mobile deployment highlights just how complex the hardware and software interactions are that can impact software performance. What Jean Ann shares highlights the importance of collaboration between developers and testers in creating applications destined to go mobile.

Jean Ann Harrison is an expert in mobile testing. Jean Ann has been in the Software Testing and Quality Assurance field for over 12 years including 4 years working within a Regulatory Environment. Her niche is system integration testing, specifically on mobile medical devices. Watch the interview here.

IBM to Expand in APAC Region

IBM has announced plans to open a new software testing center at the University of Ballarat, Victoria, to expand application development in the Asia Pacific region. The IBM Delivery Centre at Ballarat Technology Park will employ 150 staff working in areas such as software application development, application support, application management and consulting services.

IBM began partnering with the University of Ballarat in 1995 following a contract with the Victorian Government to build the Southern Region Data Centre. It currently operates three buildings in the area which provide business services, process services, technology services and administration for IBM clients around the world.

Mobile Application Testing Market Sees Growth

A new study from ABI research states that the mobile application testing will exceed $200 million in 2012, with 75% of the revenue being generated from sales automation tools. The study concluded that growth will propel the automation market to $800 million by 2017.

The study also points out that mobile testing has developed independently of traditional testing. This has provided the opportunity for start-ups to enter the space and fill market demand gaps quickly.

These findings are from ABI Research's Mobile Application Technologies Research Service (http://www.abiresearch.com/research/service/mobile-application-enabling-technologies/), which takes a deep dive into various technologies that enable new and transformative applications. This research involves identifying early signals of how advances in areas such as Augmented Reality, HTML5, NFC, and Voice Recognition are translating into developer activity, as well as visionary predictions on how these enablers may reshape the industry in the future.

BLOGGER OF THE MONTH

Mobile Test Automation: Best Practices

Test engineers face a rapidly changing mobile application landscape, making mobile test automation a necessity.

By Abhijit Phatak, mAutomate

We know that mobile apps are becoming increasingly complex along with the technological advances in tablets and smartphones.

Test engineers have to address multiple challenges while testing data-centric mobile apps, including the growing diversity of device features, platforms, and technologies.

Fortunately, mobile testing automation can help take care of the majority of the functional and non-functional intricacies of app behavior.

Test automation can be considered a mix of Environmental, Behavioral, Performance and Complete Ecosystem testing.

Environmental Testing

1 – Using Real Devices

When it comes to environmental testing, my first piece of advice is to use physical devices for testing and not emulators. While we think that various tools available in the market can allow us to run automation on emulators, it is a possibility that they may not replicate actual devices.

Testing with actual devices ensures real time evaluation of apps for various environmental and device specific factors, such as benchmarking, analysis, and resolution. You also need to keep in mind the application behavior may be affected by the network type, latency, and carrier infrastructure in the target geographies. An automation testing strategy should emphasize periodically changing the available networks so that the entire spectrum of network technologies can be tested.

2 - Automated Network Switching

Switching networks can help understand the change in the application behavior and help identify any performance bottlenecks.

3 - Auto application installation through OTA

If your application supports different platforms and configurations, you need to verify the successful installation and upgrade on the target platforms. It is even more important now as there are different ways to distribute the software, such as via the internet, from a network location, or it can even be pushed to the end user's device. An automation testing suite is incomplete if it is not providing installation of a third party application automatically over the air.

4- Manage notification services

While executing the test case automation, the process should be capable of handling or simulating events such as receiving a call, SMS or email and things like “battery low” indicators. An automation system should incorporate these types of events and then simulate the same while running the test cases. Simulation of these events are a must before certifying any application.

To write the complete automation, you should definitely have a third party application running on the devices, which opens multiple threads and multiple connections to the server, performs long running operations, and so on. This will help the automation tool to analyze the performance of original applications and help procure the exact status of memory and CPU when it is being eaten up by another service.

Behavioral Testing

1- Screen Orientation Change

Most devices now support a screen orientation feature, so your automation strategy should include orientation testing.

On some platforms - including Windows Mobile & Android - the orientation change can be triggered by extending the soft keyboard. An automation ecosystem should include automatic switching of orientation so that the application can be tested on all UI changes.

In the case of Android, automation should define complete configuration change events for all the activities, so that the UI can be changed automatically at runtime.

Your automation strategy should always include simulation of a "no network" scenario on devices, for example: when a device is switched to airplane mode. This will help to understand the behavior of an application when there is no network access.

Emulators can assist with many aspects of mobile testing but verification testing should always be done with actual devices.

Testing on actual devices ensures real time evaluation of the apps on various environmental and device specific factors.

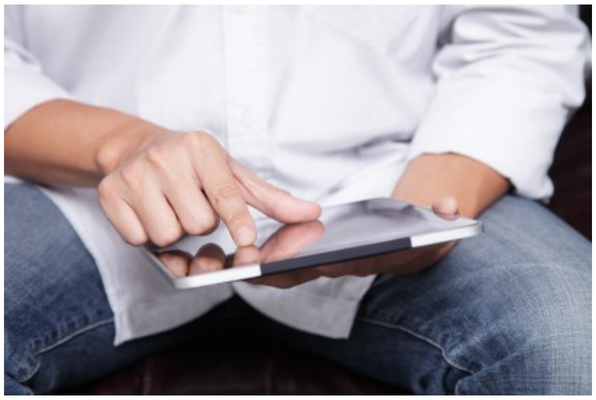
For applications that support different hardware features, we advise that you take appropriate measures to automate the use of a device's native capabilities. For example, if an application has the functionality to take photographs, the automation should cover this scenario while opening the camera interface and performing the relevant steps to take a picture.

A test of your application with active native features defines whether the application is compatible with a platform's native capabilities and also indicates whether any changes in the native features would affect your application. An automation tool or script should be capable enough to execute these hardware switches.

2 - Automatic simulation of location attributes

To explain this best practice for automation testing, I would like to share an interesting example to make you understand the use of a location attribute simulation.

Let's suppose your application's business logic has to show nearby ATMs in your current location. But if you are based outside the US and using a US server, your server would only provide the list of ATMs which are located in the US.
A good automation system should be able to simulate any difference in location. So if you run this application outside of the US, the latitude and longitude coordinates would be injected into the system from which you would get the list of ATMs.



3 - System event versus User events

System events are triggered by an application developer writing a piece of code into the application. For example: a developer writes a code to inject a button click event automatically. User events are those which are performed by the end-user, such as pressing a key on a device's keypad.

In both scenarios, the final system event as well as the user event will be triggered at the same location. Your automation tool should differentiate between them.

The stability and reliability of an application also depends on how the application handles improper user input. Try checking the behavior of the application by introducing exceptions, such as entering invalid data. For example, put characters in a field where numbers are expected. Or, introduce unusual situations, like turning off the network while downloading a file. This will help improve the stability of your application and lead you to any weak points.

To automate the rigorous flows, the automation system should provide a way that makes the test-cases data driven. This means that users can have a few sets of test data and then automatically insert multiple key value pairs.

Performance Testing

1 – Collect CPU and Memory Analytics

One of the best mobile test practices is to collect memory, CPU and battery analytics from time to time while testing . This information can be useful in identifying and addressing any problem areas.

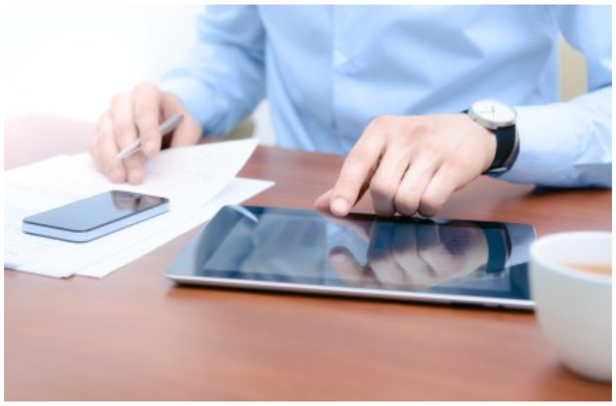
A CPU intensive, memory eating, and battery draining app is sure to be disliked by end-users and is likely to be a commercial failure. It is crucial to benchmark, measure and optimize your application based on these performance parameters.

But the question is how can automation make this analysis better? An automation testing tool should gather CPU, memory and battery statistics, as each test case executes. This way, the developer can get analytics in an application flow and can immediately identify the problem.

2 – Application Responsiveness

Another important non-functional requirement is application responsiveness. A responsive UI is snappy – it responds to inputs promptly, and doesn't leave its users hanging.
While planning the automation of your application, you should calculate the collapse time of the application's screens and the time required for screen navigation.

With the technology advancements and maturity in mobile platforms, the user experience has become a very important consideration. The app should not only look good but also perform well.



3 – UI Navigation Performance

UI should be effective and transitions should be fast. Your automation strategy should focus on UI response-time based on the actual time taken by the application to navigate and draw screens. Since the automation would be running on multiple devices, you should compare the response time on each screen and update the test team on the devices that are taking longer than others.

4 – Usage of Performance analytics tools

You can easily check the performance and memory usage of your mobile applications with any one of a myriad of tools available on the market such as Traceview and Eclipse Memory Analyzer for Android; J2ME profiler for Java phones; JDE memory Analyzer for BlackBerry; and Instruments for iOS.

These tools help by providing information about the memory usage, CPU cycles, etc. They also help debug applications and can create performance profiles.

Ideal Test Ecosystem:

Having seen the best practices of mobile application automation testing, let's see what would make an ideal test ecological system.

1 – Test & Result protocol

While talking about automation, you should clearly define the Test and Result protocol. Ensure that the automation tools/scripts define the protocol for failure with "correct reason of failure", "Screen shot from the device" and "complete logging information from the native shell".

2- Device Management

Device management is one of the most important practices recommended for an ideal test ecosystem. This infers that there should be a system to manage your devices; and categorize them on the basis of parameters like manufacturer, versions, OS, etc...

For example, there are a huge number of Android devices available in the market. If we want complete functional testing on a variety of Android devices, we need to automatically provision them via the cloud so that they can be directly used by customers and run the test cases across geographies.

3- Test Case Management

In any complex application there are a huge number of test cases ranging from hundreds to thousands and sometimes more. An ideal test automation management system should be able to store, organize, execute and view test cases and their results reports. Any number of test cases can be assigned to their target devices using automation, which in turn executes these tests automatically and provides complete results to the user.

4 – Result Reporting

This refers to reporting of crucial app statistics that help evaluate the overall application stability and readiness for the market. These reports are also helpful for iterative testing cycles.

We suggest that Result reports should be comprehensive and share the complete scenarios of testing and the reason of failure for failed test cases. ■

About Abhijit

Originator, innovator and creator for mAutomate a Cloud enabled mobile automation testing solution from Impetus Technologies. Abhijit has 10+ years of experience in the field of mobile applications and has worked on all mobile technologies starting from BREW, Symbian, Android, iOS, BlackBerry, Windows Mobile and .NET.

Cover Story

Leverage Automation to Make Mobile Testing Manageable

A high level of test automation will help assure the best quality mobile applications.

By Hans Buwalda, Aldo Kurnia, Do Nguyen and Joe Luthy

Mobile application testing can be done with minimal automation, but a high level of test automation will help assure the best quality mobile applications.

The most significant facet of mobile testing is to understand what goes on in the mobile ecosystem to be able to design tests appropriately. This sounds obvious, but due to the complex nature of the mobile ecosystem, it isn't. The multiplicity of devices and operating systems alone make for a very challenging testing matrix, but layered on that are a variety of hardware operational nuances. These include what are best described as interruption events; things like incoming call and message notification, change in network speed, and power warning notices.

The risk of having a test work for a while and then inexplicably break is high in the absence of a significant emphasis on up-front test design and structure. For example when testing an application download function, the outcome of the test is dependent on the application, the device and the OS version. What seems as basic as automatically installing an app and launching it without manual intervention can be surprisingly complex.

Design Then Test

For any automation effort the primary key to achieving success will be the approach. As a rule of thumb test automation is successful when the "5% rules of test automation" can be achieved, which means that no more than 5% of test cases should be executed manually, with no more than 5% of testing resources applied to reach that level of automation. This is aggressive, but achievable. Having this as a goal puts the focus on effective testing, not on test execution or automation scripting. Establishing an aggressive goal will help drive efficiency throughout the entire test team.

Anyone who has been testing for any length of time knows there are many ways to get a short-term boost in testing efforts. However, to achieve lasting results it is necessary to have a good high level organization of tests and an automation tool that can effectively execute tests and increase the scale of testing without adding overhead. Ideally, the automation tool should incorporate an agile keyword-driven automation framework. The proper organization and the right tool will enable mobile test processes that include all of these:

- Effectiveness. It must be robust and aggressive enough to find the hidden bugs and problems
- Efficiency. It must minimize time, costs and utilization of resources
- Manageability. Managing testing and test automation can be even more challenging than managing software development
- And...not boring or repetitive. Bored testers are not sharp testers. Monotonous execution tasks should be done by automation, so testers can focus on more creative and ambitious test scenarios

At LogiGear, our test teams are trained on the Action Based Testing method (ABT). ABT borrows from state-of-the practice object-oriented software programming that radically changed software development. The approach makes test design and development essentially one and the same.

ABT focuses on the use of action keywords to design tests for automation and the modular test design concept it promotes integrates especially well with Agile practices. In application we have found this approach enables test teams to consistently attain the 5% goal of test automation.

Test Tools

Few testing organizations are exempt from the pressure to test faster. The typical response is to focus on selecting an easy-to-use tool and then create manual tests routines and automate them, or create tests based on an existing tool. Either way can be adequate for a few hundred test cases, but it is difficult to scale. Test sequences captured with record and playback are fast to create but costly to maintain. Lengthy test scripts usually contain steps that are likely to be repeated in other tests. Any application change that includes common steps will impact all tests and increase maintenance time and/or require creating new tests. The rapid rate of change in the mobile environment almost guarantees test requirements will change. And we all know that nothing saps productivity like recreating hundreds of tests.

The TestArchitect tool set was specifically designed to support ABT, and employs a modular concept to streamline test design and management. The benefit of the modular approach is that when there is a change in an application, only the associated task(s) or function(s) in a module need to be changed, not the entire test. Take a secure logon operation as an example. With ABT a test of the secure logon would be created as an action with a keyword name. For every test requiring logon, the action would be included so the test could execute. If the logon action changed, say from a numeric entry to a motion sequence, one update to the logon action script would automatically update every test that required the logon script.

We have, and are currently working with a number of companies to improve their mobile testing efforts by incorporating ABT and TestArchitect. What follows are the test processes supported in TestArchitect that make mobile testing successful.

One Test, Multiple Targets

There are 3 dominant code bases for mobile devices: iOS, Android and Windows Mobile that most applications will be designed for. For every platform, you will likely need to test different hardware configurations - for example iPhone and iPad, Nexus S and Galaxy Tab, etc. This presents a substantial opportunity for bugs to appear between different versions, and every time you make a code change there is risk of breaking something that used to work.

The ideal solution is to create one set of tests that can be used for multiple devices and applications. This is what action keywords and modular test design enable you to do—by separating tests from the underlying code. Tests are created using actions that outline the task of the test. The automation tool assembles the test scripts from the actions, freeing test developers from tedious coding. The abstraction engine in TestArchitect translates the test cases into the appropriate code for the target device(s). There is some work required by automation engineers to map the appropriate interface, but this is a fairly minor task. The result is that 100 or 1000 test cases can be multiplied across a number of devices and OSs, with a fractional use of testing resources.

Mobile UI Testing

Mobile devices rely heavily on object oriented user interfaces for screens and controls. Routines are increasingly executed by taps, swipes and pinches and less on keypad inputs. Test cases that verify the flow and dialogs of the UI should be created and executed early in the test cycle to catch UI issues that could impact automation in the later stages. This will eliminate hours of investigative time to find the tests that need to be corrected.

There are two methods to test the UI actions--image based and object based testing. Object access to controls is a much more powerful and a maintainable option. It will expose virtually all properties of a screen or control to the test tool and also be visible to a developer. These properties can be used to identify a control or to verify the control values. For example with this approach the caption on a button will be visible as the “text” property of that button.

Object oriented testing isn’t impacted by UI dynamics such as changes in background, image size or orientation. Automating the testing requires access to the OEM provisioning files (Apple’s is ipa and Android is APK). TestArchitect installs its signature in the file to be able to interact with the object. The downside is that not all apps can access the provisioning files, for example Android security will not allow access to built-in applications.

Image-based automation will provide a true "end user approach" that verifies the tests sees the same image as the user. The test automation engine acquires the images from the on-screen display to perform the tasks required for the test. It doesn’t matter what the underlying technology is; if it is visible on the screen, image-based automation can interact with it.

Since it appears that image-based testing would work more often than object-based testing it begs the question, “Why not just use image-based testing”? The answer is that it is generally more sensitive to even minor changes in the UI or the application under test, and it is harder to let the same test run across different devices with different screen resolutions. In order to verify the app continues to work as expected when any of the parameters change, test cases with images for those specific parameters are needed. Here again, the modular test structure in TestArchitect makes it possible to minimize maintenance by creating a module just for the images. By jupdating the test in the image module, all tests containing the images will be updated.

Testing Multiple Devices

Any good mobile test automation will include testing multiple devices simultaneously. This can involve running one test on many devices in parallel, and depending on the application it may require testing devices while interacting with each other. These types of tests can be run with agents or simulators, or a combination of both.

Mobile emulators can be essential for developing and testing a quality application. Emulators are good from the standpoint of running standard UI interactions and being able to do exploratory and benchmark testing economically. The technology enables developers to verify certain functionality that is not specific to any device, carrier or operating system, and for testing usability including data input, button use, etc. – all with the convenience of a full keyboard and mouse. However, emulators and simulators don't operate in the real world and it is always recommended to test real devices in the native operating environment. Once satisfied the test parameters are correct, automation agents can execute tests on the actual devices (and run tests in parallel with emulators) for specific versions of devices and or OS so that testing effort can be maximized.

Cloud based services are also an option for testing multiple devices. The benefit is that these services have a wide range of devices and you pay only for use. The risk is uncertainty of the validity of the outcome. Many of the phones used in cloud based services have been jail-broken which can result in unintended consequences. This can be the case when an application relies on a specific carrier's network security--what worked fine in the lab may not work when released to customers. Again, a good way to validate this is to test one or more actual devices.

There are device connectivity options as well. Devices can be connected to the host machine via USB and Wi-Fi. In most cases a USB connection is necessary to set up the automation agent, but once operational the tests can be driven over Wi-Fi. This has the benefit of testing the sensitivity of the application and device to changes in speed and or network signal strength--which users will encounter in the real world.

Summary

The demands for mobile test automation will continue to grow along with the increasing adoption of mobile devices. Due to the multiple OS standards and the multitude of device providers testers face a highly complex testing environment. By adopting the right techniques and methods and employing the appropriate tools, it is possible to tackle complex automation and be able to effectively scale as the need requires. It all starts with planning. Part of that planning should include learning and implementing new methods and tools that will improve efficiency. Those who are committed to long-term success in their field recognize this and embrace it. ■

Feature

Test Automation Interfaces for Mobile Apps

What you need to know in order to have effective and reliable test automation for your mobile apps

By Julian Harty

I realized that test automation interfaces are pivotal to effective and efficient test automation, yet very few people trying to test their mobile apps seemed to know how their automated tests connected with the apps they wanted to test. Even if you use commercial products that claim to remove the need to write test automation code you may find the information enlightening, at least to understand the mechanics and some of the limitations of how these products 'work'.

This article provides an overview of the topic of test automation interfaces, starting from stuff we need to know, then things we need to learn & understand, then the concluding section covers factors worth considering in order to have effective and reliable test automation for your mobile apps. I have included various additional topics that don't fit into the know/understand/consider sections.

Like much of my material this is a work-in-progress. Comments, suggestions and feedback are all welcome to help me improve the utility and accuracy of this material.

The Many Stages of Test Automation

There are at least 3 key stages of test automation: Discovery, Design, and Execution. Each is important albeit execution may be the only one that many people outside testing recognize.

Discovery Stage

We need to discover ways our automated tests can interface with the target app.

If we can read and understand the source code perhaps there's little to discover about the app. Conversely, if we're less technically minded or when the source code isn't available, discovery can be more challenging and time-consuming. Furthermore we need to identify trustworthy identifiers that will remain useable for the test automation to survive new releases of the app. Dynamically assigned IDs that vary from release to another, sometimes unpredictably, can be the start of nightmares where the test automation needs patching and repairing virtually constantly.

Some test automation software tries to automatically detect these interfaces e.g. WebDriver's has a 'PageFactory' http://code.google.com/p/selenium/wiki/PageFactory which matches a parameter with the name or id of a web element. There's a good introduction to how GorillaLogic's MonkeyTalk identifies elements in the August 2012 issue of Test Magazine.

Alternatively we can use various tools to help discover how to interface with elements in the mobile app. Tools include FireBug for web apps; XCode's 'Instruments' for iOS; Calabash's Interactive Ruby Shell for iOS; etc.

Some test automation tools include a feature known as 'record & playback'. The recorder needs to capture a de-facto test automation interface and record the elements the user interacts with. They typically also record the values entered by the user into a single test script which can then be replayed many times to repeat the actions of the user – the 'tester'. For the purposes of this article I'll leave aside the discussion of the validity of 'record & playback' test automation as a test automation strategy; however, if we can read and understand the recorded script we can use it to help identify and discover ways to interface with the app we want to test.

Design and Execution Stages

During the design stage we create the tests and write the test automation (interface) code, which is a precursor to running, or executing, the tests. Design includes picking the most robust and appropriate test automation interface e.g. whether to use co-ordinates, ordinals, or labels to identify specific elements; and whether to add descriptive identifiers to elements that don't have anything suitable; etc.

Design also includes deciding what information we want to capture when the tests are executed (run), and making sure that information will be captured so it can be analysed later. Sometimes we may want to analyse the data much later on e.g. many months or even several years later, so good archiving techniques may also form part of the grand plan.

Finally, we need to execute, or run, the automated tests. This will involve some preparation for instance to configure the test environment, to synchronise the clocks across machines, etc. Remember to have more than enough storage for the logs and other outputs from the testing to be recorded safely.

Know the Interfaces Used by a Mobile App

A mobile phone may include lots of different interfaces, which users might choose to use as part of using a mobile app. They can be grouped into two broad groups: HCI and Sensor interfaces depending on whether humans interact with the interface, or not. HCI includes:

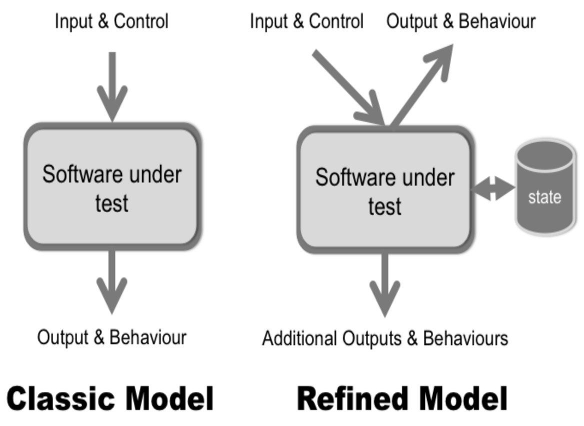
- Touch: where some devices support multi-touch, pressure, contact areas, and edge detection e.g. is the person also touching the edge of the screen? For multi-touch, some phones detect more simultaneous touches than others e.g. some may detect 5 and others 10 concurrent contacts. Imagine a piano app on a tablet device – would the app receive inputs for all 10 fingers and thumbs, or only one hand's worth?
- Proximity: proximity can be used to change the sensitivity of the screen, for instance a phone app may disable the on-screen keypad when the phone is close to a person's cheek.
- Movement: especially when playing games, motion can even be the primary interface.
- Sound: including voice input e.g. for Voice Search; or apps can 'listen' to music to name the artist and the track, etc.
- Light: for instance for face detection to unlock the phone, or to help a visually impaired user to find the exit in a dark room into a lighted room.
- Controls: these include D-pads, buttons, etc.

Sensor Interfaces include: accelerometers, magnetism, GPS, Orientation, Camera, NFC, etc.

You may have already noticed some overlap between these sensors, that's OK, the goal is to identify the range of sensors and what they're being used for in terms of the app you want to test with automated tests.

Block Diagrams

A classic model of a system under test show inputs and control which are processed by the software under test that then provides output(s) and may exhibit additional 'behaviours'. In practice, the software is likely to include additional outputs and behaviours beyond those we initially realise.

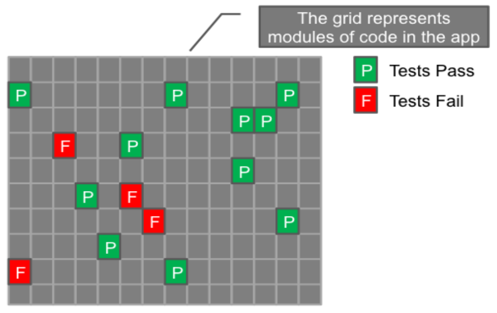


Some of these additional outputs and behaviours may be useful from a test automation perspective. For instance, one test automation team wrote code to process and parse the Android system log in their automated tests. They used the Robotium API method waitForLogMessage (String logMessage). The logs include a record of each network request and response for debugging, which are written by the application code when it is run in a particular configuration. The log messages are a proxy for actually capturing the network requests, which can be done programmatically.

Know Your Test Automation Interface

What information does the test automation interface provide? E.g. can it return the types of information we're interested in? Can it return compound information? What about error reporting, and error recovery?

UNIT TESTS OVERLAY PARTS OF THE CODE



Unit tests overlay parts of the code

Unit tests

We can classify our test automation in various ways. Unit tests live alongside the code they test. They effectively overlay the code they will test, and they are compiled together with the code into a module that is then processed by the unit-testing framework. They do not test the application.

APIs

The app may include APIs. These APIs may be an intrinsic part of the application, for instance to enable other apps to interact with it. Or the API may be added explicitly to facilitate test automation. Sometimes the API may be presented as a embedded web server. When APIs are used as a test automation interface, they should be easy to write automated tests for as they're inherently intended to be called by other software.

Structural

Structural interfaces represent and reflect structures in the software being tested, for instance the DOM of HTML5 apps. They support filtering and selection based on the structure or hierarchy of the presentation of the app. Tellurium has an interesting object location algorithm called Santa and separates the object mapping from the test scripts.

GUI

GUI test interfaces match images, possibly within a larger image, the visual representation of the app. GUI test automation is excellent for pixel-perfect matching, however there are many legitimate reasons why images may differ during tests, for instance if they include the system clock on screen. GUI test interfaces are sometimes the test automation interface of last resort, when all other mechanisms are not viable.

There are various techniques we can use in order to cope with mismatches in the images we consider irrelevant. These include searching for sub-images within a larger image, using fuzzy matching (e.g. where the test accepts 2% difference in the pixels), and ignoring some sections of the screenshot.

Characteristics of test automation tools

One of the key questions to understand is how will your chosen test automation interface detect various outputs and behaviors.

The internal mechanics of the test automation code may also be relevant. For instance WebDriver would need to send lots of individual requests to collect all the information on an element, rather than asking for the set of attributes in a single API call. Because the state of the element can change, even between the various requests, the final results may be inconsistent or inaccurate in some circumstances.

Know Your App

Ultimately, your app is particular to you and it's worth knowing your app in some depth if you want to create effective test automation.

- Decide what aspects you want to focus on testing, checking etc.
- Realise many other facets of the app's appearance and behaviour may be ignored by your automated tests.

Beware of 'Wooden Testing': testing that is limited to 1-dimensional testing of the basic functionality to show 'something' works. Testing that disregards the realistic aspects of the software's behaviour, errors, timing, events, etc. At best, your automated tests will be incomplete, leaving much of the apps behaviour and qualities untested. Worse, the team may be lulled into a false sense of security, like the French were when they'd constructed the Maginot Line before World War II, thinking your automated tests will protect the project while they're largely irrelevant and outflanked by real-world events that expose gaping problems in your app.

So, let's think about which aspects we want to focus on testing or checking, for instance – is the performance of UI responses important? If so, will your tests be able to measure the performance sufficiently accurately? In addition, how will you implement the measurements and check they are accurate?

Try to consciously identify elements your tests will not be checking, so you're able to recognize the types of bugs your test automation will not capture (because the tests are ignoring those facets of the app). If we want to test these facets, we can include some other forms of testing explicitly for these aspects e.g. interactive testing, usability testing, etc. I've yet to meet a team, project, or company who use 100% automated testing for their mobile apps – so don't be embarrassed by having some interactive testing (otherwise known as manual testing) simply learn to do it well.

What parts of the app change?

- Identify the elements on each screen in the UI
- Identify the transitions from one screen to another
- Understand what things or events causes changes in the app's behaviour or appearances

Consider ways to:

- Launch screens independently.
- Use of mock objects, test servers, etc. to reduce the effects of 'other' code on the software you want to test. Lighter-weight (fly/welter weight?) test automation.

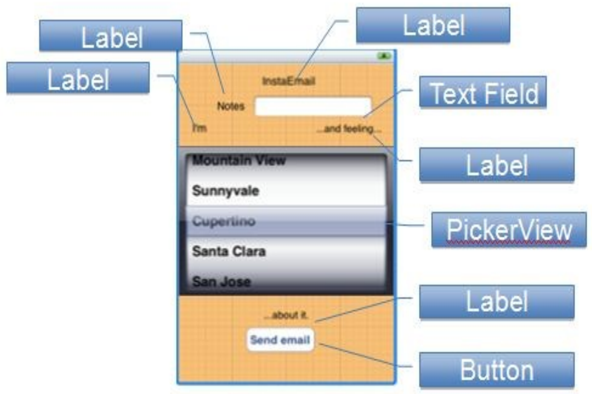
Related topics: outside the scope of this material.

- Test below (or without) the GUI. While this is a separate topic, sometimes you'll get excellent results with such testing. It can augment GUI test automation.
- Effort needed to maintain your tests.
- Ways to prepare the environment and to launch the tests.
- Ways to structure your tests to reduce maintenance effort. Note: it's worth covering aspects of how a test automation interface might be structured to reduce the maintenance effort. E.g., calabash currently has differences between the APIs (and the step definitions) for Android and iOS.

Know Your Test Data

The test data can often affect what's shown in the UI and how the app behaves. Some examples include:

- The account used to by the tests to login
- Images e.g. returned in search results, picture galleries, etc.
- Ads, particularly those that target aspects of the user's location, behavior, etc.



Unless we 'know' or can predict exactly what data to expect our tests may be adversely affected. We can chose several options e.g. to ignore the contents of various fields and areas of the UI, to inject known test data, to use heuristics to decide whether the results are acceptable, etc.

Tests that depend on production data are particularly prone to challenge the behaviour of our tests. I've encountered image-based test automation that's had to ignore virtually all aspects of the returned results because the tests couldn't realistically cope with the variety of the content returned by the production servers.

Know Your Platform & Know Your Development Tools

Each mobile platform is different and offers distinct ways to interface with an application from a test automation perspective. Some incorporate 'Accessibility' capabilities, which are sufficiently potent to form the primary interface for test automation; iOS is the most potent in this area. Others rely on having access to the source code for an app e.g. Android.

Discover; learn what the interfaces are already available for the platform you want to write automated tests for.

For Android these include:

- Android Instrumentation (primarily uses component type or resource ID)
- Android MonkeyRunner
- Image based matching
- Firing events, actions, intents, etc.

Not used:

- Accessibility features

For iOS these include:

- The Accessibility Labels
- UIScript & UIAutomation
- The interactive recorder, which is part of the Instruments developer tool provided with XCode.

Know the Devices

If you run the same tests on the same device with the same date and the same software installed you might expect the screenshots of each screen would have consistent colours, text, etc. However, we discovered a number of anomalies for some Android devices. Sometimes the size of the test for a Dialog's title element was larger in one test run and smaller for another. Also, there were numerous differences in the colours of background pixels on the screen. There were no known reasons for the changes. However the changes had the effect of causing screenshot comparisons to fail. For this client they wanted the UI to be pixel perfect so the differences in the screenshots were relevant and meant the team had to spend significant time reviewing the differences to decide if the differences were material and enough to reject a build of the app.

We're still researching whether there are patterns that trigger or cause the differences. In the meantime, the matching process includes 2 tolerances for differences:

1. Per pixel tolerance of the colour values
2. Per image tolerance of the percentage of pixels that don't match.

Additional Approaches to Test Automation

Teams can also write specific APIs and interfaces e.g. to allow remote control of their app. These APIs and interfaces may be shipped as part of the app, disabled, or removed entirely from the app's we intend to ship – the release versions. There are various ways of designing the APIs / interfaces; one of the most popular ways is to embed a HTTP server – effectively a mini- web-server as an intrinsic part of the app.

Some test automation frameworks rely on a compiled-in interface, which the development team incorporates into the testable version of their app. These include commercial offerings from Soasta and opensource offerings including Calabash-ios.

Considerations

How many ways?

See how many ways you can find to interact with your app. Sometimes the challenge is to find a single way to interface your test automation code with your app. However, once you've found at least one way to connect it's worth seeking more ways to interface the test automation with the app.

You may well discover more reliable and elegant ways to connect them together.

In the domain of test automation for web apps, WebDriver is the foremost opensource test automation tool. It provides the 'By' clause which includes a multitude of ways to interface the tests with the web site or web app. Some choices tend to be more reliable and resilient e.g. when elements have an unique identifier. Here's the current set of 'by' options for WebDriver to inspire you:

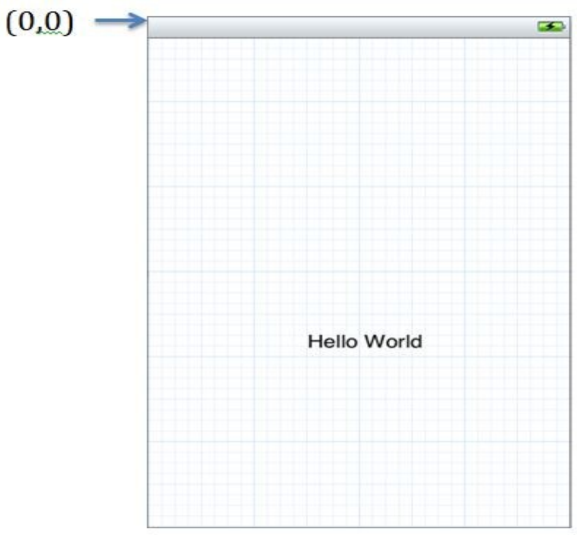
```
class By(object):
   ID = "id"
   XPATH = "xpath"
   LINK_TEXT = "link text"
   PARTIAL_LINK_TEXT = "partial link text"
   NAME = "name"
   TAG_NAME = "tag name"
   CLASS_NAME = "class name"
   CSS_SELECTOR = "css selector" (found in: py/selenium/
   webdriver/common/by.py)
```

Other options include the location of elements on the screen. The location is generally calculated using a pair of co-ordinates, the x and y values based on a 2-dimensional grid. The origin (where x is 0 and y is 0) tends to be the top left corner or the bottom left corner, depending on the platform.

Maintainability vs. Fragility of Interfaces and the Test Code

Interfaces may be fragile, for instance where tests break when the UI is changed, even marginally.

Efficient test automation is more likely when the developers collaborate and facilitate the test automation interfaces. They're more likely to create maintainable test interfaces than anyone else. Ideally the business and project team will recognise the importance of efficient and effective test automation as adding good interfaces will take some time initially, The return on investment comes after subsequent releases need to be tested - provided the tests don't need lots of maintenance.



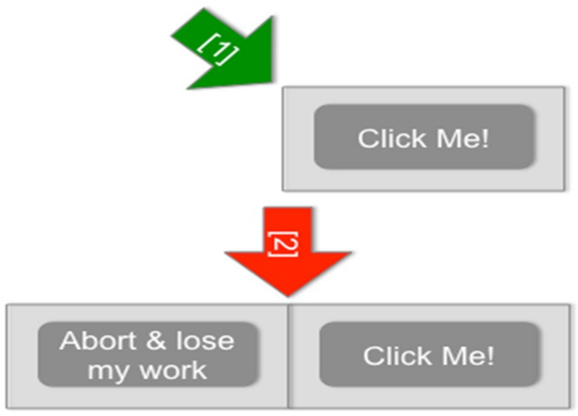
Each interface has limitations, beyond which it becomes unreliable or simply broken. For instance, one that's based on the order of similar elements e.g. the second EditText field is subject to the order of EditText fields used to create the current UI. If the designers or developers decide to change the order or add additional EditText fields before the one we intended our test would interact with, then our tests may behave in unexpected ways.

Using positioning and touch areas

Most elements can be described by a set of points, the most common set being 2 pairs of points describing the top-left point and bottom-right points of a rectangle. When someone touches the screen of their phone anywhere within the rectangle, the UI sends the touch event to that element.

Sometimes the touch area is actually larger than the dimensions of the element. This is intended to make the user-interface easier to interact with. The following figure, with the arrow labelled [1] shows how the touch area is increased for the 'Click Me!' Button. For the second example in the figure, the touch areas abut each other, as indicated by the arrow labelled [2]. So a user might inadvertently select the 'Abort & lose my work' button when they intended to press 'Click Me!' instead.

Developers may be able to explicitly specify the size of the touch area, or the platform's user-interface may even do so automatically. So, we may want to test the touch area as part of our automated tests.



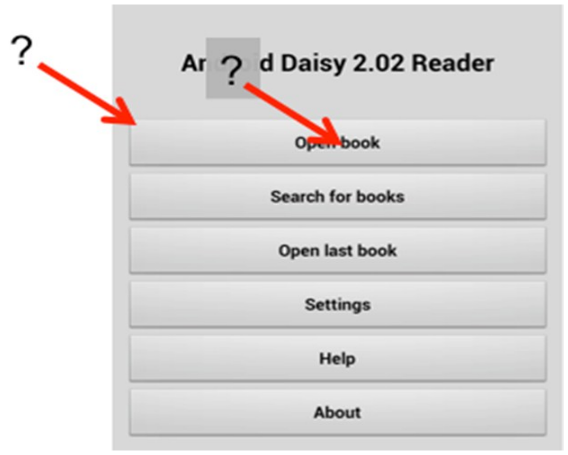
Also, if elements move slightly in the UI e.g. because the designer wants them moving slightly to the right, it's worth checking whether the current coordinates are still suitable. If we'd used the top-left corner as our touch co-ordinate then our tests may start failing mysteriously. Whereas if we'd used the center of the rectangle, they'd still work, even though they're no longer in the absolute center of the new location of the element. The following image indicates two choices of where to locate touch events: near the top-left of the button, or near the center.

Designing Test Automation Interfaces

Some of us will be fortunate and involved in the design of test automation interfaces. Here are some suggestions for what to consider.

Choosing good abstractions

A test automation interface needs to communicate with people as well as with computers. The people include the programmers, testers, and others on the team. A good abstraction can make the interface easier to comprehend and use, for instance by naming methods and parameters so they reflect the domain objects.



Higher-level interfaces, for instance Cucumber, are intended to be easy for everyone to read what the tests do; yet they can be simple to implement and even easy to use.

Preserving Fidelity

Preserving the fidelity of the app's appearance and behaviour is also important, so the tests can detect changes in the app's output or behavior. As an example there was a bug reported on Selenium, #2536 where the screenshot does not match the actual viewport for Google's Chrome web browser, but does for Firefox. The Chrome Driver is reducing the fidelity of the output, somehow, and therefore reduces our confidence in using it when we want to process or use the screenshot.

There may be a trade-off between choosing good abstractions, using higher-level test interfaces, and preserving the fidelity. Rather than me telling you what your trade-off should be, I suggest you discuss the options with the rest of the people involved in the project.

A related area is to provide sufficient capabilities to interact adequately with the app. One of the problems faced by cross platform tools, and tools that provide multiple programmatic interfaces e.g. implementations in several programming languages, is to provide the same level of capabilities for every platform and programming language.

Sometimes there may be technical reasons why some capabilities cannot be provided, if so, document these so the test automator has advanced notice rather than assuming all will be well.

Ways to increase flexibility of a test automation interface

Interfaces can be designed to allow greater flexibility and resilience to make the interface less brittle. Selections that use names rather than positions can cope with changes in the order of elements. Some selections are intended to be unique – for instance when IDs are used to select web elements . Others filter or match one or more elements: CSS selectors are a good example. XPATH selectors can reflect the hierarchy of elements and select some elements within a DOM structure. Named parameters in an API allow the order of elements to vary between the caller and the implementation, and default values for new elements provide some backwards compatibility for older test automation code.

Here are examples of each approach:

- **IDs e.g.** <element id="id">
- **CSS** <a href="#" class="classname">click me</a>
- **Xpath** /bookstore/book[1]
- **Named parameters e.g.** def info (object, spacing=10, collapse=1):

For designers of test automation interfaces, they need to consider how much flexibility is appropriate.

Decide What's Worth 'Testing'

Are the colors in the UI worth testing? How about the exact location and order of elements on the screen? And what about the individual images displayed in your picture gallery? You may decide some of these are relevant to your automated testing. If so, we need to find ways to:

1. Obtain the information from the app, in the test automation code
2. Verify the information

For attributes such as colour, obtaining the information and comparing it to the expected value may be straightforward.

Images can be much harder to obtain or compare, and there may be other technical reasons why the information may be hard to match exactly. This is particularly so when using images captured using a camera e.g. from commercial products such as Perfecto Mobile's

Recognize the no-man's land and the no-go areas

Some aspects may be impractical to test with the current tools e.g. rapid screen updates may be too fast for some test automation tools to detect (reliably). The most obvious examples include video content; also, transitions may be very hard to test robustly or reliably.

Similarly, when the test environment uses production data, the results may be hard to predict sufficiently accurately to determine if things like the search results are 'correct'.

Sometimes the most practical approach is to err on the side of reliability where the tests wait until the GUI has stabilized e.g. when the web view has finished rendering all the content before trying to interpret the contents of the GUI.

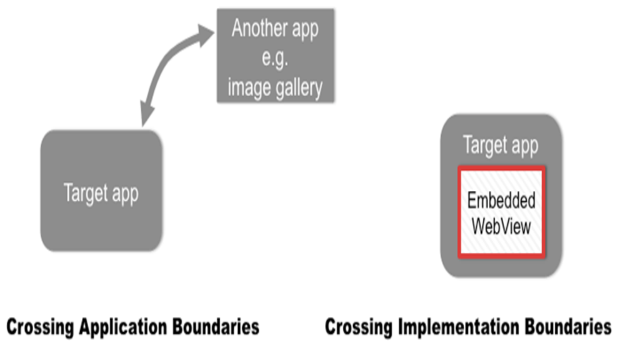
Limitations of the chosen test automation interfaces

The dynamic nature of apps, latency of the test automation, data volumes, etc. can all adversely affect the validity, reliability and costs of the tests. These factors are all worth considering when we design and implement our automated tests, otherwise we risk the tests generating incorrect results or being undependable as the conditions vary from test run to test run.

Other limitations, particularly when testing mobile apps on actual devices, is dealing with the security defenses provided by the platform.

For iOS, there are no public ways to interact with the preinstalled apps on a device. The only viable alternative is to run tests against the Simulator, using GUI test automation tools such as Sikuli.

For Android, the primary test automation approach uses something called the Instrumentation Test Runner. This relies on the tests being written alongside the code we want to test, a white-box test automation technique where identifiers for UI resources may be shared between the app and the test automation code. Instrumentation is limited, by design to interacting with the app. However some apps interact with email clients, the photo gallery, etc. for instance, to allow a user to upload a new picture of themselves online. Android Instrumentation is not permitted to interact with the gallery or with the email clients. So the test automator needs to find an alternative test automation tool to interact with these apps they didn't write. For Android they may choose to use MonkeyRunner, which is a GUI test automation tool, to interact with these separate apps. Or they can choose to write custom apps to replace the general-purpose email, photo picker, and other apps.


Another app e.g. image gallery
Target app
Target app
Embedded WebView
Crossing Application Boundaries
Crossing Implementation Boundaries

The figure above illustrates inter-app and intra-app limitations.

The custom app then provides a way to interact with them programmatically, from the test automation code.

Some apps embed a web browser. Few test automation tools can cope with interacting with the embedded web browser, which can cause significant challenges if the tests need to use it e.g. to login to the app.

Remember – the Human to Computer Interface

Ultimately, we want our apps to be useable and enjoyable for our users. Therefore we need to test on-behalf-of people we think may use the app. Remember to consider users who may be less gifted than the gifted programmers and designers; users who may be less adept at using mobile devices. Also consider users who benefit from software they can adapt to suit their needs e.g. who may use a screen reader such as VoiceOver on an iPhone, or TalkBack on Android.

Test automation can help detect some of the problems, for instance to determine the size and proximity of touch areas that are too small or too close together. Conversely test automation may 'work' but ignore problems that make an app unusable for humans, for instance where white text is displayed on a white background.

Good luck with your test automation. Let me know about your experiences. ■

Glossary:

Term - Description.

App - Abbreviation for 'Application'.

Application - Installable software intended to be visible and used directly by users. Compare with a Service or other background software, which don't have a visible user interface.

Platform - The operating system of a particular mobile device. Common platforms include: iOS for the iPhone and iPad; Android for Android phones; and BlackBerry OS for BlackBerry phones.

Special Bonus Topic

Things Which can Affect Captured Images

Colour Pallets

Although a device may be able to display 24-bit colours (theoretically allowing gazillions of hues and colours to be displayed), manufacturers may transform the specified colour slightly to reduce the number of distinct, unique colours that will actually be displayed on the screen. By reducing the number of colours they can increase the performance of rendering the graphics on the display. The trade-off of tweaking the colours was often masked by limitations in the human eye's ability to detect the changes on the device's screen.

Some mobile devices actually had several conversion processes e.g. one in software and another in hardware. The multiple conversions made it hard for the programmer or designer of an app to know the actual colour that a user would see, unless they tested their app on each combination of device model and platform revision.

Web-safe colours

Historically, desktop displays could be limited to as few as 256 colours. The industry devised a set of around 216 distinct colours which were expected to be sufficiently distinct yet able to be rendered by all desktop and laptop computers.

Dithering

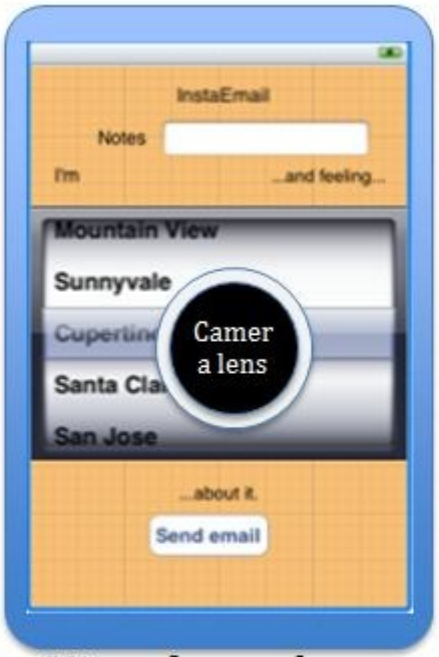
Dithering is a process that 'decides' what value to specify e.g. where the current value isn't viable. It can apply to choice of colour (as mentioned in the section on colour pallets), location e.g. of an element on screen, etc. We have discovered that some smartphone models vary the location of text slightly (by a pixel or so) between one run of an application and another run. This means the screenshots can vary even when the platform and the app remain constant!

Fonts

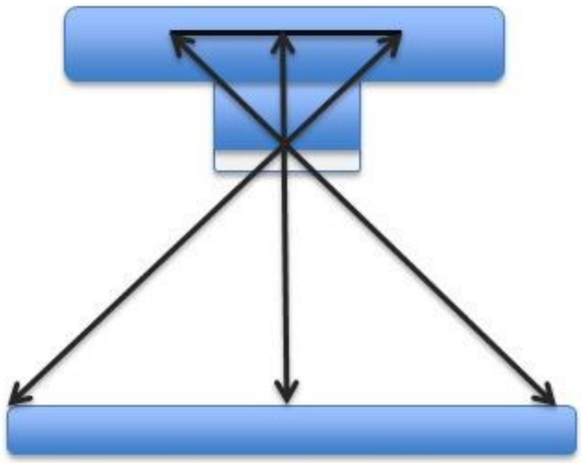
The choices of font and appearance of text (e.g. font size, *italic*, **bold**) may have a significant effect on how the text is rendered on screen and on how much of the text is visible. If the tests match the images pixel-by-pixel then virtually any change of font or appearance may cause the image matching, and therefore the test to fail.

Parallax

Parallax in terms of test automation is the effect of the positioning a camera above the screen of the device. The centre of the screen is closer, and takes more space, than the extremes of the screen. This distorts the recorded images. The following diagram shows a simplified version of why parallax occurs.



View from above



View from side of phone

Lighting and reflection

Lighting and reflection can also adversely affect test automation that relies on an external camera to capture screenshots.

Compensating for the various differences

1. Tolerances, e.g. for colours, for things like the percentage of differences to allow. Fuzzy matching e.g. to find a small image within an area of the screenshot.

2. Blanking out or ignoring parts of the image.

3. Auto-generating the images we want to match against (this could be an entire section, I expect).

4. Controlling the input data, picking distinct colours, fonts, etc. that are easier to match programmatically (in the test automation code).

5. Controlling the lighting, picking the choice of camera carefully, etc.

6. Deciding which format to capture and store images in.

About Julian

Julian has been working in technology since 1980 and over the years has held an eclectic collection of roles and responsibilities: He was the first software test engineer at Google outside the USA and a main board company director in England for a mix of companies involved in running the systems and operations for the European aspects of Dun & Bradstreet's Advanced Research and Development company. Currently, Julian's main responsibility is Tester At Large for eBay. He also works on opensource test automation projects for mobile phone applications, available at http://tr.im/mobtest.

Mobile Trends by the Numbers

Interesting statistics about the most popular mobile platforms, smart phone usage around the world, and the emergence of mobile malware.

- Nigeria has close to 100 million mobile phone lines, making it Africa's largest telecoms market.
  - Nigerian Communications Commission

- Google plans to sell 200 million Android phones in Africa. It is estimated that by 2016 there will be a billion mobile phones on the continent.
  - CNN

mPayments

- M-PESA is a mobile money transfer service launched by Safaricom, Kenya's largest mobile operator, and Vodafone, in 2007. Five years later, M-PESA provides services to 15 million Kenyans (more than a third of the population) and serves as conduit for a fifth of the country's GDP.

- In Kenya, Sudan and Gabon half or more of adults used mobile money, according to a survey by the Gates Foundation and the World Bank.
  - M-payment

- There will be 212.2 million m-payment users in 2012 (up from 160.5 million in 2011), m-payments will total US $171.5 billion in 2012 (up 61.9 percent from $105.9 billion in 2011).

- Gartner predicts that in 2016 there will be 448 million m-payment users, in a market worth $617 billion. Asia/Pacific will have the most m-payment users, but Africa will account for the highest revenues.
  - Gartner (May 2012)

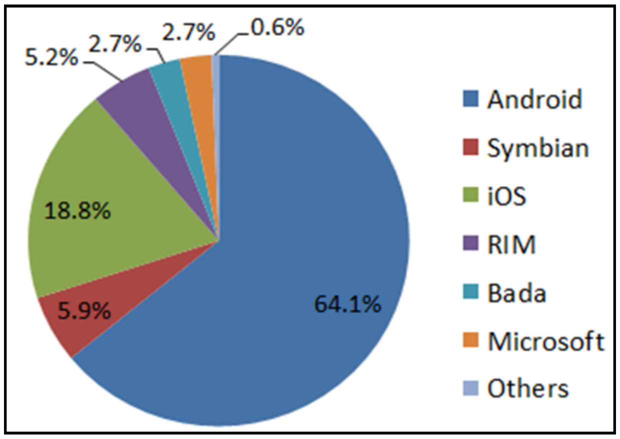
- While the rest of the world dreams of mobile payment , in Japan it is already a way of life. 47 million Japanese have adopted tap-and-go phones in three years – this is one of the fastest roll outs of electronic products in human history.
  - ComScore (February 2011)

Mobile usage in the United States, Japan and EU5 (UK, Germany, France, Spain and Italy)

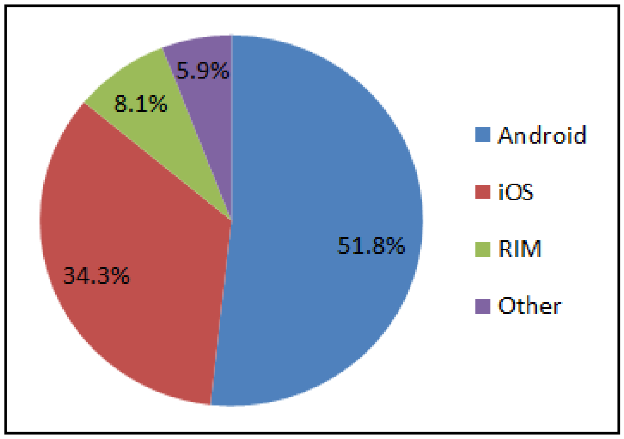
Used connected media			
	United States	Europe	Japan
(browser, app or download)	46.70%	41.10%	76.80%
Used messaging			
Sent text message	68.00%	82.70%	41.60%
Email	30.50%	22.20%	57.10%
Accessed financial services			
Bank accounts	11.40%	8.00%	7.00%

Smart phone / device usage by operating system

*Gartner's analysis of **Global** Q2*



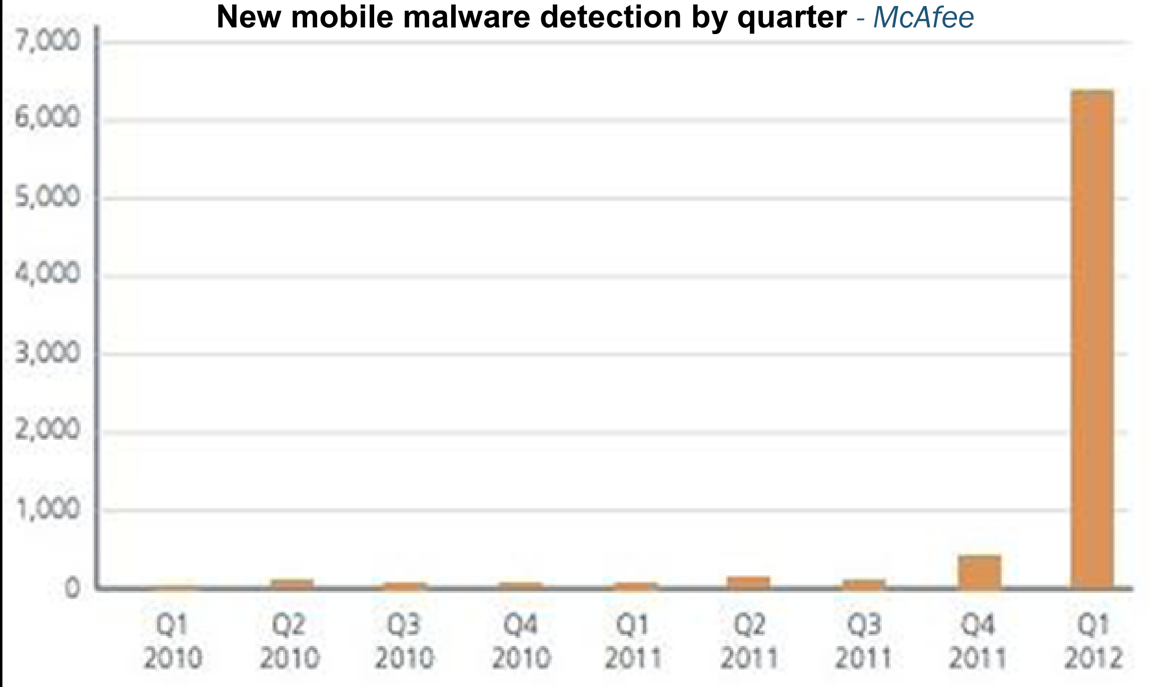
*Nielsen's figures for Q2 2012 in the **US***



25% of Americans own a tablet. Up from 11% of U.S. adults in July of 2011 to 18% in January of 2012.
- Pew Internet & American Life Project

The Emergence of Mobile Malware and Security and Performance Testing

New mobile malware detection by quarter - *McAfee*



- The **World Quality Report**, which is compiled each year by consultants **Capgemini**, software test specialist **Sogeti** and IT group HP, found that only 31% of 1,550 CFOs, CIOs, IT directors and quality assurance (QA) directors polled conduct tests of mobile applications. Even where testing of mobile software does occur it's focused on performance rather than functionality or security.

- Respondents acknowledged that security testing was only a priority for 18 percent of organizations.

- According to a report by the FBI, as of October, 2012, a whopping 25% of employed U.S. adults say they have been a victim of malware or hacking on their personal electronic device.

- "As more and more people around the world are adopting smartphones and using them to download apps, bank, and conduct business, there's more and more of an incentive for criminals to attack phones like they've attacked PCs in the past," he said.
  - *Kevin Mahaffey, CTO, Lookout*

Feature

Extending Test Automation to Mobile Applications

Organizations need to implement automated testing strategies designed specifically for mobile applications.

By Edward Hill, HP Software UK & Ireland

Mobile device usage continues to gain momentum at increasing speed. Enterprises that delay mobile adoption face the danger of becoming competitively disadvantaged. But, before jumping in headlong, you need to be fully aware of the unique challenges that can arise when developing and implementing mobile applications—and be fully prepared.

Compared to the desktop environment, mobile applications have quite different user interface (UI) requirements, business process flows, and infrastructure dependencies. Coupled with the growing variety of mobile devices and operating systems, increased strain can be put on IT to build, port and maintain mobile applications, and this heightens the risk of quality and performance problems.

Functional Validation

Potential glitches can be avoided by putting certain mobile applications through what's sometimes referred to as 'functional validation'. In simple terms, functional validation is a way of testing mobile applications to ensure that they do what they are intended to do based on testing concepts such as test asset management, collaboration, reusability, accurate reporting and analysis.

The majority of testing on mobile devices is currently done manually. The problem with manually testing mobile applications is lack of accuracy, coverage and scalability, Manual testing also has considerable associated costs, especially if an application needs to be retested multiple times to ensure development and quality.

While a test engineer can manually key in a handful of transactions, he/she cannot continuously test and monitor the connection for application availability or test for all possible scenario permutations. Additionally, a tester is not able to quickly identify problems, fix them, and rerun all the tests required to validate the fix.

Using actual devices for testing

Many industry experts argue against using handsets for testing in favor of using a browser or emulator for multiple reasons. However, in order to ensure applications function and perform, they must be tested on mobile handsets and there are several reasons ranging from the theoretical to the practical.

Usability testing

Usability testing on emulators and browsers with any extension do not represent what will be shown on the actual device. With Android specifically, emulated "vanilla" environments almost never represent the actual behavior of the phone as vendors tend to modify the "source code" of the handset and add their own customization layer.

Environment related testing

Environmental testing is critical. Mobile applications rely on the location, accelerometer and network. Simply put, the test cannot be done in a simulated environment.

Interoperability testing

A mobile handset "lives" in a communicated environment and behaves accordingly, meaning an incoming phone call always receives priority over any application in use. Testing scenarios to see what happens to an application when a phone call is received is critical, as this often an everyday occurrence.

Network-related testing

Proper network related testing must be conducted. You can't assume an application developed for Vodafone in the UK will work on the Verizon network in the US. Network neutrality has not been integrated in the cellular world, which means testing must be done taking into account the need for users to navigate multiple carriers.

Security

Security is a hot-button issue with mobile users. People are concerned about personal data, such as bank account numbers that remain on handsets, or passwords being displayed on the screen. Testing for these types of security concerns in a simulated environment is not a good use of time because what is required to be tested is the behavior of the actual handset. Not all handsets have the same security designs, so each device must be individually tested.

Performance Validation

A second issue facing IT when it comes to mobile applications is the need for performance validation - applications must be validated to ensure performance expectations are met. It's worth remembering that the user expectations from these applications may even be higher than those of their desktop counter-parts. Performance validation for mobile applications is therefore crucial and, at times, more complex than for desktop applications.

In addition to this, mobile applications are affected by mobile network conditions such as bandwidth limitations that are more pronounced than land-line networks. These restrictions may adversely affect the communication between a handset and the back-end servers.

In order to reach the large number of simulated users that will be accessing the tested application at any given moment, performance validation needs to be automated, which can be achieved through simulated users running on emulators and browsers. Without this kind of automation it is virtually impossible to create a realistic load on the application servers and infrastructure.

When replicating the appropriate network conditions you need to assure that a simulated mobile user experiences the same bandwidth throttling and network delays that a real mobile user would. In order to complete the test, real handsets executing the relevant business processes should be running parallel to the automated performance test. This will allow the end user experience to be measured while also testing the behavior of the application's servers and infrastructure under load.

In Conclusion

Automated functional and performance testing has a proven track-record in desktop-based applications. It helps companies deploy higher-quality software applications, reduce business risk, and accelerate problem to resolution. This will ultimately help organizations avoid costly interruptions and errors. The growing demand for instant information, especially through mobile applications, should encourage organizations to adopt automated testing strategies and solutions that are designed specifically for their mobile application needs. ■

Areas that should always be considered in your testing strategy:

- **Data entry:** Test for problems related to scrolling, text selection, the back button, etc..
- **Connection Speed / Carrier:** Devices are now running on multiple carriers, so testing on only one carrier is not an option.
- **Operating System:** Mobile operating systems can impact an apps performance. It may not be possible to test every device combination, but 3-5 of the most popular should be considered.
- **Screen issues:** An Android or iPhone app can be displayed an a wide range of screen sizes. Make sure that size/orientation is tested.
- **Interruptions:** Exploratory testing is the best way to find issues. At a minimum check for power level incoming call, SMS, and MMS issues, and network signal strength.

About Edward

Edward has worked at HP since 2007 is currently Sector Head—Retail and Consumer Goods at HP Software. His responsibilities include: the vertical transformation of the UK *go-to-market* strategy, sales execution for HP Software and to build a trusted, secure and vertically credible external perception of HPSW.

Glossary: Mobile Test Automation

Platform - A computing platform includes a hardware architecture and a software framework (including application frameworks), where the combination allows software, particularly application software, to run. Typical platforms include a computer architecture, operating system, programming languages, related user interface and tools. For example, Android, the most common *mobile platform,* is Google's open and free software stack that includes an operating system, middleware and key applications for mobile devices, including smartphones. A key feature of platforms is their ability to incorporate hardware characteristics and support tools.

Cross-platform - An attribute conferred to computer software or computing methods and concepts that are implemented and inter-operate on multiple computer platforms. Cross-platform software may be divided into two types; one requires individual building or compilation for each platform that it supports, and the other one can be directly run on any platform without special preparation. In mobile, for example, many test teams struggle with cross platform automation, meaning trying to use the same automation tool or scripts on iOS, Android and/or Win Mobile.

iOS - A mobile operating system developed and distributed by Apple Inc. The user interface of iOS is based on the concept of direct manipulation, using multi-touch gestures. iOS is derived from OS X, with which it shares the Darwin foundation, and is therefore a Unix operating system. iOS is Apple's mobile version of the OS X operating system used on Apple computers.

Android - A Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers. It is currently developed by Google in conjunction with the Open Handset Alliance. Android has a large community of developers writing applications ("apps") that extend the functionality of devices, written primarily in a customized version of Java. They are available for download through Google Play or third-party sites.

Windows Mobile 8 - Windows 8 introduces significant changes to the operating system's platform, primarily focused towards improving its user experience on mobile devices such as tablets to rival other mobile operating systems like Android and iOS. Windows 8 also features a new app platform with an emphasis on touchscreen input, and the new Windows Store to obtain and/or purchase applications to run on the operating system.

Symbian - Symbian is an open-source platform developed by Symbian Foundation in 2009, as the successor of the original *Symbian OS*. Symbian was the most popular smartphone OS until the end of 2010, when it was overtaken by Android.

Application - Application software, also known as an application or an **app**, is computer software designed to help the user to perform specific tasks. Some applications are designed to run on smartphones, tablet computers and other mobile devices. These mobile apps are available through application distribution platforms, which are typically operated by the owner of the mobile operating system, such as the Apple App Store, Google Play, Windows Phone Store and BlackBerry App World.

Native application - An application designed to run in the computer environment it is being run in. The term is used to refer to a locally installed application in contrast to various other software architectures. In iOS for mobile devices, native apps include mail, maps and safari. A native app may be contrasted with an emulated application written for a different platform and converted in real time to run.

Mobile Optimized Website - Websites that are built to live on the mobile web; miniaturized websites are not. A mobile-optimized site is a condensed, highly-functional, and elegant accompaniment to the main web presence.

Mini browser - A web browser designed primarily for mobile phones, smartphones and personal digital assistants. Until version 4 it used the Java ME platform, requiring the mobile device to run Java ME applications. From version 5 it is also available as a native application for Android, bada, iOS, Symbian OS, and Windows Mobile.

HTML5 - A markup language for structuring and presenting content for the World Wide Web and a core technology of the Internet. The focus of HTML5 is improving multimedia applications and redesigning the web for a much broader range of devices, particularly mobile devices. While some companies have started to use the language, it's still in its infancy and will not be standard until 2020. Facebook spent 2 years developing a mobile app in HTML5 which CEO, Mark Zuckerberg stated was, "probably the biggest strategic mistake we made."

Simulator - A computer simulation, computer model, or computational model is a computer program, run on a single computer, or network of computers, that attempts to simulate an abstract model of a particular system.

Emulator - In computing, an emulator is hardware or software or both that duplicates (or *emulates*) the functions of a first computer system (the *guest*) in a different second computer system (the *host*), so that the emulated behavior closely resembles the behavior of the real system.

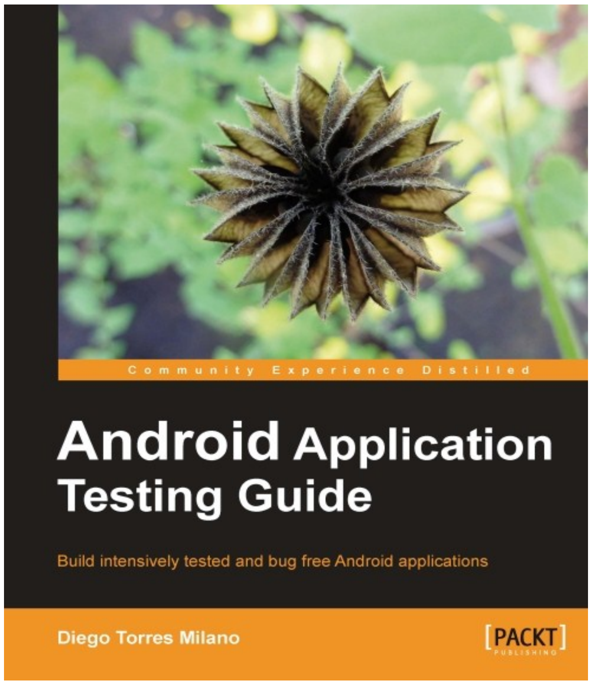
4G - The fourth generation of mobile phone mobile communications standards. A 4G system provides mobile ultra-broadband Internet access, for example to laptops with USB wireless modems, to smartphones, and to other mobile devices. Conceivable applications include amended mobile web access, IP telephony, gaming services, high-definition mobile TV, video conferencing and 3D television.

Sources: Wikipedia, PC Magazine, eWeek

Book Review

Android Application Testing Guide

By Ittichai Chamavanijakul, Motorola


Community Experience Distilled
Android Application Testing Guide
Build intensively tested and bug free Android applications
Diego Torres Milano
[PACKT] PUBLISHING

Testing appears to be the least popular topic in Android development circles based on the relatively few books on Android app testing. Most tend to focus on development because, unfortunately (but true), application testing isn't be something most developers think much about, or if they do, they don't do it systematically (I'm guilty of this as well). This book has allowed me to expand my horizon by learning from the pros.

The "Android Application Testing Guide" is a very practical book introducing available frameworks and most widely used tools & techniques to improve the quality of Android applications by engaging in quality control and testing throughout the development cycle, not just testing at the end. The Agile development concept, called the *Test Driven Development (TDD)*, relies on repeatable short cycles to catch and address potential defects (i.e., bugs) as early as possible.

The first chapter explains what's involved in the different stages of testing in Android development including unit tests, integration tests, functional or acceptance tests, system tests and performance tests. It introduces the Android testing framework extending JUnit which provides the complete testing framework suitable for the end-to-end testing strategies.

Chapter 2 deals with testing with JUnit which is the default framework for any Android testing project and is supported by Eclipse, the most widely-used IDE for Android development. The chapter jumps right into a step-by-step on how to create an Android test project separate from the development project being tested. The test project will have an independent structure and a set of its own components. Having a separate project is the best practice because from a production build's standpoint, testing codes will not be included in the actual build, thus it will be not be in the APK.

Chapter 3 dives into individual building blocks in the Android SDK tool. This covers Assertions, TouchUtils class (to simulate the touch events), Mock objects (to simulate mock objects in order to isolate the tests), TestCase class, and Instrumentation. There is an extensive explanation of each individual component accompanied by code samples.

Chapter 4 talks about the concept of *Test Driven Development*. Again, it is the strategy of performing tests along the development process – not at the end as in the traditional approach. This even includes writing test cases (and test codes) right after studying the requirements, and then writing the actual code to satisfy (having attained "pass" results) the test cases. The author claims that creating test cases early on will help ensure that tests will be performed. Delaying testing makes it highly possible tests will forgotten or ignored. I agree with the concept, but in reality, this may not work for projects. Even the author advises to use judgment and expertise in applying the approach wherever suitable. The latter part of the chapter shows examples of applying TDD in step-by-step sample applications and tests, which makes the case very compelling.

Chapter 5 introduces the *Android Virtual Device (AVD),* the next best thing to real devices. The obvious advantage of AVD is the ability to create a variety of Android configurations to run tests. The *Headless Emulator* is also mentioned. This enables automating tests via command line with no windows. I really enjoyed the samples of what you can do with command-lines, especially simulating different device conditions including network bandwidth throttling, different locales, etc. Later in the chapter, it talks about the support of the *Monkey* application (you can read more about Monkey Theorem here) which allows random event generation. It also demos the server-client setup and test scripting with Monkeyrunner.

Chapter 6 discusses the Behavior Driven Development approach, which according to author is considered the evolution of TDD and a need for *Acceptance Testing*. The approach emphasizes not only business or end users in testing, but also to use non-technical terms in the test cases that business or end users would understand. Personally, I think the concept is more abstract than practical. However, the introduction of Fitnesse and Slim tools point out this concept's usefulness especially in terms of "Wiki"-style test cases and test result documentation.

Chapter 7 contains practical, real-world samples of the disciplines and techniques you can use in different situations. This includes testing activities and applications, databases and Content Providers, UIs, exceptions, parsers and memory leaks. A couple of tools are introduced here including EasyMock which provides mock objects for interfaces, and Hamcrest which is a library of matcher objects for comparison.

Chapter 8 expands on the Agile technique with continuous integration. Similar to continuous testing, the author suggests that integration should be done frequently and early in the process in small steps. The most common practice is to trigger the build process after every submission to the source code repository. The chapter goes into detail on how to setup an automation building process using ant, version control repository using Git, continuous integration with Hudson, and test result analysis using nbandroid-utils.

Chapter 9 discusses one of the most critical components in designing Android applications - performance. As we all know, mobile devices are unique and balancing between performance and effective use of resources is something developers must keep in mind when designing an Android application. The chapter contains an outline of tools and techniques for performance testing. This includes using traditional *LogCat*logging, extending Instrumentation, Traceview, dmtracedump, and Caliper microbenchmarks.

Chapter 10 states that alternative testing tactics are not for most Android developers as it involves building Android from source code, and introduces more testing tools. Even though it is very technical, it is still a good read.

Summary*:*

I have to say that as a novice Android developer I learned a lot from reading the *Android Application Testing Guide*. Best -in-class testing techniques commonly practiced in Agile development are covered in this book. These include Test Driven Development, Behavior Driven Development, Continuous Integration as well as others. The traditional approach of testing and integration at the end of development cycle is generally opposed. However, as the author states in the book, there are no silver bullets in software development – the suggested testing approaches might not apply to certain needs or projects. It is clear that for a large, complex project, applying the methods and techniques as described will likely increase the productivity and quality of your apps.

Numerous tools are introduced in this book. This may be one of the big issues for those who are less-advanced in Android development. Since there is no common repository for these tools, extended knowledge of a variety of open-source tools, and the ability to incorporate them in the development process are vital. I wish that there were a consolidated repository for all testing tools or even better - all required testing functionality would be built-in in the SDK. But while we wait for that to happen, the *Android Application Testing Guide* is one of a few reference books I'm aware of that has the complete collection of Android testing topics. ■

About Ittichai

Ittichai has been involved in many aspects of Oracle database technologies including RAC, ASM, Data Guard and Streams. He enjoys working on database and SQL performance tunings. He's also interested in rapid web development using Oracle APEX.

He is currently an IT technical manager and data warehouse architect with Motorola Solutions.

Rain - A Blessing in Disguise

Rain is a constant in Vietnam. While sometimes it may inconvenience us, it is one of the country's most valuable natural resources.

By Brian Letwin

Rain is one of Vietnam's defining characteristics. It floods the streets, feeds the fields, cleans the cities and provides a soothing soundtrack at night. Some days we curse it, some days we embrace it, but it's a vital part of life in Vietnam.

Every year is defined by two distinct monsoon periods. The southwest monsoon extends from April to September and is accompanied by warm weather. The northeast monsoon marks the colder weather months of October through March.

The rain is hardly predictable. It can last for a few minutes or for days, but regardless, the Vietnamese have adapted and seemingly take it in stride.

When I arrived in Vietnam I was sporting a pair of white sneakers. I noticed the locals wearing sandals or flip flops pointing at me and assumed they were admiring my shoes. After a week my sneakers turned gray, and I too learned to appreciate flip flops.

At times, rain can be a huge challenge to the infrastructure, turning streets and alleys into small rivers. Buses and taxis offer some refuge from the rain, but most motorbike riders prefer their personal transportation. Donning body-encompassing raincoats, they go about their business undeterred.

The inconvenience is small in relation to the benefits the rain provides for Vietnam's economy. Rivers from China, Laos and Cambodia that begin as melt from ice of the Tibetan Plateau are continuously reinforced by heavy rains as they snake their way south and empty out across the Mekong Delta. The soil rich deposits in the vast region have helped make Vietnam a top agricultural exporter. Agriculture is a huge source of employment and makes up over 20% of Vietnam's GDP with exports of coffee, cotton, peanuts, rubber, sugarcane, and tea to countries around the world.

There are plenty of moments when I shake my fist at the sky after a solid week of rain. But in the end, I realize the benefits far outweigh the slight personal inconveniences. ■

TestArchitect™

Action Based Automated Testing Goes Mobile

testarchitect.com | 800.322.0333

LogiGear®

LOGIGEAR MAGAZINE
DECEMBER 2012 I VOL VI I ISSUE 5

United States	Viet Nam, Da Nang	Viet Nam, Ho Chi Minh City
2015 Pioneer Ct., Suite B	7th floor, Dana Book Building	1A Phan Xich Long, Ward 2
San Mateo, CA 94403	76-78 Bach Dang	Phu Nhuan District
Tel +1 650 572 1400	Hai Chau District	Tel +84 8 3995 4072
Fax +1 650 572 2822	Tel: +84 511 3655 333	Fax +84 8 3995 4076
	Fax: +84 511 3655 336	