As fast as Mobile is growing, the platform is still immature and is evolving at a very rapid pace.

While there are whole countries that have migrated large government services to mobile, countries ranging from Estonia to Turkey to Kenya have many longtime mobile users have yet to use mPay or other mobile payment systems. Users are still maturing in their trust, reliance and comfort managing their lives on mobile devices.

Mobile development companies continue to advance in supported platforms, cross platform development, greater sensor use and test automation. Yet many companies still lag in security, usability and performance testing. Mobile test strategies need to mature.

The maturing of test strategies is not only a response to development and device changes - users are more demanding and critical. Criticism in social media is rapid and damning. Bad reviews spread like wildfire and go viral before a company may have a chance to remediate a problem. Usability and performance are common criticisms and these quality aspects are commonly under tested.

The platform is evolving. New sensors from gesture barcode scanners to fingerprint scanners and more NFC (near field communication) are becoming commonplace. New integration with backend systems and data are making mobile systems more complex than the much better understood web and client/server systems. Most browsers and client systems to not have such great variation in screen size and resolution as mobile devices, do not have connection problems of dropping in and out of range, have easier update and patching control, do not use sensors for location, light, environment, IR, Bluetooth, NFC or swipe for data entry! The mobile platforms are complex and getting even more so.

How can test teams respond to these maturity problems? Problems between emulators and real devices, access to devices, test automation - especially cross-platform automation issues or roadblocks need to be resolved for very rapid deployment.

In this issue: the team at Virtual City provides a list of steps to identify weaknesses in your new app; LogiGear’s John Kane reveals how to maximize mobile test automation efficiency; Tania Lang argues that the traditional testing standards must be applied to mobile; Tatyana Mahlaeva reminds us that QA is necessary in all stages of mobile development, and the HowAboutWe team warns that even in large organizations, mobile development is behind in up-to-date testing practices.

Remember, for your reference, please read our other issues devoted exclusively to mobile testing:

Mobile Testing - November, 2011
Mobile Test Automation - December, 2012

Our next issue is on another fast growing technology topic and its impact on testing: Cloud and SAAS (Software-as-a-service) Testing. I can’t wait.

Michael Hackett
Senior Vice President, LogiGear Corporation
Editor in Chief
In this Issue

4 IN THE NEWS

5 BETA TESTING MOBILE APPS: HOW TO GET IT RIGHT

The team at Virtual City

Steps that will enable you to identify the weaknesses of your new app, its vulnerabilities and strengths.

6 OPTIONS FOR EFFICIENT MOBILE TESTING

John Kane, LogiGear Corporation

By focusing on test design, analyzing test requirements and optimizing the approach to testing, it’s possible to maximize mobile test automation cost effectively.

8 EIGHT LESSONS IN MOBILE UsABILITY TESTING

Tania Lang, Peak Usability

Mobile is no longer an area that a few UX people specialize in, and we need to start designing and testing everything for smartphones and tablets as well as computers.

11 BOOK REVIEW

David Greenlees

A review of “Tap into Mobile Testing” by Jonathan Kohl.

14 MOBILE COMPATIBILITY: THE INCREASINGLY FRACTURED MOBILE DEVICE MARKET

Some statistics and graphs that illustrate global trends in mobile testing.

17 TESTING MOBILE APPS: QA CAN MAKE OR BREAK PROJECT’S SUCCESS

Tatyana Mahlaeva, A1QA

To ensure the success of an app, QA must be involved in all stages of development.

20 FINALLY, 3 WAYS TO AUTOMATE IOS TESTING

HowAboutWe

iOS culture, even in many large organizations with skilled engineers, is behind on up-to-date testing practices.

25 MOBILE TESTING GLOSSARY

Some of the terms used when discussing mobile testing.

26 BANH MI—FROM FRENCH IMPORT TO VIETNAMESE STAPLE

Brian Letwin, LogiGear Corporation

Where the Vietnamese go, the banh mi follows. With one of the largest diaspora communities in the world, banh mi has long been a staple in overseas Vietnamese (Viet Kieu) communities.
In the News

New Report on Global Software Testing Market

MarketResearchReports.Biz recently released a report on the state of global software testing market. The report predicts that the Global Software Testing Services market will continue to grow steadily at a combined average growth rate of 5.41 percent over the next three years.

The key factors the report cited as contributing to market growth are the reduction in operation time and cost. Other contributing factors are the increased demand for testing as a number of organizations are embedding testing into the software development process as part of the adoption of the independent Test Maturity Model Integration (TMMi) as cited in a report by Experimentus.

The Global Software Testing Services market has also been witnessing an increase in the adoption of cloud-based testing services. However, there isn’t a clear trend to predict the growth of this segment of the market.

Ericsson Builds New R&D Centers for Cloud Research, Interoperability Testing

Ericsson has announced plans to expand its research and development efforts, including those in the cloud space. On Sep. 2, the Swedish networking company announced it would open two new R&D facilities in its home country this year and next, in addition to a facility in Montreal, Canada in 2015.

The three R&D centers will provide state-of-the-art hardware and software testing facilities for Ericsson's 24,000 engineers around the world. Ericsson is also opening a new hardware design building in Stockholm.

According to the company, Ericsson has plans to provide its customers with remote access into the centers for interoperability testing purposes.

LogiGear to Offer New Mobile Class

This 2-day hands-on course offers an intensive experience in a new way of thinking for mobile testing. It presents essential topics for effectively testing on the important and evolving mobile platforms. It provides, thorough lecture, discussions, reading, and hands-on examples, important ideas, perspectives, strategies and experience in state-of-the-art methods for effective mobile testing.

Focusing on testing implications unique to mobile devices, it also helps you evaluate your testing practices, including usability and compatibility testing to develop a test strategy in today’s world of rapid Agile deployment. You can access the course here.

Julian Harty Video Interview on Testing Aspects in Mobile

At VISTACon 2013, LogiGear SVP, Michael Hackett sat down with mobile testing expert, Julian Harty to discuss cross-platform development, how mobile is altering the day-to-day role of testers, if usability is a major consideration for mobile app users and how mobile is changing the human resource requirements of testing teams. Watch the video here.
So you’ve just finished developing a nifty, customisable app that can help farmers track their produce from source to market via their mobile phone. You’re elated and want to get started marketing it right away. Not to burst you bubble, but are you 100% sure that the app actually works across all mobile platforms and is scalable affordably?

As a developer, the foremost thought that should linger at the back of your mind is not whether the app is cool; yes, it solves a real need for farmers, especially those in rural areas who have no idea whether they are getting real value from their produce sales. But a question you should have an answer to is: will the app be received well by the public, the consumers of the applications – consumers in your case being the agribusiness stakeholders for example? Whether it will fly off the app store shelves or stare back at the creator with a one star rating, is something you ought to find out prior to launch phase.

Beta testing your mobile app

How to get it right is really that question that can have you going back to the drawing board. Beta testing comes in to guarantee the excellence of your app, something that can be determined while still in the testing process and save you loads of cash and embarrassment. Beta testing simply determines whether an application works: i.e is usable, functional, practical and most importantly, flawless. It also helps you better understand your target audience and capitalize on adequate business analysis.

If you have no idea how to carry out your Beta testing for your new application, here are a few ideas you could consider.

1. **Who is your target audience?**
   
   Before you even start developing the app, figure out who you are designing it for. True, beta testing is all about testing the app after it has been created, but the initial testing needs to be completed even before you start the creating process. Design usually follows purpose. If you figure out who the app is for, you are more likely to create a bug free app.

2. **Get in touch with tech bloggers**

   Some app developers go the route of hiring an app testing firm, or have a dedicated team of app testers as part of their staff. If however you are starting out, the easiest thing you could do is get in touch with- or involve tech bloggers in beta-testing the app. You can generate social media hype around the testing, that way you will not only get tech bloggers to test the app, but also create anticipation for the app. If the beta testing goes well then you’re certain from the buzz already generated your app has willing buyers or users.

3. **Value feedback and adjust**

   The reason why you involved beta-testers in the first place, was to allow you to know where your app fails and where it flourishes. Always consider that the success of your app is dependent on the feedback you receive. Understand that testing the app is about way more than just reducing the bugs in how the program runs, but also listening to your users and consumers. Be ready to embrace the criticism that will come, fix the bugs and problems, adjust, and roll with the punches. The responses, comments, and thoughts you receive from the testers are invaluable to your app creation process. The areas to concentrate on here include the usability and functionality of the app. How satisfied are your customers? Is the app interactive? etc.

In closing, the above steps will enable you to identify the weaknesses of your new app, its vulnerabilities and strengths. The process will also enable you to eliminate anything that needs eliminating and prepare your app for the app store. In the end, help you create and deliver something that the customer wants and needs.
In a previous article we outlined the importance of understanding the mobile ecosystem and test design for planning and executing mobile testing. The focus of this article is about efficient mobile test execution and test coverage maximization.

Despite the growing number of mobile testing solutions the market is far from mature, which is one of the reasons why the majority of mobile device testing is still done manually. The primary issues with manual testing for mobile are scalability and cost. Outsourcing is always an option to increase the scale of testing, and then there is crowdsourcing. Crowdsourced mobile testing is being promoted as an ultra low-cost method of outsourced mobile testing. Crowdsourcing proponents tout flexibility, multilingual testing and a broad cross section of devices, but crowdsourcing is performed by a collection of freelancers so keep in mind the oft-used advertising disclaimer: “Results may vary.”

To adequately perform testing that assures mobile applications will meet customer expectations, test automation is necessary, and the need will only grow as mobile applications increase in complexity and importance. A high degree of test coverage can be achieved using both emulators and actual devices. How much testing to perform using emulators versus actual devices will depend on the budget, the automation tool, the application and the type of testing.

We have approached mobile test automation to make it possible for testers to employ a combination of testing assets—emulator/device/cloud testing resources—to efficiently achieve testing goals. Our automation tools provide the features and functionality to scale mobile test automation along with project needs and resources.

Emulators are ideal for testing new functionality or a new component design, and have some advantages over using actual devices. Logging faults and capturing screenshots are much simpler when working from a desktop, and you can simulate conditions that are hard to duplicate on real devices, like low battery power. A nice feature of TestArchitect is the ability to perform multilingual testing with emulators. This feature allows running identical tests in multiple languages in a short period of time.

Emulators can be accessed from the respective OS (Android, iOS, Windows Phone, Blackberry etc.) vendor’s SDK for free. Unlike iOS, Android isn’t the same for every device. For the best device approximation emulators from each Android device provider should be used. Then there are tools to augment the device emulators like Fiddler, NetLimiter and Charles Proxy that simulate GSM/GPRS networks to include testing of near real-world network conditions.

The main thing to keep in mind is that emulators and simulators only provide an approximation of device performance. From a QA perspective, emulators and simulators are fine for testing base functionality, but critical application sign-off should be done by live testing.
Testing a representative sample of actual devices is always ideal. The choice here is acquiring and updating an inventory of devices or using cloud base services. For overall maximum test coverage a public cloud option will provide broadest device selection. There are a number of cloud based options, each with their pros and cons. The big considerations are avoiding the need to recreate tests and security of the devices.

The accepted norm is to perform tests on 30 to 40 different devices currently available in the market. In cases where it’s essential to test actual devices, but less than the ideal number, applying Pareto’s Principle—80% of your customers will use just 20% of the devices available—should enable you to select the minimum number of devices that will provide the maximum test coverage.

Our approach to mobile cloud test automation was to create a host/agent set-up. Agents are downloaded onto the mobile devices (via wi-fi) to be tested, and tests are run remotely from the host machine. TestArchitect allows up to 40 devices to be tested simultaneously on a host or agent equipped machine to test within accepted norms. This approach eliminates API programming and provides the potential for regional testing—important for testing apps/devices on various operators’ networks, and in multiple languages—to validate applications for use on a global scale. The additional advantage of using TestArchitect is that tests used for any initial emulator testing can be run without modification on the actual devices, whether in the cloud or physically connected.

A well thought out combination of emulators, actual devices and cloud testing can provide broad test coverage and a high degree of automation. New applications can first be tested using emulators to test base functionality and/or multiple languages and identify major bugs. Next, targeted and thorough testing can be performed on actual devices. Segregating testing in this manner can speed testing, and significantly help separate OS issues from device issues to minimize debugging time. Similarly, current applications can be tested on emulators of new or upgraded OS’s first, and then on a representative sample of next generation devices. Regression testing can be performed in the cloud across a wide range of devices with various hardware and OS versions.

Mobile devices are quickly becoming preferred to the desktop. To assure the best user experience, it is important to test mobile apps and websites with actual devices, and to keep this up requires automation. Mobile test automation is easily achievable. By focusing on test design, analyzing test requirements and optimizing the approach to testing, it’s possible to maximize mobile test automation and to do it cost effectively.

**About John**

John Kane oversees LogiGear’s Product Program Management to develop, productize and deliver LogiGear’s Testing Automation solutions to the marketplace. John has more than thirty years of experience in the software industry in the areas Electronic Design Automation (software for designing computer chips), Software Licensing, Chip Design, Government ARPA Programs, and Software Quality Consulting and Services. John Kane holds a Bachelor of Science in Computer Science from Oregon State University and a mini-MBA from Santa Clara University Leavey School of Business.
Eight Lessons in Mobile Usability Testing

Mobile is no longer an area that a few UX people specialize in, and we need to start designing and testing everything for smartphones and tablets as well as computers.

By Tania Lang, Peak Usability

**Lesson #1: How to test mobile paper prototypes**

One of my favorite quotes comes from Frank Lloyd Wright: “Fix it on the drafting board with an eraser or on the construction site with a sledgehammer.” Testing paper prototypes with users has its merits but how do you test paper prototypes of mobile screens that require scrolling?

Showing users a paper mock-up of a full screen does not really test what they would do when they only see the first screen. My solution was to develop a cardboard template of a smart phone with two slits through which a paper prototype can be inserted. This simulates what users would see on one screen. You can easily create your own or download our smartphone template for paper prototype testing and print or stick on cardboard.

**Lesson #2: Limitations with screen recorders**

The obvious benefit of using a smartphone screen recorder is that it is unobtrusive and does not get in the way of testing. Users can use the device as they typically would in any context or environment.

Unfortunately, Apple isn’t big on approving screen-recording apps for sale in the iTunes store. There are screen recording apps such as Display Recorder (from Cydia) that you can use to record screen activity but personally, I don’t want to jailbreak my iPhone in order to record a mobile test session. There are a few Android screen recorder apps available, but even if you find a screen recorder app for your smartphone, there are a few issues.

- Screen recorders don’t typically record users’ gestures, such as trying to tap 10 times on a tiny link or a user attempting to use a slider on a touch screen.
- They often have limited recording times and capabilities. You will fill your phone up very quickly if recording one-hour sessions.
- They generally can’t be used to record screen activity of another app so you may be limited to recording mobile web sessions only.
- You can’t use the test participant’s own device unless you want to install an expensive app on their iPhone, which they may not be too keen about.
- They don’t integrate with usability testing software such as Morae, which allows observers to unobtrusively observe a session in real time.
- You can’t typically record user’s facial expressions and verbal comments like you can with standard usability testing software such as Morae.

There are two iOS apps that I know of that partially address the last issue. One is called UX Recorder. It does a reasonably good job of recording screen activity, captures some user gestures (showing circles for taps and arrows for swipes), and records a user’s voice and face (picture-in-picture) using the iPhone’s own camera. Its biggest drawback is you can’t use it to test mobile apps. You can only use it to test mobile web.
The second is a relatively new iOS app called Magitest, which, in my view, is more usable than UX Recorder. Magitest captures tap gestures only at this point (although they are soon to add swipe capture, I believe), as well as user’s voice and face using the iPhone camera (picture-in-picture). More importantly, using Magitest Native you can also record screen activity using other iOS apps. It is also quite reasonably priced.

Lesson #3: Benefits of using mobile testing sleds

The alternative to screen recorders is a mobile or tablet testing sled that will allow you to mount a small camera over the mobile device. There are a few key benefits in using sleds.

- You can record users’ gestures, which are really important for mobile and tablet usability testing. We once had a user trying about 10 times to use a slider on a tablet before giving up. You can also plug your sled webcam into your computer and input into testing software such as Morae. The great thing about Morae is that it lets you input two cameras so you can also record the user’s face and voice using a second webcam. You can then record user’s screen activity, gestures, voice and stream all of it live across your network to an observation room or create video clips later to effectively communicate issues to stakeholders.

Nearly all the UX practitioners I know are using DIY sled systems that they have made themselves. We have investigated a number of mobile test sleds and developed a few of our own over the last two years. Here are some of the things to be aware of:

- The sled must be light and easy for test participants to hold.
- The sled should not slow users down or get in their way.
- The sled should accommodate different sized devices so that you can use test participants’ own devices.
- The camera mount should ideally be adjustable to accommodate user preferences and different devices.
- The sled should allow users to easily switch device as well as orientation i.e. portrait to landscape view without having to move the camera position.
- The whole sled set up must still be stable and not move during testing.

Lesson #4: What cameras work best for mobile testing

When we started out with our first sled, we used two Logitech webcams: C910 and C615. These webcams offer good quality, high definition recordings, plug into Morae test software and have easy to use drivers. The main disadvantages are the size and weight. The Logitech C910 is also difficult to connect to a sled.

We found the Microsoft Lifecam offers good 720p resolution, works with Morae, and is easy to connect to the sled as it can bolt on. The downsides of this camera are also the size and weight, plus less flexibility in the camera mount and camera software we find difficult to use.

An alternative to webcams is the IPevo document camera. What we like most about this camera is that it smaller and very lightweight, has a very high resolution, plugs into Morae, and comes with a flexible plastic arm. The down side is that it is difficult to attach to a sled and the frame rate is a bit slow and jerky as it is meant for recording documents.

Our favourite webcam is the Hue HD webcam, which is very light, offers good resolution and frame rates, comes with a flexible bendable arm, and plugs directly into a PC and Morae. It even comes in pretty colours such as pink and green, but as we don’t want to draw attention to the camera we use plain old black.

Lesson #5: There are times when you should use no technology

So far, I’ve mainly focused on mobile testing technology but there are times when I advocate using no technology (with the exception of the user’s phone, of course). As I mentioned, I recently conducted testing of a quit smoking app designed for pregnant women. We ended up recruiting mothers with babies who had recently quit smoking or were attempting to quit. Given the sensitive nature of the topic (women who had tried to quit smoking when pregnant) and who I was testing with (new mothers with babies), I needed to conduct many of the test sessions in users’ homes.
As a mother with young children, it was easy to build rapport with these women and for them to open up to me as I attempted to make the testing as casual as possible. One of the test sessions was even run on a participant’s sofa while she was breastfeeding her 5-month-old. She had baby in one arm and tested the app with her iPhone in her other hand. Obviously it was not appropriate to record these sessions in any way and the intrusion of any additional technology would have potentially affected the results. Sometimes it is better not to record at all.

Lesson #6: Context may not always be as important as you think

We hear that context is critical for mobile design but it’s more important for some types of sites or apps than others. There are some interesting mobile web statistics from comScore that actually indicate that the highest use of mobile devices is actually early in the morning and after 8pm at night, presumably when most people are home.

My advice is that you need to consider where users are likely to use your app. For instance, many users of the quit smoking app for pregnant women said they would probably use the app alone at home when they have limited resolve and their phone is on hand. We also worked on a public transport journey planner for mobile, which would much more likely be used when users are out and about, walking down the street. For that mobile site, context was critical.

Lesson #7: How to simulate users’ context and environment

In a lab environment, we may have a stable wireless connection and might not be moving about, but this doesn’t happen often in real life. People use their phones on trains, buses, and while waiting in a queue. Signals drop out. Activity often occurs in short bursts (such as during TV ad breaks). So how do we test for this?

One of our insurance clients was interested in understanding users’ expectations regarding what would happen in the event they lost their signal or left their session and returned to it 30 minutes later. Halfway through their task of getting an insurance quote, I actually stopped users and said “imagine you were doing this on a train and the signal dropped out as you went through a tunnel.” I then asked them what they would expect to happen and showed them a screenshot to help determine what they would do. So even though we could not test on a train, I managed to simulate that experience.

When I conducted testing of the quit-smoking mobile app, some users preferred to come into our test lab. Instead of using our lab’s test room with cameras, desks, and computers, I tried to recreate a “home” like environment, running the sessions on our sofa in our test observation room without using any technology to record the session. Even though this wasn’t the users’ true context, we simulated their home environment and the test outcomes were consistent with sessions I ran in other users’ homes.

Lesson #8: Employing users’ own technology will discover more issues

When we conducted testing of a mobile insurance website, we found that Galaxy and iPhone users loved the pickers for date input. However, I had one user with a HTC phone that really struggled with the date picker for inputs such as years. The year values in the date picker were very slow, and he scrolled quickly it scrolled the whole page.

When testing a public transport journey planner, we used users’ own mobile carrier network connections and some screens/search results were really slow to load. This had an effect on the user experience and satisfaction. This was good information to understand and pass on to the client prior to launch.

By employing user’s own devices and network connections we found issues that we would not have picked up if we had just used our iPhone and fast office wireless connection.

In conclusion

With nearly a third of all page views now coming from mobile and tablets in some Western countries (and even more in many developing countries) UX practitioners and researchers need to embrace mobile. Mobile is no longer an area that a few UX people specialize in, and we need to start designing and testing everything for smartphones and tablets as well as computers.

Hopefully this article will save UX practitioners new to mobile some of the pain I have been through.

About Tania

Tania Lang is the principal of Peak Usability as well as a member of the Usability Professionals’ Association, Australian Web Industry Association, and the Queensland State Representative for CHISIG in Australia. She is considered one of the leaders in her field and is passionate about usability. She has organized numerous local usability and HCI events to increase awareness and adoption of good usability and user centered design practices.
Here's some good news: Jonathan Kohl has a new book out titled *Tap into Mobile Application Testing*. This is a Leanpub release and is still in Beta, but the beauty of buying through Leanpub - you get all the author updates as they make them! However I will be keeping an eye out for the print-on-demand service in early 2013, because I love ‘real’ books and what an awesome cover!

What an amazing amount of detail in one book. As I was reading it, the test ideas were jumping out of the page; so much so that I wanted a mobile application project to test. I could dazzle the project team with all of ‘my’ wonderful test ideas (while referring to the book at the same time). Don’t worry Jonathan, I would give you credit...eventually.

I like the way the book is structured - the order of chapters maintain a great flow through the entire process of testing mobile applications and devices. From the introduction to mobile technology, through to performance and security. The book will be a definite reference point for me if developing a strategy for mobile application testing in the future!

Chapters 2 and 3 were incredibly valuable! As mentioned above, test ideas jump off the pages and into your brain.

Chapter 2 walks you through and overview of mobile technology and puts a lot of things into perspective; namely just how complex these little devices are. We all know that understanding what you are testing and the technology behind it is important right? Well this chapter goes a long way to helping you with that.

Each mobile device component is listed with examples of how they are used and what could wrong, among other things. Each of these would provide very valuable input to any mobile testing strategy.

Chapter 3 explains many different types of mobile testing tours (if you haven’t heard of these, try here and here as a start pointing). I had done some research into testing tours prior to reading this book; however I had never looked into tours for mobile testing. This chapter provides a significant list of possible tours, accompanied by a significant amount of detail for each one. The User Tour stuck in my head; possibly due to the following:

“...go to a local coffee shop and observe other people as they use the devices. Do they get frustrated? Talk to them (without being creepy) and ask them about their devices, what they like and don’t like. Watch for emotion: are they happy? Angry? Frustrated? Does technology make them feel dumb?”

I wrote about this type of concept here. I think there is immense value to be gained through such a process, and not just at the ‘end’ of the project, but also initially as the application is being designed.

This of course (using Jonathan’s excellent structure) flows directly on to chapter 4: Test Using Different Perspectives. I SLICED UP FUN is then introduced - an awesome mobile testing mnemonic; and awesome thinking tool.

If you are testing mobile devices then I urge you to look at this and use it. Jonathan walks you through this by actually testing an application (of your choice of course). I would encourage all readers of the book to actually pickup a mobile device and test while Jonathan takes you through the process (not the best word to use, but you get it).

Obviously this will take you longer, but I don’t believe this book is a ‘from start to finish’ publication. It’s a set of tools to help you test better and what other ways are there to learn how to test better than actually testing?

None.
One aspect of this chapter that stands out for me is the bug story for each letter of the mnemonic. After Jonathan has finished explaining the letter and providing testing tips he tells a bug story from his experience that relates to the letter. Story telling in a book, who would’ve thought? Seriously though, a lot of testing books don’t tell stories from experience, they preach theory. Of course they too have their place, but experience is REAL, and real adds significant value to my testing.

I would also like to quote a section of the book that I completely agree with...

―Usability is the single most important factor for a mobile app. It is so crucial, I recommend that you study mobile usability works so that you can find and report important usability problems quickly. As I’ve mentioned before, if your app isn’t very usable, people will just delete it. (The deletable offence.) If it annoys them, they may just go on to a social networking service and rant about your app. (A rantable offence.) Or, they may just quietly give your app a poor rating on application stores.‖

Now, who said that all testing can be automated? Chapter 5 takes you through some interesting information in relation to logging and diagnosing bugs. One thing that we as testers seem to be reasonably good at is logging the bugs, however I think there is a lot more work to be done in the diagnosing space! Quoting from the book, which is quoting Cem Kaner...

―The best tester isn’t the one who finds the most bugs or who embarrasses the most programmers. The best tester is the one who gets the most bugs fixed.‖ - Cem Kaner

Before you jump on me, I agree that it isn’t a tester’s job to fix bugs (if you can, then cool), but we can do enough investigation to make our bug reports so compelling that they are hard to ignore.

There is a lot of information on bug advocacy online, most prominent is Cem Kaner’s paper; Bug Advocacy: How to Win Friends, and Stomp Bugs.

Jonathan does a great job of sharing his thoughts on the subject, and of course putting a mobile testing slant on it. He focuses on the important information to put in a mobile testing bug report. One very important message that shines through is the value in being exact with your steps to reproduce.

Mobile devices are, well, mobile.

The amount of varying scenarios more than doubles to that of a desktop application where your PC is stationary. It is extremely important to be as exact as possible when recording your steps. A change of location by 10 feet could make a difference when trying to reproduce a bug (read from page 214 to see what I mean).

Chapter 6 is all about the mobile testing strategy. The question that always comes to my mind in relation to mobile testing is what platform to use? It seems it is also the most common question Jonathan gets too!

The first part of this chapter covers all sorts of different elements that you need to consider when trying to answer that question. Check out Jonathan’s four different strategies for choosing which platforms to test with.

In this chapter, Jonathan walks you through a strategy example for a particular platform choice. Even the simplest of choices quickly adds up to a huge amount of possible combinations to be considered. This is a great example of just how complex the mobile testing space can be.

The second part of chapter 6 looks at Jonathan’s ‘general’ process when creating a mobile testing strategy. There are five steps to the process, to which Jonathan explains in great detail. I enjoyed this and will definitely pull this piece out so I can carry it from project to project. In closing Jonathan talks about military strategy and it’s obvious alignment to strategies for almost anything. He runs through OODA, which is brilliant. You’ll need to read the book to see how this fits in.

Guidance and planning are next on the menu.

Chapter 7 begins by highlighting the importance of structure, which stands true for any testing project, let alone in the complex world of mobile. However, as Jonathan points out, don’t overdo it! Mobile projects in general are extremely fast moving, and you don’t want to structure things so much that they slow you down. A happy balance is required.

This is a great chapter focusing on coverage, guidance (on the move as well), checklists, maps, how-to guides, session based testing, and models (including modelling the users)!

This last one was particularly interesting for me as I had experienced it first hand.
Not too long ago I had a problem with my car and the mobile (pun not intended) mechanic paid a visit. All went well until he went to use the new tablet they had just installed in his truck.

It was a new way of recording the job real-time with data being sent straight to the server instead of the old paperwork and manual input. The screen would just not recognise his gestures. After some expletives and getting rather wet in the rain that had just started, I offered to have a look for him (being that I ‘worked’ in IT). Guess what, all my gestures were recognised straight away! Seems my hands were a hell of a lot cleaner than his. Long story short, they now have wet wipes in the truck so they can clean their hands before lodging the jobs.

Another noteworthy mention for this chapter: TAP IT UP. I’ll leave this to you and the book. I will say though, it’s awesome!

I recall when I first started reviewing the book I made a note to myself to make sure Jonathan had written about the differences in native and web applications on mobile devices. Enter chapter 8. Jonathan’s explanation of web application history is very interesting indeed. I was only a youngster when the web came about, so the majority of ‘amazing’ discoveries had been made by the time I started using it. Now I find it fascinating to read about. Jonathan walks us through how web applications work, all while reminding us how their native brothers and sisters differ.

He provides a valuable explanation of all the components that make up web applications, and while reading I had various test ideas pop into my head. All of which were ideas that I could have used on previous projects (for PC web applications). Jonathan does a good job with a pretty dry subject. It’s as though he’s telling a story instead of just throwing theory at us. My patience runs pretty thin when reading theory on certain technologies, but having my mobile device along side of me helped immensely (something that Jonathan encourages throughout the book).

One particularly good section of this chapter is Mobile Web Challenges, and each of the considerations detailed after this. Yes, you guessed it! More test ideas. In case you hadn’t noticed, this book is solid gold for those!

Chapter 9 gives you a great reminder; PS: Don’t Forget about Performance and Security! These two subjects are only an overview, as we all know that each could have their own books (or two, maybe three). It gets started with performance, and Jonathan provides a great analogy for performance benchmarks via the use of cars. Seems he’s a bit of a car nut, not unlike myself, so this was a particularly good analogy for me (and more interesting than most).

Jonathan then calls on some expertise from John Carpenter to describe some performance areas to watch for when testing mobile applications. This is a great little section that I’ll be returning to (like many others). Oh yes, there is mention of a stop watch; as silly as that may sound these days, it’s very useful. I’m actually using one on my current project, just for some baseline metrics. It’s good to know the experts still use them.

Next up we scratch the surface of security testing for mobile applications. Jonathan describes some basics of security testing, however is careful to warn readers that there is a lot more to it. Things like password protection, application input attacks, and technical analysis. A useful highlight into the VERY complex world of mobile applications security testing. Jonathan then closes out the chapter briefly covering some other common test approaches such as accessibility, localization, and duration testing (OK, not so commonly known - duration testing is using the application for extended periods of time which can yield some great results when compared to the usual short bursts of testing).

Chapter 10 captures Jonathan’s final thoughts and a very cool little section called The Seven Deadly Sins of Mobile Apps. Good fun.

But wait, there’s more - appendix 1 and 2! 1 provides a list of, and details, some basic test tools to get you started. I particularly like; Your Brain & Imagination. What better tool is there? 2 lists some really great quick tests, which will also get you off to a great start testing mobile applications.

In total, 446 pages of brilliance! Have I mentioned I’m not a huge fan of reading? Not sure I can get away with saying that anymore, with all the proof reading gigs I have on the go! However, what I’m trying to say is that this is a monster of a book for my standards, but it was easy for me to read. Jonathan has a story telling style of writing that maintained my interest, even through the ‘dry’ subjects. I now have loads of ammunition; I just need to find a mobile testing project to unleash it!

About David

David Greenlees has been testing software for over 10 years. Many of these spent in one of Australia’s largest government departments, while more recently undertaking a consultant role in multiple organisations. He is a vocal and valued member of the Context-Driven Testing Community and is extremely passionate about the betterment of the software testing craft.

He has published several articles and blogs regularly at http://martialtester.wordpress.com/ and http://helлотестworld.com/.
Mobile Compatibility Testing: The Increasingly Fractured Mobile Market

If you test and are tasked with mobile compatibility testing, Good Luck! How do you select what devices to test? What criteria do you use?

In August, 2012, OpenSignalMaps published some now famous research that found almost 4,000 distinct devices in use running on Android. Today the number is even bigger!

Can there be such a thing as a future-proof app?

Your hope is that your app will be out in the world for a while!

Mobile OS is still a battle for mind and market share.

People often compare mobile platforms to the “browser wars” of the 90s. IE vs Netscape vs AOL, etc. After 10 years or so, browsers converged into sets of standards. Even with a wider variety of browsers today- IE, Safari, Chrome, FireFox, Opera – many follow the Mozilla standard or if not, they are headed in the same direction. HTML5 compliance also has browsers converging along a set of standards.

Today, one dev team may work on all the supported desktop browsers. No problem. In the mobile world, there are likely separate app teams for iOS, Android, Windows mobile, etc. The platform is fractured and there are no signs of convergence.
What is the primary device type?

What is the primary use of the device?

What are the market factors?

How can we test? What kind of platform coverage can or should we aim for? Which devices do we test for compatibility testing? Only the latest? Prioritize according to market share? What coverage will we have?

Source: Blog.mefeedia

Video Viewing

iPhone & Android lead the way for mobile video viewing, outpacing all other devices – in the US and globally.

Source: Blog.mefeedia

What is the primary device type?

In which market?

Source: Adobe
The **risk** of poor quality

What’s the criteria for your compatibility matrix?

- Specify only a few devices
- Equivalence class analysis?
- Test with the devices “on hand”?
- Purchase devices (with limited life)?
- Buy a service?
- Accept the risk of using emulators and simulators?

Whatever your criteria, choose wisely!

Publish your list to your dev. team and get agreement! This is a big decision.
Testing Mobile Apps: QA Can Make or Break Project’s Success

To ensure the success of an app, QA must be involved in all stages of development.

By Tatyana Mahlaeva, A1QA

Guidelines to determine target devices include:

1. Figuring out what devices the application will support (phones, tablets, other devices – players, navigation);
2. Determining the earliest version of relevant operating systems to be supported.

There are two options: a special restriction, which will be further reflected in the requirements when installing the app, or selecting a lower limit operating system by device popularity among the target audience.

For example, for iOS devices it should be the first iPad with iOS 5.1.1 (if the application has no performance problems on it, there should be no problems in later versions of the device); the iPhone 3GS only runs iOS 4.3.5, so it is the least-productive platform to develop for, but the device is still popular globally.

3. Identifying the most popular models for the target audience.

4. Selecting additional devices with different screen sizes than the most popular models.

Functional requirements

The next step is introducing functional requirements. It is important to define whether the app is browser-based or installable, whether it interacts with a Web site or database, whether it interacts with other apps or social networks, or if it is completely self-contained (does not interact with the Internet and cellular network).

Information collected at this stage should focus on important app functions.

Test documentation

Then it is time to develop test documentation.

Many mobile apps only need high-level documentation, as tasks within the app are typically done with just a few clicks and do not involve complex tasks.

This simplicity means that it is not necessary to create detailed instructions for testing.

Preparatory phase

This phase begins after app development has been commissioned and is complete before the first assembly of a functioning app begins.

Important activities during this phase include:

1. Identifying target devices
2. Introducing functional requirements
3. Developing test documentation
4. Preparing the test environment

Quality Assurance (QA) plays a vital role in the development of mobile applications, but many overlook the critical nature of this piece of the app development process.

To ensure the success of an app, QA must be involved in all stages of development, from creating the concept, analyzing requirements and wishes, creating test specifications, testing early versions of the app, releasing the finished product, to the post-development review process.
For non-mobile apps as well, much depends on optimally partitioning app functionality into blocks while creating test documentation.

This allows connections between different units to be tracked accurately, and requires less time to verify various functions.

User feedback

If the app is an updated version of an existing AppStore or Google Play offering, it may be useful to analyze user reviews and feedback that has been posted on social networks and the app marketplace.

Problems are often found and documented in these environments by end users, and this information will help identify and focus on the app’s most important shortcomings. When new app versions are introduced, users expect to find specific problems have been fixed.

If they are not, it often causes them to react with anger or frustration, especially if they and others have taken the time to document or report errors.

It can be useful to add these end-user concerns to the test checklist as a separate category of problems, and pay special attention to them during testing.

Test environment

Preparing a test environment typically requires the installation of required apps on the mobile device, as well as the installation and configuration of required apps on the QA engineer’s computer.

This might include apps such as the iPhone Configuration Utility and Android SDK.

From here, the process becomes an iterative cycle. It starts after the integration of the first assembly and ends with the completion of product development. It includes testing multiple assemblies at regular intervals (once or twice a week, for example).

Development and testing are carried out simultaneously with each new build of the app.

The main idea of testing is for the QA engineers to put themselves in the shoes of the user, but with a more profound knowledge of the settings and the principles of a particular device and the features of the app being tested.

Here are the characteristics of different types of apps and operating platforms, along with areas that require special attention when it comes to testing for various mobile operating systems.

Apple iOS testing

1. Verify compliance with UI Guidelines from Apple.
2. Apps must be backward-compatible with the OS (a new version of the OS is expected to maintain full functionality of the app).
3. When multitasking, all settings and current progress of the app must be preserved when the app is minimized or when the user switches between apps.
4. You must be able to debug the app and collect logs via a USB cable connection without additional operations being performed on the device.
5. There is no need to reboot the device while testing.

Google’s Android testing

1. The app must support running multiple apps in the background.
2. For debugging via USB, you must first select “Enable USB Debugging” in the device settings, and then connect the device to the computer.
3. When using the hardware “Back” button, the app should be minimized, not closed. Otherwise, the device may require rebooting to run the app again. You can also use an app such as Task Killer as an alternative to rebooting.

Windows mobile testing

1. The app must support multiple apps running in the background.
2. You must be able to close the app through the task manager.
3. You must connect the device to a computer via USB to debug the app.

Check these, regardless of the device/OS

1. Input methods, devices and form factors (number pad, qwerty-keyboard, touch screen, side panel, external devices)
2. Different networks (WiFi, Edge, 3G, 4G, GSM / CDMA), including unstable networks

3. GPS functionality

4. Energy consumption

5. Accelerometer

6. Standards Compliance mobile development (AppleHiG, Android principles)

7. External interrupts (calls, SMS, stability in the event of a shortage of disk space, installation and restoration in the background)

8. Memory cards.


10. Using data on the device (address book, calendar).


13. Interface (animations, icons, portrait/landscape orientation, Retina (for iOS), the visibility of labels, size, ease of use).

14. Time (server time/phone time, time zones).

15. Redirections (Web → app, app → Web).

**Control phase**

The control phase is where you prepare the draft for release after product development ends.

This phase includes detailed and complete testing (during early iterative phases, a full test might not have been performed, or more regression tests may be required) to stabilize the app and uncover minor defects.

This also tests for loss of auxiliary modules (testing in the app can present opportunities to check subsidiary functions such as Log Collector and artificial recharge of game balance, which were added to facilitate or reduce testing time, but that should not appear in the final product).

The approach typically used for testing in this control phase is somewhat like using scaffolding when constructing a building.

We can add features and change settings for the sake of development and testing that will not be in the final version of the app.

For example, an app undergoing control phase testing might be set to provide endless lives or infinite ammunition for the tester.

Once the building is complete, the scaffolding is taken away – much like how settings and features used to test and develop the final version of the app are then taken away before it is made available to the public.

---

**Acceptance testing**

The final phase is acceptance testing. The focus here is to check if the app matches its acceptance criteria (which was defined at the start of the development cycle) or not.

Acceptance testing is performed based on a set of typical test cases and scenarios that are created based on the app requirements. It may specify “no major defects exist in the application,” or “all application labels and functions fully correspond to the rules of the target language.”

This phase of testing is also typically the most important, especially when further support of the app is not provided once it is made public.

Acceptance testing is used to ensure the quality of the app is perfect – or near perfect – and to ensure that the customer is happy with the results of the development process. At this step, the customer decides if the app is excellent enough to accept it as final, and that decision is primarily based on the test results.

AFTER CAREFULLY tailoring these test phases to the app being developed and meticulously carrying them out, you are certain to end up with functional product.

---

**About Tatyana**

Tatyana Mahlaeva is mobile applications QA manager for A1QA, an Austin, TX-based global software testing and quality assurance company. Reach her at t.mahlaeva@a1qa.com.
Agile development has long been all the rage; indeed, in most modern development shops the great agile methodologies are old hat. If you come from a software background like Ruby on Rails, Python, or certain Java niches, you may—until recently—have experienced a small jolt of culture shock when encountering the deep obstacles that agile development practices faced on the iOS platform.

This article outlines our experience using TDD to build HowAboutWe Dating for iPad and iPhone. We’ll describe the stack of tools we use for testing and continuous integration and how we use them to speed the delivery of quality software.

When we made automated tests a requirement for completing a feature or bug-fix ticket, our QA churn dropped radically; our crash instances plummeted; developer confidence improved because we saw the risk of making changes go down; and we could better predict our release readiness without emergency feature cuts.

Our most important tools are Kiwi for unit testing (what Xcode calls “logic tests”) our model and controller logic; KIF for integration testing of user-facing behavior; and Cruise-Control.rb for continuous integration to keep us honest. We also have some key practices that guide our use of these tools.

Tool number one: Kiwi for unit testing

If you’ve ever used RSpec, you’re familiar with the likes of:

describe RingOfPower do
  it ‘takes a name in the constructor’ do
    my_precious = RingOfPower.new('The One Ring')
    my_precious.name.should eq('The One Ring')
  end
end

Allen Ding’s Kiwi is a testing framework for iOS with an RSpec-inspired syntax. It makes slick use of Objective-C blocks and lends itself to readable, contextualized tests. Kiwi is a very complete framework, with many of the levers and knobs you’d reach for regularly in RSpec, like:

- Nestable contexts
- Blocks to call before and after each or all specs in the context
- A rich set of expectations
- Mocks and stubs
- Asynchronous testing

In addition, Kiwi is built on top of OCUnit, which means that it integrates seamlessly with Xcode logic tests and that you can reuse your old OCUnit tests, if you want to do a whole-hog migration to Kiwi. We prefer Kiwi to raw OCUnit, mainly for the elegant syntax—the nested blocks are easy to scan, and the specs are about as smooth to write as you could hope for in Objective-C.
We use kiwi

With our models (most of which are subclassed from NSManagedObject), we test all the code not generated for us. This includes parsing JSON from our API into Objective-C instances; all model-level internal logic, such as converting a user’s gender and orientation into a complementary set of genders and orientations to search for; and important inter-model interactions, as between messages and message threads.

Helpers and categories are another place where Kiwi and TDD shine. We’ve test-driven a set of CGRect helper functions that aid us in smart photo cropping: a photo cache; and a category of time- and sanity-saving methods on NSLayoutConstraint.

We’ve also been driving toward thinning out our view controllers, and a lot of that involves factoring complex code into separate, single-responsibility objects. An example: In our app’s messaging module, we offer an Inbox, a Sent messages folder, and an Archive folder. The three boxes have different behaviors (e.g., you can only archive a thread from the Inbox), and an earlier revision of the messaging view controller had a lot of if-inbox-then-do-X-else-if-sent-do-Y-else, plus a lot of code to make sure the correct message folder was loaded and visible, that Sent and Inbox were properly synced but sorted slightly differently, different empty state strings were displayed for each folder, etc.

Fat controllers and repeated if-else chains are both code smells, and we used Kiwi tests to drive out a single solution to both of them: a separate MessageStore object that handled the juggling of messages and threads. The messaging view controller tells the MessageStore when the user switches modes and queries the MessageStore for the contents of the current folder, appropriate loading and empty strings, and for yes/no answers to behavioral questions like, “Should I expose an Archive button?” The controller is slimmed and the chained if-else-if statements are replaced by data structures that will be easily extensible if we decide to add a fourth folder.

Kiwi specs were integral to building the MessageStore with minimalism and correctness. To give you a taste, here are two specs that cover message-archiving behavior:

beforeEach(^ { messageStore = [[MessageStore alloc] init]]; }); // unini-
tialized afterEach(^ { messageStore = nil; }));

it(“"says whether you can archive messages in the current folder", ^{[theBlock(^ { [messageStore canArchiveThreads]; }) should] raise]; messageStore.mode = Message-

storeInbox;

[[theValue(messageStore.canArchiveThreads) should] equal:theValue(YES)]; mes-

sageStore.mode = MessageStoreSent;

[[theValue(messageStore.canArchiveThreads) should] equal:theValue

(NO)];messageStore.mode = MessageStore-

Archive;

[[theValue(messageStore.canArchiveThreads) should] equal:theValue(NO)];

This first spec tells us that if the MessageStore is uninitial-
ized, it should throw an exception when asked whether archiving behavior should be exposed; otherwise, it should give a boolean answer appropriate to its current mode. If the user requests that a thread be archived, the MessageStore handles that, as defined in the second spec. This spec sets up an Inbox containing message objects (here represented by some dummy NSNumber objects—the MessageStore does not actually care about the type of the objects it is holding) and mimics various requests to pull an object from the Inbox and insert it into a particular place in the archive folder. For modes where the user should not be allowed to archive messages (as defined in the previous spec) or when invalid indices in the Inbox or archive collections are specified, an exception should be thrown; otherwise, the appropriate object should change folders.

The full spec is about 250 LoC, and canonical red-green-refactor TDD drove out an implementation of about 200 LoC. Visible, facile metrics like this scare some people off TDD, because they just see the cost of more code; I see this and know that I’ve written and tested a well-specified, tight bundle of logic, and I took one of the flakier, harder-to-maintain pieces of our app and broke it into solid, loosely-coupled modules that work reliably. The test-driven MessageStore and the concomitant simplification of the messaging view controller purged a whole class of hard-to-diagnose bugs from our issue tracker. When it comes to stabilizing the most-used parts of your app, 250 lines of straightforward, declarative test code is cheap.

One limitation of Kiwi is that it’s not so good for testing UIKit-derived classes or anything that touches them. This is actually a limitation of Xcode logic tests—they don’t fire up a UIApplication instance and don’t play nicely with UIKit. To test elements of the project that can’t be separated from the UI, we use automated integration tests.

Tool number two: KIF for integration testing

Kiwi helps us keep our lovely abstractions lovely, but what of the user-facing parts of the app? And how do we know
that all the pieces work together? For integration tests of user-visible behavior, we use Square's KIF library. It uses the iOS accessibility framework to simulate user interaction with the app.

Testing every facet of the app by automatically driving the app through every possible user action would be insanely costly, and it would rapidly get to the point of diminishing returns. In addition, the fact that the tests run in the simulator by faking user behavior means that the tests run at human-ish speeds, not as fast as the CPU can run through them. Integration testing in the sim requires a number of additional practices and judgment calls to make it sane and valuable.

First, the tests have to be decoupled from the outside world. We've used method swizzling to stub out all our network calls and give back dynamically generated, predictable data to drive the app.

Each of those stub methods returns a simulated API response, based on responses recorded in an actual session. This keeps us from having to stand up a web server to test the client app, and puts the inputs to the tests entirely under our control. We frequently have the stubs respond to different inputs by returning different data or exceptions so that we can simulate behaviors like paging data, network failure modes, etc.

Second, the integration tests have to be decoupled from each other. If you run 50 integration tests one after the other and make a change to the fourth test that alters the app's state in a persistent way, you risk breaking the next 46 tests. To mitigate this risk, we bundle the tests into related modules and run steps to log the test user out and clear the database between modules. Where it's important that an intermodule dependency be tested (e.g., a message sent from a user profile should show up in the logged-in user's Sent folder), we write a test for it, but otherwise we try to keep the KIF test scenarios limited to one screen or a small set of related screens, each testing a limited but meaningful set of user behavior.

Third, a lot of judgment needs to be exercised in what gets tested. It is impossible to test every possible user input, but you want to hit all your major error states as well as at least one valid input. It is impossible to test every path through the code, but you want to reasonably simulate the things a user is likely to do and spend a little more effort on the parts of the code that matter most to the user experience.

If you've read about the pros and cons of integration testing, you've heard some version of the issues I've described above (coupling-induced fragility, impossibility of total coverage, etc.) as reasons why integration tests are a bad thing. Certainly, we've found them costlier to write and maintain than unit tests. The way we've applied them, though, has given us far too much value to even consider discarding them: Where we've used unit tests to write quality code, the integration tests have been invaluable in helping us maintain it. They form our "regression firewall", and if the test board is green, then the developers, product managers, and QA all know that none of the big stuff has gone wrong. Bugs still get through, but there will tend to be stuff around the edges.

In the rare case something big gets through, we add it to the suite. It happened recently that a release made it into the wild with a 100% reproducible crash when the subscription screen was reached by a certain path.

From this short repro, you can get the feel of KIF tests: Check the state of the screen (mostly via the accessibilityLabel and accessibilityValue properties of screen elements), interact with screen elements, check the state, interact, and so on.

Our crash occurred on the last line of the repro above. We added the steps to the suite, ran it, watched it crash; then we fixed the bug, ran the test, watched it pass; then we ran the rest of the upgrade-related test module to make sure we didn't break anything. This is a much longer process than just diving in and fixing the bug as soon as you've diagnosed it, but it boosts confidence among everyone who builds or inspects the app in two ways: They trust that we haven't broken existing functionality (thanks to existing tests), and they trust that the bug being addressed in the new test won't return.

One KIF trick that comes in handy is defining our own one-off steps. Any parameterizable process that gets used more than once gets a factory method in our own category on the KIFTestStep class, but sometimes the code is made more comprehensible when a task that only happens once is defined inline with a block:

The cloud inside the silver lining

There are two major downsides to KIF. The first is the syntax—all those addStep: calls are actually building the test suite, not running it, and there's no clean way to set a break point at a particular instance of a step (unless you've defined the step yourself). We tolerate it because KIF is the best thing we've found for this type of testing. We feel it has yet to achieve true maturity, and we've extended it quite a bit for our own purposes, but it largely does what it says it will and has served as one of the pillars of our testing strategy.
The other pain point is the run time of the tests. Our full suite takes nearly 15 minutes to run, which makes it useless for fast-iterating styles of TDD/BDD. Our usual method of handling this goes something like:

- Run only the test scenario related to the feature or bug being addressed.
- Once the central test passes, run related test modules to make sure nothing was broken.
- In cases where confidence is low or a change is far-reaching, run the full suite at the developer bench. Otherwise, merge your change (after review: See below) and be ready to jump back on it if the CI board (again: See below) goes red.

This is another one of those times when developer judgment plays a key role. Running the full suite is a major break in your rhythm, especially when you're making (what feels like) a small change. The value of moving on to the next thing needs to be weighed against the risk inherent in the change being made, and the evaluation of that risk depends on one's intimacy with the code and seasoning as a programmer. In the section on practices below, I go into how we buttress individual judgment with the collective wisdom of the team.

**Tool number three: CruiseControl.rb for continuous integration**

CruiseControl.rb is a darling of the Rails community, but it's not just for Rails apps. It's quick to set up—including on a Mac, which is required to run our Xcode-based tests—and can run and extract results from arbitrary build-and-test scripts. CC.rb handles polling our GitHub repositories for changes to projects; our custom scripts do the rest, and CC.rb reports red/green for each project by checking standard Unix return values from the scripts.

First the common library shared by our iPhone and iPad projects gets built, and its Kiwi tests are run:

```
#!/bin/bash
git submodule init
git submodule update
xcodebuild -scheme HAWCommonTestsCL -sdk iphonesimulator \
-TEST_AFTER_BUILD=YES -arch i386 clean build | grep "BUILD SUCCEEDED"
```

It's that simple; the result of the final grep for Kiwi's success message is the success or failure that CruiseControl.rb reports.

Running the KIF tests for the iPhone and iPad projects is a bit more of a production. We have to do extra steps to build the common library prior to the main project, and we have to use Waxsim (we prefer Jonathan Penn's excellent fork) to run the simulator from the command line and capture console output, and sift through that console output for success or failure messages. The end result is the same, though: The return value of the script is reported as the test outcome by CC.rb.

Continuous Integration, of course, is only as good as the speed with which it gives you feedback. We have our CI set up on a Mac Mini running Screen Sharing enabled, and with the CC.rb dashboard exposed on a convenient port. CruiseControl.rb can be set up to send email, but our inboxes are plenty cluttered already.

We get the results through two main channels:
- On the developer desktop, CCMenu keeps the latest test results an eye flick away:
- To broadcast status to management and the wider team, we keep CruiseControl.rb's web dashboard on a large screen mounted on a wall overlooking the developers' corner of the shop:

The public display of results is an important driver of good testing habits. As soon as a team gets used to meeting high expectations for test reliability, a red test suite on display for all to see becomes a distracting irritant. While we don't generally suggest introducing distracting irritants into technical workflows, in this case the irritation is confluent with the worthy goal of maintaining a reliable test suite that consistently inspires confidence in everyone who builds or depends on the software.

Our best practices: The rules to guide the tools

Tools are only valuable when they are used well. We surround our tools with processes to get the most out of them, and tune those processes as we go in response to real-world feedback. Here are three simple rules that guide our use of the tools described above:

**A: TDD**

The core rhythm of TDD (and its cousins like BDD) is often described as "red-green-refactor":
- Red: You write a test describing the behavior you want, run it, and watch it fail.
- Green: You nudge your code into a state where the test passes.
- Refactor: You inspect your code (and your tests) for duplication and other issues, and remove them. The tests keep you from breaking anything.

The last step is probably the least-understood, most-skipped one in the process. People carry a lot of weird,
As well as applying a critical eye to the code. With both practices (pairing and pull requests), both parties are expected to be active collaborators; no one should be rubber-stamping someone else's decisions.

**Conclusion**

iOS culture, even in many large organizations with skilled engineers, is behind on up-to-date testing practices. Aggressive mobile strategies up against lengthy App Store release cycles and manual user app updates create pressure to jettison code and best practices that might be seen as "extras."

It's ironic that iOS development--the catalyst of the consumer web explosion of the past few years--has been a reluctant late comer to TDD, perhaps the most cherished methodology of the agile web development culture that is building the consumer Internet.

Over the platform's short history, agile methodology and TDD have been at odds in iOS development culture. The agile desire for speed has taken precedence over other concerns due to a past dearth of high-powered automated testing frameworks, and the results have often been high crash rates, long QA cycles, and a whole series of tribulations that the modern developer associates with the antiquities of waterfall development.

Our experience building HowAboutWe Dating for iPhone and iPad has shown that TDD and CI on iOS are well worth the effort. The tools are young but rapidly maturing. It is possible! Our move to a genuine culture of TDD on iOS has transformed the quality of our software and how quickly and predictably we can deliver it. So we're believers that any organization not already employing these practices should dive in and measure the results for themselves.

HowAboutWe is the modern love company and has launched a series of products designed to help people fall in love and stay in love.

**About the authors**

Aaron Schildkrout is co-founder and co-CEO of HowAboutWe, where he runs product.

Brad Heintz is the Lead iOS Developer at HowAboutWe. When he's not bringing people better love through TDD, he's tinkering, painting, or playing the Chapman Stick.

James Paolantonio is a mobile engineer at HowAboutWe, specializing in iOS applications. He has been developing mobile apps since the launch of the first iPhone SDK. Besides coding, James enjoys watching sports, going to the beach, and scuba diving.

This article was originally published in *Fast Company*.
Glossary: Mobile Testing

Some Mobile Device Input Methods

**Swipe/Swype:**
An action taken on a device’s screen that involves translational movement of touch points. A swipe recognizer makes an instantaneous decision as to whether the user's touches moved linearly in the required direction.

**Apple/iOS:**
“Touch, type, tell”

**Google/Android:**
“Type, write or speak”

**Gesture:**
Multi-touch gestures are predefined motions used to interact with multi-touch devices. An increasing number of products like smartphones, tablets, laptops or desktop computers features functions that are triggered by multi-touch gestures. Some typical gesture-function pairs are: tap, double tap, long press, scroll, pan, flick, pinch and rotate.

**QWERTY:**
QWERTY is the most common modern-day keyboard layout. The name comes from the first six keys appearing on the top left letter row of the keyboard and read from left to right: Q-W-E-R-T-Y. QWERTY is often used to describe keyboard character and number entry as well as keyboard navigation.

**Sensors**

**Ambient light sensor:**
An ambient light sensor basically adjusts the display brightness which in turn saves battery power in Smartphone; it saves power by adjusting the brightness of the display based on how much ambient light is present.

**Accelerometer:**
The accelerometer allows the device of Smartphone to detect the orientation of the device and adapts the content to suit the new orientation.

**Gyroscope:**
Gyrosopes are used in Smartphones and tablet PCs for finding the position and orientation of devices. Combining a gyroscope with an accelerometer allows the device to sense motion on six axes – left, right, up, down, forward and backward, as well as roll, pitch and yaw rotations – allowing for more accurate motion sensing abilities.

**Proximity sensor:**
The proximity sensor in Smartphone senses how close the phone is to the user's cheek/face, so that it can pause whatever activity it is in the middle of (playing music or browsing the web, for example) so the user can take a phone call. When the phone is removed from the ear after the call, the phone resumes its previous activity.

**Magnetometer:**
A magnetometer is used to measure the strength and/or direction of the magnetic field in the vicinity of the device. Sometimes certain devices or radio signals can interfere with the magnetometer requiring users to either move away from the interference or re-calibrate by moving the device in a figure 8 motion.

**NFC:**
NFC a.k.a Near Field Communication, is a short-range wireless technology used as a method of transferring data over short distances between two compatible devices.

Sources: Wikipedia, techopedia, mobileburn, knowyourmobile, mobiledeviceinsight
Vietnam View

Banh Mi - From French Import to Vietnamese Staple

While Vietnamese coffee seems to garner the most attention, there's another French import that is becoming increasingly popular worldwide – banh mi.

Banh mi literally means “bread” though it is often extrapolated to include its sandwich form which is sold on seemingly every corner and in every alleyway in Vietnam’s cities.

Bread was introduced by the French in the 18th century during their brutal occupation of Vietnam. Soon after, the Vietnamese developed a taste for the French baguette and began to pair it with local ingredients including cilantro, fish sauce, and pickled carrots.

Like most national Vietnamese food, each region puts its own spin on the dish, based on local tastes and ingredients.

The classic version, bánh mì thịt nguội, sometimes known as bánh mì đặc biệt or “special combo”, is made with various Vietnamese cold cuts, such as sliced pork or pork bellies, chả lụa, and head cheese, along with the liver pâté and vegetables.

To the delight of early risers, many shops offer a breakfast banh mi - eggs fried sunny-side-up with onions, sprinkled with soy sauce or Maggi sauce, and eaten with a fresh (and sometimes buttered) baguette.

Where the Vietnamese go, the banh mi follows. With one of the largest diaspora communities in the world, banh mi has long been a staple in overseas Vietnamese (Viet Kieu) communities. This group is also known for recreating their close-knit communities in the adopted homelands, cuisine included.

From Paris to Houston, one can find authentic banh mi, albeit with slightly higher quality ingredients than their Vietnamese counterparts. This can be seen in the price difference - in Saigon, the sandwich will only set you back VND15,000 or $.75 whereas the price can rise well above $5.00 in other countries.

After a generation in western society, trendy restaurateurs, have updated the classic bani mi to the point where it loses much of its Vietnamese qualities.

Even as Vietnam adds brands like Starbucks and McDonalds, banh mi is here to stay.
Speed and Collaboration
The winning combination for large-scale test automation

TestArchitect
Go to market with confidence

TestArchitect is the modern module-based keyword-driven software test automation tool that enables teams to create, maintain and execute large-scale test automation with groundbreaking speed.

Accelerate test creation and maintenance. TestArchitect’s module-based, keyword-driven authoring platform and library of preprogrammed actions enables team members to create and update tests without programming scripts, making your entire team more productive.

Scale resources and collaborate with ease. TestArchitect makes it easy to scale up project teams as testing demands require. By combining test design and automation into one tightly-integrated process, team members with diverse skill sets can collaborate easily.

Find out more at www.testarchitect.com
United States  
2015 Pioneer Ct., Suite B  
San Mateo, CA 94403  
Tel +1 650 572 1400  
Fax +1 650 572 2822

Viet Nam, Da Nang  
7th floor, Dana Book Building  
76-78 Bach Dang  
Hai Chau District  
Tel: +84 511 3655 333  
Fax: +84 511 3655 336

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