Mark Ha '13

Adviser: Brian Kernighan

5/1/13 Draft

**“EasyAndroid” – Developers’ Tool for Android Beginners**

Android Development Made Easy

# Table of Contents

Introduction 2

Motivation 8

Related Works 12

Design Decisions 19

Implementation 17

Limitations 61

Challenges 65

Evaluation 68

Future Steps 75

Conclusion 78

I think you could trim the implementation section a fair amount. If you add some screen shots, you won’t need nearly so many words. You can also safely skip some of the low-level details. The writing could be tighter throughout; I marked some obvious places. It might be nice to have some examples of the commandline in action; those might even let you eliminate a lot of text: e.g., show the two or three lines that make a tip calculator, then show the picture of the Android screen

\*Remember to put in Acknowledgements

\* Add number to section headers

\* Add page numbers in the Table of Contents

\* Add some references for the competing products like Andromo? Maybe footnotes for their web sites are enough.

# Introduction

## Summary of EasyAndroid – what it does, how, very *general* conclusions/results

“EasyAndroid” is a desktop application that enables users with little or no programming and computer science experience to be able to create their own Android applications without needing to know the exact details of the code that goes behind it. What follows from that is that a user with a clear design or product in mind would should? be able to transform that idea into an actual, functional application on their Android phone in a short amount of time (on the order of one to two hours for first-time users).

The current version of EasyAndroid has a graphical user interface through which users can interact to create their own Android application. The user specifies the contents of the app through the interface, and EasyAndroid translates those interactions into Java code for an Android app, which users can then choose to compile and install onto their Android phones.

I have developed a successful prototype of what EasyAndroid was meant to be. If someone with relatively little software development experience can create a practical app, then EasyAndroid is a success. For instance, the equivalent of a “Hello World” app for Android – which would simply display a screen with the text “Hello World” – would take less than a minute to complete, especially with guidance or a tutorial. Replacing “Hello World” with a dynamically generated text – for example, “Hello NAME” where NAME is based on a text field value – would take a bit more guidance or examples, but once the user understood how to do so, would take just minutes. Expanding that to something like a tip calculator – which would use text fields for the base value, a text display to display the tip, and a button to generate the tip – could feasibly be done within an half an hour for a first-time user. With further development, EasyAndroid could reasonably become a viable replacement for Android developers for simpler Android apps.

## My approach to figuring out what to build in EasyAndroid – built my own “simple” Android apps, and figuring out 1) what people would want, and 2) what is required to build those kinds of apps? I assume this will be replaced by a short heading? Or deleted?

In starting of the design process for EasyAndroid, I used an approach aimed to maximize its usefulness for average users. This approach can be broken down into two main steps. First, what do people want? Of course, there are the visionaries who have grand ideas about an app that would require a real team of developers – those ideas are meant for startups. The average user, however, usually has at least two or three simple apps that they wish their phone had – a note-taking app, a simpler tip calculator, a to-do list app, and more. In some cases, their phones come with one of those apps pre-installed, or they already exist on the App Store (for iOS users) or the Google Market (for Android users). But even if an app of similar functionality already exists, most people that I talked to were either reluctant to put in the time to search for an app that fit their needs, wished for some extra functionality or small variation to the app, or had reservations about downloading random apps due to privacy concerns (Why does it need to use personal data to calculate tip?). In short, most people – including especially non-developers – have at least one simple app idea in mind, which, given the proper tools, they could even develop themselves. And so, that gave me an idea of what kinds of apps people are looking for.

Secondly, what goes into building those apps? Various questions such as how much code is required, which parts of the Android library need to be used, and which widgets are the most useful – all of these questions are relevant and shaped how I built my “barebones” version of EasyAndroid. That is, even a barebones version of the tool would need to have the capability to create the apps described, or at least most of them. Once I knew what needed to be in the tool – that is, once I clearly defined the extent of the functionality – I could begin designing the underlying structure.

And so, my methodology in building EasyAndroid was based on real examples. In order to identify what the tool needed to be able to do, I began by building my own “simple” Android apps, based on suggestions from people. By developing example Android apps – from basic tip calculators to displays for the native accelerometer – I was able to design a template structure that would be generic enough to fit any of those simple example apps.

## Explanation of some terminology (for the purposes of the paper)

Going forward, I will use some terminology and jargon that may be unfamiliar to some readers, and so I define them here, so as to minimize confusion.

**Android app:** In full, an Android application – that is, an application meant for the Android Operating System. Though the full term is “Android application,” I will refer to it as an “Android app” or simply an “app,” in order to distinguish it from the more general term “application,” which could apply to almost any type of computer software.

**apk:** The Android “application package file” (APK) is the file format in which Android apps are distributed and installed. For the purposes of this project, the “apk file” can be thought of as the finished product – an executable that users can run on their phones to see what Android app they have built.

**API:** Short for “Application Programming Interface,” an API defines the specifics of how one should interact with a particular piece of software. It’s not “one” – that would apply more to the GUI. Perhaps “how a programmer writes code to interact…”

**Widget:** The word “widget” in itself has several different meanings and usages, including a special functionality within the context of the Android Operating System. The way I will be using “widget,” however, is closer to what one may think of as an object or an item. Specifically, when referring to items that a user can add to their Android app – such as a button, a text field, etc. – I will use the term “widget,” partly because the word “object” has a closer meaning in the context of Java, and partly out of lack of a better term for now.

**Activity**: Though not an entirely accurate definition, it is enough to think of an Activity in Android as equivalent to everything that happens on any one screen in an app. Everything from the creation of the user interface to event-triggered logic – all of these aspects of the app go into the activity. In terms of implementation, any given Activity is a Java class. Most of the user’s interactions with EasyAndroid will be geared toward creating and customizing their app’s main activity.

**Command:** Throughout this document, “commands” will essentially refer to the API of EasyAndroid. These commands will be explained in-depth in the “Implementation” section, but in general, commands are functionality that I have developed as methods of interaction with the tool.

**GUI:** Short for Graphical User Interface, the GUI is a visual medium through which the user can interact with EasyAndroid. In particular, EasyAndroid started out with a command line interface (explained in the next section), and then grew to a graphical user interface, which would be much more intuitive and user-friendly.

## Building up “Core functionality” for EasyAndroid

From the beginning, I decided that I should first develop an end-to-end version of the tool, with an emphasis on vertical integration. This meant that the tool needed to handle every step of the Android app making process, from creating the project directory to installing the apk file onto the user’s phone. In doing so, one-hundred percent of the user’s interaction with Android development could and would be done from within the tool, such that the user would never need to leave the interface. Part of the motivation for this approach is that users first of all cannot be expected to know how to open up their computer’s command line shell, and enter the correct Apache Ant build script. By containing all of the messy operations that happen “under the hood” within the tool, the required interactions from the user are reduced.

How I envisioned the “alpha” version of the tool was through an interface modeled after the command line interface. In short, the user would be able to specify through the command line what he or she wanted the Android app to have in it. So for example, a user who wanted to add a button to their app would input something like:

**button –text hello world –height 10**

This command would add the corresponding button (one that says, “hello world” with a height of 10 pixels) to the current project. Thus, through a series of commands inputted into the “command line” interface, the user can design and create the app that he or she desires, and then have the app built (compiled) and then installed onto his or her phone.

## Expanding the Core Functionality

Once the initial core functionality was complete, I would develop on top of that, adding more and more “features,” from simply more options for widgets that the user could utilize to a Graphical User Interface (GUI) that would replace the command line interface. Once I had built enough features such that I felt that a user could user the tool successfully without too much difficulty, I would have rounds of usability testing and make improvements based on the testing and user evaluations.

# Motivation

## Mobile is hot, and Android is even hotter

Upon hearing my thesis topic, there is always the question of “Why Mobile?” which is quickly followed by “Why Android?” Especially in the United States, where iPhone seems more dominant and Windows Phone is making its entrance into the market, Android may seem like an odd choice. But in reality, Android actually has a larger market share, even in the United States, where iPhones do their best[[1]](#footnote-1). But in my particular case, Android is an even more obvious choice, because I have a full year’s worth of Android experience (two Junior Papers and a summer internship, along with other assorted projects), compared to zero iPhone experience. So not only would an Android tool appeal to a larger audience, but also, it would better suit my skills.

## Entry cost for Android development is high

The name, “EasyAndroid,” sums up my objective for beginner Android developers with this tool – facilitating Android development. As the title of this section suggests, the barrier for beginning Android development is quite high. For first-time users, the typical setup process to even begin Android development – making the basic “Hello World” application, for example – though much more well-documented and well-packaged than it was a year or two ago, requires an assortment of downloads and installations. Specifically, the user needs to have the Java SDK, the Android SDK, Eclipse (or some other comprehensive, powerful editor that can compile and run Android), and finally, any other additional Android SDK tools, extras, platforms, depending on the user's particular needs (i.e. an app meant for phones running Android 4.0 would need the corresponding Android 4.0 SDK). Once they have all of the software downloaded and installed, they can finally open up the editor and *attempt* to create their first Android app.

After that, there are plenty of tutorials and resources online to guide developers through the “Hello World” step, but for first-time Android developers, especially those who come from non-technical backgrounds, the barrier of entry between “Hello World,” and making their own application can be quite significant. While browsing Google search links will eventually answer the question, “How do I make an application that does \_\_\_?” it is a process that can be both time-consuming and incredibly frustrating, especially when there is no guarantee of the validity of the “solutions” one finds. The Android library is complex and comprehensive – for instance, one can see the drawbacks of potentially needing to go through dozens of library classes in order to find the Global Positioning System (GPS) API. Likewise, there are a plethora of complications when dealing with backwards compatibility – that is, making an Android app that is available to as many versions of Android as possible. Aside from these fairly advanced topics, there are also questions fundamental to Android that a first-time developer would ask and find difficult to answer – for example, what is an “Activity”? What is a “Context”? What is the “res” folder for? Though many of these basic questions are covered in the Android tutorial, it takes both reading and experience to fully understand how to use these properly – mishandling “Context” objects, for instance, can easily lead to memory leaks and eventual crashes in Android apps.

And so, on top of all the software setup that is required to begin Android development, learning how to program specifically for Android – even for an experienced Java programmer (Android apps are built on the programming language Java) – consumes a considerable amount of additional time. If the user also needs to learn Java, then even more time is required. In short, the combination of overhead costs presents a substantial impediment to any curious individuals considering starting Android development; EasyAndroid reduces that setup cost.

## Limited “Try out” Options for Android Development

In terms of existing tools for “trying out” Android development, AppInventor is probably the best option right now. The main problem with AppInventor, however, is that it is essentially a completely new visually-oriented “language,” and does not carry over to real Android development. Moreover, AppInventor does not offer any export source code option, so there is no way to see the source code equivalent of one’s app in AppInventor. In short, if you start using AppInventor, you have to continue to use AppInventor, or else lose your invested time if you choose to transition to traditional development.

Other products that provide “Android app building” services are similar in that usage of the product or tool does not carry over to real Android development. So, what trying out Android becomes is downloading and installing the relevant software, and then progressing through the Android tutorials on the Android website or some other website.

## Nothing Quite Like It

When I initially chose my thesis topic, I knew of no other product that simplified the Android app development process, aside from AppInventor. Since then, I discovered products such as Andromo, which completely remove the coding from Android development. “DroidDev,” a senior thesis from last year, sought to reduce the barriers to Android development by simplifying the setup process; it did so by hosting the necessary software on a server (cloud). This meant no installations or downloads required. The drawback of everything being hosted online, of course, is that the tool required a steady Internet connection, and that nothing could be accessed without Internet (there was no local platform). While an Internet connection is becoming more and more reasonable to assume that people have at any given time, there is still an advantage to being able to develop apps in times of unstable or no Internet connection, such as while traveling or commuting.

EasyAndroid differs in a few ways. Unlike DroidDev, which simplified the download and installation process, EasyAndroid simplifies the actual coding process. It essentially replaces Java code with a set of property definitions, which are then turned into Java code. Additionally, EasyAndroid is run locally, not requiring any Internet connection after the initial installation. Furthermore, in contrast to all of these other products, the Java and XML code, and the project directory generated by EasyAndroid are all available for the user to view and edit. One could easily use EasyAndroid as a starting point to create an Android app, and later import it into an editor like Eclipse to continue development. Lastly, EasyAndroid, while constrained in exactly how much it can do, gives users more freedom to create their own apps that have the look and feel that they want.

# Related Works

\* Add pictures to this section

In terms of related products, projects, and tools that work toward facilitating the Android development process, there are a few different types. Perhaps the most commonly used, and with good reason, is the Android library source code API and documentation – “Java docs” for the Android library essentially – which is available online. In addition to the documentation, Google provides an abundance of Android tutorials, ranging from the simplest topics like “Building your first app” to the most advanced and recent additions to the Android library, such as the ActionBar. Both of these documents provide information and often sample code to the user, saving the user from at least part of the pain of the development process. But there are also fully developed money-charging products available that either reduce the coding and development process, or remove it altogether – applications that seek to, on some level and to some degree, simplify the programming experience for fresh Android developers. Because these applications are closest to EasyAndroid, I take a look at several of them in this section.

## Scratch

Scratch is a programming language designed for early or new programmers (specifically young children), developed at MIT[[2]](#footnote-2). The main advantage of Scratch is that it removes the actual coding aspect and converts that into a graphical interface through which users interact in order to build their program. Scratch uses shapes and drag and drop, rather than words or text to represent many of the logical paradigms that introductory computer science courses typically teach. All syntax and code is converted into either blocks or some graphic, which are considerably easier to understand and interact with for beginners, than the standard syntax and coding rules that one learns in an introductory computer science class.

Perhaps the nicest characteristics of Scratch are its simplicity and intuitiveness. One small series of scripts – which take the form of readable English, such as “Say hello” or “Move 10 steps” – can compose the entire program. Furthermore, because Scratch is geared toward younger audiences, it has a heavy emphasis on learning to *think* like a computer scientist and problem solve both more creatively and more logically. From this perspective, it is easy to see that Scratch was a model for EasyAndroid, not only in turning minimal user interaction into maximal program development, but also in focusing on the user learning to develop rather than trying to re-create the full developing environment that one finds in Eclipse.

## App Inventor

Perhaps one of the more prominent Android programming tools is App Inventor, a project originally headed by Google, but now under the care of the Massachusetts Institute of Technology[[3]](#footnote-3). In part based on and quite related to Scratch, App Inventor takes a unique approach to programming, presenting programming and code to the user as puzzle-piece-like blocks. Instead of writing line after line of code, developing in App Inventor looks more like connecting puzzle pieces together, though there is no end picture to guide the development process. One example is that in place of the typical Java “for” loop, there is a “for” *block*, which takes the necessary arguments that determine its lifetime; more importantly, however, the block is shaped such that another block – one that might normally be considered the body of the “for” loop – fits inside and connects to the “for” block. By converting all code into puzzle pieces, App Inventor provides a more visually intuitive experience for new Android developers, and also removes the barrier of not knowing Java, or how to code.

As with Scratch, App Inventor emphasizes the *learning* process for the user, introducing Android development as a block-based language. Furthermore, neither Scratch nor App Inventor is meant to be as powerful a language as say, Java, or C, or even Python. For instance, Scratch is limited to one-dimensional arrays only, and String manipulation is limited as well. Along the same lines, App Inventor does not easily lend itself to larger scale applications, due to its inability to include external jars (Java Archives) or create true multi-screen applications (though you can technically clear the screen and redraw it). Both also have their own upfront setup costs, such as downloads, installation, and learning to use the tool. While it would be false to say that one cannot make useful and relevant applications using these programming tools, it is apparent that neither is meant to be a full-fledged programming language or a replacement for real programming. Still, both are arguably much easier to learn and acquire for beginner programmers, who have yet to learn the ins and outs of syntax and coding.

EasyAndroid is in part inspired by App Inventor and Scratch, particularly in the way they appeal and are much more effective for beginner developers. While neither can create an actual, complex product as well as Java or Python, both are even better than those powerful languages for inexperienced developers. EasyAndroid takes these models and strives to facilitate simple development for newer Android programmers, keeping the entire process as efficient and intuitive as possible.

## Droid Dev

Droid Dev was a senior thesis last year, which also sought to simplify the Android development process. The main difference is that rather than reducing or simplifying the actual development process, the major advantage of Droid Dev was that it essentially eliminated the setup cost of downloading and installing software. It did so by hosting the necessary software on a server (cloud) – this meant no installations or downloads required. The drawback of everything being hosted online, of course, is that the tool required a steady Internet connection, and that nothing could be accessed without Internet (there was no local platform). While an Internet connection is becoming more and more reasonable to assume that people have at any given time, there is still an advantage to being able to develop apps in times of unstable or no Internet connection, such as while traveling.This sort of repeats something a few pages back; might tighten it up a bit. With the exception of Twill, all of the products described in the following sections are provided via the Internet, and so have the same disadvantages as described here.

## Twill

Twill is a language written in Python which brings the Web-browsing experience to the command line interface[[4]](#footnote-4). The main purpose in mentioning Twill is that it served as an inspiration for the core functionality of EasyAndroid, which has a command line interface. In the same way that Twill simplified browsing the Web to command line interactions, the alpha version of EasyAndroid simplified Android development to command line interactions.

## If This Then That

If This Then That (IFTTT) is perhaps the ideal model and inspiration in terms of simplifying “programming.” IFTTT allows users to establish if-trigger-then-action combinations through an extremely simple interface that is broken into seven steps, but can really be though as two steps[[5]](#footnote-5). All the user has to do is choose the *this* trigger and the *that* action – two items. And so, the user does minimal work, oblivious to what goes on behind the scenes, and creates an extremely useful automated service. Officially, the seven steps to creating a “recipe” (if-trigger-then-action combination) are:

1. Choose Trigger Channel (a service such as Gmail, Facebook, etc.)

2. Choose Trigger (mail received, message received, etc.)

3. Define Trigger properties (such as mail from *a particular person*)

4. Choose Action Channel (again, a service)

5. Choose Action (send me a chat message, send me a text message, etc.)

6. Define Action properties (such as title, message body, etc.)

7. Create and activate (accept and confirm the recipe properties)

An example recipe would be, if (Gmail) any new mail received, then (SMS) send a text to ### with a message body “New Mail: [Subject] from [Sender].” Of course, setting up this type of service requires login or registration of the utilized services, but otherwise, the design is incredibly simple and easy-to-use.

EasyAndroid strives to have that kind of simplicity. For example, the act of adding a new button can be simplified into the process of:

1. Choose widget (Button)

2. Define widget properties (name, text, height, width, action)

3. Confirm addition of the widget (button) to the app

As with IFTTT, the drawback of this extremely simplified process is that there is no customization available, outside of defining the properties. In IFTTT’s case, there is a finite set of triggers and a finite set of actions to choose from – what IFTTT offers is exactly what it offers, nothing more, and nothing less.

## Andromo

The product most similar in functionality to EasyAndroid is Andromo, a web application through which users can completely build their own Android app[[6]](#footnote-6). Perhaps the strongest characteristic of Andromo is that the user does absolute no programming or coding in creating his or her app. In a sense, the app creation process requires that the user decide and specify what kind of content he or she wants in the app, but there is never any need for code of any sort on the user’s part – even the user interface and the look and feel are already designed for the app, though the user can select between a few different themes, as one would when creating a website. From the perspective of eliminating any need for software experience or a technical background before building an Android app, Andromo does an amazing job.

On the other hand, no product is without flaws, and Andromo’s major drawback is that users are extremely limited in the style and type of Android app that they can produce using Andromo. In other words, because there is “No Coding Required,” as advertised by their slogan[[7]](#footnote-7), Andromo – like IFTTT – is restricted in its the capacity for customization. As with IFTTT, the options and template provided by Andromo is what the user has to work with – anything outside of those options is impossible. In particular, almost every Android made by Andromo has a dashboard menu screen that leads to content screens – the navigation flow is the same across all apps, and options for content are restricted essentially to displaying web or local content and links to social media pages. Because of this limitation, Andromo apps end up being extremely generic.

## AppsGeyser

AppsGeyser is a product that converts existing websites or web content into an Android app in a process that requires almost no user interaction other providing the website URL. As a result of this minimal effort required by the user, the product is both extremely convenient and straightforward to use, and extremely limited and inflexible. AppsGeyser is perhaps the extreme of “reducing” the Android development process in that the user has zero say in how the Android app will turn out, allowing for practically no “development.”

## Appcelerator Titanium

Appcelerator Titanium is a platform that takes JavaScript code, analyzes, preprocesses, and pre-compiles it into symbols that the native mobile compiler understands[[8]](#footnote-8). In other words, it converts Javascript to mobile source code (for both iPhone and Android). And though the conversion is decent enough to serve as a prototype app, the costs are also quite high. First of all, there is no guarantee that the Appcelerator Titanium converts the code in the best way possible, which leads some developers to learn how to code the app natively anyway. Secondly, Appcelerator Titanium has an API that users must learn to use and interact with. Despite these drawbacks, Appcelerator Titanium can definitely be seen as an effective transition tool and stepping stone for Javascript developers seeking to move into mobile apps in a quick and efficient manner.

# Design Decisions

## Mission Statement & Defining Project Scope

As mentioned, EasyAndroid seeks to simplify the process of developing an Android app. Since there are several methods to do so and several fronts on which the process can be simplified, I had to make decisions along the way about which methods and which fronts to pursue. As with most design decisions, each option has its particular advantages and disadvantages – rarely is there an option that is strictly better than all other options. And so in the following sections, I will present the problem that I encountered, the various potential solutions for that particular problem and each of their pros and cons, and finally, which solution I decided to go with and the rationale behind that decision.

## Using Existing Software

A pivotal decision made early on was to take full advantage of the software available and typically used for Android development. That is, rather than seek to simplify the upfront setup costs, EasyAndroid aims to simply the actual development process. With that goal in mind, the dependence of EasyAndroid on the various SDKs and tools that typical Android development depends on became not as big of an issue – rather, eliminating that dependence on the “third-party” software was not as important as first developing a tool that simplified the coding.

## Target audience

It was important to be realistic and realize that it would be impossible to create a tool that fit the needs of every developer – a product designed for everyone would be effective for no one. I decided early on that my target audience would be first-time Android developers, particularly those with no or relatively little general programming experience. I could have targeted more experienced Android developers, who know well the shortcomings of the Android library and development process as a whole. Likewise, I also could have target programmers new to Android, but otherwise experienced, but decided that the learning curve for those developers is not as significant, and therefore, a tool designed for them is not as crucial, nor would it be as impactful. On the opposite end of the spectrum, however, there are the people with no programming experience whatsoever who struggle with the setup costs and learning curve described above, and as a result, quit before they have even started real development. Ultimately, I decided to target these kinds of users, hoping that EasyAndroid would make it such that even without significant general programming experience – around the level of completing an introductory computer science course – someone with a clear design or specification for an Android app in mind would be able to create their own Android app with relatively little effort.

## Priority of the Graphical User Interface

As I approached the deadline for my thesis submission, I realized that I could not implement all of the features that I had originally planned to. This meant that I would need to prioritize the features and abandon some of the lower-priority features. Because of how I defined “core functionality” for EasyAndroid, completion of core functionality already meant developing a command line interface through which to interact with the tool. If I were targeting computer science majors and technically experienced users, then it would have been enough to leave it at the command line, and continue expanding the functionality of the tool. But a command line interface is far from intuitive for a first-time user, and for a target audience of people with minimal computer science background and likewise minimal experience with a command line interface, a Graphical User Interface is necessary. As a result, my first priority upon completing a stable version of core functionality for EasyAndroid was to build a GUI for it.

## “Help” Documentation

One of the advantages of having a GUI is that rather than forcing the user to look up the documentation to figure out their options, the GUI can just have a list of all options available to the user, which is what it does with the “Options” pane. Even with a GUI that lists all of the options, however, it is still unclear what exactly each of those options represents, how it works, and what kind of input its properties take. Therefore, there is still need for a “Help” page.

## Designing for Benign Users vs. Malicious users

One major design decision I made was to design the tool assuming that the user is “benign,” as opposed to “malicious.” In other words, there is a large difference between an inexperienced but innocent user simply trying to use the tool and a user that is deliberately trying to break the tool. The main area that this decision affected was the design for EasyAndroid’s interface, especially in considerations such as how to parse, interpret, and in some cases, reformat user input.

## General approach: Abstraction

Heading into the implementation of the tool, the first question was what the structure of the tool would look like. On one end of the spectrum, everything could be hard-coded to function according to my specifications, and each next step would require additional hard-coding. On the other hand, I could establish abstract models to represent the different parts of the tool and the various properties of an app. Next steps would follow these abstract models.

In general, the more desirable and better-scaling solution is to establish the base models and build everything in a systematic manner. The structure of EasyAndroid would ideally be extremely flexible and easy to change – furthermore, expanding the tool would be simple and efficient, because of the underlying structure. But of course, it is impossible to think a hundred of steps ahead and plan for everything – thus, my approach became, 1) think as far ahead as possible, and then 2) implement an appropriate amount of abstraction and flexibility given the time limitations and importance of each feature. In the end, EasyAndroid is very much a mix of “hard-coded” or temporary solutions – such as with the command parser – and abstractions – such as with much of the widget interactions.

## “Design By Example”

In approaching building and designing EasyAndroid – what functionality would it have, with what features – I took a “design by example” approach. What this meant was creating example or sample Android apps with actual Java code, and using those sample apps to determine what EasyAndroid as a programming tool needed to be able to do. Thus, the functionality of EasyAndroid was largely driven by looking at several simple Android apps, and then generalizing them in a way that could be made into a tool and a single set of widgets. Among sample apps that I made to set up the structure for widgets were: the traditional “Hello World” app, a tip calculator, an accelerometer display, a camera launcher, a contacts list searchable by regular expression, and a GPS location tracker. Using these sample apps, I was able to see the general structure that most Android widgets follow, and define the general widget for EasyAndroid based on that.

## Java Code over XML Layout Files to Define Layouts

Typically, when developing an Android application, there are several different types of resources that the developers would have to manage. In fact, Android project directories automatically separate these different resources upon creation. There are the “res” or resources files, which are a combination of visual assets such as logos or images that are used in the app (some would use the term “assets”), and XML layout files, which developers can use to define the template for the look and feel or a particular screen. There are also the “src” or source files, which is generally Java code.

The reason I mention these is that in defining the look and feel for their app, developers have two methods through which they can do so. The first, and more commonly used approach, is through XML layouts, where developers can specify a static version of their look and feel. Should the developer choose to define their user interface through XML, the Android OS will take that XML file at runtime and “inflate” it onto the user’s screen. The second approach is to define the layout at runtime through Java code. In this approach, when an activity is created, there will be nothing to begin with – the screen’s UI will be populated as the Java code is run.

The XML approach has several advantages over the Java approach: XML is much quicker and easier to manage for the developer, and, perhaps most important of all, it separates the definition of the UI from the logic and other computations that might typically be done in the Java code. Furthermore, XML is more flexible in terms of providing an easy way to switch out one XML layout file for another depending on the screen-size, language settings, orientation, and other miscellaneous properties. In a traditional developing environment such as Eclipse (*the* developing environment for Android, as it is), using XML layout files for UI definition has a significant advantage over Java in that Eclipse (and many other editors) have a graphical layout tool that shows the user a preview of what the screen will look like based on the XML file. In order to see what a user interface coded in Java looks like, however, the developer would actually need to build the project and launch it on their phone (or emulator).

On the other hand, the main advantage of using Java to define layouts programmatically (as it is often called) is that the definition happens dynamically. Thus, in situations where the user interface varies depending on external factors, defining the layout dynamically makes more sense, since attempting to define it statically (through XML) would just result in adjusting it at runtime, and would then be a waste of time (and processor power).

That said, using XML layout files for UI definition is generally advised, though from a performance efficiency perspective, the difference is insignificant, if any. In terms of the limitations of each approach, there are very few cases where a property is available for use in XML but not Java, or vice versa, but in general, the two can produce the exact same results.

For EasyAndroid, underneath the GUI and command parsing, I define widgets through Java code, as opposed to XML, for a couple of reasons. The main reason is that by defining things in Java, I can keep most of the code in one file. The main activity file – set to “MyActivity.java” as the default – is necessary for the app to run and function. The XML layout file – “main.xml” in most Android tutorials – that normally goes with that Java file, however, is convenient but not necessary. And so, instead of main.xml, the user interface for the app can simply be defined directly in MyActivity.java. Not only does that mean less work in terms of file management – creating and deleting multiple files in the actual project directory, but also, the user benefits because they can actually see the changes that they made in full as Java code. While I could have a multi-window or multi-view preview panel that shows both the Java source code file and the corresponding XML file, combining them into the one Java file is both easier and simpler. Furthermore, even if a layout is defined in XML, dynamic interaction with that layout such as updating a value displayed still requires Java code, and so there is no way to avoid some additional Java code, unless the layout is completely static.

To elaborate a bit more on what the definition of UI actually looks like in EasyAndroid, I will present an example, and then explain how that applies to the general case. Suppose that the user adds a TextView to the layout, with default properties other than “Hello World” set as the text. In Java, this translates to:

**final TextView widget1 = new TextView(this);**

**widget1.setText("Hello World");**

**widget1.setLayoutParams(new LinearLayout.LayoutParams(ViewGroup.LayoutParams.WRAP\_CONTENT, ViewGroup.LayoutParams.WRAP\_CONTENT));**

**rootView.addView(widget1);**

Just as a note of comparison, the XML layout file version of this segment of the code would be:

**<TextView**

**android:layout\_width=" wrap\_content"**

**android:layout\_height="wrap\_content"**

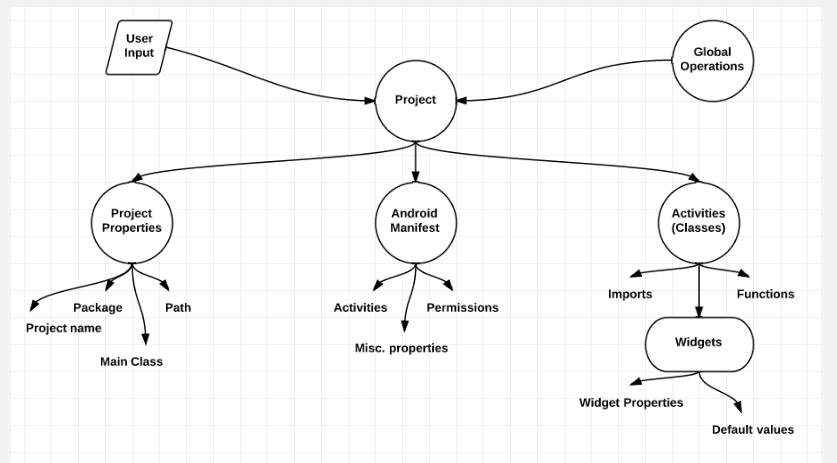
**android:text="Hello World"**

**/>**

And naturally, a more complicated widget would require more lines of code. In this particular example, however, we first initialize the new widget, with a default name of “object1”, and then set its properties – what the text is and what the dimensions are. Lastly, we add it to “rootView”, which, as the name suggests, is root view object of the activity’s layout, which is designed in a tree hierarchy structure. To elaborate on rootView, it is a LinearLayout object, which is a type of container for Android layouts. Other examples of Android layouts would be RelativeLayout or FrameLayout – depending on the situation, one layout might be better than another. In my experience, however, while RelativeLayout is perhaps the most flexible and most precise, LinearLayout is the simplest and perhaps most efficient in terms of time spent defining the layout and the resulting usefulness of the layout. For that reason, I decided to make rootView a vertically-oriented LinearLayout object, with the idea that it would be the most intuitive for the user.

As such, every widget addition can be broken into three steps, regardless of how simple or complicated the widget is: 1) declaration and initialization, 2) definition of properties, and 3) addition to the rootView. More complicated widgets may have more lines of code per section, but ultimately, all widgets follow the same pattern.

## Architecture of an EasyAndroid Project



I took a very object-oriented approach in designing the project structure for EasyAndroid – that is, EasyAndroid maintains essentially a “project object “ which contains all of the relevant properties and items that go into a typical Android project. The EasyAndroid project structure is shown in the figure above. The basic flow is that incoming user input modifies the project object, which includes operations that can modify the AndroidManifest, imports, custom functions, permissions, and general project properties. Defining a new widget enters that new widget into a list in the project object, which, during the build process, gets translated into Java code.

# Implementation

\* break this section up? 40 pages... Seems pretty long

The following sections describe some of the more interesting or challenging parts of EasyAndroid’s implementation, as well as the fundamentals. Some sections give more background on the decision-making process to implement that particular part of EasyAndroid in that way, while some simply give an in-depth explanation of the implementation of a particular functionality.

## Language of Choice: Java

EasyAndroid is one-hundred-percent written in Java. The first question I had when beginning implementation was which programming language to use. I debated early on whether other languages would be easier, or perhaps more effective, but soon realized that doing things with a little extra effort in Java, the language I am most comfortable with, would probably be better than learning a new language *and* learning how to do the same things in that new language, even if the new language was specialized. In the end, Java was a fine fit for my needs, and I did not have to waste any time familiarizing myself with its syntax or peculiarities.

Additionally, EasyAndroid expands across several different types of modules (customized Android data structures, parser, app abstraction as an object, GUI), which makes it unlikely that there would be an all-purpose language in which all of those would be easy to implement. While perhaps using a specialized language for one module might have been easier –I considered it for the GUI in particular – I decided that Java would suffice.

Because EasyAndroid is coded in Java and compiled completely in a regular Java environment, it is enough to simply run EasyAndroid as a Java program. Before the GUI had been implemented, this meant interacting with EasyAndroid through standard input – completion of the GUI meant that the user could simply interact naturally with the GUI.

## Android Manifest Generator

An essential file – in a sense the most essential file – in every Android project is the AndroidManifest.xml file. Written in XML code, the AndroidManifest defines the properties of the Android app that is being built, as well as describing the contents of the app. This extends from the “project properties” such as build target and app name to a list of the activity classes that are used in the app. It can be thought of as an abstract or pre-definition of the app’s contents. Unlike many of the other contents of the project directory, there is no convenient command or script to auto-generate the AndroidManifest– in the typical development process, the developer must update the file manually.

Thus, I created a “AndroidManifestGenerator” class to populate the AndroidManifest file. Because the file needs to be in XML format, simply printing the equivalent String into the file causes problems and fails to function properly. Instead, I used a combination of the DocumentBuilder class and Element class (as my XML tags) to build the contents of the file. The advantage of the file being in XML format is that the contents themselves follow a structure and a format – there is a finite number of valid elements, sub-elements, and attributes that can be contained in the file. As a result, those elements and sub-elements can be systematically generated, which is exactly what my AndroidManifestGenerator class does.

### Permissions

One part of the contents of the AndroidManifest.xml file which required a bit of extra effort is the permissions element. All Android apps request a particular set of permissions from the user, such as permission to read the phone’s contacts list, permission to use the GPS data, permission to use the network (Internet), and others. On the developer side, the permissions that the developer would like to request for his or her app are specified in the AndroidManifest. As a result, I had to establish a method of specifying which permissions the app required. One might wonder, “Why not just always ask for every permission to be allowed?” The answer to this is that, when Android phone users install Android apps, they are shown a list of the permissions that the app requests – if the app were to blindly request all permissions, users who are sensitive about privacy would justly wonder why a Contacts app requests permission for Internet, GPS data, Alarm settings, and all of the other permissions. Thus, the best policy is to ask for exactly what the app needs, and nothing more.

To describe the implementation of this permissions handler (called “PermissionManifestObject”), the short version is that it is a list of boolean variables, each of which corresponds to one of the available Android permissions. This implementation obviously does not scale very well; at the same time, there are a limited number of permissions – one-hundred thirty to be exact Good grief! – and an even smaller number that are commonly used, and an even smaller number than that, that the functionality of EasyAndroid could actually employ (currently two). As a result, this method of implementation works quite well for the time being – as the functionality of EasyAndroid expands, one could simply continue adding on to the PermissionManifestObject class. This class outputs a list of XML elements designed to be injected into the AndroidManifest.

### Activities

Aside from the permissions elements, the only noteworthy part of the AndroidManifest from an implementation standpoint is its list of activities. In short, activity classes have to be “registered” in order to be used in the app – opening an “unregistered” activity causes the app to crash and force close. The declaration of the activity classes is quite straightforward – the one exception is the “main activity”, or what I call in EasyAndroid the “main class”. The “main activity” of an Android app is simple the starting or home activity; it is marked specially in the AndroidManifest. In typical development, the activities can also be labeled with special properties, such as orientation, but because EasyAndroid currently only deals with simple apps, those special properties were not implemented in the AndroidManifestGenerator.

Aside from the permissions and activities elements, the exact details of the contents of an AndroidManifest are unimportant and uninteresting. Thus, further details of the AndroidManifestGenerator’s implementation are omitted – it is enough to understand that it is a class created in order to correctly generate the crucial file that is AndroidManifest.xml.

## “Global” versus “Local” Commands

\* Note to self: check consistency of quotes usage. “build” vs. build \* Activity – check consistency on capitalization

\* Check consistency of “app” and “project”

\* Do I capitalize commands if they are the beginning of a sentence? “Build” vs “build” – “Build”

would not work in the tool, and so is inaccurate... (also with punctuation)

Partway through my thesis, I realized that it would make more sense to separate “global” commands and “local” commands. Global commands can be thought of as operations that do not affect the app directly, and are more administrative in a sense. This includes the “open” and “save” file and “reset” commands, which change *which* app the user is working on; the “create”, “build”, “install”, and “run” commands, which convert the app into a file directory and an apk file (for “install”); the “help” command, which shows the user the list of available commands; and the debug commands – “print”, “debugfunctions”, “debugimports”, “debugmanifest” – which show the user the Java and XML code for different portions of the app. Local commands, on the other hand, are operations that in general change the app’s contents. This includes operations on the app’s properties (project name, path, main class, and package name); the “addFile” command, which adds the current class with all of its widgets and functions to the app; the custom commands (“customfunction” and “customimport”), which allow for custom functionality in the app; and lastly, the widget commands, which manipulate the widgets that are in the current Activity class of the app. In short, global commands do not necessarily change the state of the app itself, whereas local commands manipulate the properties and contents of the app.

It should be noted that the descriptions of the commands that follow generally describe the commands as if they were being inputted via command line. While the GUI makes it such that the user never actually uses these commands explicitly, almost all of the GUI’s functionality is built on top of the original command line interface and its parser. Thus, when using these commands, the GUI takes the user input into the GUI and formats it as a command as if it were meant for the original command line interface parser; this updates the underlying project data structure appropriately.

## Global Commands

### Open File, Save File

The open and save file feature is more supplementary in nature, because users can do everything in one session, but still convenient and useful, and thus, valuable. What this feature does is allow the user to save their progress during the development process, and quit the tool; upon restarting or reopening the tool, everything is reset to the default settings of a fresh project, but the open command allows the user to load his or her progress from a saved session.

For the implementation of this feature, I immediately saw two approaches that I could take: a “dirty” solution, and an “elegant” solution. The dirty solution is to record all of the user’s valid “local” input as it comes; upon executing the “save” command, those inputs are stored into a file on disk, in the user’s “.saves” folder, with the filename by default set to the project name if unspecified during execution. The “open” command does the opposite – it takes the commands stored in a file (filename must be specified during execution), and re-executes all of them. By doing so, the project is brought back to state at which the user saved.

The elegant solution implements the concept of a state – rather than keeping track of each step, this method takes a “snapshot” of the moment the user executes the “save” command. The state would include every changeable aspect of a project – from the project name to the list of widgets. All of those would need to compressed into some parseable data structure or format, which, when read, would restore the state of the project.

Both approaches have their advantages and disadvantages. The most significant advantage of the dirty solution is that it is simple to implement – “save” literally only requires adding to an ArrayList of Strings for every local command inputted by the user. The implementation of “open” is also practically already in place – because the saved data is the exact same as the user input, it can be processed correctly by the existing parser. The only extra work is to create a function that pulls the data from the saved file, and feeds it into the parser. At the end of that, the project will have returned to the point at which the user had originally saved the project.

Because of the nature of the dirty solution, however, obsolete commands will still be executed upon reload. For example, if the user changes the project name twenty times, then in the process of loading that project, the project name would be changed twenty times again, when it would be enough to change it once (to the last change). This particular example can be optimized for the project properties such that new commands of the same type replace old commands, but doing so would first of all complicate the implementation – detracting from its main advantage – and secondly, not necessarily work for other aspects of the project, such as a deleted or re-positioned widget. Furthermore, the more complicated the project, the more commands there will be, and as a result, the longer loading the project will take. Clearly, re-executing every command would not scale to larger projects very well.

On the other hand, the elegant solution avoids both of those problems – it both scales extremely well, and eliminates redundancy. The primary advantage of the elegant solution is in the efficiency of having a state data structure. When the user saves state, the project’s exact state – nothing more, nothing less – is saved. Then, when the user opens a saved state, there are no extraneous operations, which lets this method scale much better than the dirty solution with complexity. While one might think that a state may include some superfluous properties, such as an empty function list, those costs are relatively small, and still out-scale the dirty solution quite quickly.

Still, every elegant solution has its disadvantages. This one in particular is complicated, and requires updating as the project structure changes. Moreover, while this approach certainly would have been feasible to implement, the definition of the project state would have required some careful planning and design, especially considering that a project can have any number of classes, each of which consist of any number of functions and widgets, each of which consist of any number of properties. In addition to creating the state structure, a parser – though not necessarily a complex one – would need to be created as well. All of this sums up to a considerable amount of work and effort.

That said, due to its much better scaling, the elegant solution would still be a mandatory feature for a final iteration of EasyAndroid. But because time was the limiting resource, and because of the upfront cost of designing the state structure and implementing a new state-customized parser, as well as the ongoing cost of updating the structure, I chose to use the dirty solution for the time being. Though the two are structurally different (and so, the implementation of the dirty solution does not carry over), the operations for each are so minimal that the user would only notice a performance difference for much larger projects. The dirty solution, while perhaps not the best solution, took a relatively small amount of time to implement, and gets the job done.

### Reset

The “reset” command is almost self-explanatory, and mainly a convenience function, originally geared toward the command line interface. “Reset” resets the working project to its base state, with all of the default values. In terms of EasyAndroid, this means clearing all existing activities, widgets, and custom functions, clearing any changes to the AndroidManifest and project properties, and clearing the history of saved user input. The effect is the same as closing the tool and then re-opening it, but it serves as a convenience method to restart the app without leaving the tool itself. As with most programs, resetting clears all unsaved progress, but saved progress is still left on disk.

### Create

Though “create” is not a *necessary* step in the development and design process of the app, it becomes necessary once the user wishes to translate his or her work into a functioning app on an Android phone. Furthermore, running “create” at the start of the project is usually in the best interests of the user, because it initializes everything properly. In this case, the step of “creating” the project in essence translates to generating the project directory and its contents. It is equivalent to the “New Project...” function in any project-based editor, such as Eclipse.

With the use of Android tools, the implementation of “create” becomes much simpler. Implementing the entire process of creating a customized directory filled with the right files with correctly-formatted content would have taken not only significantly more code, but also much more research. That said, among the software provided by Android (Google) is an exceedingly convenient command – “android create project” – that generates a basic Android project, with some specifiable properties (project name, directory path, main activity, and package name). Then, “create” is reduced to formatting this command, and then managing the child process that executes it.

### Build

“Building” the project is actually composed of several steps, the compilation itself being only one of those steps. Compilation is done through running an Apache Ant build script, so before the project can be compiled, the build script needs to be generated and all of the appropriate project files need to be created, properly populated, and placed in the correct directories.

The first step is to create and format the AndroidManifest.xml file, using the AndroidManifestGenerator class described earlier. After setting the fields for the AndroidManifestGenerator, and describing the app’s contents in full, the AndroidManifestGenerator creates a well-formatted AndroidManifest.xml file with the appropriate contents, customized to the project’s specifications. Because the AndroidManifest.xml file is a key asset for compilation – compilation fails if the file does exist, and an incorrectly defined AndroidManifest.xml file can cause runtime errors and crashes – the first part of building the app is to use the AndroidManifestGenerator to create a correct AndroidManifest.xml file on disk.

Next, the current class file needs to be written on disk. This step simply calls “addFile” (described later) on the current class. In short, this involves creating a file, and writing the appropriate contents to that file (using a BufferedWriter stream and FileWriter). Determining those contents is equivalent to the “print” function (also described later), which converts the user’s development in the project up to that point into a Java class file. Thus, this step is mainly a compilation of two other, smaller commands.

Finally, once everything is in place, the Ant build script – generated by the Android tool – is run. Since Ant takes care of everything, there is little work on my part, aside from finding the build.xml file to run, and managing the child process that executes the script (the exact command is “ant debug -f [path]\build.xml”). Thus, this last step of the build process is reduced to simply using Ant to run the provided build script.

### Install

Thanks to Apache Ant and the Android tools again, the “install” step is also relatively simple and straightforward. Assuming that the user always “builds” before attempting to “install”, the only thing that “install” needs to do is put a pre-existing apk file onto the connected Android device (the exact command is “ant installd -f [path]\build.xml”). Again, I have to locate the build.xml file, and manage the child process that executes this command, but other than that, all of the work is done by Apache Ant.

### Run (2.0)

One detail to note is that running the build script (for “build”) takes a noticeable amount of time, even if it is just a few seconds. Because of that, if the user were to try to run “install” *immediately* after starting the building process, the building process would potentially not have finished, the install would likely fail, or install a previously built version of the project. To eliminate that possibility, I decided to intentionally allow the child process for “build” to lock the main thread – of course, ideally the child process would be run “in the background” and there would be a popup that informs the user, “Building...” but in terms of the effect that it has, telling the main thread to wait on the child process has the same effect in forcing the user to wait for the project to finish building. Perhaps the optimal solution would be if compilation was instantaneous, but, especially as the project size grows, that solution is simply unrealistic.

Furthermore, I realized that there are few situations in which the user would want to “build”, but not “install”. Thus, I decided that it would be better if I simply made it a “one-step” process, which I decided to call “run”. The command “run” is essentially a combination of “create”, “build”, and “install”, as described above. The main difference is that, rather than running two separate Ant commands for “build” and “install”, one command that does both the building and installation is run (the exact command is “ant debug install -f [path]\build.xml”).

### Help

The command “help” (formally “--help”) is a work in progress, mainly by necessity. This command can be thought of as the documentation of the available commands. In a typical command line interface fashion, malformed input is responded to with an error message, which is sometimes accompanied by a suggestion, or an example of well-formed input. What “help” does for the user is provide a list of available commands, along with descriptions of how to use each command. In a sense, “help” is not crucial to functionality of EasyAndroid, but realistically, there needs to be documentation for the tool somewhere, or else users would never be able to figure out how exactly the tool works, other than by trial and error. For that reason, some would argue that “help” is the most important feature of the tool – it enables the successful usage of all of the other features.

The current implementation of “help” is designed to complement the command line interface. As a result, it is omitted from the GUI – were it to be included, it would most likely be an item in the menu bar, and activating or selecting that item would launch a new popup window with the text from “help”, which would inform the user of the various functionality and usage of different parts of the tool.

### Debug Commands: Print, Debugfunctions, Debugimports, Debugmanifest

In addition to these commands, there are a few commands geared toward debugging during the development process. Though most users would not be expected to try to debug the app and thus would not get much value out of these commands, users seeking to use EasyAndroid as a stepping stone for actual development would most likely find these debug commands useful.

The “print” command is perhaps the most useful – it prints out the actual Java code that is compiled for the current Activity in full. “In full” includes the package declaration, import statements, function definitions, and widget declarations and initializations. Because the file is printed in full, the “print” command functionality also doubles as a means for obtaining the contents of an Activity’s Java source code file, which is extremely useful when writing the file on disk (see “addFile” later). In other words, one could cut the output from the “print” command and paste it into a Java file; barring any missing dependencies (i.e. external classes that need to be in the build path), the code would compile and function perfectly in an Android project.

“debugfunctions” gives the full list of all of the user’s custom functions defined in the current Activity. This is mainly a convenience method for the user, in case he or she forgets what functions are defined in the project. Of course, “print” includes the definitions of the custom functions, but “debugfunctions” gives only the custom functions in particular.

“debugimports” does the analog for the Activity’s imports, giving the full list of imports in the current Activity. Just as the user is able to add custom functions to the Activity, the user can also add custom imports to the class, which would typically be used in combination with a custom function that uses an external library. Again, one could “print” to see the Activity in full, which would include the imports, but this convenience command allows the user to see the imports as a separate module.

“debugmanifest” shows the String of the current state of the AndroidManifest.xml file. Due to the actual AndroidManifest being generated as XML with proper character encoding, the “debugmanifest” version of the file is not a completely accurate one-to-one representation. The differences, for the most part, are in alignment and ordering of content – otherwise, all of the content is the same, and “debugmanifest” can be seen as faithfully representing the actual AndroidManifest file. This command is probably one of the more useful debug commands, as the AndroidManifest can be both complicated and tricky to deal with, particularly for new developers – having a good model or template can save users a significant amount of time when they start real Android development.

All of these debug commands were originally designed for the command line interface, where it was impossible to display everything at once. The current implementation of the GUI does not utilize these debug commands, except for “print” (a preview of the class’s Java code is shown and updated as the user changes their app), but future implementations could certainly have a “Show Imports” or “Show AndroidManifest” option that would display the relevant text in the same manner as “help” is shown.

## Local Commands

### Project Name

The “projectname” command sets the project (app) name to the first argument following “projectname”. The project name is essentially the public identity of app – it is the display name for the app in a phone’s app list, and the last extension of the app’s package. Naturally, the default project name is set to “helloworld”.

### Path

“path” changes the base path of the EasyAndroid app’s file directory to the value of the first argument. Developing and building an app does not necessarily require changing the path ever; in fact, it makes the most sense, and the tool works best, if the default path is never changed. Nevertheless, the option is available to users who prefer some alternate path. The default path is set to “C:\\easyandroid”, which contains all of the metadata and apps for EasyAndroid.

### Package Name

The “packagename” command updates the package name of the app to the value of the first argument. The package name mainly serves to uniquely identify the app – no two apps with the same package name can exist simultaneously on the same phone. Attempting to install an app with the same package name as another app would result in either failure, or replacing the other app. Furthermore, the package name serves to identify the source code files locations in the file directory as well. The default package name is set to “com.example” – so the full package extension of a project would be “com.example.[projectname]”, or in the default example for EasyAndroid, the full extension would be “com.example.helloworld”.

### Main Class

The “mainclass” command changes the main class of the app to the value of the first argument. The main class of an app is, in other words, the starting or home activity of the app. For users who make an app using EasyAndroid that want more than one activity class, the “mainclass” commands enables them to change which the activity from which the app first starts. The main class also affects the definition of the AndroidManifest – actually, the AndroidManifest is the only place where the main class is indicated. The default main class is the default name of the app’s first activity: “MyActivity”.

### Add File

“addfile” is an essential feature for the tool, particularly because of its usage as a part of other commands. By itself, the “addfile” command creates the Java source code file on disk for the current activity class that is being manipulated by the user. The path and the package name of the current class determine its location in the file directory. Because it simply operates on the current class, “addfile” takes no arguments. As previously mentioned, “addfile” takes advantage of the “print” command – it takes the output from the “print” command and uses that to populate the new source code file on disk. If the file already exists, running the “addfile” command overwrites the old source code. The command also adds the activity to the AndroidManifest data structure – every activity class needs to be declared in the AndroidManifest in order to be used by the app. The underlying data structure automatically denies the addition of duplicate classes to the manifest. Because it is more likely than not that a first-time user would not think to run the “addfile” command, the “build”, “install”, and “run” commands all run “addfile” before running their respective Ant build scripts.

### Class Name

The “classname” command allows the user to rename the current activity class that he or she is developing. The first argument of the command is the new value of the argument. This functionality is most important for apps with more than one activity class; single-activity apps can simply stick to the default name (“MyActivity”), and it will not change the app’s functionality at all. For apps with multiple activity classes, however, changing the class name is a necessary command, or else the user would essentially never be able to create a “new” class (it would overwrite the old class that has the same name). One potential bug that could easily arise as a result of this command is that if the user changes the class name of the main class, but does not update the main class app property, then the main class of the app – as far as it can tell – will not exist.

### Custom Function

The more experienced Java programmers also have the option of defining their own Java functions using the “customfunction” command, or “Create a custom function” in the GUI. This is provided mainly as a convenience to the user, so that if there is some Java code that he or she wishes to use in multiple places, rather than re-typing it several times, there is the option to define a function once, and call the function after that. One example of a suitable use for this feature is for a tip calculator that has three buttons that calculate 15% tip, 18% tip, and 20% tip. Rather than the user defining the actions in full for each of those buttons, he or she can define a function that takes the pre-tip amount and tip percentage as arguments, does the calculations, and returns the post-tip amount (or the tip amount alone). While this is not the best example, one could see a more complex function could be invaluable to the user.

Custom functions are defined in a similar manner to the definition of widgets. They take as parameters the function name, the return type, the parameters, and the body, all read in as Strings. This is the only option that actually *requires* fields to be filled in – mainly because it does not make sense to create a function with no return type, or no body (though one could make a case for “void” being a viable default return type). More importantly, if a user defines a function without one of its fields, then the result will probably not be what the user intended – in that case, it is better to simply stop the user from doing so. The function name and the return type simply expect Strings (spaces or invalid symbols will result in a bug in the app). Parameters are parsed as pairs; the first “word,” or String is the data type, and the second word is the name of the parameter. Both the parts of each pairs and the pairs themselves are separated by spaces (for example, “int a int b String text”). Lastly, the body expects compilable Java code – copy-pasting the body of a real function would work perfectly. Aside from that, there is nothing extraordinary about the “custom function” feature. The successful creation of a function adds it to the user’s app, and allows him or her to use it in the future.

#### Default Functions

Along with allowing the user to define their own custom functions, I provide a few functions that are available by default to the user. The first are the basic arithmetic functions: addition, subtraction, multiplication, and division. Each function returns the result of its respective arithmetic operation. Ideally, these default functions would be able to take in any form of number input and accept as many variations of user input as possible. The current implementation accepts two double inputs for all of the functions, and has return type of double as well. Because Java will automatically cast any integer input to double, but not the other way around, double is the more versatile data type in terms of accepting input.

In addition to these arithmetic functions, I provide by default a “sendsms” function. The function accepts two String parameters – the first being the phone number of the recipient, the second being the body of the SMS message (text). The return type of the function is void, and the body uses the Android library to send a text message to the phone number specified, with the message specified. For example, usage of the function would look something like:

**sendsms(“6091234567”, “Hello”);**

The main purpose of these default “custom” functions is to provide the user with useful functionality that a typical user would not be able to implement themselves. While the arithmetic functions are mostly for convenience, since most Java programmers would know how to perform the basic arithmetic operations without a separate function, the sendsms function in particular is something that no programmer would be able to implement without researching the Android library. In the future, my goal would be to provide more default functions that have the same type of utility – packaging useful and commonly desired Android features into functions that users can utilize in their own custom Android apps.

### Custom Import

As a complement to the “customfunction” command, the “customimport” command allows the user to add their own custom import statements to the current activity class. The command simply takes the value of the first argument as the package for the import statement to be added – for example, in order to add the custom import statement for the Java utility package, one would type, “customimport java.util.\*” or simply enter “java.util.\*” as the value of the import in the GUI. This would be translated in Java as “import java.util.\*” As previously mentioned, the main usage of this command would be to support a custom function that was made, but used some external library. While EasyAndroid takes care of the typical and default import statements – namely for the various widgets – it cannot know which imports are necessary for user-defined functions. As a result, this command is necessary in order to allow users to use the “customfunction” command to its fullest potential.

### Widget Commands: Add [Widget Type], Up, Down, Remove

#### (Add) [Widget Type]

Because the commands to add widgets are fundamentally the same, it is easier to first define them generally, and them point out the unique features of each afterward. Currently, there are four unique widgets available to the user: the TextView, EditText, Button, and ContactsList widgets. Each widget has a set of properties associated with it – for example its widget name, or the text that it displays. More specifically, *all* widgets have a height, a width, and a widget name associated with them. Most widgets have additional properties, but any basic widget has at least those three properties. Furthermore, each widget also has a particular Java source code output associated with it and its properties. The format of the commands is perhaps the most complex of all of the tool’s commands – the general format looks like this:

**[widget type] [-property] [argument value] [-property #2] [argument value #2]...**

So for example, a TextView displaying the text “HelloWorld” might look like:

**textview -name helloWorldText -text HelloWorld**

All properties have a default value, which is generally the most suitable or all-around best value for that property. In the previous example, neither height nor width – both mandatory properties for any widget – are specified. No specification by the user means that the widget uses the default values for that property, which for height and width are “wrap” (“wrap content” in full) – a special value for widgets which resizes the widget to match its contents. Had no name been specified (also a required property), the default widget name would have been “widget#” where # is replaced by the number widget that that widget is. So the default names would be “widget1”, “widget2”, and so on.

TextView is the Android name for the widget which simply displays text. In addition to the basic properties, and TextView widget also has a text property. The value of the text property is the text (String) that is displayed in the app. The default value of the text property is an empty String, or nothing. An example of the definition of a TextView was given above.

EditText is the Android widget equivalent for what is more commonly known as a text field – an editable space in which the user can input text. In addition to the basic properties, an EditText also has a default text property, and a hint property. The default text property is the initial (default) value of the EditText, which is editable. The hint property is the text that is displayed (typically in gray) if the field is empty at the time – examples of common hints are: “Enter message here,” or “Type here,” or “name.” Its function is typically to suggest a particular type of user interaction or user input. The default value of both of these properties is an empty String, or nothing. EditText and TextView are very similar, and so definition of an EditText looks quite similar:

**edittext -name myEditText -hint enter text here**

The Button widget is exactly what it sounds like: a button. Aside from the basic properties, the Button widget, like the TextView and EditText widgets, also has a text property, which is the display text of the button. Again, the text property has a default value of no text. Uniquely for the Button widget, however, there is also the action property, which is the Java code that is run whenever the button is clicked. Note that it the value of the action property is the *Java code* that is run – this means that users using this property need to input correct and compilable Java code as its value. Naturally, the default value of the action property is nothing. An example of a Button widget definition follows:

**button -name button1 -text print HelloWorld -action System.out.println(“Hello World”)**

Finally, a ContactsList widget is a scrolling list of the phone’s contacts. The ContactsList widget is the most dissimilar of the four existing widgets. In addition to the basic properties, the ContactsList widget also has an action property, a hasName property, a hasNumber property, and a divider property. The action property is exactly the same as with the Button widget, except that the Java code is run every time one of the contacts in the contacts list is clicked. The hasName and hasNumber properties are both boolean values, which are true by default. Actively specifying either property as anything but “1” results in a value of false. Finally, the divider property takes a String value – the default is nothing – which serves as the dividing text between a contact’s name and his or her number. An example definition follows:

**contactslist -divider ||| -hasNumber no**

#### Up, Down, & Remove

Likewise, the “up”, “down”, and “remove” commands are all similar – and simple – in nature, and so I explain them here together as a group, while pointing out the specifics of each. The “up” and “down” commands simply manipulate the position of the widget in the app, in relation to the other widgets in the app. Currently, widgets are shown in the app as a vertical list; “up” moves the widget up one spot in the vertical list, and “down” moves the widget down on spot in the list. The edge cases (widget is either at the top or at the bottom) are also handled – if the widget being operated on is at one of the edges and cannot be moved further, then the commands do nothing. As expected, “remove” deletes the widget from the app completely. For all three commands, the value of the first argument (an integer index) is the position of the widget that is to be relocated or removed.

In the GUI, the list of widgets displayed is the same order that the widgets appear in the app; thus, the index of the widgets of the displayed list is the same as the index of the widgets in the app list. This makes it extremely simple to translate the user interaction with the GUI into a parseable command for the underlying data structure.

## Graphical User Interface

\* cut this section down

The final major component of EasyAndroid is the Graphical User Interface (GUI). From the outset, I knew that the GUI would be perhaps the most important component of the tool, if not at the very least, an essential part. Although I could have left the tool as a “command line interface” application and spent the rest of my time enlarging the scope and functionality of the tool, because my target audience is not the old-school experienced programmer who starts programs through command line instead of double clicking, a GUI seemed imperative, or else EasyAndroid would simply be unusable by the target audience. Also, as important as it is for the underlying structure to be well-made and versatile, what the user spends the majority of his or her time looking at is the GUI. If the user does not understand how to interact with the tool, then any features or capability that it might have are completely useless. Consequently, I made a GUI for my tool, rendering the command line interface obsolete.

### Design and Functionality

In designing the GUI, I first laid out what the essential pieces were. Among these pieces were the project properties, a list of widgets the user could add, a panel for the definition and creation of a widget, a list of the current available functions, a list of the currently existing widgets, and lastly, a preview of the Java code equivalent of the user’s project. I also originally planned to have a “display AndroidManifest.xml” pop-out, but did not have time to implement that.

The GUI can be divided into three major sections: the menu, the header properties, and the components. Each of these parts is completely separate from the other parts in the GUI itself, and appears vertically from top to bottom in the order mentioned.

The top bar menu is intended to contain all of the “global” commands described in the previous section. Those include commands such as save file, open file, exit, build, install, and eventually help. A submenu “File” contains the “Save”, “Open”, “Reset”, and “Exit” commands, and a second submenu “Build” contains the “Build Project” and “Install Project” commands. As the tool was developed further, more submenus and more items in the submenus would be added of course. Furthermore, this design follows the typical Windows user interface model, in which administrative actions and settings are available through the menu, and content-related displays are in the “main window.”

The header properties section of the GUI maintains all of the project properties: “Project Name”, “Package Name”, “Path”, and “Main Class”. Each of these properties is displayed with editable text fields, though they are set to the default or current values of each property. Editing these properties and then building the project will result in a change in the project’s properties.

The last section, where the majority of content is, is the “components” section. This is the most complicated section of the three, itself further divided into (currently) five sections. From left to right, the panes are: options, functions, parameters, hierarchy, preview. Excluding the functions pane, all of these are related to each other and manipulating one potentially changes others. More specifically, the parameters pane determines what properties to ask the user to define based on which option is selected; choosing to add a widget after defining its properties will add the widget to the hierarchy pane; and the preview pane is a synthesis of all of the content from the project, so any persisted change in any of the panes will change the preview. A more in-depth explanation of each of the five panes follows.

The options pane is what it suggests: it is the options that a user has for widgets that could be added to the Android app. Currently, the available options are, “Add Button,” “Add TextView,” “Add EditText,” “Add Contacts List,” and “Create a Custom Function.” The last one, “Create a Custom Function,” is the clear outlier, but because it follows the same structure in which then defines the properties of the custom function, and then adds it to the project, it fits well as an “option,” though it is not a widget that will physically appear in the app. Additionally, “Create a Custom Function” is the only option that does not have default properties that replace empty input. This is because if the user were to attempt to add a custom function without defining all the properties, such as having an empty body to the function, then the function would most likely not fulfill the user’s expectations – in other words, it is probably a mistake. Accordingly, I force the user to enter values for the name, the return type, and the body of the function (functions can have no parameters, so I allow that). The rest of the options, however, are simply widgets that the user has the ability to add to his or her Android app. Because each widget – as well as the custom function option – has a different set of properties that can or need to be defined, choosing any of the options will trigger an update to the parameters pane if necessary.

The second pane is the functions pane, which lists all of the currently defined functions in the project that the user has available to him or her. These are all essentially helper functions, which are generally meant to be used in the “action” part of widgets’ properties. At the moment, this pane does nothing but display the name of the existing functions, but in the future, I would also have a display that shows the user an example of how to use whichever function is selected, or alternatively, shows the definition of the function. The user also has the option to remove any existing functions through a “Remove” button, in case he or she made a mistake in defining the function. In order to add functions to the list (and to the project), the user must define and add the function through the options and parameters panes.

Third, the parameters pane is where all of the properties for a particular widget appear. The parameters, or properties, appear as name-value pairs, with the name describing the property that the user is defining, and the value being an empty text field through which the user can specify a value. The contents of the parameter pane – that is, which name-value pairs appear – varies depending on which option is selected from the options pane. Lastly, above the parameters, is an “Add” button, which takes the widget defined in the pane and commits it to the underlying structure, if possible. Upon a successful addition, the properties are cleared, and the hierarchy and preview panes are updated.

The fourth pane is the hierarchy pane, in which the current list of existing widgets is displayed. Naturally, the list starts off as empty, but as the user adds widgets to the project, the names of the widgets are displayed in the list, in the order in which they will appear in both the code and the Android app. At the top of the pane is also a field for the current activity or class name, which by default, is “MyActivity”. If the user wants to give their activity a different name, they can alter it here. At the bottom of the pane are three buttons: “Up”, “Down”, and “Remove”, which allow the user to manipulate the placement of the widgets. “Up” moves the currently selected widget (if there is one) up one, “Down” moves it down one, and “Remove” deletes it from the project. Any changes made by these buttons are automatically reflected in the preview pane. In the future, this pane would most likely be replaced by a graphical layout designer of some sort, in which the user can simply drag and drop the different widgets in order to specify their placement in the app. Because implementing that is difficult, however, the current design simply lists the widgets in vertical order, and allows the user to manipulate their vertical placement.

Finally, the fifth pane is the preview pane, which shows the user a preview of the Java code associated with the project. Because users have the edit the code (though the changes are not actually reflected), the pane also includes a “Refresh” button to restore the state of the code, or in cases where the code may not be updated for some reason, the button updates the display.

### Implementation

Since I was already using Java for the underlying structure, I decided to stick with Java and use Java Swing for the implementation of the GUI as well. Because it was my first time using Java Swing without a book telling me what to type line by line, there was considerable time spent upfront – and throughout the development process – in learning the Java Swing API, and figuring out which classes to use and how to use those classes.

Before continuing, it is worth describing in detail the different Java Swing layout managers. Firstly, similar to Android, Java Swing uses the concept of containers – each of which is attached to a layout – which compose a “containment hierarchy,” the tree hierarchy of containers and their layouts. Moreover, the Java Swing library has a total of eight different types of layout managers, each with different limitations and advantages, and each for different purposes.

I started off using the GridBagLayout layout manager for the root container, largely because it seemed to be the most flexible, and my GUI was somewhat designed in a grid structure. My goal in using GridBagLayout was to reduce the depth of the hierarchy tree, and instead, define everything in terms of the root layout manager. The main problem, however, was that, in a GridBagLayout, each grid block needs to be the same size as every other block in that grid. This resulted in some sections of the GUI taking up more space than necessary with meant awkward spacing between different panes, and other panes requiring more space than was given to them, which made those panes cramped. Consequently, I switched to SpringLayout, which allows me to position the different items in the GUI in relation to each other. SpringLayout proved to be much easier to work with in general, and is what most of the panes in the GUI use. As an example, the header section and the components section are positioned in relation to each other through SpringLayout. Lastly, I also use BoxLayout, which lists its components as a single row or a single column – BoxLayout is useful for extremely simple panes that simply list objects. So for example, the five component panes that appear in a row – with the options pane on the far left and the preview pane on the far right – are arranged in a BoxLayout.

The main Java class file is SwingAppFrame.java, in which almost all of the GUI code lies (about 900 lines). SwingAppFrame extends the JFrame class, which is essentially the window class for Java Swing. The main function of this class simply initializes a new SwingAppFrame object and displays it. The constructor for SwingAppFrame sets some basic properties for the JFrame, including the total size of the window, the type of layout the frame uses, and the exit behavior. It also initializes a couple of instance variables: the main layout manager, the root container itself, and the CommandLineObject (referred to as CMD in the future), which manages all of the actual project infrastructure and data. Finally, the constructor adds the actual content of the GUI to the frame – the menu, the header pane, and the various components – and then sizes the window appropriately.

Implementing the JMenuBar (the menu bar) was probably the most straightforward part of the GUI, especially because the way I use the menu is not complicated. That is, while some menu bars might have submenus within submenus, my menu bar simply has submenus with items that, once clicked, perform an action directly. Perhaps as commands are added, the menu bar would need to become more complex, but as it stands, the menu bar is fairly simple, and thus, the implementation of the menu bar is also fairly simple.

First, I initialize the menu bar object and define the menu bar’s general properties: opacity, background color, and size. Following that, I define each of the “File” and “Build” submenus, and add the appropriate menu items to each one. For each of the menu items, I specify the ActionListener – that is, what happens when that particular menu item is clicked. Because each menu item corresponds to a particular command for CMD, the ActionListeners essentially send their namesake as a command to CMD. The only exception is the “Build Project” menu item, which, before telling CMD to “build”, first sends commands to update the project properties based on the text fields from the header pane and also the class name from the hierarchy pane, then sends the command “create” in case it is a first time build, and finally, tells CMD to “build.” All of this together results in the whole menu bar for EasyAndroid.

The header pane below the menu bar is also quite simple in implementation, despite taking a long time to reach this type of implementation. As previously mentioned, the header pane contains four project properties. For the display of these properties, each property is given a label and text field combination, which functions as a name-value pair. The array of the text fields is maintained through an instance variable, so that the values of these text fields can be retrieved later, upon project compilation. The properties are initialized, and then set in a “two-by-two” grid (because the labels and text fields each count as separate elements, the grid is technically two-by-four). In addition to these project properties, the header pane also includes a light blue divider line, using the class JSeparator, which serves to visually separate the header and component sections. And aside from positioning the header pane itself in relation to the rest of window, that is all of the header.

Finally, the components pane, to no surprise is the most complicated of the three sections. Specifically, the components pane is itself a container of five smaller items, some of which themselves are containers of more items. As mentioned above, this outside container utilizes a horizontally-aligned BoxLayout in order to simply display its child items in a row. While this could be managed through SpringLayout, using BoxLayout is more natural – adding a new pane, for example, would require a minimal amount of additional code, since the positioning of the pane would be taken care of. Other than that, the outside container is quite simple, aligned below the header pane. This outside container (the components pane) is held as an instance variable for the purposes of updating the interface upon various events.

Furthermore, I define several helper functions to reduce repetitive code in dealing with the components panes, since they can all be managed similarly. In particular, I define a method “generatePanel” which takes an ArrayList of JComponents as a parameter, and returns a container for the list items with a SpringLayout that aligns the components vertically, in the order that they appear in the ArrayList. BoxLayout could also be used here, but BoxLayout resizes its components to fill the area, making the configuration slightly more complicated, contrary to expectations. All of the component panes use this static helper function to convert the list of components for that particular pane into one container pane, which gets added to the outside container.

Each component pane, with the exception of the options pane, also has a “refresh” helper function associated with it. Because the options pane never changes, there is never a need to refresh it. The rest of the panes, however, are dynamically updated based on the user’s actions and input, and every time the user changes the project, the appropriate panes need to be updated to reflect those changes. These refresh helper functions are all defined in the same way: first, the outdated pane is removed from the outside container. A new pane is then created based on the current project state. The new pane is added to the appropriate position in the outside container, and then lastly, a call is made for the outside container to “revalidate,” or refresh itself, so that the new, updated pane is shown. These refresh helper functions are used whenever the project data has changed, and the UI needs to be updated.

In implementing the options pane, I considered a few different options before deciding to use a JList – that is, a list of selectable items. Some other options that I initially implemented were a ComboBox, which is essentially a drop-down menu, and a column of Buttons, one for each option. Ultimately, the JList was the easiest to use, and most scalable as well. In terms of how to use the JList, the JList class has a convenient constructor that creates a list based on an array of Strings; I define an array of Strings with all of the options (statically defined, or “hard-coded,” since the options never change and are not project-specific), and then JList takes care of the rest. While I could have defined a model that the JList can inflate, it was simpler to just use a String array. Implementing the ListSelectionListener (what happens when an item is clicked) was probably the most interesting part of this pane – I maintain which item is currently selected as an instance variable and update it in the listener. The reason why it is interesting is that when a user clicks an item in the JList, the listener, by default, registers both the user’s action of clicking down, and the action of releasing. What this translates to is two events for every one click, which triggers the listener and causes processing to occur twice as much as expected. Fortunately, there is method to distinguish between the two actions, and acting only on the release events solved the problem. Finally, the JList is wrapped in a scroll pane, so that if they were too many items in the list, it would be scrollable. Also, all of the component panes have a title at the top of the pane, which is simply a JLabel that displays the title text. Thus, the options pane container only actually holds two children – the JLabel that displays the title of the pane, and the JList that holds all of content. Using the helper function, these two items are arranged through SpringLayout, and added to the outside container.

The functions pane is almost identical to the options pane in implementation, also using a JList to display the content. The main difference is that while the options pane’s JList is populated from a static String array that I hard-code, the functions pane is dynamically populated based on the project state. To do so, I defined a helper in the CommandLineObject class that returns the project’s list of custom functions – from that list, I create a String array of the functions’ names, and use that array to populate the JList. Currently, this list’s ListSelectionListener does nothing, but, as mentioned in the design section, a future version of the functions pane would include a text display of some sort that offers guidance for the user on how to use functions – once that were implemented, the listener would trigger changes to that text display. The other major difference from the options pane is that the functions pane includes a “Remove” JButton at the bottom of the pane, which is simply positioned below the JList using SpringLayout. Clicking the button triggers the removal of the selected function from CMD, and then updates the relevant parts of the GUI (the function pane and the preview pane). The last step in the creation of the functions pane is, of course, using the helper function to place these components into a single container.

Thirdly, the parameters pane is probably the most dissimilar and most complicated of the panes. Like the other panes, it has a title display at the top, defined in the same manner. Below the title is the “Add” button, which when clicked, pulls the values of the text fields from the user interface, and processes them. Included in the processing is converting the text field values into a command line input format, and sending that to CMD to handle. For the options that add a widget to the project, clicking the “Add” button changes the project’s list of widgets, and so the hierarchy pane that displays that list is also updated. The main part of this section, however, are the parameters themselves, which are generated based on which item from the options pane is selected. Once it is determined which parameters are available for the particular item, for each parameter, a label and text field pair is added to a JPanel that holds all of these pairs – in the case of parameters that require more than just a few words or one line of text, a larger text area is attached to the label, instead of the text field. When the panel is fully populated, this part of the whole parameters pane goes below the “Add” button. And again, the helper function arranges these items in a single container.

Next, the hierarchy pane is extremely similar to the functions pane in implementation. Perhaps the most interesting part is the additional label and text field combination for the “Class Name” of the current file in the project. Changes to this text field are persisted upon build, but otherwise, it does not automatically update. Below the “Class Name” item, which is below the typical title, the list of currently existing widgets is shown, implemented in an analogous way to the functions pane. And, just like the functions pane, below the widgets list is a “Remove” button to manipulate the currently selected widget. Different from the functions pane, however, are the “Up” and “Down” buttons, which manipulate the currently selected widget’s position in the project. These buttons simply serve as an interface to CMD, which updates the project appropriately when either button is clicked. The three buttons are contained within a horizontally-aligned BoxLayout container. The main pane itself thus contains four child components: the title, the “Class Name” item, the list of widgets, and the item containing the buttons – these components are of course aligned using the helper function.

Last is the preview pane that shows a preview of the Java code that makes up the project. The most important and most noticeable part of the pane is the actual preview window itself, which is a scrollable text area, pre-populated by CMD’s Java code. Because the Java code does not have a character limit per line, nor a line limit, the code naturally requires the scrolling capability. The “Refresh” button above the text area simply re-generates the preview panel through the refresh helper function. Including the title component, there are three total components in this pane: the title, the “Refresh” button, and the preview text area. As always, the helper function takes these three components and puts them into a container, which is then added to the “main” outside container of the components pane.

That is the complete implementation of the Java Swing GUI for EasyAndroid. While it took quite a bit of time and effort to reach the solution that is currently being used, I feel that the key steps in implementing the GUI were 1) having a clear design from the start in terms of what to include, the functionality of each piece, and the placement of everything; 2) implementing helper functions and in general, approaching each section of the GUI in as similar a way to the other sections as possible; and 3) keeping things as simple as possible – I started off trying to use GridBagLayout, which led to more complications than it was worth; switching to using exclusively BoxLayout and SpringLayout made everything much simpler. Though the GUI is lacking in many regards, it still does the job – that is, a user should be able to build a simple app solely through the GUI.

# Limitations

Despite the usefulness of EasyAndroid, there are a plethora of limitations and weaknesses that still exist. Needless to say, EasyAndroid is incomplete; many of these limitations are due to a shortage of time. This section seeks to address the known limitations of the tool, and offer potential solutions or at least a rationale for those limitations.

## Simple Apps... Only

In its current state, the functionality of EasyAndroid is quite limited – the extent of its usefulness is creating Android apps that use only the four types of widgets available, and nothing more. While the number of apps included in that description is not trivial – an extremely useful TipCalculator, for example – it certainly inhibits the user’s creativeness. At the same time, the current version of EasyAndroid is perhaps better thought of as a prototype. Because of the way the tool’s structure is set up, adding new widget types to the tool does not take a large amount of time; the real work is in clearly defining the new widget and its properties. Of course, the fact that the process of adding a new widget type to EasyAndroid is manual means that it does not scale very well. But the simple explanation for why there are so few widget types is that I ran out of time and prioritized the GUI over expanding the widget list.

Even with dozens of widgets, however, EasyAndroid would still only be able to create “simple” apps, and that is something that is inherent in its design. EasyAndroid is not trying to make a complex, full-feature app, like the Facebook app or Angry Birds. In other words, the tool offers a finite set of functionality – not *everything* in the Android library. Developers who want to make those kinds of large-scale apps are better off learning how to use the Android library in full, rather than the abridged version presented in EasyAndroid. As a result, EasyAndroid will never be able to make an extremely complex app, but on the other hand, simple apps can prove extremely useful too (think of apps for tip calculators, alarms, to-do lists, etc.).

## Still Have the Problem of Setup Costs

Because EasyAndroid focuses on simplifying the actual process of programming for Android, it does not actually reduce the upfront download and installation costs. While EasyAndroid does in a sense “replace” Eclipse, and thus, reduce the installation size by a few hundred megabytes, the tool still depends on the user installing the Java SDK, the Android SDKs, and Apache Ant. Since EasyAndroid is a Java program, it does not require additional software, but this still leaves the problem of downloading and installing other software for a couple of hours, and then also needing to deal with problems such as setting the environment and PATH variables – there are plenty of steps at which the installation process could go wrong.

This is naturally the result of keeping the program local, and deciding to avoid hosting everything on the cloud. While EasyAndroid could definitely be hosted on the cloud and do all of the building and compilation online, doing so would mean that the apk is also built on the cloud. That would mean that the user would have to download and install the apk file separately, which can be slightly complicated, but also causes a break in the development process – it would not have the end-to-end aspect that EasyAndroid currently has. In short, however, EasyAndroid does *not* solve the issue of the painful setup process for Android development that was part of the motivation for this project. Instead, it aims to solve the step after setup, which is the development of the Android app itself.

## The Elegant Solution for Open File/Save File

As described earlier, the current implementation of open and save file is very much a “quick fix.” The “elegant solution,” as described earlier, would scale much better. Again, the main reason for choosing the “dirty solution” was that I found myself short on time, and so went with the easiest solution possible. As a result of implementing the dirty solution over the elegant solution, complicated apps would take a long time to load, and process obsolete commands.

## Lack of Error Detection

Another major missing feature for the tool is feedback about errors. First of all, the build step assumes that the tool’s code for the Android app is correct. Erroneous code causes the build process to fail silently. In a sense, the tool assumes that the user uses all of its features properly. Though the tool does its best to interpret user input, if the user input is malformed, such as uncompilable Java code for the custom function body, then the app will continue to function normally, and simply fail to build. Secondly, while some errors are outputted to standard output, that output does not manifest it in any way in the GUI, and so even those errors are never really displayed to the user. Thus, not only is there a lack of error detection to catch errors before build time, but also, errors that arise are not shown to the user – instead, the tool continues operating as if everything is normal, and fails silently.

## Vertical Layout Only

One major limitation of EasyAndroid right now is that the look and feel of its apps is relatively inflexible and underdeveloped. For simplicity’s sake, I made it such that the app simply displays a vertical list of its widgets. Though the order of the widgets is currently adjustable, the manner in which they appear is the same for all apps. A quick solution to offer more options for the appearance of the app would be to have a set of styles that the user could choose from – for example, a vertical list, a horizontal list, a grid layout. Perhaps the ideal solution, however, would be to implement a drag and drop layout builder, which would allow the user to arrange the app’s widgets exactly in the way he or she wants. Of course, more complicated configurations requires more complicated code, and would be more difficult to generalize, but as a part of the app building process, giving the user full control over the layout design is important.

## “Best Coding Practices”

Any experienced Java developer who looks at the Java code generated by EasyAndroid can easily tell that it hardly follows “best coding practices.” First of all, there are no comments explaining the code, which I find to be one of the most important features that is missing. But also, in terms of declaring variables efficiently (both in terms of lines of code and memory), naming conventions, modular programming, EasyAndroid makes no special effort to put any of those best coding practices to use. Instead, the focus is on generating Java code that gets the job done in the simplest manner possible.

At the same time, however, I feel that the absence of these best coding practices is justified – the tool is geared toward people who have barely seen code, and have no idea that a concept such as best coding practices even exists. While there is something to be said for teaching users how to code “right,” there is also the problem that most users using EasyAndroid will only vaguely understand the Java code preview, and for those that do, those for which employing the best coding practices matters are quite few. Moreover, someone who knows and has time to worry about the best coding practices, probably has the capacity to skip using EasyAndroid and learn Android directly.

# Challenges

This section mentions some of the main challenges I encountered during the creation and implementation of EasyAndroid.

## New Java Libraries

As expected, working with unfamiliar libraries had its difficulties. In particular, dealing with the input and output buffer classes (BufferedReader, FileWriter, BufferedWriter classes) took some research, as well as trial and error to get working correctly. Particularly in dealing with the open file/save file feature, properties that originally required additional user input – namely, the custom function and action properties – had to be reworked to process input from a single BufferedReader object, rather than a new one as I had originally done.

The other library that I primarily worked with was the Java Swing library, which came with its complications. Choosing between the options for LayoutManagers probably cost the most time; after working with Swing for a few weeks, I was familiar enough with the library to switch from the complex, yet limited GridBagLayout to the better, simpler option of combining SpringLayout and BoxLayout to achieve a better result that also improved scaling. Ultimately, because of my unfamiliarity with the Swing library, the GUI was one of the most difficult features to implement for EasyAndroid.

## Keeping the Design Abstract and Flexible

One of the challenges that existed from the beginning and remained throughout the development process was simply establishing the design for the tool. As mentioned before, there was a constant struggle to balance between keeping the tool’s structure as abstract and flexible as possible, while not wasting too much time setting up the structure. On the one hand, developing a specialized solution for some of the problems – such as hard-coding the available properties for each widget – was more realistic and would save unimaginable amounts of time for implementation. On the other hand, however, generalized and more elegant solutions would scale much better in the future, and would make future adjustments easy, or perhaps make more sense with the existing structure. Furthermore, as the tool continued to be developed, changes to accommodate unforeseen issues were made to the original structure, which rendered some older parts of the design obsolete. Waiting to implement those parts, then, would have saved me some time. But on the other hand, it would be foolish to wait forever – at some point, the line needs to be drawn, between waiting for the fuller picture so that the design can be optimized, and implementing the current design, so that the project can move forward. At the end of the day, there are plenty of tradeoffs to be considered, and it is impossible to account for every possible issue that might come up, and develop with every possible future feature in mind, and so much of the “wasted” or “lost” time was really necessary and unpreventable step in the process of refining EasyAndroid.

## Focus

As mentioned briefly in the last section, the direction and vision for EasyAndroid changed several times over the course of its developments. Perhaps more accurately, there are several more features that EasyAndroid could have included – and still can – and also multiple angles from which EasyAndroid could be portrayed. Just as an example, the question of who the target audience of EasyAndroid would be was crucial to shaping its functionality. *Because* EasyAndroid targeted inexperienced or new programmers in particular, the GUI was even more important. Had its target audience been computer scientists in love with Unix, then developing the command line interface further would have been a more appropriate next step. Similarly, the importance of a feature such as the Java code preview would be considerably less if the focus of the tool was not so much on helping the user learn Android development.

In the middle of developing EasyAndroid, there was the realization of just how much future versions *could* do, and all the different functionality that it could provide. Unfortunately, it would have been impossible to pursue all of those possibilities, and it was essential to prioritize the features that would have the most impact in bringing EasyAndroid closer to its goal and mission of simplifying the Android development experience for Android beginners.

# Evaluation

## First Iteration

For my first iteration of usability testing, I had three users test the tool. The experience was positive and encouraging, confirmed some suspicions I had about areas in which EasyAndroid was lacking, and exposed some unexpected shortcomings as well.

### Test Setup

The tests were not quantitative in nature; the only numbers that I recorded from this test were times for how long each part of the test took for each user. My main objective in doing usability testing was to find ways in which EasyAndroid could improve, and also confirming that it was viable and fulfilled its purpose. That said, the tests were quite informal – I allowed for roughly five minutes where I said essentially nothing, aside from answering questions in the most objective and unhelpful way possible (that is, encouraging the user to figure out the answer alone, rather than giving the answer). After that time, I interacted with the test subject much more, explaining what the function of different parts of the tool were, and guiding the user toward the “right answer” if he or she wandered for too long.

In terms of setup, I used exclusively my own equipment – my computer and my phone, which allowed for me to have everything started and running for the user when he or she began. That is, my phone was hooked up to my computer via USB cable, and EasyAndroid was running when the user began testing. It was also essential that I use my computer, because of the software that needed to be downloaded, as well as configurations that can be troublesome (i.e. path and environment variables, directory structure).

The first task presented to the user was to create a “Hello World” app – an app that simple displays the text, “Hello World”. Upon successful completion and installation of that introductory app, the user was asked to make an app with a button that texted themselves (from my phone) a predetermined (hard-coded) message, which they specified during development. The last task was to create an app that had a text field, and a button that texted themselves the contents of that text field.

All three of the users had at least some computer science background – Subject 1 had the least amount of experience (currently taking COS 126), Subject 2 had slightly more (currently taking COS 226 with AP Computer Science credit from high school), and Subject 3 had the most experience (has taken all of the COS introductory courses).

### Results

As far as quantitative results go, Subject 1 took 15 minutes, 15 minutes, and 7 minutes, for the three tasks, in their respective order. Subject 2 took 10 minutes, 2 minutes, and 2 minutes. Subject 3 took 3 minutes on the first task, 5 minutes on the second task, and 4 minutes on the last task. It should noted that test subjects performed these tasks *while* exploring the tool for the first time, and so the times vary between tasks depending on at which point the test subjects decided to explore the tool more.

|  |  |  |  |
| --- | --- | --- | --- |
| Task | Hello World | Static Text Message | Dynamic Text Message |
| Subject 1 | 15 minutes | 15 minutes | 7 minutes |
| Subject 2 | 10 minutes | 2 minutes | 2 minutes |
| Subject 3 | 3 minutes | 5 minutes | 4 minutes |

### Feedback and Takeaways

Subject 3 especially breezed through the tasks, even taking his time in between the tasks to unnecessarily rename his widgets and define irrelevant functions. He described the experience as both fun and easy. It was clear that he had a comfortable understanding of how the tool worked, and overall positive review. His main feedback was that the steps that required Java code were not really explained anywhere, which is both accurate and valuable. Subjects 1 and 2 likewise responded positively – all of the subjects rated the experience 5 out of 5.

As expected, the purpose of the “functions” part of the tool was unclear. Subject 1 inferred from the default function names (addition, subtraction, multiplication, division, and sendsms) that the first four functions composed a form of calculator, and the sendsms function was related to chat. All three users also struggled with determining how to use functions on their own; with some explanation, however, they were all able to successfully incorporate the functions into their apps. Subjects 2 and 3 in particular understood immediately once I explained to them how to use it.

Because I had no written documentation available, the test subjects were sometimes uncertain about the functions and usage of most things in general. For example, Subject 2 was confused about how to fill in the “height” and “width” properties – “What unit is this in? What should I put here?” As a result, she initially placed extremely low values arbitrarily, like 4 and 2. Because there was nothing to tell her that no input would result in a generally better default value, or that a number would be processed in pixels, she had no idea what kind of input the text fields were expected. Subject 1 behaved similarly, asking, “What is height and width?” – while she was fully aware of their English meaning, she wanted some feedback or some hint toward their function in EasyAndroid.

Contrary to my expectations, the actual function of the “Action” parameter was not intuitive to the test subjects at all. Subject 2 had an interesting take on Action: for the “Hello World” program, she entered as her action, “say hello world” (quotation marks mine). Similar to an in-line in Java, she used Action as a description for the widget. In particular, because some text fields in EasyAndroid require Java Code, and others require plain text, it is confusing how the “Action” parameter is supposed to be filled.

Another surprising user behavior was the tendency to fill in the “name” property with the value that would be more appropriate for the “text” property. In particular, when doing the “Hello World” app, all three test subjects naturally filled in “Hello World” as the name of the widget; Subject 2 also left the “text” property blank. And so, the function of the “name” and “text” properties was unclear – the first instinct of the test subjects was to fill “name” in with the value that they wished to be displayed in the app.

Additionally, the “build” and “install” menu items were not as obvious as I had expected them to be. Though Subject 3 was certainly aware of the menu bar, he, along with the other two test subjects, took some extra time in finding the build function. Instead, the test subjects excepted the build and run functionality to be somewhere in the main window. Furthermore, as mentioned in the “Implementations” section, I noticed that the asynchronous nature of running the “build” and “install” commands in the background did not give the program enough time to compile. In particular, users would click “Build Project”, which would process the command and start the background operation, and then because the GUI did not freeze up and wait for the background operation, the user would immediately attempt to “Install Project” afterward. This resulted in a failure, because the build process had simply not yet completed.

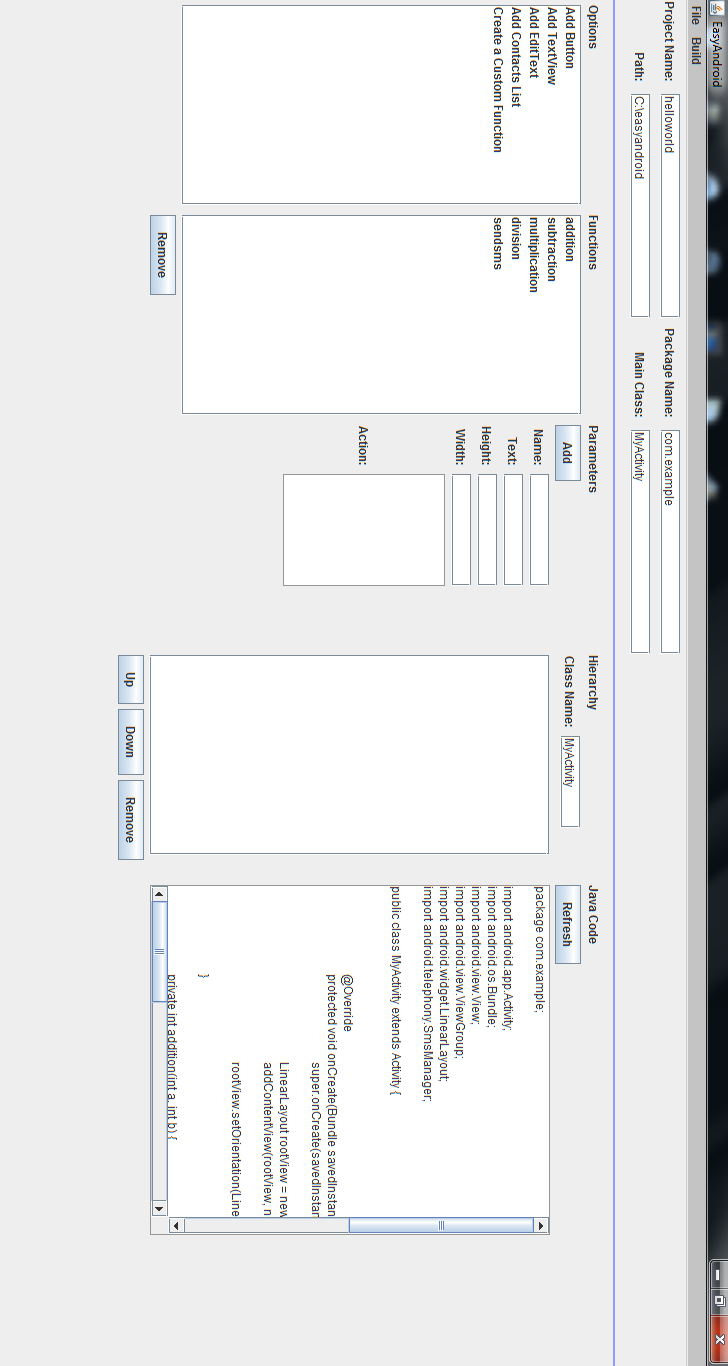
In general, this round of usability testing provided useful insights, both into how users interact with EasyAndroid, and into the shortcomings of its functionality and terminology. Still, the fact that the test subjects all completed all three tasks – the longest being just 37 minutes – and had an enjoyable experience while doing so, demonstrated that the first iteration of EasyAndroid as a prototype was a success.

## Improvements

As a result of the usability testing, I made some changes – hopefully improvements – to EasyAndroid. Among them was a bug not mentioned earlier, which appeared when the user tried to add multiple ContactsList widgets to the app. Aside from that bug, with the goal of emphasizing the direct nature of the relationship between the “Options” pane and the “Parameters” pane, I rearranged the panes from “Options”-“Functions”-“Parameters” to “Options”-“Parameters”-“Functions.” Though the change is somewhat minor, I expect it to clarify the role of the “Options” pane and the “Parameters” pane’s dependence on it.

To address the issue of the confusion of the “name” and “text” properties, I made very subtle changes to the names of those properties. Specifically, the “name” property is now shown with the label “Widget Name”, and the “text” property has the label “Display Text” (“Default Text” in the case of the EditText widget). Hopefully, these small changes give the user a better sense of exactly what those properties are.

Perhaps the most significant change, however, is the addition of the “Run” button, and the “Run” command. As explained in the “Implementations” section, the “Run” command is essentially “create”, “build”, and “install”, all in one command. The reasoning behind having it all as one command is that users will rarely want to build but not install – so rather than forcing the user to make run three commands (or click two menu items, in the case of the GUI), I made the process into a single command, and a single button, which – this is important – is extremely prominent at the top right of GUI’s main window. My expectation for the “Run” button is that it will make it almost offensively obvious and simple as to how to turn the development in the tool into an actual Android app.



# Future Steps

If this report has not yet made this clear, I will say it again: EasyAndroid has a lot of potential. More formally, the potential for further development and expansion of features and functionality of EasyAndroid the programming tool is more than enough to keep a startup busy for at least a year. In this section, I address some of the ways in which EasyAndroid could have been improved even further, given more time and resources.

One feature that benefits EasyAndroid in terms of scaling for larger or more complicated apps is the capability to read and process external files into the project. While users can technically do this manually via accessing and adding files to the project directory, incorporating it as a native feature for EasyAndroid would not only make it easier for the user, but would also ensure that the new class files are added properly to the project – in particular, if the user were to try to add a new activity class file, using it would require that he or she add it to the AndroidManifest, which is a complication that really hinders the development. Of course, this is very much a feature for more advanced users, and therefore, fairly low priority.

Naturally, the GUI can be always be improved, as proven by the usability testing. Among the features that would have a major impact is the drag and drop layout design feature that was mentioned in the “Limitations” section. The drag and drop feature is crucial both in that it improves the user experience significantly, but also makes it easier to portray what the user wants to build as their app. For example, position widgets in relation to each other is fairly unintuitive to describe via text – having a drag and drop feature, however, would facilitate that process quite a bit. Aside from that major feature, little details such as renaming the terminology of the GUI, and changing the look and feel and the size of some of the text windows could also improve the user experience.

Additionally, as mentioned in previous sections, the “elegant solution” for the open file/save file feature would implement the concept of a state, which would capture all of the properties of an EasyAndroid “app.” Because this has been explained in detail already, I omit the mundane details of exactly how the implementation would differ – in short, however, the elegant solution simply scales significantly better than the current implementation.

An “Android docs fetcher” – that is, a data scraper to convert Android docs into a data format that my program can parse – would be helpful as the tool scales to include as much of the native Android library functionality as possible. Having this tool would mean that the process of deciding what properties a particular widget has, and which widgets are available in EasyAndroid, completely automated. While there are situations where one might want to manually override the output of the automated process, the docs fetcher would be particularly helpful for including the more obscure properties that widgets can sometimes have – such as text color, or font, or “scale type” for images (like I said, obscure).

Furthermore, the implementation of a “meta app” would prove extremely useful. Actually, as the EasyAndroid tool is itself a sort of “meta app” in that it generates code for an Android app, this new level of “meta app” would be a “meta meta app.” This “meta meta app” would take input from the “Android docs fetcher” described earlier, and generate the Java code that represents the custom “Android widget” classes of the EasyAndroid tool. Having this “meta meta app” would automate the process of creating those custom Android widget classes – even without the “Android docs fetcher” tool, simply reading from a formatted text file that can be appended would reduce the marginal cost of adding new widget types. Using the “Android docs fetcher” and the “meta meta app” in combination, however, would completely automate the process of going from the Android documentation to Java Object classes that the EasyAndroid tool can use.

Another feature that I considered early and was actually a part of my original timeline was an Android emulator style “preview” screen. It would essentially show the user what their app would look like on an Android phone – I postponed implementation of this feature because I decided that it was non-essential (users can just build and install the app onto the phone or an emulator). However, having this preview screen *while* developing could certainly be valuable, especially because the user would be able to see the results of their changes to the app in real-time.

As mentioned near the beginning of this paper, the project structure can always be abstracted as much as possible. Taken as far as possible, more complicated code can be abstracted or simplified such that an inexperienced user can use even complicated code. For example, more “default functions” can always be added to provide shortcuts for the user. Particularly in the case of services that are commonly interacted with – YouTube, Twitter, Facebook, for example – providing a shortcut function for the user would prove quite helpful, especially since it is unlikely that the user would be able to do so themselves.

Likewise, introducing a much more event or trigger-based style of app is definitely doable, and would probably make the most sense with mobile apps, which are largely built around user interaction (events). The only other triggers in mobile apps are really environmental or time-based triggers, or on start-up.

All of these ideas are future steps for EasyAndroid – some could be implemented immediately, while others are much more long-term in nature. The main takeaway, however, is that EasyAndroid has plenty of potential for growth, expansion, and improvements.

# Conclusion

In conclusion, my progress with EasyAndroid both pleasantly surprised me with its success and fell short of my expectations. As explained in the previous section, the potential for this development tool is quite significant, and I could honestly see it becoming something revolutionary. Ultimately, completing everything that I had initially planned and imagined to accomplish simply would have required a much greater time commitment. Plenty of the “future” features could have been added – such as extra “shortcut” functions – had I simply put more time into EasyAndroid’s development.

One of the key lessons and a constant struggle that I carried with me throughout the development of EasyAndroid was the importance of maintaining a narrow focus. What my original timeline boils down to is essentially, trying to do too much with too little time. Had I had a clearer idea of exactly what I wanted to do from the start, I could have saved myself a lot of time along the way, bouncing between options. The lack of focus led to my implementing tools that I did not necessarily need, and failing to account for tools that I did need in some cases as well.

Another key lesson is that I should have emphasized research on related products and tools early on. I assumed that the four or so related tools that I discovered quickly would be enough and would background EasyAndroid the best, but then I discovered Andromo – the closest product to EasyAndroid – months into development. Perhaps, had I known about Andromo from the start, I could have shaped the design of EasyAndroid differently to further accentuate its advantages over products like Andromo.

I also learned that you can never account for everything. Even spending as much time as I did on design and planning, there were still issues that I missed, which forced me to backtrack and change EasyAndroid. For example, in specifying custom functions, I originally created a new BufferedReader to capture the user’s definition of the custom function. That method, however, did not work well with open file/save file feature, and so I had to restructure that whole method so that I could parse a list of “saved” commands seamlessly. Many other examples of my failing to account for potential issues manifested themselves in similar ways, consuming my time.

Creating, developing, and implementing EasyAndroid taught me that “abstracting” an entire platform or tool is not easy, and in most cases, impossible (or at least, impossible to do in a year as a senior thesis / fifth class). The entire process showed me the importance of understanding my limits – while being ambitious, optimistic, and a visionary are all important in their own ways, it is also important to understand the limits of my capabilities. If I look my original timeline, I technically only completed about seventy percent of what I had originally planned to do. Even though I had consciously planned optimistically, being realistic and setting priorities – and cutting the non-essentials – is also a lesson I learned through this experience.

I sincerely believe that if I were developing this tool full-time, it could be a real, valuable, even profitable product; certainly, its educational value would be significant. All in all, I am proud to say that the product and technology that I created is pretty cool. Seeing users who had never touched Android code before in their life – and a couple of users who were still struggling to grasp the basics of Java – seeing the excitement of those people as they created their own Android apps (pretty useful ones too, to be honest), was probably the highlight of this entire process. I suppose it was the fruit of a yearlong adventure – but nothing felt quite as good as the moment when I realized, “It works!” And now, curious people who otherwise might have never made their own Android app, can get to say that, “I made an Android app. And it was easy.”

1. <http://www.theregister.co.uk/2013/04/05/android_market_share_slipping/> [↑](#footnote-ref-1)
2. <http://info.scratch.mit.edu/About_Scratch> [↑](#footnote-ref-2)
3. [http://appinventor.mit.edu](http://appinventor.mit.edu/) [↑](#footnote-ref-3)
4. <http://twill.idyll.org/> [↑](#footnote-ref-4)
5. <https://ifttt.com/wtf> [↑](#footnote-ref-5)
6. (“Andromo - Make an Android App. No Coding Required”) [↑](#footnote-ref-6)
7. (“Andromo - Make an Android App. No Coding Required”)same as previous footnote [↑](#footnote-ref-7)
8. <http://www.appcelerator.com/> [↑](#footnote-ref-8)