BY Julian Harty & Marc van't Veer

Testing

There are many parallel worlds for mobile apps on mobile devices. Meanwhile mobile devices evolve incredibly rapidly compared to most worldly goods. Testing mobile apps needs to keep pace even as the pace of change continues to accelerate.

The fate of mobile apps hangs in the balance where users, like crowds in amphitheatres back in Roman times, often make the ultimate decision of whether an app lives to fight another day, or dies. And similarly, unremarkable apps are likely to languish as a statistic in the App Store, negating much of the hard work involved in conceiving, nurturing and launching it. Furthermore, the stigma of a poor rating has a long half-life which is hard to recover from.

Testing might be seen as an impediment but failures in your app can be all too public. And recovering your credibility is hard when your app has a poor score in the app store. So you could wait for users to decide the fate, testing your mobile apps can adjust the balance in your favour. You have the opportunity to help equip you and your testing team so they can help test your app more effectively.

Beware of Specifics

Platforms, networks, devices, and even firmware, are all specific. Any could cause problems for your applications. There are several ways to identify the effects of specifics, for instance, a tester may notice differences in the performance of the app and the behaviour of the UI during interactive testing with different devices. QuizUp used automated tests which helped them find five significant issues in their Android app triggered by differences in the devices, and one bug specific to Android 4.0.4. The automated tests ran across 30 devices in 30

minutes which made multi-device testing practical and useful, rather than spending 60 hours trying to do manual testing of the app on 30 devices[1]. You need to know about these specifics to be able to decide whether to address undesirable differences by modifying the app before it is launched.

Conversely, Mobile Analytics can help identify differences in various aspects including performance and power consumption when the app is being used by many users on the vast variety of their devices. Some compelling examples of differences in behaviour and on ways issues were addressed in a paper published by computer scientists from the University of Wisconsin[2]. A book is also available from HP Enterprises on the confluence between mobile analytics and testing for mobile apps, details are available at *themobileanalyticsplaybook.com*. (co-written by Julian Harty, one of the authors of this chapter.)

This chapter covers the general topics; testing for specific platforms is covered in the relevant chapter.

Testing Needs Time - You Need a Strategy

The strategy defines how much test time is spent to the different parts of the mobile app and during the different development phases. There are tradeoffs on how best to spend whatever time you have. For instance, testing features in more detail versus testing on a wider variety of devices versus testing various quality aspects including performance, usability

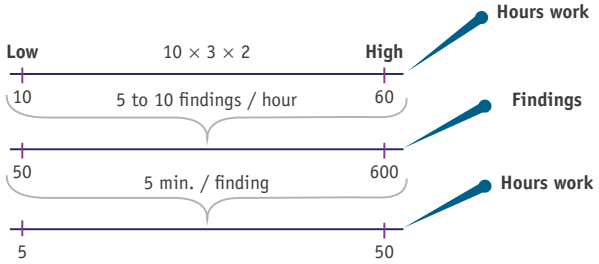
1 blog.testmunk.com/quizup-mastering-android-device-fragmentation-automated-testing/

2 "Capturing Mobile Experience in the Wild: A Tale of Two Apps", available as a download at static.googleusercontent.com/media/research.google.com/en//pubs/archive/41590.pdf

and security. The conditions a mobile app need to operate are vast and to factor in these conditions into the testing is challenging. There may be more productive ways to obtain some information, for instance by using feedback from end-users and from mobile analytics. However the risks of deferring data gathering (versus testing internally) need to be actively considered. An effective test strategy aims to balance both approaches. With the risk analysis, the quality perspectives and the available time the test plan can be created.

Calculation Example

Let us assume your app has ten epics (also called user stories in this context) and every epic needs to be tested using between one and three perspectives (functionality, usability, user scenarios) at a rate of one test per hour. If you test each epic with one perspective and one test case it takes ten hours to test the app. On the other hand, if you test every epic using all three perspectives with two test cases testing takes 60 hours. Typically, testing finds defects and other work worth reporting, which takes more time. You then choose to spend time addressing some of the findings, for instance, to fix a defect or otherwise improve the app. Let us assume that every executed test case results in 5 to 10 findings. The time needed to process the defects lies between 5 and 50 hours work. So in the most positive scenario it takes 15 hours to test the app (only functionally) and in the more complex scenario, it can take up to 110 hours to test the app using three perspectives.



You may also need to factor in much more time to test the app on a variety of compatible devices, particularly for web apps and for Android native apps.

Continuous Testing

Continuous delivery needs continuous testing. Viable apps need to be updated on an ongoing basis in production. Updates may include fixes for new platform versions or device models, new functionality and other improvements. Therefore, testing is not a one-off task; high quality apps befit ongoing, optimised testing, including testing in production. Production testing includes testing engagement and validation as well as early detection of potential problems before they mushroom.

Manage your Testing Time

Testing as you have discovered can take many hours, far more than you may want to do, particularly if you are close to a deadline such as a release date. There are various ways you can manage time spent in testing, in parallel testing can be made more interesting, rewarding, and more productive.

- **Reduce setup time**: Find ways to deploy apps quickly and efficiently. Implement mechanisms to provide the appropriate test data and configuration on both the mobile device and the relevant servers. Aim to have devices and systems 'ready to test'.
- **Reduce time needed for reporting & bug analysis**: Data, screenshots, and even video, can help make bugs easier and faster to investigate. Data can include logs, system configurations, network traffic, and runtime information. Commercial tools, such as HP Sprinter[3] can record actions and screenshots to reduce the time and effort needed to report and reproduce problems.
- **Risk Analysis**: You can use the risk analysis to decide how and when to allocate testing effort. Risks are hard to determine accurately by the tester or developer alone; a joint effort from all the stakeholders of the mobile app can help to improve the risk analysis. Sometimes, the mobile app tester is the facilitator in getting the product risk analysis in place.
- **Scaling Testing**: Increasing the throughput of testing by scaling it, for instance using test automation, cloud-based test systems, and more humans involved in the testing can help increase the volume, and potentially the quality, of the testing. Using static analysis tools to review code and other artefacts can also help the team to find and fix problems before the app is released.

3 hp.com/go/sprinter

Involve End-Users in your Testing

Development teams need a mirror to develop a useful mobile app. Early user feedback can provide that mirror. You do not need many end-users to have good feedback[4]. Bigger value is gained with early involvement, multiple users, regular sessions, and multiple smaller tests. Testers can guide and facilitate the end-user testing, for instance, by preparing the tests, processing log files and analysing results. They can also retest fixes to the app.

Whenever others are involved in testing an app, they need ways to access and use the app. Web apps can be hosted online, perhaps protected using: passwords, hard-to-guess URLs, and other techniques. Installable apps need at least one way to be installed, for instance using a corporate app store or specialist deployment services. Possible sources of end-users can be Crowdsourcing[5].

When the app is closer to being production-ready, users can test the more mature version of the mobile app in Alpha & Beta tests phases. A development team or organisation can offer an online community to give end-users early access to new releases, give loyalty points, ratings. This community should be a friendly ecosystem to receive feedback before the mobile app is released into the app store.

Effective Testing Practices

Testing, like other competencies, can be improved by applying various techniques and practices. Some of these need to be applied when developing your mobile app, such as testability,

4 nngroup.com/articles/why-you-only-need-to-test-with-5-users

5 service providers include www.applause.com, PassBrains.com, and TestBirds.de

others apply when creating your tests, and others still when you perform your testing. Testdroid offers a good checklist[6] on getting the right testing expertise into your team.

Mnemonics Summarizing Testing Heuristics

Heuristics are fallible guidelines, or rules-of-thumb, that tend to be useful. Several have been created specifically to help test mobile apps and some use mnemonics to help you consider particular aspects of software. Each letter is the initial letter of a word representing a key word.

— **I SLICED UP FUN**[7]**: I**nput (Test the application changing its orientation (horizontal/vertical) and trying out all the inputs including keyboard, gestures etc.), **S**tore (Use appstore guidelines as a source for testing ideas), **L**ocation (Test on the move and check for localisation issues), **I**nteraction/Interruption (See how your app interacts with other programs, particularly built-in, native apps), **C**ommunication (Observe your app's behaviour when receiving calls, e-mails, etc.), **E**rgonomics (Search for problem areas in interaction, e.g. small fonts), **D**ata (Test handling of special characters, different languages, external media feeds, large files of different formats, notifications), **U**sability (Look for any user actions that are awkward, confusing, or slow), **P**latform (Test on different OS versions), **F**unction (Verify that all features are implemented and that they work the way they are supposed to), **U**ser Scenarios (Create testing scenarios for concrete types of users), **N**etwork (Test under different and changing network conditions)

6 testdroid.com/testdroid/6336/get-the-superb-expertise-in-your-testingqa-team

7 kohl.ca/articles/ISLICEDUPFUN.pdf

— **COP FLUNG GUN**[8] summarizes similar aspects under **C**ommunication, **O**rientation, **P**latform, **F**unction, **L**ocation, **U**ser Scenarios, **N**etwork, **G**estures, **G**uidelines, **U**pdates, **N**otifications.

Implementing Testability

Start designing and implementing ways to test your app during its development already; this applies especially for automated testing. For example, using techniques such as Dependency Injection in your code enables you to replace real servers (slow and flaky) with mock servers (controllable and fast). Use unique, clear identifiers for key UI elements. If you keep identifiers unchanged your automated tests require less maintenance.

Separate your code into testable modules. Several years ago, when mobile devices and software tools were very limited, developers chose to 'optimise' their mobile code into monolithic blobs of code, however the current devices and mobile platforms mean this form of 'optimisation' is unnecessary and possibly counterproductive. These two topics are both covered in a useful article on the Google Testing Blog, Android UI Automated Testing[9].

Provide ways to query the state of the application, possibly through a custom debug interface. You, or your testers, might otherwise spend lots of time trying to fathom out what the problems are when the application does not work as hoped.

Tours for Exploratory Testing

A tour is a type of exploratory testing, a way to more structure the exploratory test sessions. Tours help you focus your testing, Cem Kaner describes a tour as "... a directed search through the

8 moolya.com/blogs/2014/05/34/COP-FLUNG-GUN-MODEL

9 googletesting.blogspot.co.uk/2015/03/android-ui-automated-testing.html

program. Find all the capabilities. Find all the claims about the product. Find all the variables. Find all the intended benefits. Find all the ways to get from A to B. Find all the X. Or maybe not ALL, but find a bunch..."[10]. With the combination of different tours in different perspectives (see the I SLICED UP FUN heuristics) coverage and test depth can be chosen.

Examples of Tours[11] include:

- **Configuration tour**: Attempt to find all the ways you can change settings in the product in a way that the application retains those settings.
- **Feature tour**: Move through the application and get familiar with all the controls and features you come across.
- **Structure tour**: Find everything you can about what comprises the physical product (code, interfaces, hardware, files, etc.).
- **Variability tour**: Look for things you can change in the application - and then you try to change them.

Personas

Personas can be used to reflect various users of software. They may be designed to reflect, or model, a specific individual or a set of key criteria for a group of users. Regardless of how they are created each persona is singular, not a group of people. Personas can be used to have a clear picture of various end-users to include so that representative tests are executed for those end-user. Various research material are available at *personas.dk*.

10 kaner.com/?p=96; also see developsense.com/blog/2009/04/of-testing-tours-and-dashboards/

11 from michaeldkelly.com/blog/2005/9/20/touring-heuristic.html

Testing on Various Devices

Some bugs are universal and can be discovered on any mobile device. Others, and there are plenty of them, are exposed on a subset of devices, and some belong to specific devices. An example of device specific problems with Android on Samsung is *verybadalloc.com/android/2015/12/19/special-place-for-samsung-in-android-hell/*. All this means we need devices to test on and to test on various devices.

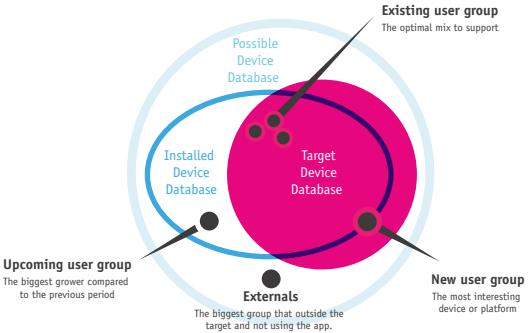
Physical and Virtual Devices

Physical devices are real, you can hold them in your hands. Virtual devices run as software, inside another computer. Both are useful hosts for testing mobile apps.

Virtual devices are generally free and immediately available to install and use. Some platforms, including Android, allows you to create custom devices, for instance with a new screen resolution, which you can use for testing your apps even before suitable hardware is available. They can provide rough-and-ready testing of your applications. Key differences include: performance, security, and how we interact with them compared to physical devices. These differences may affect the validity of some test results. Beside the android platform virtual devices you can use *GenyMotion.com*, a faster and more capable Android emulator, for instance, to control sensor values.

The set of test devices to use needs to be reviewed on an ongoing basis as the app and the ecosystem evolve. Also, you may identify new devices, that your app currently does not support, during your reviews. The following figure illustrates these concepts.

Ultimately your software needs to run on real, physical, phones, as used by your intended users. The performance characteristics of various phone models vary tremendously from



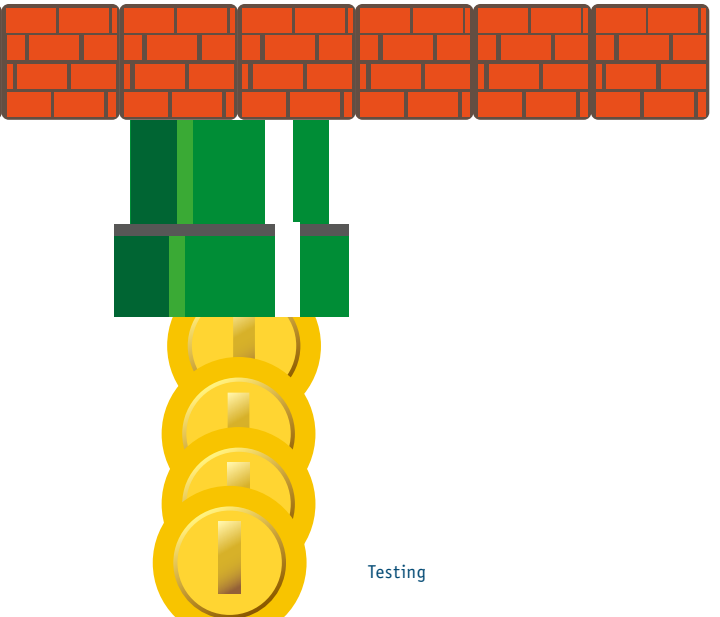
each other, and from virtual devices on your computer. So: buy, beg, borrow phones to test on. A good start is to pick a mix of popular, new, and models that include specific characteristics or features such as: touch screen, physical keyboard, screen resolution, networking chipset, et cetera. Try your software on at least one low-end or old device as you want users with these devices to be happy too.

Here are some examples of areas to test on physical devices:

- **Navigating the UI:** for instance, can users use your application with one hand? Effects of different lighting conditions: the experience of the user interface can differ in real sunlight when you are out and about. It is a mobile device – most users will be on the move. Rotate the screen and make sure the app is equally attractive and functional.
- **Location:** if you use location information within your app: move – both quickly and slowly. Go to locations with patchy network and GPS coverage to see how your app behaves.
- **Multimedia:** support for audio, video playback and recording facilities can differ dramatically between devices and their respective emulators.

— **Internet connectivity:** establishing an internet connection can take an incredible amount of time. Connection delay and bandwidth depend on the network, its current strength and the number of simultaneous connections. Test the effects of intermittent connectivity and how the app responds.

As mentioned before, crowdtesting can also help to cover a wide range of real devices, but you should never trust on external peoples' observations alone.



Remote Devices

If you do not have physical devices at hand or if you need to test your application on other networks, especially abroad and for other locales, then one of the 'remote device services' might help you. They can help extend the breadth and depth of your testing at little or no cost.

Several manufacturers provide this service free-of-charge for a subset of their phone models to registered software developers. Samsung[12] (for Android and Tizen) provide restricted but free daily access.

You can also use commercial services of companies such as *SauceLabs.com*, *testdroid.com*, *PerfectoMobile.com* or *DeviceAnywhere.com* for similar testing across a range of devices and platforms. Some manufacturers brand and promote these services however you often have to pay for them after a short trial period. Some of the commercial services provide APIs to enable you to create automated tests.

You can even create a private repository of remote devices, e.g. by hosting them in remote offices and locations.

Beware of privacy and confidentiality when using shared devices.

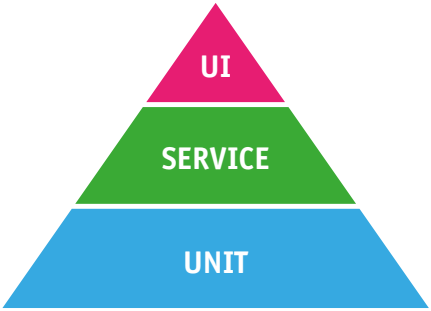
Test Automation

Automated tests can help you maintain and improve your velocity, your speed of delivering features, by providing early feedback of problems. To do so, they need to be well-designed and implemented. Good automated tests mimic good software development practices, for instance using Design Patterns[13], modularity, performing code reviews, et cetera. To automate

12 developer.samsung.com/remotetestlab/rtlDeviceList.action

13 en.wikipedia.org/wiki/Design_Patterns

scripting and coding skills are needed. The level of skills is dependent on the chosen tool. Test automation tools provided as part of the development SDK are worth considering. They are generally free, inherently available for the particular platform, and are supported by massive companies. Test automation can be performed at different levels, see the automation pyramid figure below. It is a strategic choice what should be automated in the unit tests, what on the service or API level and what scenarios on the UI level of the application. The pyramid represents trust that is buildup from the unit test to the higher levels. Multiple test levels are needed to prove that the app works.



GUI Level Test Automation

The first level of automation are the tests that interact with the app via the Graphical User Interface (GUI). It is one of the elixirs of the testing industry, many have tried but few have succeeded. One of the main reasons why GUI test automation is so challenging is that the User Interface is subject to significant changes which may break the way automated tests interact with the app.

For the tests to be effective in the longer term, and as the app changes, developers need to design, implement and

support the labels and other hooks used by the automated GUI tests. Both Apple, with UI Automation[14], and more recently Android[15] use the Accessibility label assigned to UI elements as the de-facto interface for UI automation.

Some commercial companies have open sourced their tools, e.g. SauceLabs' appium[16] and Xamarin's Calabash[17]. These tools aim to provide cross-platform support, particularly for Android and iOS. Other successful open source frameworks include Robotium[18] which now offers a commercial product - a test recorder. Several other tools have effectively disappeared, perhaps the industry is now maturing where the only the stronger offerings survive?

Service Level Test Automation

There is a lot of business logic implemented inside an API. Changes in this logic or in the backend system can be monitored with automated API tests. The focus of the test can be functional-regression but also reliability, performance and security. For functional regression testing a tool like Postman is useful[19].

Several tools can help with API testing. They include Fiddler

14 developer.apple.com/library/tvos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UIAutomation.html

15 developer.android.com/tools/testing/testing_ui.html

16 github.com/appium/appium

17 github.com/calabash

18 github.com/robotiumtech/robotium

19 blog.getpostman.com/2014/03/07/writing-automated-tests-for-apis-using-postman/

by Telerik[20] and Charles[21]. Both enable you to view and modify network traffic between your mobile device and the network.

Unit Level Test Automation

Unit testing involves writing automated tests that test small chunks of code, typically only a few lines of source code. Generally, they should be written by the same developer who writes the source code for the app as they reflect how those individual chunks are expected to behave. Unit tests have a long pedigree in software development, where JUnit[22] has spawned similar frameworks for virtually all of the programming languages used to develop mobile apps.

BDD Test Automation

BDD is a Behavior-Driven Development approach with the Test Driven Development family[23]. The behaviour is described in formatted text files that can be run as automated tests. The format of the tests are intended to be readable and understandable by anyone involved with the software project. They can be written in virtually any human language, for instance Japanese[24], and they use a consistent, simple structure with statements such as **Given, When, Then** to structure the test scripts.

The primary BDD framework to test mobile apps is Calabash for Android and iOS[25]. Various others have not been developed

20 telerik.com/fiddler

21 Proxycharlesproxy.com/documentation/getting-started/

22 en.wikipedia.org/wiki/JUnit

23 en.wikipedia.org/wiki/Behavior-driven_development

24 github.com/cucumber/cucumber/tree/master/examples/i18n/ja

25 github.com/calabash

or maintained in the last year and can be considered defunct for all but the most persistent developers. General purpose BDD frameworks may still be relevant when they are integrated with frameworks, such as appium, that use the WebDriver wire protocol (a W3 standard)[26].

Automation can also help the manual testing, for instance, to replace manual, error-prone steps when testing, or to reduce the time and effort needed, for instance, to automate a collection of screenshots. Developers can help testers to be more efficient by providing automated tools, e.g. app deployment via ADB[27].

Testing Through The Five Phases of an App's Lifecycle

Software is developed in phases, which are called steps in the life cycle. A mobile app tester can be part of the development team, but can also be responsible to facilitate the user experience tests in production. Depending on which phase(s) you are involved in the life cycle, there are different tasks to be performed. For example, when joining a development team, the task can be the analysis of the error in the log files on a device. When joining a beta test phase, a task can be the analysis of usability tests results like recording movies. The complete lifecycle of a mobile app fits into 5 phases: implementation, verification, launch, engagement and validation.

Improvement Cycles

Testing applies to each phase. Some of the decisions made for earlier stages can affect your testing in later stages. For instance, if you decide you want automated system tests in

26 w3.org/TR/webdriver/

27 thefriendlytester.co.uk/2015/11/deploying-to-multiple-android-devices.html

the first phase they will be easier to implement in subsequent phases. The five phases might suggest that they follow one after the other and form a logical flow of water down the river. This is not the case. Every step in the different phases provides the possibility to learn and improve. When testing the team learns both how good the mobile app product is and also about areas for improvement in how the app is produced. Mobile app development is a challenging complex, dynamic activity that does not go perfectly the first time, therefore, teams should incorporate an improvement cycle so they can learn and actively improve what they do.

Phase 1: Implementation

This includes design, code, unit tests, and build tasks. Traditionally testers are not involved in these tasks; however good testing here can materially improve the quality and success of the app by helping us to make sure the implementation is done well.

In terms of testing, you should decide the following questions:

- Do you use test-driven development (TDD)?
- Help review designs on what are the main, alternative and negative user flows
- Which test data do you use to validate the user flows?
- Will you have automated system tests? If so, how will you facilitate these automated system tests? For instance by adding suitable labels to key objects in the UI.
- How will you validate your apps? For instance, through the use of Mobile Analytics? Crash reporting? Feedback from users?

Question the design. You want to make sure it fulfills the

intended purposes; you also want to avoid making serious mistakes. Phillip Armour's paper on five orders of ignorance[28] is a great resource to help structure your approach.

Phase 2: Verification

Review your unit, internal installation, and system tests and assess their potency: Are they really useful and trustworthy? Note: they should also be reviewed as part of the implementation phase, however, this is a good time to address material shortcomings before the development is considered 'complete' for the current code base.

For apps that need installing, you need ways to deploy them to specific devices for pre-release testing. Some platforms (including Android, iOS and Windows) need phones to be configured so development apps can be installed. Based on your test strategy you can decide on which phones, platforms, versions, resolutions are in scope of testing and support.

System tests are often performed interactively, by testers. You also want to consider how to make sure the app meets:

- Usability, user experience and aesthetics requirements
- Performance, particularly as perceived by end users[29]
- Internationalisation and localisation testing

28 www-plan.cs.colorado.edu/diwan/3308-07/p17-armour.pdf

29 A relevant performance testing tool is ARO (Application Resource Optimizer) by AT&T: developer.att.com/application-resource-optimiser, an open source project at github.com/attdevsupport/ARO

Phase 3: Launch

For those of you who have yet to work with major app stores be prepared for a challenging experience where most aspects are outside your control, including the timescales for approval of your app. Also, on some app stores, you are unable to revert a new release. So if your current release has major flaws you have to create a new release that fixes the flaws, then wait until it has been approved by the app store, before your users can receive a working version of your app.

Given these constraints, it is worth extending your testing to include pre-publication checks and beta tests of the app such as whether it is suitable for the set of targeted devices and end-users. The providers of the main platforms now publish guidelines to help you test your app will meet their submission criteria. These guidelines may help you even if you target other app stores. The guideline can be used as a checklist during the implementation phase.

Apple	developer.apple.com/appstore/resources/approval/guidelines.html
Android	developer.android.com/distribute/googleplay/publish/preparing.html#core-app-quality
Windows Phone	msdn.microsoft.com/en-us/library/windowsphone/develop/hh394032
BlackBerry	developer.blackberry.com/devzone/appworld/tips_for_app_approval.html

Phase 4: Engagement

This includes search, trust, download and installation. Once your app is publicly available users need to find, trust, download and install it. You can test each aspect of this phase in before and in production. Try searching for your app on the relevant app store, and in mainstream search engines. On how many different ways can it be found by your target users? What about users outside

the target groups - do you want them to find it? How will users trust your app sufficiently to download and try it? Does your app really need so many permissions? How large is the download, and how practical is it to download over the mobile network? Will it fit on the user's phone, particularly if there is little free storage available on their device? And does the app install correctly? - there may be signing issues which cause the app to be rejected by some phones.

Phase 5: Validation

This includes payment, usage and user feedback. As you may already know, a mobile app with poor feedback is unlikely to succeed. Furthermore, many apps have a very short active life on a user's phone. If the app does not please and engage them within a few minutes it is likely to be discarded or ignored. And for those of you who are seeking payment, it is worth testing the various forms of payment, especially for in-app payments.

Consider finding ways to test the following as soon as practical:

- Problem detection and reporting. These may include your own code, third-party utilities, and online services.
- Mobile Analytics. Does the data being collected make sense? What anomalies are there in the reported data? What is the latency in getting the results, et cetera?

Learn More

Testing mobile apps is becoming mainstream with various good sources of information. Useful online sources include:

- *blog.testmunk.com* Testmunk's blog has a wide range of relevant articles on testing mobile apps.
- *enjoytesting.files.wordpress.com/2013/10/mobile_testing_ready_reckoner.pdf* contains short, clear testing ideas with examples, mainly for Android devices.
- *developers.google.com/google-test-automation-conference/2015/presentations* In 2015, the Google Test Automation Conference (GTAC) includes at least 5 presentations related to testing mobile apps, worth watching.
- *handsonmobileapptesting.com/* which links to the book: Hands-on Mobile App Testing, by Daniel Knott. A well-written book on various aspects of testing mobile apps. A sample chapter is available from the web site.
- *testdroid.com/blog*, a fertile blog on various topics including testing mobile apps. They also have a series on testing mobile games[30].
- *appqualitybook.com*, the website about Jason Arbon's interesting book based on the experiences of testing and analysing vast numbers of mobile apps.
- *appqualityalliance.org/resources,* the official App Quality Alliance AQuA website including their useful app testing guidelines.

30 testdroid.com/testdroid/7790/best-practices-in-mobile-game-testing