27

Designing and Developing Bulletproof Android Applications

In this chapter, we cover tips and techniques from our years in the trenches of mobile software design and development. We also warn you—the designers, developers, and managers of mobile applications—of the various and sundry pitfalls you should do your best to avoid. Reading this chapter all at one time when you're new to mobile development might be a bit overwhelming. Instead, consider reading specific sections when planning the parts of the overall process. All our advice might not be appropriate for your particular project, and processes can always be improved. Hopefully this information about how mobile development projects succeed (or fail) gives you some insight into how you might improve the chances of success for your own projects.

Best Practices in Designing Bulletproof Mobile Applications

The "rules" of mobile application design are straightforward and apply across all platforms. These rules were crafted to remind us that our applications play a secondary role on the device. Often Android devices are, at the end of the day, phones first. These rules also make it clear that we do operate, to some extent, because of the infrastructure managed by the carriers and device manufacturers. These rules are echoed throughout the Android Software Development Kit (SDK) License Agreement and those of third-party application marketplace terms and conditions.

These "rules" are as follows:

- Don't interfere with device phone and messaging services.
- Don't break or otherwise tamper with or exploit the device hardware, firmware, software, or OEM components.

- Don't abuse or cause problems on operator networks.
- Don't abuse the user's trust.

Now perhaps these rules sound like no-brainers, but even the most well-intentioned developer can accidentally fall into some of these categories if they aren't careful and don't test the application thoroughly before distribution. This is especially true for applications that leverage networking support and low-level hardware APIs on the device, and those that store private user data such as names, locations, and contact information.

Meeting Mobile Users' Demands

Mobile users also have their own set of demands for applications they install on their devices. Applications are expected to

- Be responsive, stable, and secure
- Have straightforward, intuitive user interfaces that are easy to get up and running
- Get the job done with minimal frustration to the user
- Be available 24 hours a day, 7 days a week (remote servers or services always on, always available)
- Include a Help and/or About Screen for feedback and support contact information

Designing User Interfaces for Mobile Devices

Designing effective user interfaces for mobile devices, especially applications that run on a number of different devices, is something of a black art. We've all seen bad mobile application user interfaces. A frustrating user experience can turn a user off your brand forever, and a good experience can win a user's loyalty for the long term. It can also give your application an edge over the competition, even if your functionality is similar. A great user interface can win over users even when the functionality is behind the competition. Here are some tips for designing great mobile user interfaces:

- Fill screens sparingly; too much information on one screen overwhelms the user.
- Be consistent with user interface workflows, menu types, and buttons. Consider the device norms with this consistency, as well.
- Make Touch Mode "hit areas" large enough and spaced appropriately.
- Streamline common use cases with clear, consistent, and straightforward interfaces.
- Use big, readable fonts and large icons.
- Integrate tightly with other applications on the system using standardized controls, such as the quick Contact badge, content providers, and search adapters.
- Keep localization in mind when designing text-heavy user interfaces. Some languages are lengthier than others.
- Reduce keys or clicks needed as much as possible.

- Do not assume that specific input mechanisms (such as specific buttons or the existence of a keyboard) are available on all devices.
- Try to design the default use case of each screen to require only the user's thumb. Special cases might require other buttons or input methods, but encourage "thumbing" by default.
- Size graphics appropriately for phones. Do not include oversized resources and assets because they use valuable device resources and load more slowly, even if they resize appropriately. Also consider stripping out unnecessary information, such as EXIF or IPTC metadata, using tools such as ImageMagick or PNGOptimizer. You can also use the Draw 9 Patch tool to help optimize your Android graphics files.
- In terms of "friendly" user interfaces, assume that users do not read the application permissions when they approve them to install your application. If your application does anything that could cause the user to incur significant fees or shares private information, consider informing them again (as appropriate) when your application performs such actions.

Note

We discuss how to design Android applications that are compatible with a wide range of devices, including how to develop for different screen sizes and resolutions, in Chapter 25, "Targeting Different Device Configurations and Languages."

Designing Stable and Responsive Mobile Applications

Mobile device hardware has come a long way in the past few years, but developers must still work with limited resources. Users do not usually have the luxury of upgrading the RAM and other hardware in Android devices such as phones. Android users can, however, take advantage of removable storage devices such as SD cards to provide some "extra" space for application and media storage. Spending some time upfront to design a stable and responsive application is important for the success of the project. The following are some tips for designing robust and responsive mobile applications:

- Don't perform resource intensive operations on the main UI thread. Always use asynchronous tasks or threads to offload blocking operations.
- Use efficient data structures and algorithms; these choices manifest themselves in app responsiveness and happy users.
- Use recursion with care; these functional areas should be code reviewed and performance tested.
- Keep application state at all times. Android activity stack makes this work well, but you should take extra care to go above and beyond.
- Save your state and assume that your application will be suspended or stopped at any moment. If your application is suspended or closed, you cannot expect a user to verify anything (click a button, and so on). If your application resumes gracefully, your users will be grateful.

- Start up fast and resume fast. You cannot afford to have the user twiddling thumbs waiting for your application to start. Instead, you need to strike a delicate balance between preloading and on-demand data because your application might be suspended (or closed) with no notice.
- Inform users during long operations by using progress bars. Consider offloading heavy processing to a server instead of performing these operations on the device because these operations might drain battery life beyond the limits users are willing to accept.
- Ensure that long operations are likely to succeed before embarking upon them. For example, if your application downloads large files, check for network connectivity, file size, and available space before attempting the download.
- Minimize use of local storage, as most devices have very limited amounts. Use external storage, when appropriate. Be aware that SD cards (the most common external storage option) can be ejected and swapped; your application should handle this gracefully.
- Understand that data calls to content providers and across the AIDL barrier come at a cost to performance, so make these calls judiciously.
- Verify that your application resource consumption model matches your target audience. Gamers might anticipate shorter battery life on graphics-intensive games, but productivity applications should not drain the battery unnecessarily and should be lightweight for people "on the go" who do not always have their phone charging.

Tip

Written by the Google Android team, Android Developers blog (http://android-developers.blogspot.com) is a fantastic resource. This blog provides detailed insight into the Android platform, often covering topics not discussed in the Android platform documentation. Here you can find tips, tricks, best practices, and shortcuts on relevant Android development topics such as memory management (such as Context management), view optimization (avoiding deep view hierarchies), and layout tricks to improve UI speed. Savvy Android developers visit this blog regularly and incorporate these practices and tips into their projects.

Designing Secure Mobile Applications

Many mobile applications integrate with core applications such as the Phone, Camera, and Contacts. Make sure you take all the precautions necessary to secure and protect private user data such as names, locations, and contact information used by your application. This includes safeguarding personal user data on application servers and during network transmission.

Tip

If your application accesses or uses private data, especially usernames, passwords, or contact information, it's a good idea to include an End User License Agreement (EULA) and a Privacy Policy with your application.

Handling Private Data

To begin with, limit the private or sensitive data your application stores as much as possible. Don't store this information in plain text, and don't transmit it without safeguards. Do not try to work around any security mechanisms imposed by the Android framework. Store private user data in private application files, which are private to the application, and not in shared parts of the operating system. Do not expose application data in content providers without enforcing appropriate permissions on other applications. Use the encryption classes available in the Android framework when necessary.

Transmitting Private Data

The same cautions should apply to any remote network data storage (such as application servers or cloud storage) and network transmission. Make sure any servers or services that your application relies upon are properly secured against identity or data theft and invasion of privacy. Treat any servers your application uses like any other part of the application—test these areas thoroughly. Any private data transmitted should be secured using typical security mechanisms such as SSL. The same rules apply when enabling your application for backups using services such as Android Backup Service.

Designing Mobile Applications for Maximum Profit

For billing and revenue generation, mobile applications generally fall into one of four categories:

- Free applications (including those with advertising revenue)
- Single payment (pay once)
- Subscription payments (pay on a schedule, often seen with productivity and service applications)
- In-application payment for content (pay for specific content, such as a ringtone, a Sword of Smiting, or a new level pack)

Applications can use multiple types of billing, depending on which marketplaces and billing APIs they use (Android Market, for example, limits billing methods to Google Checkout). Although there are currently no billing APIs in the Android framework, there are rumblings that this might change in a future version. With Android in general, third parties can provide billing methods or APIs, so technically the sky's the limit.

When designing your mobile applications, consider the functional areas where billing can come into play and factor this into your design. Consider the transactional integrity of specific workflow areas of the application that can be charged for. For example, if your application has the capability to deliver data to the device, make sure this process is transactional in nature so that if you decide to charge for this feature, you can drop in the billing code, and when the user pays, the delivery occurs, or the entire transaction is rolled back.

Note

You learn more about the different methods currently available to market your application in Chapter 29, "Selling Your Android Application."

Leveraging Third-Party Standards for Android Application Design

There are no certification programs specifically designed for Android applications. However, as more applications are developed, third-party standards might be designed to differentiate quality applications from the masses. For example, mobile marketplaces could impose quality requirements and certainly programs could be created with some recognized body's endorsement or stamp of approval. Developers with an eye on financial applications would do well to consider conformance requirements.

Warning

With Android, the market is expected to manage itself. Do not make the mistake of interpreting that as "no rules" when it really means "few rules imposed by the system." There are strong licensing terms to keep malware and other malicious code out of users' hands.

It can be highly beneficial to examine the certification programs available in other mobile platforms and adjust them for Android. You might also want to examine the certification programs for desktop applications and other mobile platforms; consider how the requirements from these programs can be applied to Android applications. For example, if a specific type of encryption is recommended to meet certification requirements, and that type of encryption is feasible within Android, you should consider using it within your application.

Designing Mobile Applications for Ease of Maintenance and Upgrades

Generally speaking, it's best to make as few assumptions about the device configurations as possible when developing a mobile application. You'll be rewarded later when you want to port your application or provide an easy upgrade. You should carefully consider what assumptions you make.

Leveraging Network Diagnostics

In addition to developing adequate documentation and easy-to-decipher code, you can leverage some tricks to help maintain and monitor mobile applications in the field. Many of these tricks work best with mobile applications leveraging an application server, but you can sometimes gather information from third-party reports, such as application sales figures from mobile marketplaces or by including optional feedback gathering features in your application. Developers can also gather statistics from the bug reports that users can send in when a crash occurs on the device.

For networked applications, it can be highly useful to build in some lightweight auditing, logging, and reporting on the server side to keep your own statistics and not rely solely on third-party information. For example, you can easily keep track of

- How many users launch the application for the first time?
- How many users regularly use the application?
- What are the most popular usage patterns and trends?
- What are the least popular usage patterns and features?
- What devices (determined by application versioning or other relevant metrics) are the most popular?

Often you can translate these figures into rough estimates of expected sales, which you can later compare with actual sales figures from third-party marketplaces. You can streamline and make more efficient the most popular usage. You can review and improve the least popular features. Sometimes you can even identify potential bugs, such as features that are not working at all, just by noting that a feature has never been used in the field. Finally, you can determine which device targets are most appropriate for your specific application and user base.

Tip

Never collect personal data without the user's knowledge and consent. Gathering anonymous diagnostics is fairly commonplace, but avoid keeping any data that can be considered private. Make sure your sample sizes are large enough to obfuscate any personal user details, and make sure to factor out any live QA testing data from your results (especially when considering sales figures).

Designing for Easy Updates and Upgrades

Android applications can easily be upgraded in the field. The application update and upgrade processes do pose some challenges to developers, though. By update, we're talking about modifying the Android manifest version information and re-deploying the updated application on users' devices. By upgrading, we're talking about creating an entirely new application package with new features and deploying it as a separate application that the user needs to choose to install.

From an update perspective, you need to consider what conditions necessitate an update in the field. For example, do you draw the line at crashes or feature requests? You also want to consider the frequency in which you deploy updates—you need to schedule updates such that they come up frequently enough to be truly useful, but not so often that the users are constantly updating their application.

Tip

You should build application content updates into the application functionality as a feature (often network-driven) as opposed to necessitating an over-the-air actual application update. By enabling your applications to retrieve fresh content on-the-fly, you keep your users happy longer and applications stay relevant.

When considering upgrades, decide what manner you will migrate users from one version of your application to the next. Will you leverage Android backup features so that your users can transition seamlessly from one device to the next or will you provide your own solution? Consider how you will inform users of existing applications that a major new version is available.

Leveraging Android Tools for Application Design

The Android SDK and developer community provide a number of useful tools and resources for application design. You might want to leverage the following tools during this phase of your development project:

- The Android emulator is a good place to start for rapid proof of concept, before you have specific devices. You can use different Android Virtual Device (AVD) configurations to simulate different device configurations and platform versions.
- The DDMS tool is very useful for memory profiling.
- The Hierarchy Viewer in Pixel Perfect View enables accurate user interface design.
- The Draw Nine Patch tool can create stretchable graphics for mobile use.
- Use real devices for feasibility research and application proof-of-concept work, whenever possible. Do not design solely using the emulator.
- The technical specifications for specific devices, often available from manufacturers and carriers, can be invaluable for determining the configuration details of target devices.

Avoiding Silly Mistakes in Android Application Design

Last but not least, here is a list of some of the silly mistakes Android designers should do their best to steer clear of:

- Designing or developing for months without performing feasibility testing on the device
- Designing for a single device, platform, language, or hardware
- Designing as if your device has a large amount of storage and processing power and is always plugged in to a power source
- Developing for the wrong version of the Android SDK (verify device SDK version)
- Trying to adapt applications to smaller screens after the fact by having the phone "scale"
- Deploying oversized graphics and media assets with an application instead of sizing them appropriately

Best Practices in Developing Bulletproof Mobile Applications

Developing applications for mobile is not that different from traditional desktop development. However, developers might find developing mobile applications more restrictive, especially resource constrained. Again, let's start with some best practices or "rules" for mobile application development:

- Test assumptions regarding feasibility early and often on the target devices.
- Keep application size as small and efficient as possible.
- Choose efficient data structures and algorithms appropriate to mobile.
- Exercise prudent memory management.
- Assume that devices are running primarily on battery power.

Designing a Development Process That Works for Mobile Development

A successful project's backbone is a good software process. It ensures standards, good communication, and reduces risks. We talked about the overall mobile development process in Chapter 26. Again, here are a few general tips of successful mobile development processes:

- Use an iterative development process.
- Use a regular, reproducible build process with adequate versioning.
- Communicate scope changes to all parties—changes often affect testing most of all.

Testing the Feasibility of Your Application Early and Often

It cannot be said enough: You must test developer assumptions on real devices. There is nothing worse than designing and developing an application for a few months only to find that it needs serious redesign to work on an actual device. Just because your application works on the emulator does not, *in any way*, guarantee that it will run properly on the device. Some functional areas to examine carefully for feasibility include

- Functionality that interacts with peripherals and device hardware
- Network speed and latency
- Memory footprint and usage
- Algorithm efficiency
- User interface suitability for different screen sizes and resolutions
- Device input method assumptions
- File size and storage usage

We know; we sound like a broken record but, truly, we've seen this mistake happen over and over again. Projects are especially vulnerable to this when target devices aren't yet available. What happens is that engineers are forced closer to the waterfall method of software development with a big, bad surprise after weeks or months of development on some vanilla-style emulator.

We don't need to explain again why waterfall approaches are dangerous, do we? You can never be too cautious about this stuff. Think of this as the preflight safety speech of mobile software development.

Using Coding Standards, Reviews, and Unit Tests to Improve Code Quality

Developers who spend the time and effort necessary to develop efficient mobile application are rewarded by their users. The following is a representative list of some of the efforts that you can take:

- Centralizing core features in shared Java packages (or, if you have shared C or C++ libraries, consider using the Android NDK)
- Developing for compatible versions of the Android SDK (know your target devices)
- Using built-in controls and widgets appropriate to the application, customizing only where needed

You can use system services to determine important device characteristics (screen type, language, date, time, input methods, available hardware, and so on). If you make any changes to system settings from within your application, be sure to change the settings back when your application exits or pauses, if appropriate.

Defining Coding Standards

Developing a set of well-communicated coding standards for the development team can help drive home some of the important requirements of mobile applications. Some standards might include

- Implementing robust error handling and handle exceptions gracefully.
- Moving lengthy, process-intensive, or blocking operations off the main UI thread.
- Releasing objects and resources you aren't actively using.
- Practicing prudent memory management. Memory leaks can render your application useless.
- Using resources appropriately for future localization. Don't hardcode strings and other assets in code or layout files.
- Avoiding obfuscation in the code itself; comments are worthwhile. However, you should consider obfuscation later in the development process to protect against software piracy. Ideally, use obfuscation settings that preserve filename and line number details, so that you can easily track down application defects.

- Considering using standard document generation tools, such as Javadoc.
- Instituting and enforce naming conventions—in code and in database schema design.

Performing Code Reviews

Performing code inspections can improve the quality of project code, help enforce coding standards, and identify problems before QA gets their hands on a build and spends time and resources testing it.

It can also be helpful to pair developers with the QA tester who tests their specific functional areas to build a closer relationship between the teams. If testers understand how the application and Android operating system functions, they can test the application more thoroughly and successfully. This might or might not be done as part of a formal code review process. For example, a tester can identify defects related to type-safety just by noting the type of input expected (but not validated) on a form field of a layout or by reviewing Submit or Save button handling function with the developer.

Developing Code Diagnostics

The Android SDK provides a number of packages related to code diagnostics. Building a framework for logging, unit testing, and exercising your application to gather important diagnostic information, such as the frequency of method calls and performance of algorithms, can help you develop a solid, efficient, and effective mobile application. It should be noted that diagnostic hooks are almost always removed prior to application publication because they impose significant performance reductions and greatly reduce responsiveness.

Using Application Logging

In Chapter 3, "Writing Your First Android Application," we discuss how to leverage the built-in logging class `android.util.Log` to implement diagnostic logging, which can be monitored via a number of Android tools, such as the LogCat utility (available within DDMS, ADB, and Android Development Plug-in for Eclipse).

Developing Unit Tests

Unit testing can help developers move one step closer to the elusive 100 percent of code coverage testing. The Android SDK includes extensions to the JUnit framework for testing Android applications. Automated testing is accomplished by creating test cases, in Java code, that verify that the application works the way you designed it. You can do this automated testing for both unit testing and functional testing, including user interface testing.

Basic JUnit support is provided through the `junit.framework` and `junit.runner` packages. Here you find the familiar framework for running basic unit tests with helper classes for individual test cases. You can combine these test cases into test suites. There are utility classes for your standard assertions and test result logic.

The Android-specific unit testing classes are part of the `android.test` package, which includes an extensive array of testing tools designed specifically for Android applications.

This package builds upon the JUnit framework and adds many interesting features, such as the following:

- Simplified Hooking of Test Instrumentation (`android.app.Instrumentation`) with `android.test.InstrumentationTestRunner`, which you can run via ADB shell commands
- Performance Testing (`android.test.PerformanceTestCase`)
- Single `Activity` (or `Context`) Testing (`android.test.ActivityUnitTestCase`)
- Full Application Testing (`android.test.ApplicationTestCase`)
- Services Testing (`android.test.ServiceTestCase`)
- Utilities for generating events such as Touch events (`android.test.TouchUtils`)
- Many more specialized assertions (`android.test.MoreAsserts`)
- `View` validation (`android.test.ViewAsserts`)

If you are interested in designing and implementing a unit test framework for your Android application, we suggest working through our tutorial called "Android SDK: Unit Testing with the JUnit Testing Framework."You can find this tutorial online on our blog at http://androidbook.blogspot.com/2010/08/unit-testing-with-android-junit.html.

Tip

For more information on JUnit, check out http://www.junit.org, or books on the subject.

Handling Defects Occurring on a Single Device

Occasionally, you have a situation in which you need to provide code for a specific device. Google and the Android team tell you that when this happens, it's a bug, so you should tell them about it. By all means, do so. However, this won't help you in the short term. Handling bugs that occur only on a single device can be tricky.You don't want to branch code unnecessarily, so here are some of your choices:

- If possible, keep the client generic, and use the server to serve up device-specific items.
- If the conditions can be determined programmatically on the client, try to craft a generic solution that enables developers to continue to develop under one source code tree, without branching.
- If the device is not a high-priority target, consider dropping it from your requirements if the cost-benefit ratio suggests that a workaround is not cost effective.
- If required, branch the code to implement the fix. Make sure to set your Android manifest file settings such that the branched application version is installed only on the appropriate devices.
- If all else fails, document the problem only and wait for the underlying "bug" to be addressed. Keep your users in the loop.

Leveraging Android Tools for Development

The Android SDK comes with a number of useful tools and resources for application development. The development community adds even more useful utilities to the mix. You might want to leverage the following tools during this phase of your development project:

- The Eclipse development environment with the ADT plug-in
- The Android emulator and physical devices for testing
- The Android Dalvik Debug Monitor Service (DDMS) tool for debugging and interaction with the emulator or device
- The Android Debug Bridge (ADB) tool for logging, debugging, and shell access tools
- The `sqlite3` command-line tool for application database access (available via the ADB shell)
- The Hierarchy Viewer for user interface debugging of views

There are also numerous other tools available as part of the Android SDK. See the Android documentation for more details.

Avoiding Silly Mistakes in Android Application Development

Here are some of the frustrating and silly mistakes Android developers should try to avoid:

- Forgetting to add new application activities and necessary permissions to the `AndroidManifest.xml` file
- Forgetting to display `Toast` messages using the `show()` method
- Hard-coding information such as network information, test user information, and other data into the application
- Forgetting to disable diagnostic logging before release
- Distributing live applications with debug mode enabled

Summary

Be responsive, stable, and secure—these are the tenets of Android development. In this chapter, we armed you, the software designers, developers, and project managers, with tips, tricks, and best practices for mobile application design and development based upon real-world knowledge and experience from veteran mobile developers. Feel free to pick and choose which information works well for your specific project, and keep in mind that the software process, especially the mobile software process, is always open to improvement.

References and More Information

Android Best Practices: Designing for Performance:
http://developer.android.com/guide/practices/design/performance.html
Android Best Practices: Designing for Responsiveness:
http://developer.android.com/guide/practices/design/responsiveness.html
Android Best Practices: Designing for Seamlessness:
http://developer.android.com/guide/practices/design/seamlessness.html
Android Best Practices: User Interface Guidelines:
http://developer.android.com/guide/practices/ui_guidelines/index.html

28

Testing Android Applications

Test early, test often, test on the device. That is the quality assurance mantra we consider most important when it comes to testing mobile applications. Testing applications need not be an onerous process. Instead, you can adapt traditional quality assurance techniques such as automation and unit testing to Android with relative ease. In this chapter, we discuss our tips and tricks for testing Android applications. We also warn you—the project managers, software developers, and testers of mobile applications—of the various and sundry pitfalls to do your best to avoid.

Best Practices in Testing Mobile Applications

Like all QA processes, mobile development projects benefit from a well-designed defect tracking system, regularly scheduled builds, and planned, systematic testing. There are also plentiful opportunities for white box (or gray box) testing and some limited opportunities for automation.

Designing a Mobile Application Defect Tracking System

You can customize most defect tracking systems to work for the testing of mobile applications. The defect tracking system must encompass tracking of issues for specific device defects and problems related to any centralized application servers (if applicable).

Logging Important Defect Information

A good mobile defect tracking system includes the following information about a typical device defect:

- Build version information, language, and so on.
- Device configuration and state information including device type, platform version, screen orientation, network state, and carrier information.

- Steps to reproduce the problem using specific details about exactly which input methods were used (touch versus click).
- Device screenshots that can be taken using DDMS or the Hierarchy Viewer tool provided with the Android SDK.

Tip

It can be helpful to develop a simple glossary of standardized terms for certain actions on the devices, such as touch mode gestures, and click versus tap, long-click versus press-and-hold, clear versus back, and so on. This helps make the steps to reproduce a defect more precise to all parties involved.

Redefining the Term Defect for Mobile Applications

It's also important to consider the larger definition of the term *defect*. Defects might occur on all devices or on only some devices. Defects might also occur in other parts of the application environment, such as on a remote application server. Some types of defects typical on mobile applications include

- Crashing and unexpected terminations.
- Features not functioning correctly (improper implementation).
- Using too much disk space/memory on the device.
- Inadequate input validation (typically, button mashing).
- State management problems (startup, shutdown, suspend, resume, power off).
- Responsiveness problems (slow startup, shutdown, suspend, resume).
- Inadequate state change testing (failures during inter-state changes, such as an unexpected interruption during resume).
- Usability issues related to input methods, font sizes, and cluttered screen real estate. Cosmetic problems that cause the screen to display incorrectly.
- Pausing or "freezing" on the main UI thread (failure to implement asynchronous threading).
- Feedback indicators missing (failure to indicate progress).
- Integration with other applications on the device causing problems.
- Application "not playing nicely" on the device (draining battery, disabling power-saving mode, overusing networking resources, incurring extensive user charges, obnoxious notifications).
- Using too much memory, not freeing memory or releasing resources appropriately, and not stopping worker threads when tasks are finished.
- Not conforming to third-party agreements, such as Android SDK License Agreement, Google Maps API terms, marketplace terms, or any other terms that apply to the application.

- Application client or server not handling protected/private data securely. This includes ensuring that remote servers or services have adequate uptime and security measures taken.

Managing the Testing Environment

Testing mobile applications poses a unique challenge to the QA team, especially in terms of configuration management. The difficulty of such testing is often underestimated. Don't make the mistake of thinking that mobile applications are easier to test because they have fewer features than desktop applications and are, therefore, simpler to validate. The vast variety of Android devices available on the market today makes testing different installation environments tricky.

> **Warning**
>
> Ensure that all changes in project scope are reviewed by the quality assurance team. Adding new devices sometimes has little impact on the development schedule but can have significant consequences in terms of testing schedules.

Managing Device Configurations

Device fragmentation is perhaps the biggest challenge the mobile tester faces. Android devices come in various form-factors with different screens, platform versions, and underlying hardware. They come with a variety of input methods such as buttons and touch screens. They come with optional features, such as cameras and enhanced graphics support. Many Android devices are smartphones, but not all. Keeping track of all the devices, their abilities, and so on is a big job, and much of the work falls on the test team.

QA personnel must have a detailed understanding of the functionality available of each target device, including familiarity with what features are available and any device-specific idiosyncrasies that exist. Whenever possible, testers should test each device as it is used in the field, which might not be the device's default configuration or language. This means changing input modes, screen orientation, and locale settings. It also means testing with battery power, not just plugged in while sitting at a desk.

> **Tip**
>
> Be aware of how third-party firmware modifications can affect how your application works on the device. For example, let's assume you've gotten your hands on an unbranded version of a target phone and testing has gone well. However, if certain carriers take that same device, but remove some default applications and load it up with others, this is valuable information to the tester. Just because your application runs flawlessly on the "vanilla" device doesn't mean that this is how most users' devices are configured by default. Do your best to get test devices that closely resemble the devices users will have in the field.

One hundred percent testing coverage is impossible, so QA must develop priorities thoughtfully. As we discuss in Chapter 26, developing a device database can greatly reduce the confusion of mobile configuration management, help determine testing priorities, and keep track of physical hardware available for testing. Using AVD configurations, the

emulator is also an effective tool for extending coverage to simulate devices and situations that would not be covered otherwise.

Tip

If you have trouble configuring devices for real-life situations, you might want to look into the device "labs" available through some carriers. Instead of loaner programs, the developer visits the carrier's onsite lab where they can rent time on specific devices. Here, the developer installs the application and tests it—not ideal for recurring testing but much better than no testing, and some labs are staffed with experts to help out with device-specific issues.

Determining Clean Starting State on a Device

There is currently no good way to "image" a device so that you can return to the same starting state again and again. The QA test team needs to define what a "clean" device is for the purposes of test cases. This can involve a specific uninstall process, some manual clean-up, or sometimes a factory reset.

Tip

Using the Android SDK tools such as DDMS and ADB enables developers and testers access to the Android file system, including application SQLite databases. You can use these tools to monitor and manipulate data on the device and the emulator. For example, testers might use the `sqlite3` command-line interface to "wipe" an application database or fill it with test data for specific test scenarios.

Mimicking Real-World Activities

It is nearly impossible (and certainly not cost-effective for most companies) to set up a complete isolated environment for mobile application testing. It's fairly common for networked applications to be tested against test (mock) application servers and then go "live" on production servers with similar configurations. However, in terms of device configuration, mobile software testers must use real devices with real service to test mobile applications properly. If the device is a phone, then it needs to be able to make and receive phone calls, send and receive text messages, determine location using LBS services, and basically do anything a phone would normally do.

Testing a mobile application involves more than just making sure the application works properly. In the real world, your application does not exist in a vacuum but is one of many installed on the device. Testing a mobile application involves ensuring that the software integrates well with other device functions and applications. For example, let's say you were developing a game. Testers must verify that calls received while playing the game caused the game to automatically pause (keep state) and allow calls to be answered or ignored without issue.

This also means testers must install other applications on to the device. A good place to start is with the most popular applications for the device. Testing your application not only with these applications installed, but also with real use, can reveal integration issues or usage patterns that don't mesh well with the rest of the device.

Sometimes testers need to be creative when it comes to reproducing certain types of events. For example, testers must ensure that their application behaves appropriately when mobile handsets lose network or phone coverage.

Tip

Unlike some other mobile platforms, testers actually have to take special steps to make most Android devices lose coverage above and beyond holding them wrong. To test loss of signal, you could go out and test your application in a highway tunnel or elevator, or you could just place the device in the refrigerator. Don't leave it in the cold too long, though, or it will drain the battery. Tin cans work great, too, especially those that have cookies in them: First, eat the cookies; then place the phone in the can to seal off the signal.

Maximizing Testing Coverage

All test teams strive for 100 percent testing coverage, but most also realize such a goal is not reasonable or cost-effective (especially with dozens of Android devices available around the world). Testers must do their best to cover a wide range of scenarios, the depth and breadth of which can be daunting—especially for those new to mobile. Let's look at several specific types of testing and how QA teams have found ways—some tried-and-true and others new and innovative—to maximize coverage.

Validating Builds and Designing Smoke Tests

In addition to a regular build process, it can be helpful to institute a build acceptance test policy (also sometimes called build validation, smoke testing, or sanity testing). Build acceptance tests are short and targeted at key functionality to determine if the build is good enough for more thorough testing to be completed. This is also an opportunity to quickly verify bug fixes expected to be in the build before a complete retesting cycle occurs. Consider developing build acceptance tests for multiple Android platform versions to run simultaneously.

Automating Functional Testing for Build Acceptance

Mobile build acceptance testing is typically done manually on the highest-priority target device; however, this is also an ideal situation for an automated "sanity" test. By creating a bare-bones functional test for the emulator that, as desktop software, can be used with typical QA automation platforms such as Borland SilkTest (www.borland.com/us/products/silk/silktest/index.html), the team can increase its level of confidence that a build is worth further testing, and the number of bad builds delivered to QA can be minimized.

Testing on the Emulator Versus the Device

When you can get your hands on the actual device your users have, focus your testing there. However, devices and the service contracts that generally come with them can be expensive. Your test team cannot be expected to set up test environments on every carrier or every country where your users use your application. There are times when the Android emulator can reduce costs and improve testing coverage. Some of the benefits of using the emulator include

- Ability to simulate devices when they are not available or in short supply
- Ability to test difficult test scenarios not feasible on live devices
- Ability to be automated like any other desktop software

Testing Before Devices Are Available Using the Emulator

Developers often target up-and-coming devices or platform versions not yet available to the general public. These devices are often highly anticipated and developers who are ready with applications for these devices on Day 1 of release often experience a sales bump because fewer applications are available to these users—less competition, more sales.

The latest version of the Android SDK is usually released to developers several months prior to when the general public receives over-the-air updates. Also, developers can sometimes gain access to preproduction phones through carrier and manufacturer developer programs. However, developers and testers should be aware of the dangers of testing on preproduction phones: These phones are beta-quality. The final technical specifications and firmware can change without notice. These phone release dates can slip, and the phone might never reach production.

When preproduction phones cannot be acquired, testers can do some functional testing using emulator configurations that attempt to closely match the target platform, lessening the risks for a compact testing cycle when these devices go live, allowing developers to release applications faster.

Leveraging Automated Testing Opportunities Using the Emulator

Android testers have a number of different automation options available to choose from. It's certainly possible to rig up automated testing software to exercise the software emulator and there are a number of testing tools (monkey, for example) that can help with the testing process. Unfortunately, there are not really a lot of options for automated hardware testing, beyond those used with the unit testing framework. We can certainly *imagine* someone coming up with a hardware testing solution—in our minds, the device looks a lot like the automated signature machine U.S. presidents use to sign pictures and Christmas cards. The catch is that every device looks and acts differently, so any animatronic hand would need to be recalibrated for each device. The other problem is how to determine when the application has failed or succeeded. If anyone is developing mobile software automated testing tools, it's likely a mobile software testing consultancy company. For the typical mobile software developer, the costs are likely prohibitive.

Understanding the Dangers of Relying on the Emulator

Unfortunately, the emulator is more of a "generic" Android device that pretends at many of the device internals—despite all the options available within the AVD configuration.

Tip

Consider developing a document describing the specific AVD configurations used for testing different device configurations as part of the test plan.

The emulator does not represent the specific implementation of the Android platform that is unique to a given device. It does not use the same hardware to determine signal, networking, or location information. The emulator can pretend to make and receive calls and messages, or take pictures or video. At the end of the day, it doesn't matter if the application works on the emulator if it doesn't work on the actual device.

Testing Strategies: White Box Testing

The Android tools provide ample tools for black box and white box testing:

- Black box testers might require only testing devices and test documentation. For black box testing, it is even more important that the testers have a working knowledge of the specific devices, so providing device manuals and technical specifications also aids in more thorough testing. In addition to such details, knowing device nuances as well as device standards can greatly help with usability testing. For example, if a dock is available for the device, knowing that it's either landscape or portrait mode is useful.
- White box testing has never been easier on mobile. White box testers can leverage the many affordable tools including the Eclipse development environment, which is free, and the many debugging tools available as part of the Android SDK. White box testers use the Android Emulator, DDMS, and ADB especially. They can also take advantage of the powerful unit testing framework, which we discussed in detail in the previous chapter. For these tasks, testers require a computer with a development environment similar to the developer's.

Testing Mobile Application Servers and Services

Although testers often focus on the client portion of the application, they sometimes neglect to thoroughly test the server portion. Many mobile applications rely on networking or "the cloud." If your application depends on a server or remote service to operate, testing the server side of your application is vital. Even if the service is not your own, you need to test thoroughly against it so you know it behaves as the application expects it to behave.

> **Warning**
>
> Users expect applications to be available any time, day or night, 24/7. Minimize server or service down times and make sure the application notifies the users appropriately (and doesn't crash and burn) if the services are unavailable. If the service is outside your control, it might be worthwhile to look at what Service Level Agreements are offered.

Here are some guidelines for testing remote servers or services:

- Version your server builds. You should manage server rollouts like any other part of the build process. The server should be versioned and rolled out in a reproducible way.
- Use test servers. Often, QA tests against a mock server in a controlled environment. This is especially true if the live server is already operational with real users.

- Verify scalability. Test the server or service under load, including stress testing (many users, simulated clients).
- Test the server security (hacking, SQL injection, and such).
- Ensure that your application handles remote server maintenance or service interruptions gracefully—scheduled or otherwise.
- Test server upgrades and rollbacks and develop a plan for how you are going to inform users if and when services are down.

These types of testing offer yet another opportunity for automated testing to be employed.

Testing Application Visual Appeal and Usability

Testing a mobile application is not only about finding dysfunctional features, but also about evaluating the usability of the application. Report areas of the application that lack visual appeal or are difficult to navigate or use. We like to use the walking-and-chewing-gum analogy when it comes to mobile user interfaces. Mobile users frequently do not give the application their full attention. Instead, they walk or do something else while they use it. Applications should be as easy for the user as chewing gum.

Tip

Consider conducting usability studies to collect feedback from people who are not familiar with the application. Relying solely on the product team members, who see the application regularly, can blind the team to application flaws.

Leveraging Third-Party Standards for Android Testing

Make a habit to try to adapt traditional software testing principles to mobile. Encourage quality assurance personnel to develop and share these practices within your company.

Again, no certification programs are specifically designed for Android applications at this time; however, nothing is stopping the mobile marketplaces from developing them. Consider looking over the certification programs available in other mobile platforms, such as the extensive testing scripts and acceptance guidelines used by Apple iPhone and BREW platforms and adjusting them for your Android applications. Whether you plan to apply for a specific certification, making an attempt to conform to well-recognized quality guidelines can improve your application's quality.

Handling Specialized Test Scenarios

In addition to functional testing, there are a few other specialized testing scenarios that any QA team should consider.

Testing Application Integration Points

It's necessary to test how the application behaves with other parts of the Android operating system. For example:

- Ensuring that interruptions from the operating system are handled properly (incoming messages, calls, and powering off)

- Validating Content Provider data exposed by your application, including such uses as through a Live Folder
- Validating functionality triggered in other applications via an `Intent`
- Validating any known functionality triggered in your application via an `Intent`
- Validating any secondary entry points to your application as defined in the `AndroidManifest.xml`, such as application shortcuts
- Validating alternate forms of your application, such as App Widgets
- Validating service-related features, if applicable

Testing Upgrades

When possible, perform upgrade tests of both the client and the server or service side of things. If upgrade support is planned, have development create a mock upgraded Android application so that QA can validate that data migration occurs properly, even if the upgraded application does nothing with the data.

> **Tip**
>
> Users receive Android platform updates over-the-air on a regular basis. The platform version your application is installed on might change over time. Some developers have found that firmware upgrades have broken their applications, necessitating upgrades. Always re-test your applications when a new version of the SDK is released, so that you can upgrade users before your applications have a chance to break in the field.

Testing Product Internationalization

It's a good idea to test internationalization support early in the development process—both the client and the server or services. You're likely to run into some problems in this area related to screen real-estate and issues with strings, dates, times, and formatting.

> **Tip**
>
> If your application will be localized for multiple languages, test in a foreign language—especially on a verbose one. The application might look flawless in English but be unusable in German where words are generally longer.

Testing for Conformance

Make sure to review any policies, agreements, and terms to which your application must conform and make sure your application complies. For example, Android applications must by default conform to the Android Developer Agreement and the Google Maps terms of service (if applicable).

Installation Testing

Generally speaking, installation of Android applications is straightforward; however, you need to test installations on devices with low resources and low memory and test installation from the specific marketplaces when your application "goes live." If the manifest install location allows external media, be sure to test various low or missing resource scenarios.

Backup Testing

Don't forget to test features that are not readily apparent to the user, such as the backup and restore services and sync features discussed in Chapter 23, "Managing User Accounts and Synchronizing User Data."

Performance Testing

Application performance matters in the mobile world. The Android SDK has support for calculating performance benchmarks within an application and monitoring memory and resource usage. Testers should familiarize themselves with these utilities and use them often to help identify performance bottlenecks and dangerous memory leaks and misused resources.

Testing Application Billing

Billing is too important to leave to guesswork. Test it. You notice a lot of test applications on the Android Market. Remember to specify that your application is a test app.

Testing for the Unexpected

Regardless of the workflow you design, understand that users do random, unexpected things—on purpose and by accident. Some users are "button mashers," whereas others forget to set the keypad lock before putting the phone in their pocket, resulting in a weird set of key presses. A phone call or text message inevitably comes in during the farthest, most-remote edge cases. Your application must be robust enough to handle this. The Exerciser Monkey command-line tool is a good way to test for this type of event.

Testing to Increase Your Chances of Being a "Killer App"

Every mobile developer wants to develop a "killer app"—those applications that go viral, rocket to the top of the charts, and make millions a month. Most people think that if they just find the right idea, they'll have a killer app on their hands. Developers are always scouring the top-ten lists, trying to figure out how to develop the next big thing. But let us tell you a little secret: If there's one thing that all "killer apps" share, it's a higher-than-average quality standard. No clunky, slow, obnoxious, or difficult-to-use application ever makes it to the big leagues. Testing and enforcing quality standards can mean the difference between a mediocre application and a killer app.

If you spend any time examining the mobile marketplace, you notice a number of larger mobile development companies publish a variety of high-quality applications with a shared look and feel. These companies leverage user interface consistency, shared and above-average quality standards to build brand loyalty and increase market share, while hedging their bets that perhaps just one of their many applications will have that magical combination of great idea and quality design. Other, smaller companies often have the great ideas but struggle with the quality aspects of mobile software development. The inevitable result is that the mobile marketplace is full of fantastic application ideas badly executed with poor user interfaces and crippling defects.

Leveraging Android Tools for Android Application Testing

The Android SDK and developer community provide a number of useful tools and resources for application testing and quality assurance. You might want to leverage these tools during this phase of your development project:

- The physical devices for testing and bug reproduction
- The Android emulator for automated testing and testing of builds when devices are not available
- The Android DDMS tool for debugging and interaction with the emulator or device, as well as taking screenshots
- The ADB tool for logging, debugging, and shell access tools
- The Exerciser Monkey command-line tool for stress testing of input (available via ADB shell)
- The `sqlite3` command-line tool for application database access (available via ADB shell)
- The Hierarchy Viewer for user interface navigation and verification and for pixel-perfect screenshots of the device
- The Eclipse development environment with the ADT and related logging and debugging tools for white box testing

It should be noted that although we have used the Android tools such as the Android emulator and DDMS debugging tools with Eclipse, these are stand-alone tools that can be used by quality assurance personnel without the need for source code or a development environment.

Avoiding Silly Mistakes in Android Application Testing

Here are some of the frustrating and silly mistakes and pitfalls that Android testers should try to avoid:

- Not testing the server or service components used by an application as thoroughly as the client side.
- Not testing with the appropriate version of the Android SDK (device versus development build versions).
- Not testing on the device and assuming the emulator is enough.
- Not testing the live application using the same system that users use (billing, installation, and such). Buy your own app.
- Neglecting to test all entry points to the application.
- Neglecting to test in different coverage areas and network speeds.
- Neglecting to test using battery power. Don't always have the device plugged in.

Outsourcing Testing Responsibilities

Mobile quality assurance can be outsourced. Remember, though, that the success of outsourcing your QA responsibilities depends on the quality and detail of the documentation you can provide. Outsourcing makes it more difficult to form the close relationships between QA and developers that help ensure thorough and comprehensive testing.

Summary

In this chapter, we armed you—the keepers of application quality—with real-world knowledge for testing Android applications. Whether you're a team of one or one hundred, testing your applications is critical for project success. Luckily, the Android SDK provides a number of tools for testing applications, as well as a powerful unit testing framework. By following standard quality assurance techniques and leveraging these tools, you can ensure that the application you deliver to your users is the best it can be.

References and More Information

Android Dev Guide: Testing & Instrumentation:
http://developer.android.com/guide/topics/testing/testing_android.html
Android Dev Guide: Testing:
http://developer.android.com/guide/developing/testing/index.html
Android Tools: UI/Application Exerciser Monkey:
http://developer.android.com/guide/developing/tools/monkey.html
Wikipedia on Software Testing:
http://en.wikipedia.org/wiki/Software_testing
Software Testing Help:
http://www.softwaretestinghelp.com