The Report Committee for Kristina Denise Hager
certifies that this is the approved version of the following report:

Patterns for Reusable Android Development

APPROVED BY

SUPERVISING COMMITTEE:

Adnan Aziz, Supervisor

Christine Julien

Patterns for Reusable Android Development

by

Kristina Denise Hager, B.A.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2015

This report is dedicated to my partner Jean Krejca for her encouragement and support over the last three years, and to my friends and coworkers for their patience, and to my parents for their support.

Acknowledgments

I would like to thank my supervisor, Dr. Adnan Aziz, for his enthusiasm and wisdom, his invaluable feedback, and for teaching one of best and most challenging classes I've ever taken. In addition I would like to thank Dr. Christine Julien, Mr. Bill Bard, Dr. Kathleen Barber, and all of the other excellent instructors in the CLEE program.

Patterns for Reusable Android Development

Kristina Denise Hager, M.S.E.
The University of Texas at Austin, 2015

Supervisor: Adnan Aziz

Software libraries, encapsulating functionality behind a clearly defined interface, are a key component of all modern software development. In Java and in other mature technologies, the development of a library is nearly as easy as and as standard as the development of an application. However, the Android development environment is not as oriented towards library development. As a result, much of the powerful and open source code Android developers have created and published is embedded within and highly coupled to its original application, so it cannot be re-used by other applications. Also, the Android library format has recently been enhanced to encapsulate Android application components extending the possibilities for an Android library to be more than what was essentially just a Java API. This gives the Android developer opportunities to create even more powerful libraries. However, in contrast to the Java API, the interfaces to these libraries are not well defined, not subject to automatic and comprehensive type checking, and not able to be documented in a standard way. Android and these more powerful libraries

are a relatively new mobile technology. However, these shortcomings must be addressed for Android and its libraries to become mature technology and become as powerful as promised.

After introducing the reader to select Android development concepts pertinent to this paper, I will present a brief case study of an existing Android library and application and cover lessons I learned there which motivated the work I present here. I subsequently review the tenets of library development, derived from the literature, from the perspective of Android. I created example libraries and applications using those libraries to illustrate best practices and key points on challenges in the Android ecosystem that hinder reusable development. A reader will also gain knowledge of how to create and publish his or her own library for use by others. Finally, I package the Android library I originally studied into a JAR file for the benefit of the wider development community.

Table of Contents

List of Figures

List of Source Listings

Chapter 1

Introduction

1.1 Introduction

The open source movement is an undeniable force today in software development. Developers spend hours of their own time and hours of paid time per week contributing to open source projects. Many large companies, such as Apple, Facebook, Google, and more, have released software they have created internally to solve complex problems to the open source community [62]. Android itself is an open source project, so Android development is well poised to be a significant force in open source software.

Although third-party mobile development for Android devices started only in 2008 [4], mobile development is rapidly growing in importance. The sale of mobile devices continues to grow, while the sales of desktops and laptops are in decline [9]. Using tagged questions at StackOverflow [1] as an informal proxy for development trends, the interest in mobile development platforms is growing relative to traditional desktop languages as shown in Figure 1.1. In this graph, I use the Android and IOS tags to represent questions regarding mobile development and Linux, C++, and Java to represent languages most strongly associated with desktop development.

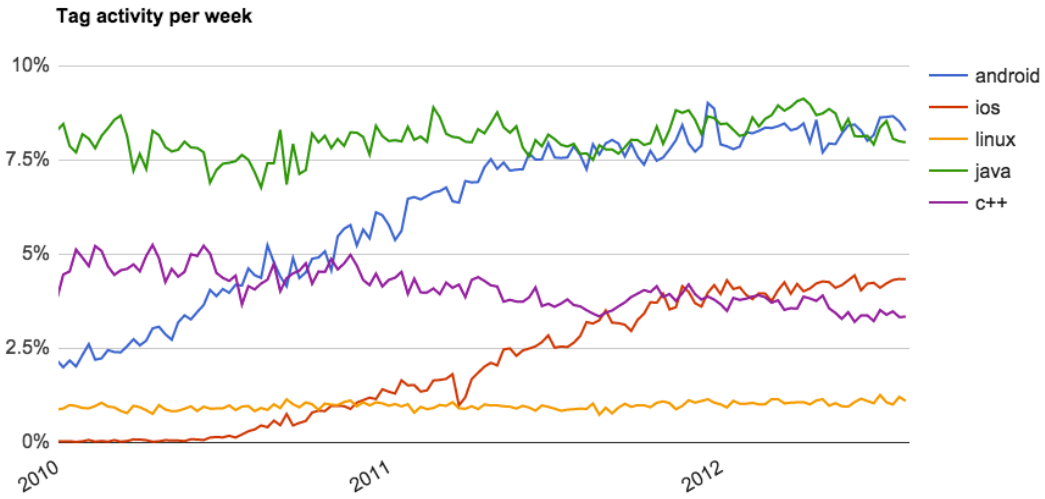

Figure 1.1: Tag activity on StackOverflow since 2010 [52]

Code re-use has been a significant player in speeding up software development and spreading its impact for at least the past few decades of software development. Software libraries, encapsulating functionality behind a clearly defined interface and easily used by applications, are a key component of this re-use, along with other powerful concepts such as object oriented design, design patterns, templates, and more. In Java and in other mature technologies, the development of a library is nearly as easy as and as standard as the development of an application. As I will show in this paper, the Android development environment is not as oriented towards library development. As a result, much of the powerful and open source code Android developers have created and published is embedded within and highly coupled to its original application, so it cannot be re-used by other applications. Also, the Android library format has recently been enhanced to encapsulate Android application

components extending the possibilities for an Android library to be more than what was essentially just a Java API. This gives the Android developer opportunities to create even more powerful libraries. However, in contrast to the Java API, the interfaces to these libraries are not well defined, not subject to automatic and comprehensive type checking, and not able to be documented in a standard way.

1.2 Structure of This Paper

The goal of this paper is to present best practices of and patterns for reusable Android development drawing from existing tenets of library development, This report will also point out aspects of the existing Android development ecosystem that hinder library development.

Chapter 2 introduces the reader to select Android development concepts that are necessary background for the discussion in this paper. These concepts include aspects of the Android development environment, selected Android terms and technologies, and an overview of Android library artifacts and publication.

Chapter 3 presents a case study of how difficult it is to re-use existing Android functionality from an Android library and an Android application. This chapter will cover lessons I learned studying these examples, which motivated the work in this paper.

Chapter 4 reviews the tenets of library development, referencing well-

established literature, from the perspective of Android. It also introduces challenges of library development particular to the Android development environment.

Chapter 5 presents my best practices for Android library development via example libraries and utilizing applications. This work includes prescriptive steps for creating, publishing, and using an Android library and provides concrete examples of challenges in the Android ecosystem that hinder reusable development.

Chapter 6 summarizes my contributions including my work packaging an existing, previously more difficult to re-use, Android library for the benefit of the broader software development community.

1.3 Source Code and Demonstration Resources

I published the source code I created in the course of this project online at GitHub. Section 6.3 lists these contributions in more detail, and Section 6.4 lists code published by others that I referenced in this paper.

Chapter 2

Background of Android Development

2.1 The Android Development Environment

In this section, I will introduce some of the fundamental components of the Android Development environment that are discussed in this paper. A reader experienced in Android development may wish to skip past those sections where he or she is already knowledgeable.

2.1.1 Integrated Development Environment

An Android developer needs to use a number of different tools in the various phases of Android application or library development. The following is a partial list of tools I have used in the creation of the simplest Android application: Android Virtual Device Manager (AVD), Android emulation, android (command line tool), lint, SDK manager, adb, and finally git for revision control.

While an Android developer is free to use only text editors and the command line to bring together all the components required for building his or her Android programs, doing so would be extremely tedious. For example, when I am using the Integrated Development Environment (IDE) to create and

debug an Android application, I rarely explicitly invoke any of the above listed tools. The IDE integrates the necessary tools into a uniform graphical interface, makes them intuitive to use, and automates routine tasks [64]. Therefore, Android has long supported an official IDE and a Software Development Kit (SDK) that has included a debugger, development libraries, sample code and more.

The default structure the IDE creates for a new program is the structure the majority of Android programs will use. The tools run automatically by the IDE are naturally, then, the tools that the average or new Android developer will use. While a more experienced developer may, of course, stray away from the defaults by changing configuration files by hand and using plugins or other techniques to extend or customize the IDE, the net effect is that the IDE has a large influence on how Android programs end up being structured.

In the course of this paper, the IDE I use and will give examples from is Android Studio version 1.3.2 with Java runtime 1.7.

2.1.2 Build System

The standard build tool for Android projects is now Gradle. Gradle is built on a Domain Specific Language (DSL) based on Groovy, and it is quite easily extended. The community around Gradle has built innumerable plugins to automate common tasks and that are easily used. I would strongly encourage any Android developer to take some time to understand the basics of Gradle, as that will definitely ease the task of customizing Gradle files to

specific needs. In Chapter 5 I will provide Gradle configurations that automate the task of building, testing, and promoting library artifacts.

2.1.3 Native Code

Android supports integration of 'native code', code written in another language like C or C++, into Android applications via the Java Native Interface (JNI) standard. JNI provides a mechanism for Java applications or libraries to handle the following situations [54]:

- Some functionality cannot be written entirely in Java because it needs to use platform dependent features.
- A library is already available but written in a language other than Java.
- Native code can better meet performance requirements.

The Android Native Development Kit (NDK) provides the required support for developing and packaging native code for usage in Android. At the moment, the Android NDK is transitioning between Android Studio and Eclipse. Android Studio has only beta level support for native code development with breaking changes to the build system expected [33]. Therefore, current Android NDK documentation still points the developer to the Eclipse NDK support. Since implementation details will change, my coverage here will stick to high-level concepts.

In order to create a native code application or library, the developer will need to have or modify these source components [34]:

Manifest Declaration The application or library must declare itself to be native in the Manifest (see Section 2.2.1).

Native Source Code Typically this is C or C++ code.

Java Source Code A native code package can include implementation in Java. Also, any functions implemented in native code will need to have a declaration in Java like: `public native int add (int x, int y);`. The 'native' keyword indicates that this function is implemented natively.

The compilation of native code will create these additional outputs:

Shared and/or Static Libraries These are *.so or *.a files built by the NDK tools.

Application Binary Interface (ABI) Different ABIs correspond to different architectures. In other words, the developer will need to build a set of shared or static objects per architecture the library will support such as 'armeabi' or 'x86'.

To package the native code into an application, the NDK will combine the source into an APK file which contains the shared and/or static libraries, the '.dex' from the Java and any other files needed for your app to run [34]. To package the native code into a library, the project's build scripts will need to create an archive with at least the manifest, shared or static libraries, '.class'

files created from the Java, and any other needed files. In Section 6.1.1, I will show how to create and publish a native library JAR from the shared objects, manifest, and Java source using Gradle and Android Studio.

2.1.4 Command Line Tools

While the vast majority of a typical developer's time is spent in the IDE's graphical interface, much, or perhaps all, of the tools used to create and work with Android applications or libraries have a command-line interface. In particular the command-line program `android` has the capability to create and update Android application projects, library projects and test projects. Since the vast majority of Android developers and especially novice Android developers will stick to the graphical interface of Android Studio, I did not use the command line for the examples in this paper to create library projects. However, the command-line approach may be an efficient way to create library projects [31]. In subsequent examples, I also use the 'terminal' interface available within Android Studio to issue custom Gradle build commands as in Chapter 5.

2.2 Introduction to Android Terms and Technologies

2.2.1 The Manifest and Intents

The Android Manifest is required for each application, library or module and must have the name `AndroidManifest.xml`. This file gives the Android operating system information that it needs to know before it can run

your application. Among many other purposes, the manifest describes the components of the application and names the Java classes that implement these and their capabilities. The manifest also declares permissions that the application or library desires or needs such as Internet access, access to contacts, and so on. This manifest has much functionality beyond what is listed here, and I would refer the reader to the Android documentation [25] for a more complete reference.

An Android project will typically have multiple manifest files. Specifically, it will have a manifest for each build flavor and for each library, module or other dependency. When a library is published or an application is built, Gradle will merge the manifests into one according to merge rules [32]. The Android developer must understand the purposes and implications of the manifest and of manifest merging.

2.2.2 Application Components: Activity, Service, and Broadcast Provider

The Android developer's guide describes three primary application components:

Activity An activity provides a screen with which users can interact to perform a task. It typically fills the entire screen, or it can be smaller and float on top of other screen.

Service A service has no user interface and performs long running operations in the background such as playing music or uploading large files [29].

Content Provider Content Providers encapsulate and manage access to a structured set of data. For example, Android includes content providers to manage data such as audio, video and personal contact information [26].

Although the official documentation describes these as 'application' components, I will show in this paper how these can also be packaged into a library with examples in Section 5.3.2.

2.2.3 Intents and Intent Filters

An activity, service or broadcast provider in a library or application is activated by an `Intent` which is a messaging object describing a desired action and, optionally, parameters for the action [27]. An intent can explicitly specify the component to perform the desired action. Alternately, an intent can simply specify the action and optionally parameters for the action it desires, and the operating system will try to find a component that can perform the action. This is known as an 'implicit intent' whereas the former is an 'explicit intent'.

To be implicitly invoked, any activity, service or broadcast provider component must specify the type of action it can perform and any parameters it needs for that action via an `<intent-filter>` in the manifest file [25]. The `<intent-filter>` is not required if the component is only explicitly invoked. A component can specify that it can be only started within its own application by setting the `exported` attribute to 'false'. This setting is suggested

for service components to ensure that another application cannot call your application's service with malicious intent or data [27].

2.2.4 Resources

The Android developer guide encourages developers to externalize 'resources' such as images, colors, strings, styles and more in the project's `res/` folder. This externalization lets the developer more easily provide alternatives for different screen sizes, screen resolutions, or internationalization of strings [28]. The developer can also maintain these resources separately from the source code and XML files where they are used. For example, the developer can refer to a string resource from the Java with `R.string.helloworld` and from XML files with `@string/helloworld` instead of using, for example, a constant string value in the Java source.

2.3 Android Libraries

2.3.1 Library Artifacts

Android Libraries were originally published in the JAR (Java ARchive) format. A JAR file is essentially an aggregate of multiple files into a single file that is typically compressed to save size [58]. Today, it is commonly used to distribute Java executable files, plain Java libraries, Android libraries, or simply to archive data into one file. However, the JAR file cannot include Android 'resources' (see Section 2.2.4), which is a severe restriction on the library developer.

More recently, Android has introduced the AAR (Android ARchive) format for distributing Android libraries. The AAR format is quite similar to the JAR format in that it is based on the zip file format. However, it opens up the door to more types of Android libraries such as those based on applications components described in Section 2.2.2 since it can contain Android resources, assets, NDK shared libraries and the Android manifest [19]:

```
/AndroidManifest.xml (mandatory)
/classes.jar (mandatory)
/res/ (mandatory)
/R.txt (mandatory)
/assets/ (optional)
/libs/*.jar (optional)
/jni/<abi>/*.so (optional)
/proguard.txt (optional)
/lint.jar (optional)
```

Listing 2.1: Contents of AAR (Android ARchive).

Although a developer can put together scripts to build either a JAR or AAR file containing their Android library, I would strongly discourage anyone from doing so. Gradle offers powerful features to automate the build of either a JAR or AAR. Anyone building an Android specific library, as opposed to a plain Java library, should be using the AAR format today unless they still need to support Eclipse users since Eclipse does not directly support the AAR format.

2.3.2 Publishing Android Libraries

The next step after creating a library is to share the library either to the general developing public or internally with others in your organization.

Since a JAR or an AAR is just a file, a primitive way to share the library is to publish it online and allow others to access it via http or ftp or to copy it somewhere to a shared file system. However, this technique puts the task of library maintenance on the developer, and it also makes it much more difficult for others to find libraries and manage library versions. When using Maven repositories, the individual developer does not need to download JARs and AARs explicitly nor manage them directly. The developer simply needs to point the Gradle scripts to a Maven repository, specify the needed libraries, and Maven will manage the dependencies.

Maven has two types of repositories: local and remote. A local repository is essentially just a local file system cache of library artifacts that have been downloaded already and any artifacts a developer has generated that have not yet been released. A remote repository is simply any repository that is not local. It can, for example, be located online and accessed via protocols like http, or it can be located in a company's internal, private file system [53].

An organization or individual can set up their own remote Maven repositories in their local file system, on their own servers, or on a hosted site. JCenter [2], hosted by Bintray, offers free hosting for open source software (OSS) libraries and paid hosting for closed source libraries. For example, these snippets of Gradle build scripts handle specifying a dependency on Android's app compatibility library and telling Gradle to find that repository at JCenter:

```
repositories {
    jcenter()
}
```

```
dependencies {
    compile 'com.android.support:appcompat-v7:22.2.1'
}
```

Listing 2.2: Compile dependency specification in Gradle.

Chapter 3

Motivation For This Work

3.1 Motivation

I was developing an Android application that is typical of many other Android applications in that it would be building a set of features from function provided by an open source library. My application would provide select GIS functions and rely on Spatialite, an open source library for representing and querying GIS data. In fact, an existing Android application, Geopaparazzi, using the same library and providing a subset of my selected GIS features had already been built and published as open source. Unfortunately, I encountered unexpected issues using the open source library and attempting to use the source code of the existing application. Therefore, I undertook this study on patterns of reusable Android development so that I and other developers may learn from and improve on the existing situation. In this chapter, I will present a case study of my experiences with the open source library, Spatialite, and trying to re-use code from the existing application, Geopaparazzi. Geopaparazzi has many elements common to Android applications, and thus is representative of the challenges in Android of developing code for reuse.

3.2 Case Study: Spatialite on Android

Spatialite [22] is an on open source extension to SQLite for representing spatial data and enabling spatial queries on GIS map data. I found it very easy to install the required Spatialite components in a Docker container with an `apt-get` command. After installation, I followed tutorials online for performing spatial queries. The process to install the tools and get started learning the software took an hour or less. For the interested reader, I posted a Docker container and instructions on how to invoke the container and start running example Spatialite SQL queries at my GitHub site [47].

I found an existing build of Spatialite for Android [8] and a tutorial with example code [16], so I was optimistic that I would be able to just as easily get started on my own Android application built around this library. While following the tutorial, the first obstacle I encountered is that the tutorial showed how to install the components of Spatialite Android using a much older, and significantly different, Android IDE. Spatialite Android is Android native code as Spatialite is written in C, so its components, outlined in Section 2.1.3, are fairly different from normal Android modules. Therefore the project set up and build instructions no longer applied, rendering much of the purpose of the tutorial moot. If Spatialite Android components had been packaged into a JAR file, then it would have been far simpler to use the older tutorial despite some IDE differences since online sources document well how to use a JAR in Android Studio.

In my attempts to get the Spatialite Android components installed, I

followed several false leads from StackOverflow [1] about how to add the native library components to Android Studio. After ten or twenty hours of effort, I found a solution to this problem and was able to create a JAR to contain the Spatialite Android libraries. I will present the details of this in Section 6.1.1.

3.3 Case Study: Geopaparazzi

After I was able to use the Spatialite Android code in a simple example that I published at [46], I began researching how to draw a map from a Spatialite database. I found Geopaparazzi [57], a fairly sophisticated Android application built on Spatialite, which allows the user to do various GIS survey tasks including viewing and manipulating a map. This application is open source and hosted on GitHub [56], so I browsed its code to see if I could re-use parts in my project. Geopaparazzi is organized into five modules, and I was very excited to find one named `geopaparazzispatialitelibrary`. After a discussion online with a contributor [13], I found out that this module includes functions I would need for my application namely reading Spatialite databases and rendering those databases into a viewable map.

In my analysis of and attempts to re-use the code from this module, I made some discoveries about its dependencies and structure as follows:

- It directly includes the source of two components that should be external dependencies:
  - the Spatialite Android JNI files and shared objects

  - the Java source for the JTS library [60]

- It is tightly coupled with other modules in the Geopaparazzi application such as the `geopaparazzilibrary` module.

- `geopaparazzilibrary` groups together too much functionality, such as a logger and support for `Bluetooth`, `SMS`, or `GPS`. These and other functions in the module can and should be separated.

The Spatialite Android files and JTS should be imported as libraries since they are separate components and authored, at least originally, by separate organizations. JTS provides spatial operations and also, important for this project, geometric algorithms [61] which enable drawing of the geometries represented in the Spatialite database.

Since the `geopaparazzispatialitelibrary` module includes the majority of the function I would like to re-use, I spent considerable effort, e.g., ten to twenty hours, trying to make this module self-contained. In parallel, a contributor to Geopaparazzi worked on a branch towards the same goal although he later revoked that work [13]. In net, two people expended many hours in attempts to reduce the coupling between this module and the others. These partial efforts included changing tens of Java source files, several XML files and a handful of Gradle build files.

The `geopaparazzilibrary` module contains twenty-two folders which represents roughly that many pieces of functionality, many of which should be

independent components. For example, it provides custom logger functionality, used by the entire Geopaparazzi application, which could stand quite on its own and should not be bundled with functionality like `Bluetooth`, `SMS`, or `GPS` [56]. Therefore, this module exhibits poor separation of concerns.

3.3.1 Learning From Geopaparazzi

After examining and working with the Geopaparazzi code, I found that it contained some very well thought out and robust code that I would have been glad to re-use. However, it suffered too much from the problems of coupling between its modules and lack of separation of concerns within its modules to be able to do so without extreme effort. Specifically, I suggest the following:

- The JTS and Spatialite Android sources should be tracked in their own repositories and distributed as JAR files in JCenter.
- Geopaparazzi's modules should be carefully separated from each other and packaged as tested, standalone libraries.

Including the Spatialite Android and JTS sources in the application source code makes it difficult to track improvements the Geopaparazzi contributor has made to these libraries and to share those changes with others. In the worst case, the inclusion of this source tempts developers to add dependencies on some other part of Geopaparazzi such as a utility or logger. If this happened, the reusability of these previously standalone components outside of Geopaparazzi would be destroyed. These components should be kept in

their own projects and distributed as JAR files. This work should not be too difficult or risky for the Geopaparazzi application.

The more difficult work will be in separating, packaging and testing the modules in Geopaparazzi to facilitate re-use and leverage the open source community in maintaining and improving these components. This work is best done gradually to reduce risk to the Geopaparazzi application. I would suggest using a coupling analysis to choose the least coupled components to separate out first.

The next natural question regards how the Geopaparazzi project ended up in this state. I suspect JTS and Spatialite Android were not included as separate JARs primarily because the original authors of those libraries did not distribute them this way and also because they did not set these projects up in a way that others could easily contribute such as publishing them on GitHub. It was simply easier for the developer of Geopaparazzi to pull in the code for these components and update that code as needed. Similarly, Geopaparazzi's modules most likely ended up in a state of high coupling because creating a library that is well documented, reusable, and tested requires an order of magnitude more developer effort than is required to create a piece of code that works well enough for a single application. In my professional experience, programming projects that result in the creation of well crafted libraries and applications with low coupling and clear dependencies simply do not happen without a heavy dose of discipline that is seen in well run commercial organizations or in well organized open source projects. In

Geopaparazzi, one developer has contributed 92% of the commits [24], and he clearly did not see the need for the above changes on his own. Nobody suggested such changes until other developers took an interest in the project, which was late enough to make the changes difficult and risky [13].

Chapter 4

Overview Of Android Libraries

An Application Programming Interface (API) is any well-defined interface that that defines the service one software component provides to another software component [20]. The literature refers to these components as libraries, modules, or applications. I will also use these terms interchangeably since it does not change the purpose of this discussion. Since Android code is implemented primarily in Java, an Android library can provide any interface or functionality a Java library can provide. With the new AAR format, an Android library can also provide functionality from Android application components, such as an activity, service, or broadcast provider with the interface provided essentially via intents. In this chapter, I will review the definition and attributes of an API, and provide my suggestions on best practices for developing an API with a focus on Android.

4.1 Attributes of a Well-Written API

4.1.1 Hiding of Implementation Details

A good library hides as many implementation details and data structures as possible from the developer using it [17, 55]. This hiding leads to loose

coupling, a desirable attribute, between the library's code and the code using the library [21]. Loose coupling allows the library to evolve its implementation and internal data representations with much less risk of perturbing the code that uses it. Furthermore, when a developer relies on a well-written library to perform some required function, he or she is introducing another desirable attribute, separation of concerns, into the project that makes the code easier to understand and maintain.

4.1.2 Well Defined and Stable Interface

Gauthier pointed out as early as 1970 that a crucial attribute of a module, or API, is that the inputs and outputs must be well defined [23]: *"At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other [system] modules."* Joshua Bloch pointed out that *"Public APIs, like diamonds, are forever. You have one chance to get it right so give it your best."* [17]. A well-defined API helps the user of the library know what to expect of it. Also, the interfaces of the library must be stable as any changes may break code that is using it.

4.2 Suggested Practices for Developing an API

Many good books are available regarding the best practices for developing a Java API with a particular focus on the nuances of that language. I will not cover those Java oriented practices here so that I may focus on more generic principles that also apply well to developing an Android library. I refer

to the reader to resources like [59] or works by Joshua Bloch.

4.2.1 Develop User Stories as Example Applications First

Bloch [17] emphasizes the importance of developing the user stories for your API before you begin developing the API. These user stories can help you clarify the requirements for your library and iterate quickly on API development. Later on, these user stories can be developed into example applications that will serve as examples of how to use your library, the basis for tutorials, and a starting point for creating tests. Bloch emphasizes that example applications must use your library in the most recommended way, as many other developers will use these as a starting point for creating their own code. Therefore, these example applications and user stories must be maintained as your library evolves.

4.2.2 Work With Others

Consult with others early on while developing your use cases and your API [17]. Other developers will be able to question your assumptions and see things you have missed. Even if you are developing an API largely on your own, you will benefit greatly from the feedback of others who may be using your library and its documentation or reviewing your code. Please see Section 3.3.1 for examples of problems that might have been avoided had the developer been collaborating with others earlier on.

4.2.3 Test Early and Thoroughly

A library must also be thoroughly tested to maintain not only the stability of the API but also a stability of function. For code with minimal dependencies on the Android operating system, JUnit [6] tests are appropriate and recommended. A mocking framework, such as Mockito, can be used to mock parts of the Android system as necessary. JUnit tests are highly efficient as they can be run on the local development environment, saving the time of starting up an Android device or emulator [11]. However, the developer must be aware that these unit tests will run with the development machine's JVM and not the Android virtual machine. If a library provides any user interface (UI), then UI testing is necessary. These tests simulate user activity without requiring the time of a human tester [15]. The Android development environment also supports testing complex interaction events between components or with the operating system [12]. This testing must also be run automatically as part of the development and release process to be effective. Writing and maintaining these tests will be a significant investment of time and effort for the development team. Including tests early in the development reduces this burden. Although a detailed investigation of Android testing is beyond the scope of this paper, testing is a critical attribute of a well-maintained library.

4.2.4 Document

Bloch [17] asserts that a good API should be self-documenting such that a user rarely needs to consult documentation to read or write code written

against it. Therefore, your APIs should be sensibly named and generally do what a reader would expect of them. I would also assert that if you want others to use your library, then you should document it well and provide some good examples of using it. I will discuss in Section 4.3.3 some documentation challenges unique to Android.

4.2.5 Publish

You will need to distribute your library, its documentation and optionally its source for use by others. While you can simply publish your library archives on a website, you will be taking on tasks like managing versions and maintaining a website while making it difficult for others to find your library. I would recommend using JCenter or MavenCentral as discussed in Section 2.3.2 to publish any open source libraries as these services make it very easy for someone else to use your library. If your library can be used by any Java code, then you should publish a JAR. However, if your library is specific to Android projects, you should take advantage of the newer and more capable AAR format. Finally, if your project is open source, your source code can be easily published online via services like GitHub.com or BitBucket.com. These services provide useful features for the open source project such as discussion boards and more.

4.3 Challenges in Creating Android Libraries

4.3.1 IDE Obstacles

Android Studio defaults to helping the user create applications and modules within applications. When a developer adds a module to an application, it is all too likely that the module will be highly coupled to other parts of the application and represents no more than code organization for the convenience of the developer in finding source files. It is only when a module is contained in its own project that the temptation to couple its code to code in adjacent modules is removed and the developer is forced to make the interfaces to the module explicit and to publish the module as a library. In order to make a module that exists alone, in its own project outside of an application, the developer must take extra steps in Android Studio and in Gradle. I will outline those steps in Section 5.3.1.

4.3.2 Challenges around Automatic Archive Publishing

In Section 2.3.1, I briefly explained the JAR and AAR artifacts, which are the older and newer formats for an Android library. While a developer can write custom scripts to create a JAR or AAR file, I would recommend strongly against that. Instead, a developer should use Gradle and its plugins to automate the build and publication of library archives.

Although Gradle is designed to be easy for Java developers to understand [41], I found that completing, or at least reading, some of the tutorials at [37] to be invaluable later on when I was customizing build scripts to pro-

mote libraries. These tutorials will cover the plugins that add tasks to your projects to handle routine events and provide an introduction to customizing existing tasks and creating new tasks.

Configuring your Gradle file to create a library file or application is so easy, thanks to the Java and Android plugins, that few novice developers truly need to understand the details of how it is accomplished. However, a developer must configure the build scripts much more extensively in order to promote a library either to a local file system or online repository I will cover examples, in detail, of the Gradle configurations required in Chapter 5.

4.3.3 Lack of Documentation Standards Beyond Javadoc

The natural and usual way to document the Java interface for your library is to use the well-known Javadoc standard [5]. Javadoc embeds documentation within the source code, increasing the likelihood documentation will stay current as code evolves. With the advent of the AAR, Android libraries can now support interfaces beyond regular Java such as intent interfaces to components like activity, service, and broadcast provider. The aspects of the intent interface to components like activity, service and broadcast provider can, after a fashion and also optionally, be documented via `<intent-filter>`'s in the Android manifest. However, that documentation, if you can call it that, is decoupled from the code in the library that actually takes in the intent messages. A library owner's primary recourse seems to be documenting these interfaces on a web page or in tutorials for the library. I would suggest that

documentation for parameters retrieved from the Intent and returned with an Intent is embedded in the source code, in a manner similar to Javadoc. The burden will remain on the developer to keep that documentation up to date and complete. However, if he or she keeps the source code that retrieves parameters from the Intent or adds return data to an Intent in a single place, then this burden will be similar to the burden of maintaining Javadoc.

Chapter 5

My Best Practices for Android Library Development

In this chapter, I will present working examples of Android and Java libraries. The functionality of my examples will be extremely simple so that I may focus on creating exemplary build scripts, documentation, and usage examples. During my research, I did not find any complete examples of how to build, publish and use an Android library, so a major contribution of this chapter will be the example Java and Android libraries hello-planet, planetary-system, and android-planetary-viewer and the Android application example view-solar-system.

5.1 Gradle is Declarative and Uses Build By Convention

Gradle is declarative [35], so its scripts rarely include information on *how* to perform a task, i.e., they rarely include explicit compiler commands as many other build scripts do. Rather, Gradle scripts declare *what* should be built, such as a Java project. The Java plugin for Gradle includes tasks that know how to compile Java code, package a JAR and build and run tests [40]. You also declare dependencies in a concise format as in Listing 2.2.

Gradle also places a heavy emphasis on 'build by convention' [35]. As long as a project is structured in line with existing conventions, Gradle will automatically find the source files and resources for it. However, a Gradle user can easily extend or customize Gradle for a project, so the project layout is not limited by convention assumptions or existing functionality. For example, if your project is not using the conventional directory layout, then you can configure Gradle for your project's source layout as introduced in [36]. You can also write your own custom plugin or extend existing tasks as introduced in [38, 39].

In the following sections of this chapter, I will include step-by-step instructions for creating, testing, and publishing Java and Android libraries. I will rely almost exclusively on conventional project structure and published plugins both to encourage readers to stick with convention as much as possible and to keep the examples simple and easy to follow.

5.2 Creating Plain Java JAR Libraries

In this section, I will provide two example plain Java libraries. The first example will be a simple Java library that has no dependencies. The second example will be another simple Java library that includes a dependency on the first example library.

Although I do emphasize the importance of documentation and testing in this paper, I will not specifically review the Java source, Javadoc or JUnit source for these examples. Java documentation and unit testing is well docu-

mented. My focus in this section will be on the complete Gradle configuration as that will be foundational for the next examples.

5.2.1 Hello Planet: No Dependencies

I created a simple, plain Java library called 'Hello Planet' [42] to use as a starting point for my examples. In the next sections, I will go through the required configuration to create a Java library step by step.

5.2.1.1 Gradle Configuration: Create a Java JAR

The first step in creating a Java library is to create the source, the unit tests, and finally the Gradle configuration to build a JAR file locally in the project. The Java Gradle plugin includes the task `build`, which runs all tasks necessary to build a JAR file including compilation and execution of unit tests. The successful completion of the command `gradle build` then produces a JAR file that has passed all existing unit tests.

The code at Listing A.1 configures the build to:

1. Declare a Java project
2. Configure the JAR file with version numbers and a name that follows convention
3. Indicate unit test dependency on JUnit and where Gradle may find that dependency
4. Configure the JAR with an appropriate manifest

5.2.1.2 Gradle Configuration: Create Javadoc and Sources JAR

A Java library should be distributed with Javadoc, and open source libraries should be distributed with the source. Listing A.2 configures the tasks creating the Javadoc and sources JAR to be included in the build of archives.

5.2.1.3 Gradle Configuration: Publish Library Locally

This library can be more easily used by example applications if it is published to a local Maven repository. The code at Listing A.3 supports publishing the source, Javadoc and library JAR to a Maven repository in the local file system when you invoke `gradle uploadArchive`. I noticed when observing the Gradle output that the `test` and `build` tasks were not being invoked with `gradle uploadArchive` as they were with `gradle build`. Since I am very interested in tests always passing and all artifacts being up to date before publishing any JAR files, I added the line at Listing A.4 that automatically includes all `check` and `assemble` tasks [36]. Gradle tracks the freshness of builds much like other build systems, so adding this dependency does not add extra work unless underlying files are out of date.

The hello-planet library is now ready to use in examples developed on the local file system. Publishing locally is useful to the original library developer or developers as it is faster than publishing to an online repository. Also, a solo developer in particular can play fast and loose with version numbers and breaking changes when he or she is iterating locally.

5.2.1.4 Gradle Configuration: Publish Library Online

Once the hello-planet library is ready for review and testing by other developers, you should publish it online. The code at Listing A.5 supports publishing the library to JCenter when you invoke `gradle bintrayUpload`. Here, I also added a dependency on the `build` task in the last line to force unit tests to pass before publication.

On a successful `gradle bintrayUpload`, your library will be awaiting publication on JCenter. On the JCenter website, you can choose to 'Publish' or 'Discard' based on how happy you are with the library as published. Once you publish, you can view the files associated with your library on JCenter, and any other developer will be able to download these files for their own use. Please note that JCenter discourages you from deleting published libraries.

You may view the complete build.gradle file online [43].

5.2.2 Planetary System: Handling Dependencies

I created the Java library 'Planetary System' to illustrate a plain Java library that is dependent on another library. This library consists of Java code using the hello-planet API along with the appropriate JUnit tests and Javadoc.

5.2.2.1 Gradle Configuration: Include a Dependency

In the 'planetary-system' library, the 'hello-planet' library is a compile time and runtime dependency. This dependency is specified on Line 2 in

Listing A.6. Please note that Gradle will automatically include all compile time dependencies into the runtime dependencies.

Since I am using a library published only to my personal space and not indexed into the wider JCenter, I need to specify the path to my Maven repository published on JCenter as on Lines 2 to 4 in Listing A.7. I also encourage a developer to publish a library under development locally in order to quickly iterate with example usages of that library. The code in Listing A.8 specifies how to pick up the dependent library from the local file system.

My 'planetary-system' library is now compiled and producing a JAR. I can choose between using the local Maven repository or the JCenter Maven repository to retrieve the hello-planet dependency. In order to publish this library locally and online, I must include the previously documented Gradle configuration, changing only names as necessary. Please review the sections on hello-planet for a reminder of these steps. The complete Gradle file for planetary-system is available online [44].

5.3 Creating Android AAR Libraries

5.3.1 Creating and Publishing Android Libraries

In this section, I will review how to create and publish an Android library using Android Studio and Gradle. My examples will use the Android library viewplanetsactivity, which has a dependency on my previously presented planetary-system Java module.

5.3.1.1 Android Studio and Gradle: Create an Android Studio Project with Library

The first steps to create an Android project that will produce a library are:

1. Create a new Android project with a module that will become your library.
2. Remove the default application component.
3. Verify that your project creates an AAR, and adjust the Gradle configuration as needed.

From Android Studio, create a new project with sensible defaults for your target usage. Next, create a new module, which will become your library, in the project. Android Studio automatically created an application component during project creation, so remove this component with `rm -rf app` at the terminal. Also, remove the reference to `app` from the project `settings.gradle` file so that it now contains only the name of your module.

If after you build your project, you see an AAR file under `<module> /build/outputs/aar` then your project is set up correctly. If it is not present, please compare your project's `build.gradle` to the example files I have published online [50], or review current Android Studio and Gradle documentation.

5.3.1.2 Gradle Configuration: Publishing an Android Library Locally

I again need to publish my newly created Android library to a local Maven repository so that I may begin iterating on a usage example. To do this I will need to:

1. Create sources and Javadoc JAR files as in Listing A.9.
2. Configure the uploadArchives task to publish the library to a local Maven repository as in Listing A.10.

I was lucky to find a blog entry at [18] with clear examples of how to publish an Android library to a local Maven repository as there are a few more configuration requirements for the AAR than for the JAR. As in the hello-planet example, I added a dependency on `build` to the task `uploadArchives` to make sure that all publishable artifacts are up to date and all tests pass. I have published the module's `build.gradle` online [49] as it stands after this step. The project's `build.gradle` has not changed.

5.3.1.3 Gradle Configuration: Publishing an Android Library Online

Fortunately, the Gradle bintray plugin supports publishing Android archives as well as Java archives. Unfortunately, I could not directly apply the examples that bintray provides for publishing an AAR [10]. I spent several hours working on the Gradle configuration to publish my Android library

AAR to JCenter. I finally found a solution by mixing together resources from the Gradle bintray plugin examples with an example I found on GitHub at [14]. My solution to this is at Listing A.11. On a successful run of `gradle bintrayUpload`, you will again have the option to publish or discard on the JCenter website.

5.3.2 ViewPlanetsActivity: An Intent-Based Library

I created an Android library built around an activity, which is an example of an Android application component started by an `Intent` as explained in 2.2.2. Thanks to the relatively new AAR format, a developer may now package these type of components into an Android Library. This raises, at the very least, some interesting questions about how to document the interface to this type of library. I will first review how an intent accepts and returns inputs and then discuss options for documenting this interface.

5.3.2.1 Intents: Passing Parameters from Application to Intent

When an application starts an activity via intent, it can include information in the intent such as the action to perform, data to act on and its type, category, extras, or the name of the component to start [27]. Other than the name of the component, the information in the intent is essentially parameters for the activity. When the name of the component to start is specified, this is an explicit intent. Otherwise, the intent is implicit. When an intent is explicit, the Android operating system will not verify any of the intent infor-

mation that is passed along to the activity. The activity code itself will need to verify, at runtime, the information in the intent. In order to be started implicitly, an activity must declare intent filters in the manifest specifying the information it expects, and the operating system will find an activity to handle the request that matches the information in the intent. The Android operating system therefore does offer some checking of the data, action, and category parameters when an intent is started implicitly. However, it never checks extras [27].

Extras are a very versatile way to pass parameters to an activity with the intent. These are key value pairs where the key is always a string and the value can be any one of a variety of types [30]. The Android framework provides string constants for common keys, but a developer may also define custom keys. The convention when defining custom keys is to use a string constant defined by the activity as shown in Listing A.12. Listing A.13 shows an example of the invoking application passing a simple string parameter and an array of strings parameter using these string constants as extras. The receiving activity then extracts the extra parameters from the intent via the same string constant keys. The burden is on the receiving activity to verify, at runtime, that the value data passed with the key is the same data type it is expecting. In Listing A.14, the library's activity is retrieving the parameters by the same constants and checking for their validity before using them.

Passing parameters via intent is very flexible but has certain disadvantages when you consider that the interface to a library should be clearly docu-

mented. Namely, the set of required and optional parameters and their types are not clearly or fully delimited outside of the activity's source code nor is there any comprehensive, automatic checking of these parameters. Therefore, extra work is placed on the Android library developer to write runtime checks for required and optional parameters checking not only for their presence but also that the correct data types have been passed. Also, since there is no standard way to document these parameters, it is up to each library developer to create his or her own mechanism to document these for users or else users will be forced to read the library source code. With a Java API the presence and type of parameters will be automatically checked at compile time. Also, if an a Java API developer embeds and maintains standard Javadoc documentation in the source code, then easy to read HTML documentation will be available to those using the API.

5.3.2.2 Intents: Returning Values

The application starting an activity via an intent can also receive data back from the activity. However, the activity must be designed to send back a result via an intent, and it is up to the code receiving the result to correctly extract the data from the intent. In Listing A.15, the calling application indicates that it wants a result back from the launched activity, Listing A.17 sends the result, and Listing A.16 receives the result.

Returning data via intent has very similar disadvantages documented in the previous section for passing data via intent. Namely, there is no automatic

checking to verify that an activity started for a result will send a result at all nor of any particular type. There is also not any standard for documenting this interface.

5.3.2.3 Intents: Documenting Inputs and Outputs

In the prior two sections, I discussed passing data, e.g., parameters, to and from an activity and reviewed the weaknesses of this interface with respect to lack of automatic, comprehensive checking of parameters and documentation standards. Those weaknesses, in summary, are:

- No automatic compile or runtime type or correctness checking on all parameters passed to the activity via intent.
- No automatic compile or runtime checking of return values expected from an activity.
- No standard method of documenting the above.
- Some parameters can be defined via an intent filter, if that is present. Since the intent filter only covers some types of parameters, is optional, is used only when starting an activity implicitly, and can be considered source of the activity, this cannot be considered complete, user friendly documentation.
- Minimal runtime checking of some parameters only when starting an intent implicitly.

- Custom keys for extras are, at best, string constants defined by the activity. With IDE auto completion, a user may discover some of these keys, but he or she will not know what values are expected.

- The burden is on the activity to check, at runtime, not only for the presence of required or optional parameters but also to verify the type of those parameters with `try...catch` statements. Otherwise, incorrect parameters may result in runtime exceptions.

In this work, I am considering the case where the activity is included in a library. However, the above weaknesses apply whether the activity is in a library, included in an application source, or in another application. In comparison to the Java API, where the Java compiler will automatically check parameter correctness and where Javadoc is the standard means of documenting interfaces, the interface to an activity is very weakly verified and documented.

Chapter 6

Conclusions

6.1 Summary of My Contributions

This paper presents my patterns and best practices for developing an Android library in the interest of promoting Android code re-use, and it points out aspects of the Android development ecosystem that do not comply with established best practices for library development. These shortcomings are as follows:

- A lack of automatic, comprehensive checking and documentation of parameters for intent interfaces weakens this recently available Android library capability.
- The Android IDE is oriented towards standalone application development and not library development, which misses an opportunity to encourage Android developers to create more reusable code.

The former shortcoming is a material hindrance to the development and adoption of Android libraries that use the intent interface. The latter is more minor, but it will be an obstacle to developers more novice to Android or general software development.

Other contributions in this paper include a review of select Android concepts, a review of established best practices for library development, and a case study of an existing Android library and application with code that proved very difficult to re-use.

6.1.1 Preparing the Spatialite Android JAR

As I reported in Section 3.2, I faced a great deal of difficulty early on when trying to use the unpackaged components of the Android port of Spatialite. Therefore, I packaged the Spatialite Android code as a JAR so that others did not have to repeat this. A summary of this effort follows.

The Spatialite JNI source declares itself via `package jsqlite;` placing it in the global scope, which is against Java conventions. Android Studio does not allow you to create a project with a global namespace during the set up for a new project. As a relatively novice Java or Android developer would do, I put in some value for the namespace. Although my new project would compile and build a JAR, code attempting to use this JAR would not compile. After many hours of debugging, I found that I had to modify the module configuration so that it would be in the global namespace as illustrated in Listing A.18.

After addressing the namespace issue, I worked to package the Spatialite Android code into a library for easier distribution. Although I have previously advocated that new Android libraries should publish in the AAR format to take advantage of new features there, the Spatialite Android code was created well before these new Android features and it does not need any of

those new features. Therefore, for simplicity and to facilitate greater adoption, such as adoption by developers who remain on Eclipse, I packaged Spatialite Android into a JAR and published it online with these major steps:

1. Create a new Android project with a module named `spatialite-android`.
2. Copy the Java JNI source and `libjsqlite.so` shared objects to the new module.
3. Configure the module's Gradle to create a JAR and publish it online.

First, I created a new Android project and a module within that project named `spatialite-android` following the steps I presented in Section 5.3.1.1. I copied the `libjsqlite.so` objects and JNI source over from the Geopaparazzi project into my module at `src/main/java/jniLibs` and to `src/main/java/jsqlite` respectively. I then modified the module's `AndroidManifest.xml` as in Listing A.18.

Listing A.19 [63] shows the Gradle configuration required to build NDK components into a JAR. Beyond this, the Gradle configuration required to build and publish the JAR to JCenter are similar to that presented in Section 5.3.1. I have published the project online [3] and an example of using this JAR at [46].

6.2 Future Work

Mobile development is growing very rapidly in importance in the software development community as outlined in Chapter 1. However, the Android development environment has several shortcomings in the area of library development as summarized in Section 6.1. These issues are somewhat understandable since third-party development started for Android applications only since 2008 [4], which is recent compared to the long-standing Java and C++ development communities. However, these shortcomings must be addressed for the Android development community to be able to take full advantage of reusable software components. The following is a summary of what I consider to be the most important issues to address in the Android development environment:

- Automatic and comprehensive checking of the intent interface to an application component
- A documentation standard for the above so that library developers may communicate what parameters are required or optional and users of the library may easily know what is expected

I would also suggest extending the examples I produced for the paper in the following ways:

Testing Although testing is a key component of creating a library, I did not address this topic since scope did not allow.

Schema Validation Schema validation is another Android library feature enabled by the AAR that I found almost by accident. Similarly to the intent interface, this feature lacks compile or build time checking and any documentation standards.

Finally, I would suggest further case studies of open and closed source Android libraries and applications, beyond my study of a single library and application, to gain more insight into common patterns and pitfulls of Android development.

6.3 Source Code and Demonstration Resources

I have created and published the following examples to illustrate points in this work:

- A demonstration of using Spatialite tools from a Docker container to perform example queries is available at [47].
- A demonstration of creating a Spatialite-Android JAR file from NDK components is available [3].
- A demonstration of using the Spatialite-Android JAR file I created is available at [46].
- My simple Java library, hello-planet, is available at [42].
- My Java library, planetary-system, which depends on hello-planet is available at [45].

- My example Android library, android-planetary-viewer, which illustrates how to package an AAR including dependencies is available at [51].
- My example Android application, view-solar-system, which illustrates how to use and communicate with android-planetary-viewer is available at [48].

6.4 Acknowledgements

My work would not have been possible without the following open source work:

- Geopaparazzi, an open source application built on Spatialite [56]
- libjsqlite-spatialite-android, builds the Spatialite Android components for Geopaparazzi [7]
- Gradle bintray plugin examples for building an AAR [10]

Appendices

Appendix A

Code Listings

A.1 Hello Planet Gradle Listings

```
//declare this as a Java project
apply plugin: 'java'

//customize properties
version = '1.1.0'
sourceCompatibility = 1.7
targetCompatibility = 1.7
jar.baseName = 'hello-planet'

//fetch any dependencies from jcenter
repositories {
    jcenter()
}

//declare dependencies
dependencies {
    testCompile group: 'junit', name: 'junit', version:
        '4.+'
}

//declare required manifest attributes for JAR file
jar {
    manifest {
        attributes 'Implementation-Title': 'hello-
           planet example',
            'Implementation-Version': version,
            'Main-Class': 'com.example.HelloPlanet'
    }
```

```
}
```

Listing A.1: Gradle: Create Java JAR.

```
//add tasks to create sources JAR
task sourcesJar(type: Jar, dependsOn: classes) {
    classifier = 'sources'
    from sourceSets.main.allSource
    //set basename of sources jar to be same as regular
        jar
    baseName = jar.baseName
}

//create javadoc JAR
task javadocJar(type: Jar, dependsOn: javadoc) {
    classifier = 'javadoc'
    from javadoc.destinationDir
    //set basename of javadoc jar to be same as regular
        jar
    baseName = jar.baseName
}

//add javadoc and sources JAR to artifacts
artifacts {
    archives sourcesJar
    archives javadocJar
}
```

Listing A.2: Gradle: Locally Promote Java JAR.

```
def localReleaseDest = "/Users/kristina/localRepos"
//publish to Maven repo on local filesystem
uploadArchives {
    repositories {
        flatDir {
            dirs "file://${localReleaseDest}"
        }
    }
}
```

Listing A.3: Gradle: Locally Promote Java JAR.

```
tasks.uploadArchives.dependsOn(build)
```

Listing A.4: Gradle: Introduce Dependency on Build Task.

```
//new for online promote.
//This must be at the top of the file
plugins {
    id "com.jfrog.bintray" version "1.3.1"
}
//new plugin for online promote
apply plugin: 'maven-publish'

publishing {
  publications {
        //name 'MyPublication' corresponds to '
           publications'
        //setting in bintray section
    MyPublication(MavenPublication) {
      from components.java
            groupId 'example.com'
            artifactId jar.baseName
            version version

            //add sources and javadoc jars to upload
            artifact(sourcesJar)
            artifact(javadocJar)
    }
  }
}

//bintray / jcenter configuration
//NOTE: I set credentials in my SHELL environment
//rather than here for security reasons
//There is more than one way to achieve this security.
//use command: gradle bintrayUpload
//to upload the archives
bintray {
    user = System.getenv('BINTRAY_USER')
    key = System.getenv('BINTRAY_KEY')
    //links to 'publishing' section above:
    publications = ['MyPublication']
```

```
    //package info. repo, name, licenses, vcsUrl are
       required
    pkg {
        repo = 'kh-examples'
        name = jar.baseName
        licenses = ['Apache-2.0']
        vcsUrl = 'https://github.com/kristina-hager/
           hello-planet.git'
        version {
            name = version
        }
    }
}

//again, add a dependency on :build to :bintrayUpload
tasks.bintrayUpload.dependsOn(build)
```

Listing A.5: Gradle: Promoting AAR to JCenter.

A.2 Planetary System Gradle Listings

```
1 dependencies {
2     compile group: 'example.com', name: 'hello-planet',
          version: '1.1.0'
3     testCompile group: 'junit', name: 'junit', version:
          '4.+'
4 }
```

Listing A.6: Gradle: Configuring Dependencies.

```
1 repositories {
2     maven {
3         url  "http://dl.bintray.com/kristina-hager/kh-
              examples"
4     }
5     jcenter()
6 }
```

Listing A.7: Gradle: Include dependency on JCenter.

```
repositories {
    jcenter()
```

```
    //use below if you want to test jars locally,
    //e.g. to test before uploading
    flatDir {
        dirs "file://${localReleaseDest}"
    }
}
```

Listing A.8: Gradle: Specify Local Maven Repository in Dependencies.

```
task androidJavadocs(type: Javadoc) {
    source = android.sourceSets.main.java.srcDirs
    classpath += project.files(android.getBootClasspath
       ().join(File.pathSeparator))
}

task androidJavadocsJar(type: Jar, dependsOn:
   androidJavadocs) {
    classifier = 'javadoc'
    from androidJavadocs.destinationDir
}

task androidSourcesJar(type: Jar) {
    classifier = 'sources'
    from android.sourceSets.main.java.srcDirs
}

artifacts {
    archives androidSourcesJar
    archives androidJavadocsJar
}
```

Listing A.9: Gradle: Create Android Sources and Javadoc Jars.

```
apply plugin: 'maven'

//ext is a gradle closure allowing the declaration of
   global properties
ext {
    PUBLISH_GROUP_ID = 'com.android_space'
    PUBLISH_ARTIFACT_ID = 'viewplanetsactivity'
    PUBLISH_VERSION = '1.0.0'
```

```
}

def localReleaseDest = "/Users/kristina/
   localAndroidRepos"

//publish to Maven repo on local filesystem
uploadArchives {
    repositories.mavenDeployer {
        pom.groupId = PUBLISH_GROUP_ID
        pom.artifactId = PUBLISH_ARTIFACT_ID
        pom.version = PUBLISH_VERSION
        //Add other pom properties here if you want (
           developer details / licenses)
        repository(url: "file://${localReleaseDest}")
    }
}

//require :build tasks such as :test,:assemble before
   upload archives
tasks.uploadArchives.dependsOn(build)
```

Listing A.10: Gradle: Promote Android AAR to Local Maven Repository.

```
apply plugin: 'com.jfrog.bintray'
apply plugin: 'maven-publish'

def siteUrl = 'https://github.com/kristina-hager/
   android-planetary-viewer'
def gitUrl = 'https://github.com/kristina-hager/android
   -planetary-viewer.git'

bintray {
    user =  System.getenv('BINTRAY_USER')
    key = System.getenv('BINTRAY_KEY')

    publications = ['MyPublication']
    pkg {
        repo = 'kh-examples'
        name = PUBLISH_ARTIFACT_ID
        desc = 'viewplanetsactivity aar publishing
           example'
```

```
        websiteUrl = siteUrl
        vcsUrl = gitUrl
        licenses = ['Apache-2.0']
        labels = ['aar', 'android', 'example']
        publicDownloadNumbers = true
        version {
            name = PUBLISH_VERSION
        }
    }
}

//adding maven information
publishing {
    publications {
        //name 'MyPublication' corresponds to '
           publications' setting in bintray section
        MyPublication(MavenPublication) {
            artifact "${project.buildDir}/outputs/aar
               /${project.name}-release.aar"
            groupId PUBLISH_GROUP_ID
            artifactId PUBLISH_ARTIFACT_ID
            version PUBLISH_VERSION

            //add sources and javadoc jars to upload
            artifact(androidSourcesJar)
            artifact(androidJavadocsJar)
        }
    }
}
//again, add a dependency on :build to :bintrayUpload
tasks.bintrayUpload.dependsOn(build)
```

Listing A.11: Gradle: Promote Android AAR to JCenter Maven Repository.

```
//class variables
public final static String SYSTEM_NAME = "com.
   android_space.viewplanetsactivity.SYSTEM_NAME";
public final static String PLANETS_ARRAY = "com.
   android_space.viewplanetsactivity.PLANETS_ARRAY";
```

Listing A.12: Java: Typical declaration of Intent Parameter Strings.

```
Intent intent = new Intent(this, com.android_space.
   viewplanetsactivity.ViewPlanetsActivity.class);
intent.putExtra(ViewPlanetsActivity.SYSTEM_NAME, "mini
   solar system");

Bundle planetBundle = new Bundle();
planetBundle.putStringArray(ViewPlanetsActivity.
   PLANETS_ARRAY,
    new String[]{"venus", "mercury", "earth", "mars"});
intent.putExtras(planetBundle);

startActivityForResult(intent,
   SOLAR_SYSTEM_RESULT_REQUEST);
```

Listing A.13: Java: Pass parameters to an Intent.

```
//code in onCreate
Intent intent = getIntent();
String systemName = intent.getStringExtra(this.
   SYSTEM_NAME);
Bundle planetsBundle = this.getIntent().getExtras();
if (systemName != null && planetsBundle != null) {
    buildPlanetarySystem(systemName, planetsBundle.
       getStringArray(this.PLANETS_ARRAY));
} else if (systemName != null) {
    buildPlanetarySystem(systemName, null);
}
```

Listing A.14: Java: Get parameters from an Intent.

```
startActivityForResult(intent,
   SOLAR_SYSTEM_RESULT_REQUEST);
```

Listing A.15: Java: Ask for Result from an Intent.

```
@Override
protected void onActivityResult(int requestCode, int
   resultCode, Intent data) {
    if (requestCode == SOLAR_SYSTEM_RESULT_REQUEST) {
        if (resultCode == RESULT_OK) {
            Bundle bundle = data.getExtras();
```

```
            String result = bundle.getString(Intent.
               EXTRA_TEXT);
            Log.i(TAG, "got result: " + result);
        }
    }
}
```

Listing A.16: Java: Get Result from an Intent.

```
public void sendActivityResult(View view) {
    String systemName = (planetarySystem != null) ?
       planetarySystem.getName() : "no planetary system
       !";
    Intent result = new Intent("com.android_space.
       RESULT_ACTION");
    result.setAction(Intent.ACTION_SEND);
    result.putExtra(Intent.EXTRA_TEXT, systemName);
    result.setType("text/plain");
    setResult(Activity.RESULT_OK, result);
    finish();
}
```

Listing A.17: Java: Send Result from an Intent.

```
<manifest xmlns:android="http://schemas.android.com/apk
   /res/android"
    package="spatialite_android">
</manifest>
```

Listing A.18: Module spatialite-android Manifest

```
ext {
    PUBLISH_GROUP_ID = 'com.example'
    PUBLISH_ARTIFACT_ID = 'spatialite-android'
    PUBLISH_VERSION = '1.0.0'
}

task androidJar(type: Jar, dependsOn: ['assemble']) {
    group 'Build'
    description 'spatialite-android JNI jarFile'
    from zipTree('build/intermediates/bundles/release/
       classes.jar')
```

```
    from(file('src/main/java/jniLibs')) {
        into 'lib'
    }
    //give jar a jar-like name with version number
    baseName = PUBLISH_ARTIFACT_ID + '-' +
       PUBLISH_VERSION
}
```

Listing A.19: Gradle Configuration to Build Android NDK Jar

Bibliography

[1] http://stackoverflow.com/. [Online; accessed 2015].

[2] https://bintray.com/bintray/jcenter. [Online; accessed 2015].

[3] https://github.com/kristina-hager/spatialite-android-database-driver/. [Online; accessed 2015].

[4] Android version history. https://en.wikipedia.org/wiki/Android_version_history#Android_1.0_.28API_level_1.29. [Online; accessed Nov 11, 2015].

[5] Javadoc Tool. http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html. [Online; accessed Nov 16, 2015].

[6] JUnit Home Page. http://junit.org/. [Online; accessed Nov 16, 2015].

[7] libjsqlite-spatialite-android. https://github.com/geopaparazzi/libjsqlite-spatialite-android. [Online; accessed Aug-2015].

[8] Splite-Android. https://www.gaia-gis.it/fossil/libspatialite/wiki?name=splite-android. [Online;

accessed 2015].

[9] Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013. http://www.gartner.com/newsroom/id/2408515, 2013. [Online; accessed Nov 11, 2015].

[10] bintray-examples/gradle-aar-example/build.gradle. https://github.com/bintray/bintray-examples/blob/master/gradle-aar-example/build.gradle#L115, 2014. [Online; accessed Aug-2015].

[11] Building Local Unit Tests. http://developer.android.com/training/testing/unit-testing/local-unit-tests.html, 2015. [Online; accessed 2015].

[12] Creating Unit Tests. http://developer.android.com/training/activity-testing/activity-unit-testing.html, 2015. [Online; accessed 2015].

[13] Example (Current) "hello world" usage of this library / libjsqlite? https://github.com/geopaparazzi/libjsqlite-spatialite-android/issues/12, 2015. [Online; accessed 2015].

[14] hellocharts-android/hellocharts-library/build.gradle. https://github.com/lecho/hellocharts-

android/blob/master/hellocharts-library/build.gradle, 2015. [Online; accessed Aug-2015].

[15] Testing UI Components. http://developer.android.com/training/activity-testing/activity-ui-testing.html, 2015. [Online; accessed 2015].

[16] Andrea Antonello. Spatialite on Android: a quick tutorial. https://www.gaia-gis.it/fossil/libspatialite/wiki?name=spatialite-android-tutorial, 2012. [Online; accessed 09-Oct-2015].

[17] Joshua Bloch. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

[18] Blundell. [TUT] Locally release an Android Library for JCenter or Maven Central inclusion. http://blog.blundell-apps.com/locally-release-an-android-library-for-jcenter-or-maven-central-inclusion/, 2014. [Online; accessed Aug-2015].

[19] Android By Code. Building an AAR Library in Android Studio. https://androidbycode.wordpress.com/2015/02/23/building-an-aar-library-in-android-studio/AAR, 2015. [Online; accessed 08-Oct-2015].

[20] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development. *SIGSOFT Softw. Eng. Notes*, 29(6):221–230, October 2004.

[21] Jim des Rivières. Eclipse apis: Lines in the sand. *EclipseCon Retrieved March*, page 2004, 2004.

[22] Alessandro Furieri. Spatialite-Tools-Docker. `http://www.gaia-gis.it/gaia-sins/`. [Online; accessed Aug-2015].

[23] Richard L Gauthier and Stephen D Ponto. Designing systems programs. 1970.

[24] GitHub. Contributors to Geopaparazzi. `https://github.com/geopaparazzi/geopaparazzi/graphs/contributors`. [Online; accessed 10-Oct-2015].

[25] Google. Android: Develop: API Guides: App Manifest. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`. [Online; accessed 05-Oct-2015].

[26] Google. Android: Develop: API Guides: Content Providers. `http://developer.android.com/guide/topics/providers/content-providers.html`. [Online; accessed 05-Oct-2015].

[27] Google. Android: Develop: API Guides: Intents and Intent Filters. http://developer.android.com/guide/components/intents-filters.html. [Online; accessed 05-Oct-2015].

[28] Google. Android: Develop: API Guides: Resources Overview. http://developer.android.com/guide/topics/resources/overview.html. [Online; accessed 05-Oct-2015].

[29] Google. Android: Develop: API Guides: Services. http://developer.android.com/guide/components/services.html. [Online; accessed 05-Oct-2015].

[30] Google. Android: Develop: Reference: Intent. http://developer.android.com/reference/android/content/Intent.html#putExtra(java.lang.String,android.os.Bundle). [Online; accessed 05-Oct-2015].

[31] Google. Android: Develop: Tools: Managing Projects from the Command Line. http://developer.android.com/tools/projects/projects-cmdline.html. [Online; accessed 04-Oct-2015].

[32] Google. Android: Develop: Tools: Manifest Merge. https://developer.android.com/tools/building/manifest-merge.html. [Online; accessed 05-Oct-2015].

[33] Google. Android NDK Preview. http://tools.android.com/tech-docs/android-ndk-preview. [Online; accessed 04-Oct-2015].

[34] Google. NDK: Guides: Concepts. https://developer.android.com/ndk/guides/concepts.html. [Online; accessed 04-Oct-2015].

[35] Gradle. Fully Programmable Builds. https://gradle.org/why/powerful-yet-concise-logic/. [Online; accessed Oct-2015].

[36] Gradle. Gradle 2.7 / User Guide / Chapter 23. The Java Plugin. https://docs.gradle.org/current/userguide/java_plugin.html. [Online; accessed Oct-2015].

[37] Gradle. Gradle 2.7 / User Guide / Chapter 3: Tutorials. https://docs.gradle.org/current/userguide/tutorials.html. [Online; accessed Oct-2015].

[38] Gradle. Gradle 2.7 / User Guide / Chapter 6. Build Script Basics. https://docs.gradle.org/current/userguide/tutorial_using_tasks.html. [Online; accessed Oct-2015].

[39] Gradle. Gradle 2.7 / User Guide / Chapter 61. Writing Custom Plugins. https:

//docs.gradle.org/current/userguide/custom_plugins.html. [Online; accessed Oct-2015].

[40] Gradle. Gradle 2.7 / User Guide / Chapter 7: Java Quickstart. https://docs.gradle.org/current/userguide/tutorial_java_projects.html. [Online; accessed Oct-2015].

[41] Gradle. Gradle 2.7 / User Guide / Chapter 2: Overview. https://docs.gradle.org/current/userguide/overview.html, 2013. [Online; accessed Oct-2015].

[42] Kristina Hager. Hello-Planet source. https://github.com/kristina-hager/hello-planet, 2015. [Online; accessed Aug-2015].

[43] Kristina Hager. Hello-Planet source. https://github.com/kristina-hager/hello-planet/blob/master/build.gradle, 2015. [Online; accessed Aug-2015].

[44] Kristina Hager. Hello-Planet source. https://github.com/kristina-hager/planetary-system/blob/master/build.gradle, 2015. [Online; accessed Aug-2015].

[45] Kristina Hager. Hello-Planet source. https://github.com/kristina-hager/planetary-system, 2015. [Online; accessed Aug-2015].

[46] Kristina Hager. Hello-Spatialite-Jar. https://github.com/kristina-hager/hello-spatialite-jar, 2015. [Online; accessed Sept-2015].

[47] Kristina Hager. Spatialite-Tools-Docker source. https://github.com/kristina-hager/spatialite-tools-docker, 2015. [Online; accessed Aug-2015].

[48] Kristina Hager. View SolarSystem GitHub Repository. https://github.com/kristina-hager/view-solar-system, 2015. [Online; accessed Aug-2015].

[49] Kristina Hager. Viewplanetsactivity local promote gradle. https://github.com/kristina-hager/android-planetary-viewer/blob/master/intermediate_gradle_files/AndroidPlanetaryViewer/viewplanetsactivity/build.step_local_promote.gradle, 2015. [Online; accessed Aug-2015].

[50] Kristina Hager. Viewplanetsactivity step1 gradle. https://github.com/kristina-hager/android-planetary-viewer/blob/master/intermediate_gradle_files/AndroidPlanetaryViewer/viewplanetsactivity/build.step1.gradle, 2015. [Online; accessed Aug-2015].

[51] Kristina Hager. Viewplanetsactivity step1 gradle. https://github.com/kristina-hager/android-planetary-viewer, 2015. [Online; accessed Aug-2015].

[52] Greg Hewgill. Stack Overflow Tag Trends. http://hewgill.com/~greg/stackoverflow/stack_overflow/tags/#!android+ios+linux+java+c%2B%2B, 2015. [Online; accessed Nov 11, 2015].

[53] Apache Maven. Apache / Maven / Introduction to Repositories. https://maven.apache.org/guides/introduction/introduction-to-repositories.html. [Online; accessed 08-Oct-2015].

[54] Oracle. Java Native Interface Overview. http:///docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp725. [Online; accessed 04-Oct-2015].

[55] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[56] HydroloGIS S.r.l. Geopaparazzi source. https://github.com/geopaparazzi/geopaparazzi. [Online; accessed 2015].

[57] HydroloGIS S.r.l. Geopaparazzi 4.4.0 documentation. http://geopaparazzi.github.io/geopaparazzi/, 2015. [Online; accessed 10-Oct-2015].

[58] Sun. JAR File Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html. [Online; accessed 08-Oct-2015].

[59] Jaroslav Tulach. *Practical API design: Confessions of a java framework architect*. Apress, 2008.

[60] Vividsolutions. JTS Downloads. `http://www.vividsolutions.com/jts/JTSHome.htm`. [Online; accessed 2015].

[61] Vividsolutions. JTS Topology Suite Technical Specifications. `http://www.vividsolutions.com/jts/bin/JTS%20Technical%20Specs.pdf`, 2003. [Online; accessed 10-Oct-2015].

[62] Matt Weinberger. How to get a software developer to work for free. `http://www.businessinsider.com/why-developers-contribute-to-open-source-projects-2015-6`, 2015. [Online; accessed Nov-2015].

[63] Emanuele Zattin. Writing Android Libraries. `https://realm.io/news/writing-android-libraries/`, 2015. [Online; accessed 2015].

[64] Andreas Zeller and Jens Krinke. *ESSENTIAL OPEN SOURCE TOOLSET: PROGRAMMING WITH ECL*. John Wiley & Sons, 2005.

Vita

Kristina Hager was born in Denton, Texas on 12 April 1979, the daughter of Kenneth Hager and Deborah Williams Caraway. She received the Bachelor of Arts degree in Computer Science from the University of Texas at Austin. She works as a software engineer in Austin, Texas and began graduate studies in Software Engineering at the University of Texas at Austin in January 2013.

Permanent address: 4806 Savorey Lane
Austin, Texas 78744

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.