EasyAndroid: A Developers' Tool for Android Beginners

Android Development Made Easy

Mark Ha '13
Adviser: Brian Kernighan

0: Honor Code

This thesis represents my own work in accordance with University Regulations.

Mark Ha

0.0: Acknowledgements

Thanks to **God**, through whom all things are possible.

"With God all things are possible."
"If you believe, you will receive whatever you ask in prayer."

Thanks to my **parents** and **siblings** who have supported and loved me unconditionally for twenty years, even if I still do not fully understand and appreciate what that means.

Thanks to my gracious, humble, and patient adviser, **Brian Kernighan**, who never said a single negative remark despite my lack of progress, postponed meetings, and failure to meet goals. I don't think I quite made what you wanted, but I'll revert it to the command line version so that your seventy-year-old bones can play with it. I'll miss my almost-weekly meetings with you, whether we were talking about my thesis, or some tangent of a tangent of my thesis. To a year and a half to being mentored by one of the greatest and most influential computer scientists in history, cheers.

Thanks to my second reader, **Margaret Martonosi**, who agreed to be my second reader last minute and will surely enjoy these one hundred or so pages (there are pictures I promise).

Thanks to my ~~guinea pigs~~ test subjects, to whom I owe at least ten of these pages: **Rebecca Tang**, **Elizabeth Yang**, **Lucas Mayer**. You all affirmed that I made something worth making.

Thanks to **Miss Diane Cho**, who somehow proofread my thesis incredibly, despite knowing nothing about computer science. I was actually amazed by the corrections you made, despite technical contexts. I wish you the most pleasant of turtles and houses.

Thanks to **Stephanie Park,** who helped me with those citation things, which I still don't really understand.

Thanks to **Kristen Kim**, who met me the first day of school and has never left me since. You know more about me than I'm comfortable admitting, but that's okay.

Thanks to the rest of the **Manna seniors**, who shared in my thesis struggle for half of the time, fed me for the rest, and loved me throughout.

Thanks to **Lydia**, for being there, for understanding, for loving, for everything. Meow.

```
...
Hello, World!
```

0.1: Table of Contents

1. Introduction	5
2. Motivation	11
3. Related Works	14
4. Design Decisions	28
5. Implementation	37
6. Demo	72
7. Limitations	80
8. Challenges	84
9. Evaluation	87
10. Future Steps	94
11. Conclusion	97
Appendices	99

1: Introduction

"EasyAndroid" is a desktop application that enables users with little to no programming or computer science experience to be able to create their own Android applications without needing to know the details of the code that goes behind it. This way, a user with a clear design or product in mind should be able to transform his or her 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 is meant to be. If a person with relatively little software development experience can create a practical app, then I consider EasyAndroid 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 mere minutes 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 additional guidance or perhaps examples, but once the user understands how to do so, would only take a minute. Applying that model 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 half an hour for a first-time user. With further development, EasyAndroid could reasonably become a viable replacement for Android developers, for small-scale Android apps.

1.0: The Process of Building EasyAndroid

When starting the design process for EasyAndroid, I used an approach aimed to maximize its usefulness for average users. The approach can be broken down into two main steps. First, what do people want? Of course, there are the visionaries with grand ideas about an app that might make millions, but would require a real team of developers – those ideas are meant for startups. Average users, however, usually have 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 a couple of those apps pre-installed, or they already exist in the App Store (for iOS users) or the Google Market (for Android users). Even if an app with similar functionality already exists, however, most people whom I talked to were reluctant to put in the time to search for the app that best fit their needs, wished for some extra functionality or small variation to the app, or had reservations about downloading uncertified apps due to privacy concerns (i.e. "Why does the app need my personal data to calculate tip?"). In short, most people – especially non-developers – have at least one simple app idea in mind, which, given the proper tools, they could even develop themselves. That is what gave me an idea of what kinds of apps people are looking for.

Secondly, what goes into building those apps? 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 the way I built the "barebones" version of EasyAndroid. By "barebones," I mean that even a barebones version of the tool would have the capability to create a majority of the apps described. 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.

1.1: Terminology (for the purposes of the paper)

Going forward, I will use some terminology and jargon that may be unfamiliar to some readers; I define them here 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. In particular, EasyAndroid is itself a desktop application, and so referring to simply “the application” can be confusing – am I talking about the Android application that the user is building, EasyAndroid, or some other software application? Thus, I refer to the EasyAndroid application by its name, or as a “tool,” and I refer to Android applications as “apps” or “Android apps.”

Additionally, when talking about the implementation structure, the directory structure that represents and will eventually become the “app” is called the “project.”

apk: The Android “application package file” (APK) is the file format in which Android apps are distributed and installed. For the purposes of this paper, the “apk file” can be thought of as the finished product – an executable file 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 a programmer writes code to interact with a particular piece of software.

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, a spinning icon, etc. – I will use the term "widget," partly because the word "object" has a special meaning in the context of Java, and partly out of lack of a better term. Each of these widgets also has a set of "properties" which define it; the properties will be described in more detail later.

Activity: Though not an entirely accurate definition, it is enough to think of an "Activity" in the context of Android application as equivalent to the container for 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 an Activity. In terms of implementation, any given Activity is a Java class, hence my occasional reference to "the Activity class." Most of the user's interactions with EasyAndroid will be geared toward creating and customizing their app's "main" Activity, which will be described in fuller detail later. Again, while the definition of an "Activity" is actually much more complex, in the context of this paper and EasyAndroid, it is easier to consider an "Activity" as synonymous with a particular screen of the app.

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

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

Version 2 or 2.0: Refers to a change that was made or added for the second iteration of the EasyAndroid code – that is, improvements introduced after the first (and only) round of usability testing.

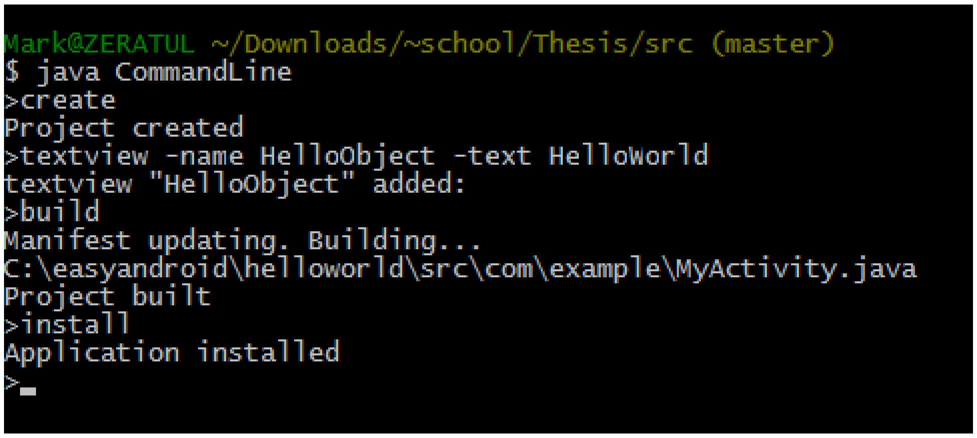
1.2: Building Up "Core functionality" for EasyAndroid

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 cannot be expected to know how to open up their computer's command line shell, and enter the correct Apache Ant (a software tool) build script. While it might be feasible to have step-by-step instructions for that part, there are several variables – for example, the build path, whether the build is meant for debugging or release, and behavior if an app of the same type is encountered – that the user probably would not understand and thus, input incorrectly. By containing all of the messy operations that happen "under the hood" within the tool, the required interactions from the user are reduced, and user experience as a whole is improved.

I envisioned the "alpha" version of the tool as 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 100
```

This command would add the corresponding button (one that says, “hello world” with a height of 100 pixels) to the current project. Thus, through a series of commands that would come through the “command line” interface, the user would be able to design and create the app that he or she desired, and then have the app built (compiled) and installed onto his or her phone.



```
Mark@ZERATUL ~/Downloads/~school/Thesis/src (master)
$ java CommandLine
>create
Project created
>textview -name HelloObject -text HelloWorld
textview "HelloObject" added:
>build
Manifest updating. Building...
C:\easyandroid\helloworld\src\com\example\MyActivity.java
Project built
>install
Application installed
>
```

1.3: Expanding the Core Functionality

Upon completion of the initial core functionality, I would develop on top of its structure, adding more and more “features,” from simply more types of widgets, to a GUI that would replace the command line interface. Once I had built enough features so that a user could use the tool successfully without too much difficulty, I would run usability tests and make improvements based on the testing and user evaluations.

2: Motivation

2.0: Mobile is Hot – Android is Even Hotter

Upon hearing my thesis topic, people usually ask, “Why Mobile?” and quickly follow with, “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]. In my particular case, Android is an even more obvious choice, because I have a full year’s worth of Android development experience (a couple full-feature Android apps, along with several smaller projects). So not only would an Android tool appeal to a larger audience, but it also better suits my skills.

2.1: High Entry Cost for Android Development

The name, “EasyAndroid,” sums up the objective of this tool for beginner Android developers: 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 – requires an assortment of downloads and installations, although the process is much more well-documented and well-packaged than it was a year or two ago. Specifically, the user needs to have the Java SDK, the Android SDK, Eclipse (or some other powerful editor that can run Android), and, depending on the user's particular needs (for example, an app meant for phones running Android 4.0 would need the corresponding Android 4.0 SDK), other additional Android SDKs, tools, platforms, and extra software. 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.

[1] Rik Myslewski. (Apr 5, 2013). *Android's US market share continues to slip*. Retrieved Apr 10, 2013 from http://www.theregister.co.uk/2013/04/05/android_market_share_slipping/.

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 significant. Extensive Google searching may eventually answer the question, "How do I make an application that does ___?" but 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" found online. 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 compensates for older Android phones. 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 tutorials, it takes both reading and experience to fully understand how to use these Android-specific features properly – mishandling "Context" objects, for instance, can easily cause memory leaks and eventual crashes in Android apps.

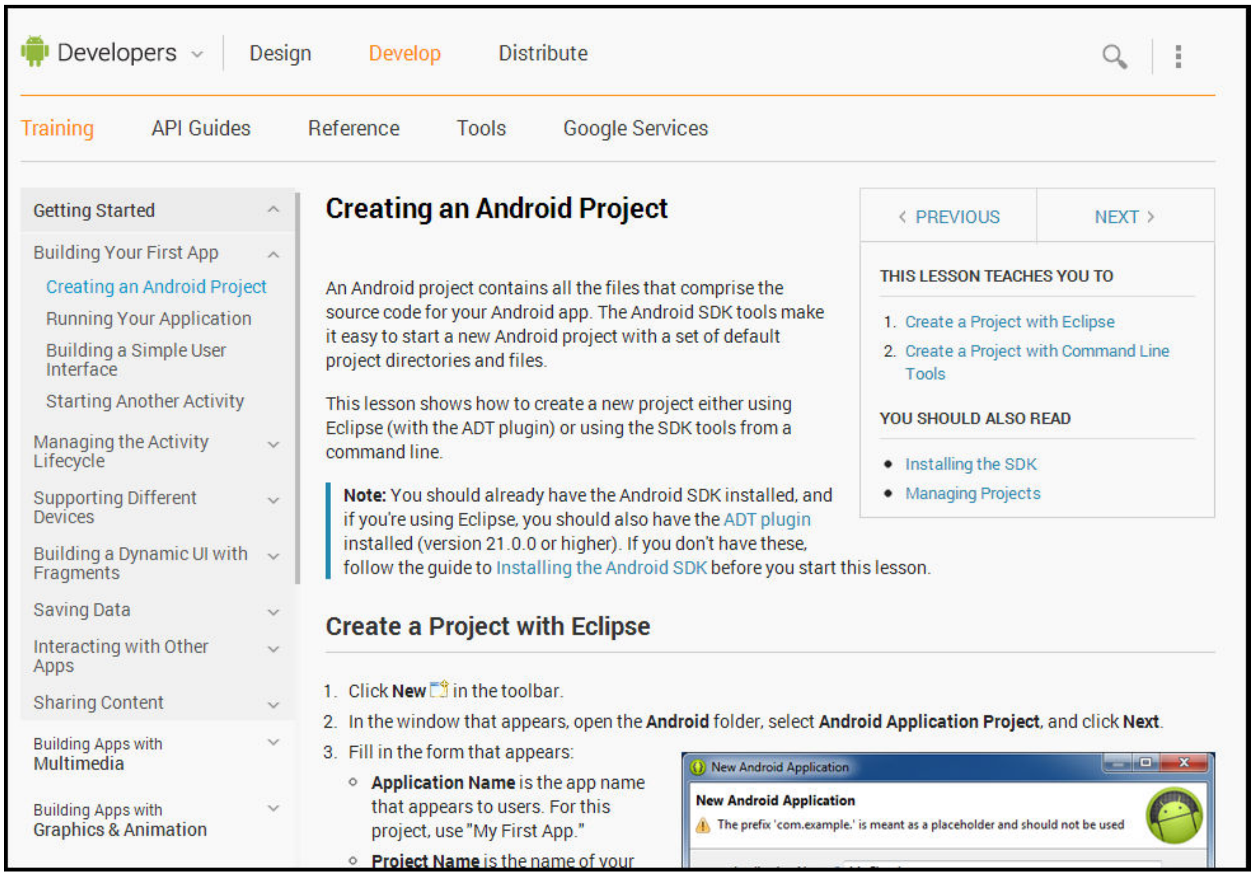
Therefore, on top of the substantial software setup that is required to begin Android development, learning how to program specifically for Android – even for an experienced Java programmer (Android apps uses the programming language Java) – consumes a considerable amount of additional time. If the user also needs to learn Java, then even more time is required. In short, the combination of overhead costs presents a serious impediment to any curious individual contemplating starting Android development. EasyAndroid reduces those costs.

2.2: Limited "Tryout" 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. Meaning, if you start using AppInventor, you have to continue to use AppInventor, or else lose your invested time if you choose to transition over to traditional Java development.

Other products that provide "Android app building" services are similar in that the product does not carry over into real Android development. So, "trying out" Android app development essentially becomes downloading and installing the relevant software, and then progressing through the Android tutorials on the Android website or some other website.

3: Related Works


Developers
Design
Develop
Distribute
Training
API Guides
Reference
Tools
Google Services
Getting Started
Building Your First App
Creating an Android Project
Running Your Application
Building a Simple User Interface
Starting Another Activity
Managing the Activity Lifecycle
Supporting Different Devices
Building a Dynamic UI with Fragments
Saving Data
Interacting with Other Apps
Sharing Content
Building Apps with Multimedia
Building Apps with Graphics & Animation
‹ PREVIOUS
NEXT ›
New Android Application

Creating an Android Project

An Android project contains all the files that comprise the source code for your Android app. The Android SDK tools make it easy to start a new Android project with a set of default project directories and files.

This lesson shows how to create a new project either using Eclipse (with the ADT plugin) or using the SDK tools from a command line.

Note: You should already have the Android SDK installed, and if you're using Eclipse, you should also have the ADT plugin installed (version 21.0.0 or higher). If you don't have these, follow the guide to Installing the Android SDK before you start this lesson.

THIS LESSON TEACHES YOU TO

1. Create a Project with Eclipse
2. Create a Project with Command Line Tools

YOU SHOULD ALSO READ

- Installing the SDK
- Managing Projects

Create a Project with Eclipse

1. Click **New** in the toolbar.
2. In the window that appears, open the **Android** folder, select **Android Application Project**, and click **Next**.
3. Fill in the form that appears:
   - **Application Name** is the app name that appears to users. For this project, use "My First App."
   - **Project Name** is the name of your

New Android Application
The prefix 'com.example.' is meant as a placeholder and should not be used

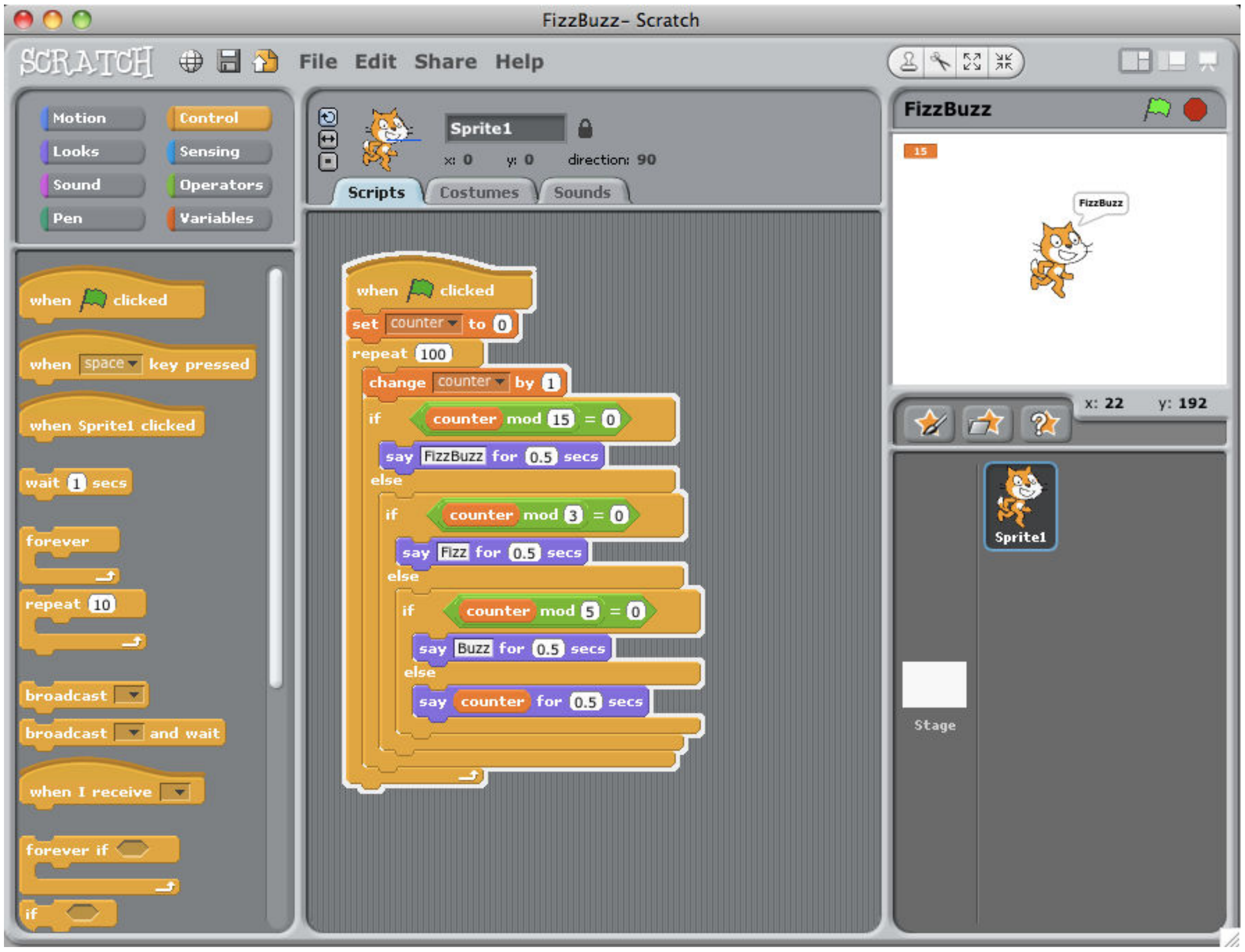
What Google Offers Its First Time Android Users[2]

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, for good reason, is the Android library source code API and documentation – essentially "Java docs" for the Android library – which is available online for free. 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[3]. Both of these documents provide information and often sample code to the user,

---

[2] *Creating an Android Project*. (n.d.). Retrieved April 25, 2013, from http://developer.android.com/training/basics/firstapp/creating-project.html.
[3] http://developer.android.com/develop/index.html

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 types of applications are the most similar to EasyAndroid, I consider several of them in this section.



```
when clicked
set counter to 0
repeat 100
    change counter by 1
    if counter mod 15 = 0
        say FizzBuzz for 0.5 secs
    else
        if counter mod 3 = 0
            say Fizz for 0.5 secs
        else
            if counter mod 5 = 0
                say Buzz for 0.5 secs
            else
                say counter for 0.5 secs
```

Scratch: Block-Based Programming[4]

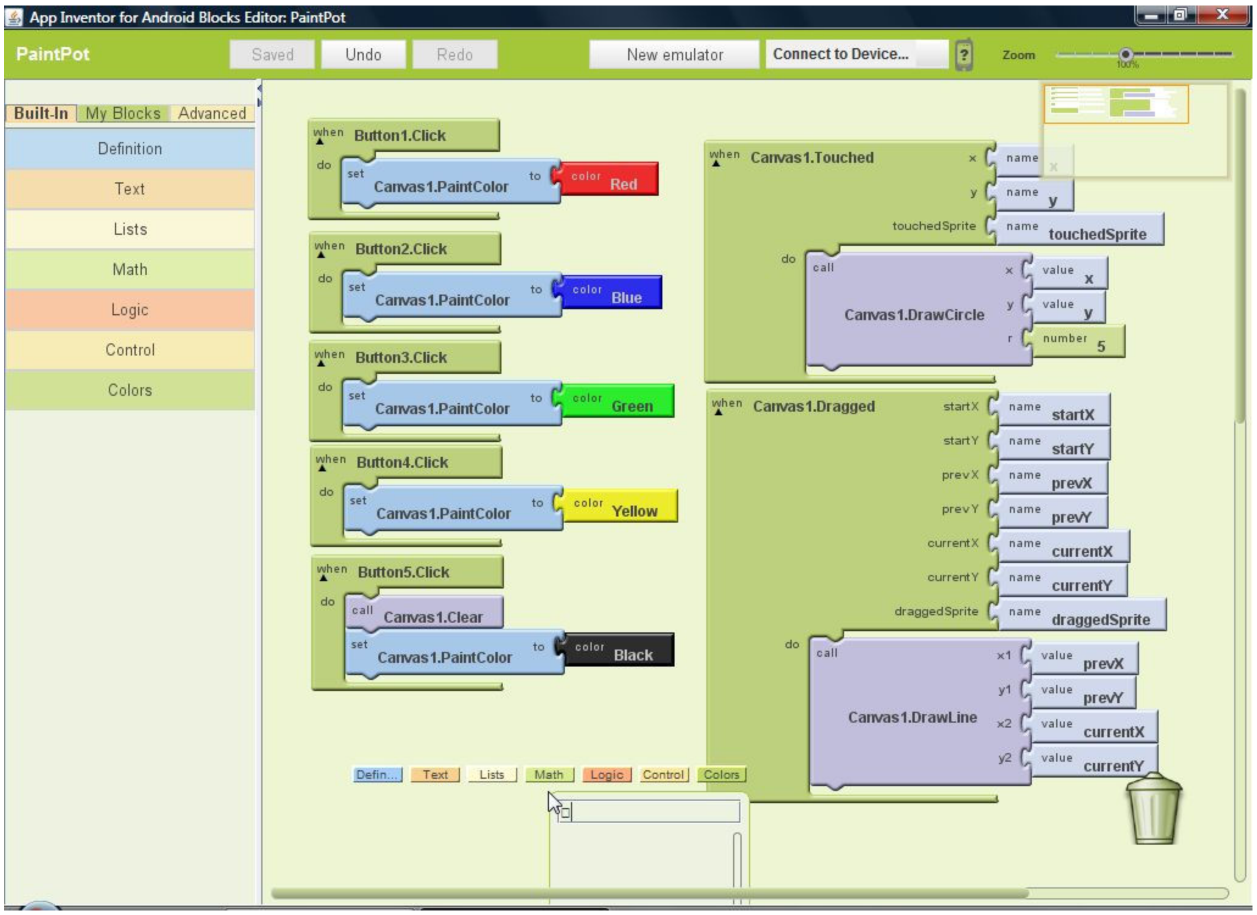
[4] *Scratch.* (n.d.). Retrieved Apr 15, 2013 from http://programming.dojo.net.nz/languages/scratch/index.

3.0: Scratch

Scratch is a block-based programming language designed for early or new programmers (specifically young children), developed at the Massachusetts Institute of Technology[5]. The main advantage of Scratch is that it removes the actual coding aspect and converts it into a graphical interface through which users 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. Likewise, all syntax and code is converted into either blocks or other graphics, which are considerably easier to understand and interact with for beginners, rather than the standard syntax and coding rules.

Perhaps the best 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 teaching the user how to develop rather than trying to re-create the full developing environment that one finds in Eclipse.

[5] *About Scratch*. (n.d.). Retrieved Apr 30, 2013 from http://info.scratch.mit.edu/About_Scratch.


App Inventor for Android Blocks Editor: PaintPot
PaintPot
Saved
Undo
Redo
New emulator
Connect to Device...
Zoom
Built-In
My Blocks
Advanced
Definition
Text
Lists
Math
Logic
Control
Colors
when Button1.Click do set Canvas1.PaintColor to color Red
when Button2.Click do set Canvas1.PaintColor to color Blue
when Button3.Click do set Canvas1.PaintColor to color Green
when Button4.Click do set Canvas1.PaintColor to color Yellow
when Button5.Click do call Canvas1.Clear set Canvas1.PaintColor to color Black
when Canvas1.Touched x name x y name y touchedSprite name touchedSprite
do call Canvas1.DrawCircle x value x y value y r number 5
when Canvas1.Dragged startX name startX startY name startY prevX name prevX prevY name prevY currentX name currentX currentY name currentY draggedSprite name draggedSprite
do call Canvas1.DrawLine x1 value prevX y1 value prevY x2 value currentX y2 value currentY
Defin...
Text
Lists
Math
Logic
Control
Colors

App Inventor: Visual Programming for Android[6]

3.1: 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[7]. 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

[6] JP. (Jan 2, 2012). *App Inventor Transition.* Retrieved Apr 20, 2013 from http://www.androidfeeds.com/2012/app-inventor-transition/.
[7] http://appinventor.mit.edu/

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.

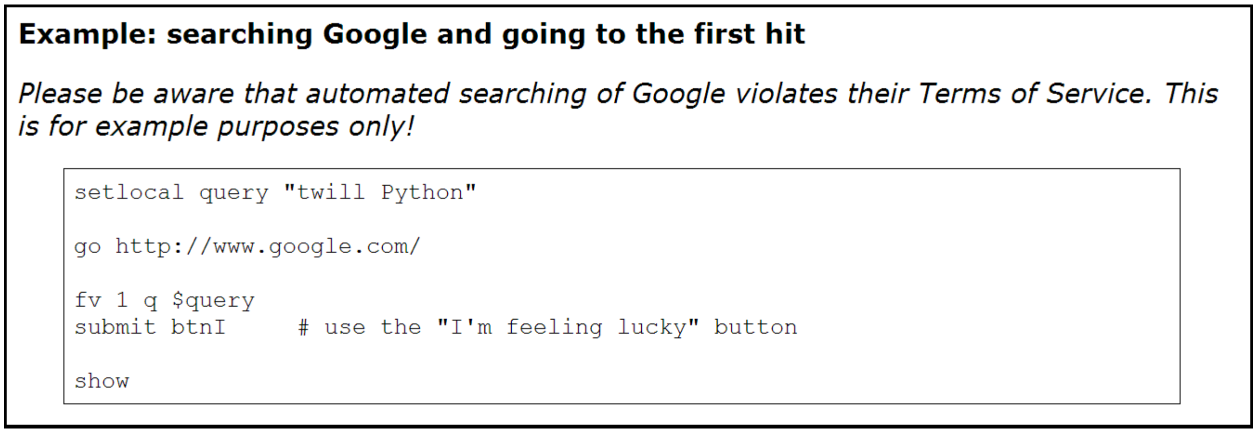
As with Scratch, App Inventor emphasizes the *learning* process for the user, introducing Android development as a block-based language. Furthermore, neither Scratch nor App Inventor is meant to be as powerful a language as say, Java, or C, or even Python. For instance, Scratch is limited to one-dimensional 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 to and assist beginner developers. While neither can create an actual, complex product as well as Java or Python can, both can be consider more effective than those powerful languages for inexperienced developers. EasyAndroid takes these two models and strives to facilitate simple development for newer Android programmers, keeping the entire process as efficient and intuitive as possible.

3.2: Droid Dev

Droid Dev was a senior thesis last year, that also sought to simplify the Android development process. The main difference is that rather than reducing or simplifying the actual development process, Droid Dev 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 were 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 it has become more reasonable to assume that people have an Internet connection at any given time, there is still an advantage to being able to develop apps where there is unstable or no Internet connection, such as while traveling. With the exception of Twill, all of the products described in the following sections are provided via the Internet, and therefore have the same advantages and disadvantages described here.

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 code, XML code, and 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.



Example: searching Google and going to the first hit

Please be aware that automated searching of Google violates their Terms of Service. This is for example purposes only!

```
setlocal query "twill Python"

go http://www.google.com/

fv 1 q $query
submit btnI     # use the "I'm feeling lucky" button

show
```

Twill: Redefining the Browsing Experience[8]

3.3: Twill

Twill is a language written in Python which brings the Web-browsing experience to the command line interface[9]. 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.

[8] *Twill examples.* (n.d.) Retrieved Apr 25, 2013, from http://twill.idyll.org/examples.html.
[9] http://twill.idyll.org/

If This Then That: Simplicty at Its Finest[10]

3.4: 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 thought of as two steps[11]. All the user has to do is choose the *this* trigger and the *that* action – two items. This way, the user does minimal work, oblivious to what goes on behind the scenes, but still 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)

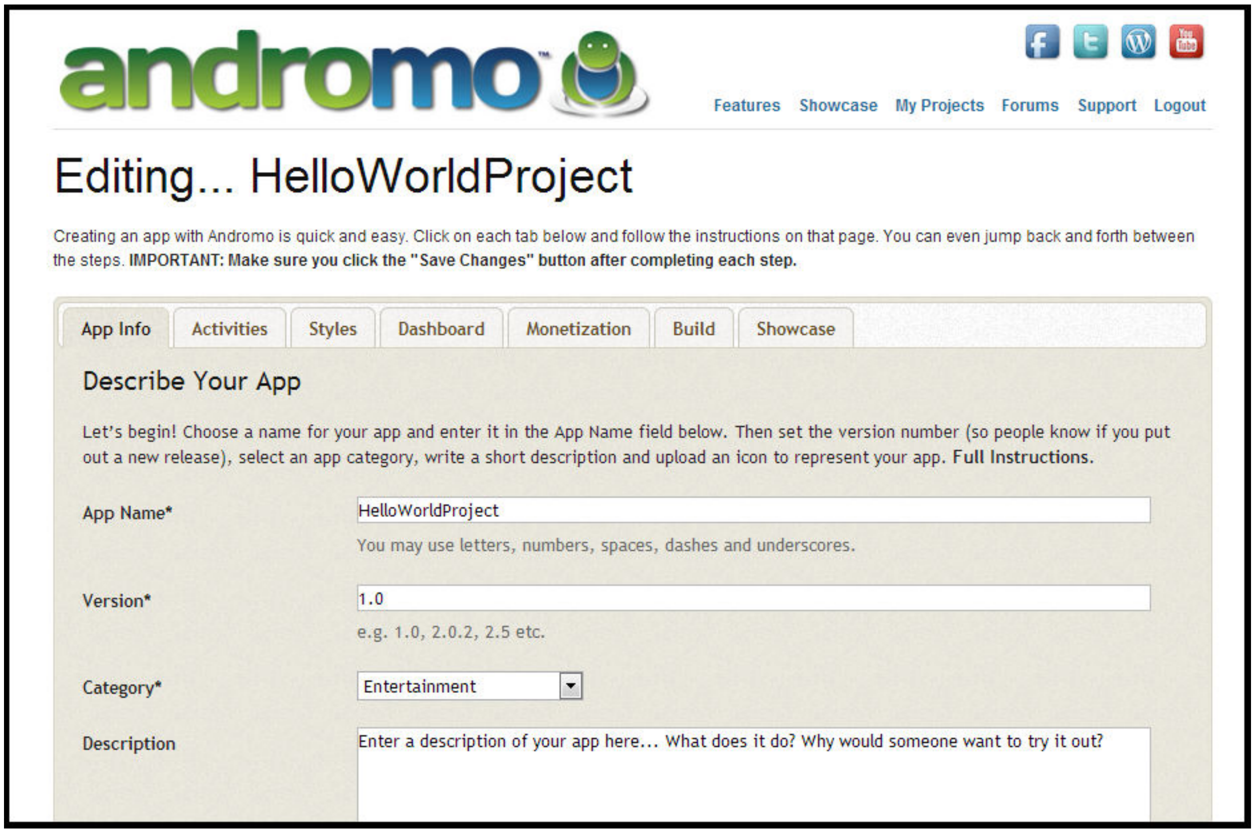
[10] https://ifttt.com/
[11] https://ifttt.com/wtf

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. See Appendix 1 for images of a full example.

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

The drawback of IFTTT’s extremely simplified process is that there is no customization available, outside of defining the properties. In other words, there is a finite set of triggers and a finite set of actions to choose from.


andromo
Features Showcase My Projects Forums Support Logout
App Info Activities Styles Dashboard Monetization Build Showcase

Editing... HelloWorldProject

Creating an app with Andromo is quick and easy. Click on each tab below and follow the instructions on that page. You can even jump back and forth between the steps. **IMPORTANT: Make sure you click the "Save Changes" button after completing each step.**

Describe Your App

Let's begin! Choose a name for your app and enter it in the App Name field below. Then set the version number (so people know if you put out a new release), select an app category, write a short description and upload an icon to represent your app. **Full Instructions.**

App Name* HelloWorldProject
You may use letters, numbers, spaces, dashes and underscores.

Version* 1.0
e.g. 1.0, 2.0.2, 2.5 etc.

Category* Entertainment

Description Enter a description of your app here... What does it do? Why would someone want to try it out?

Andromo: "Make an Android App. No Coding Required"[12]

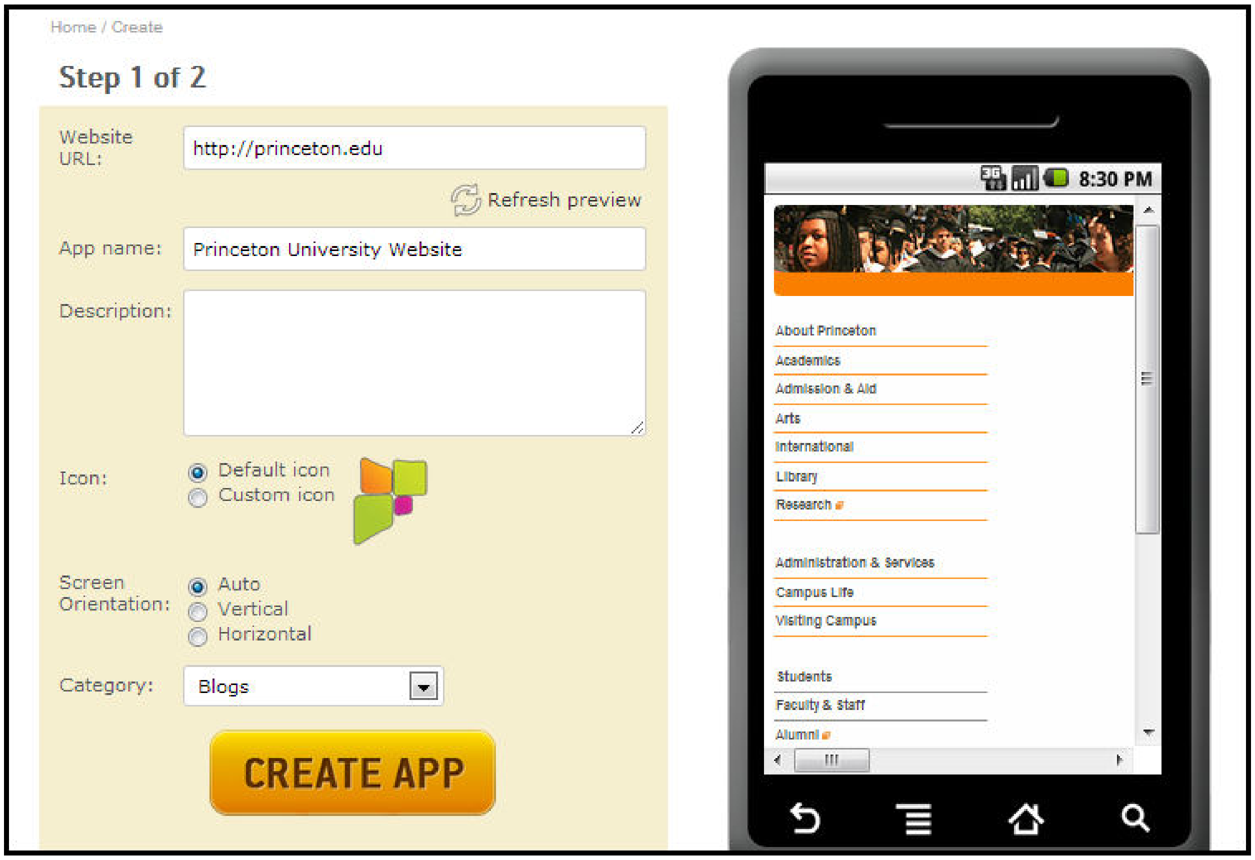
3.5: Andromo

The product most similar in functionality to EasyAndroid is Andromo, a web application through which users can build their own complete Android app[13]. Perhaps the strongest characteristic of Andromo is that the user does absolutely 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 any need for the user to code in any way or form – 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. See Appendix 2 for a full example.

[12] http://www.andromo.com/
[13] Ibid.

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[14], Andromo – like IFTTT – is restricted in its 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 essentially restricted to displaying web or local content and links to social media pages. Because of this limitation, Andromo apps end up being extremely generic.

[14] Ibid.



Step 1 of 2

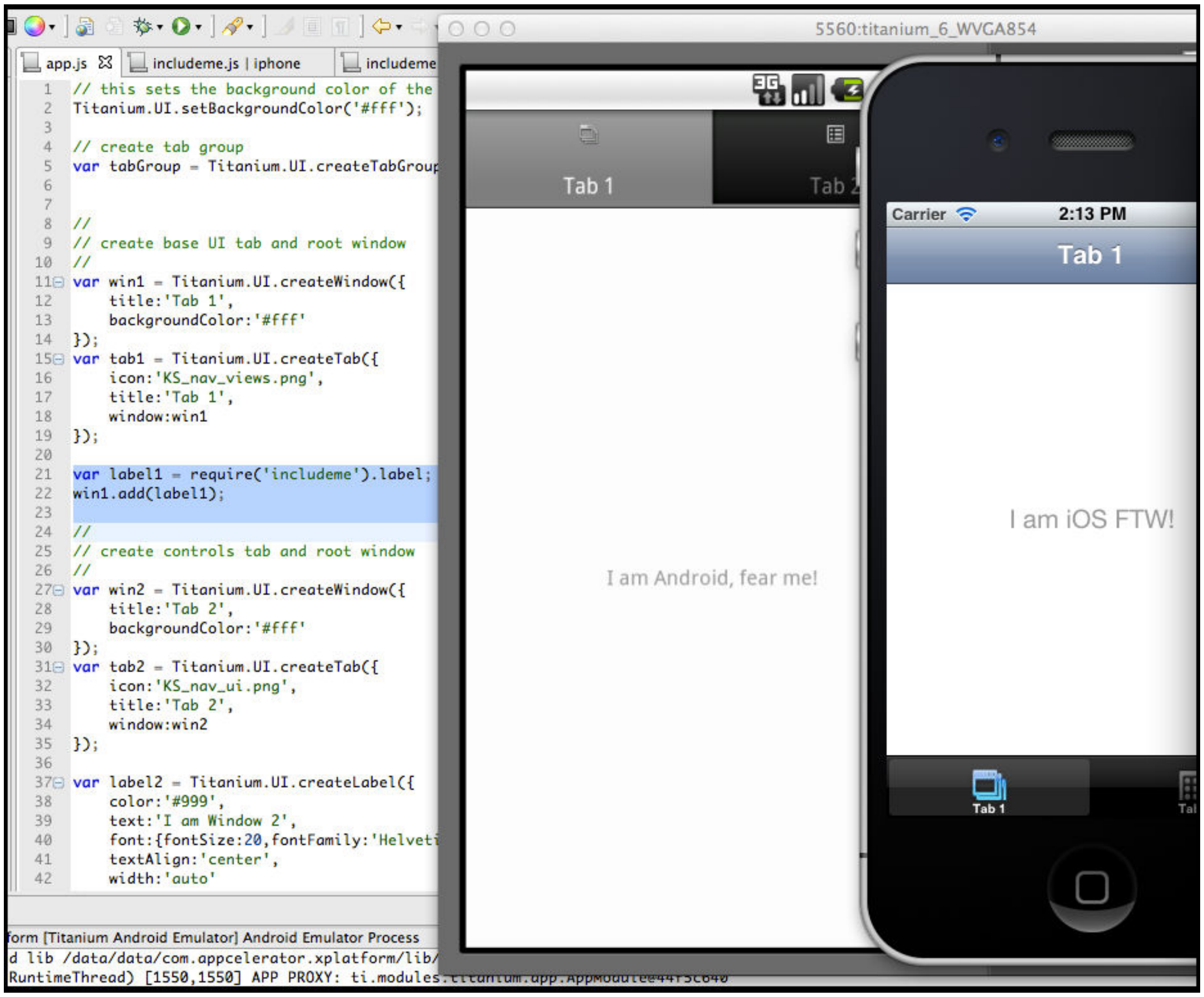
Website URL:	http://princeton.edu
App name:	Princeton University Website
Description:	
Icon:	[✓ Default icon] Custom icon
Screen Orientation:	[✓ Auto] Vertical Horizontal
Category:	Blogs

AppsGeyser: From Website to Android App in Two Steps[15]

3.6: 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[16]. As a result of this minimal effort required by the user, the product is extremely convenient and straightforward to use, and at the same time, extremely limited and inflexible. AppsGeyser is perhaps the extreme of "reducing" the Android development process in a way that gives the user zero say in how the Android app will turn out, allowing for practically no "development." See Appendix 3 for the full version of the image above.

[15] http://www.appsgeyser.com/create/url/
[16] http://www.appsgeyser.com/about/



```
// this sets the background color of the
Titanium.UI.setBackgroundColor('#fff');

// create tab group
var tabGroup = Titanium.UI.createTabGroup


//
// create base UI tab and root window
//
var win1 = Titanium.UI.createWindow({
    title:'Tab 1',
    backgroundColor:'#fff'
});
var tab1 = Titanium.UI.createTab({
    icon:'KS_nav_views.png',
    title:'Tab 1',
    window:win1
});

var label1 = require('includeme').label;
win1.add(label1);

//
// create controls tab and root window
//
var win2 = Titanium.UI.createWindow({
    title:'Tab 2',
    backgroundColor:'#fff'
});
var tab2 = Titanium.UI.createTab({
    icon:'KS_nav_ui.png',
    title:'Tab 2',
    window:win2
});

var label2 = Titanium.UI.createLabel({
    color:'#999',
    text:'I am Window 2',
    font:{fontSize:20,fontFamily:'Helveti
    textAlign:'center',
    width:'auto'
```

```
form [Titanium Android Emulator] Android Emulator Process
d lib /data/data/com.appcelerator.xplatform/lib/
RuntimeThread) [1550,1550] APP PROXY: ti.modules.titanium.app.AppModule@44f3c640
```

Appcelerator Titanium: Making Mobile Easy for JavaScript Programmers[17]

3.7: 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[18]. 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,

[17] *Supporting Multiple Platforms in a Single Codebase*. (n.d.). Retrieved Apr 25, 2013,from http://docs.appcelerator.com/titanium/3.0/?_escaped_fragment_=/guide/Supporting_Multiple_Platforms_in_a_Single_Codebase.
[18] http://www.appcelerator.com/

which leads some developers to learn how to code the app via the traditional Android software described earlier anyway. Secondly, Appcelerator Titanium has an API that users must learn to use and interact with. Viewing this product from the context of EasyAndroid, it is also clear that this platform targets a much more advanced audience than EasyAndroid does. 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.

4: Design Decisions

4.0: 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, the solution I decided to go with and the rationale behind that decision.

4.1: Using Existing Software

A pivotal decision that was made early on was to take full advantage of the software that is typically available and used for Android development. In other words, rather than seeking to simplify upfront setup costs, EasyAndroid was aimed to simply the actual development process. With that goal in mind, eliminating the dependence of EasyAndroid on the various "third-party" software (such as the Android SDKs and tools) was not as important as first developing a tool that simplified the coding.

4.2: 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 that was designed for everyone would be optimal 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 the development process as a whole. Likewise, I also could have targeted programmers that were new to Android, but otherwise experienced in software development, but decided that the learning curve for those developers was not as significant. Therefore, a tool designed for experienced developers was 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 enable someone with a clear design or specification for an Android app in mind to create their own Android app with relatively little effort, regardless of his or her programming experience level.

4.3: 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. As a result, I needed 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 users 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.

4.4: “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 his or her options, the GUI can just have a list of all options available to the user, which is what it does with the “Options” fragment. 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.

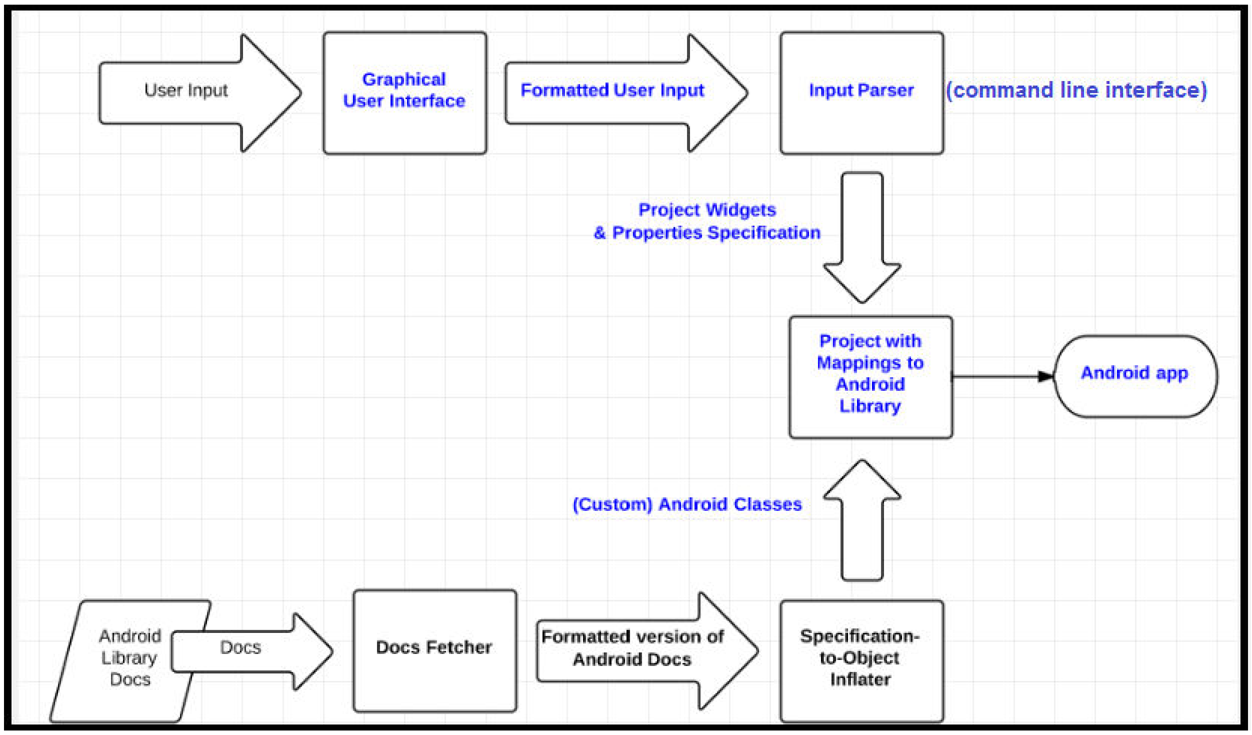
4.5: Designing for Benign Users vs. Malicious users

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 innocuous 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.

4.6: 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. Further 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, thanks to the underlying structure. But of course, it is impossible to think a hundred 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.



The Full "Abstraction" System (bottom row not implemented)

4.7: "Design By Example"

In approaching building and designing EasyAndroid – the functionality and features it would have – I took a "design by example" approach. As previously mentioned, this meant 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 then define the general widget for EasyAndroid.

4.8: Java Code over XML Layout Files to Define Layouts

4.8.0: Background

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 of 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. The first and more commonly used approach is using XML, where developers can specify a static version of their look and feel, which the Android OS takes at runtime and "inflates" onto the user's screen. The second approach is to define the layout 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 by the Java code. Furthermore, XML is more flexible in terms of providing an easy way to dynamically 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 graphical layout tools that show the user a preview of the Android screen, 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, such as incoming data or even just the time of day, 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 be redundant.

That said, using XML layout files for UI definition is generally advised, though from a performance efficiency perspective, there is little difference. 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. In general, the two can produce the exact same results. Developers typically end up using a mix of the two, though XML generally sees more usage.

4.8.1: UI Layouts in EasyAndroid

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 is convenient but not necessary. Thus, 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, users benefit because they can actually see, in full, the changes that they made 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, 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 (a display text) 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_CONTE
     NT, ViewGroup.LayoutParams.WRAP_CONTENT));
rootView.addView(widget1);
```

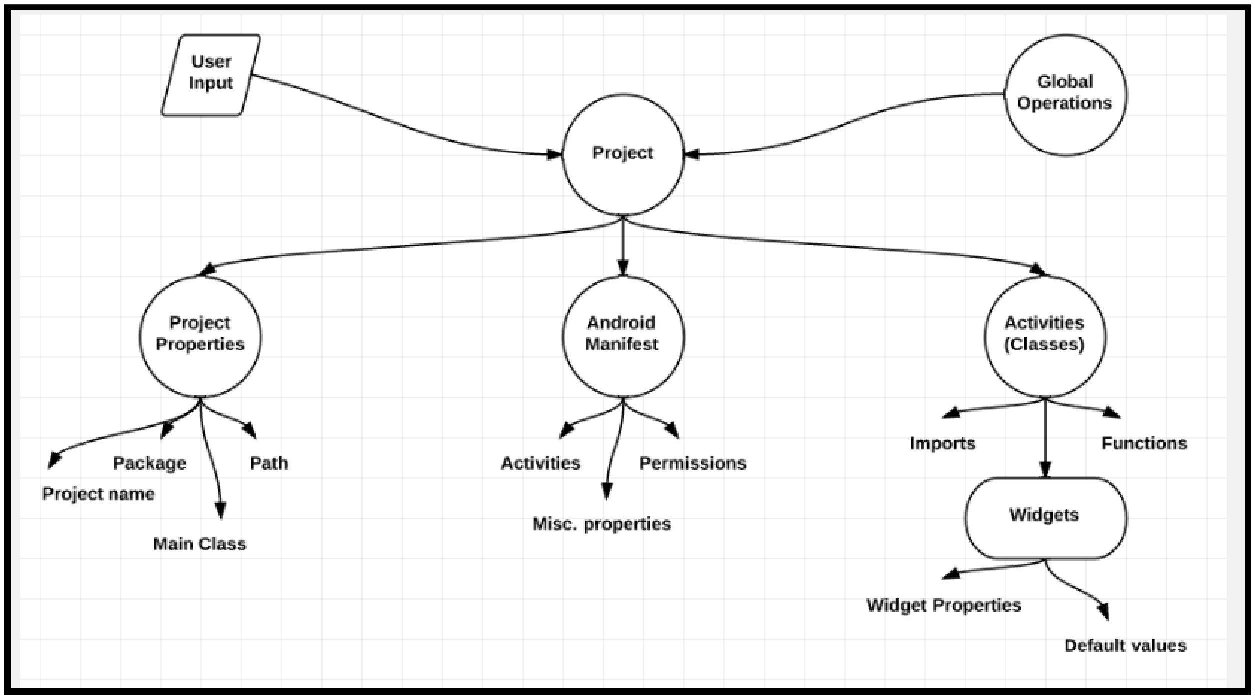
Just as a note of comparison, the XML layout file version of this segment of the code would be:

```
<TextView
     android:layout_width=" wrap_content"
     android:layout_height="wrap_content"
     android:text="Hello World"
     />
```

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 *widget1*, 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, *rootView* 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 the 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, expecting it to 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*, of which each Activity has exactly one. More complicated widgets may have more lines of code per section, but ultimately, all widgets follow the same pattern.

4.9: Architecture of an EasyAndroid Project



EasyAndroid Project Structure

I took a very object-oriented approach in designing the project structure for EasyAndroid – that is, EasyAndroid essentially maintains 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 or global operations (that is, “global commands,” described later) modify 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 Activity object (a model for Activities), of which any given project has at least one. The build process converts all of this into Java code.

5: Implementation

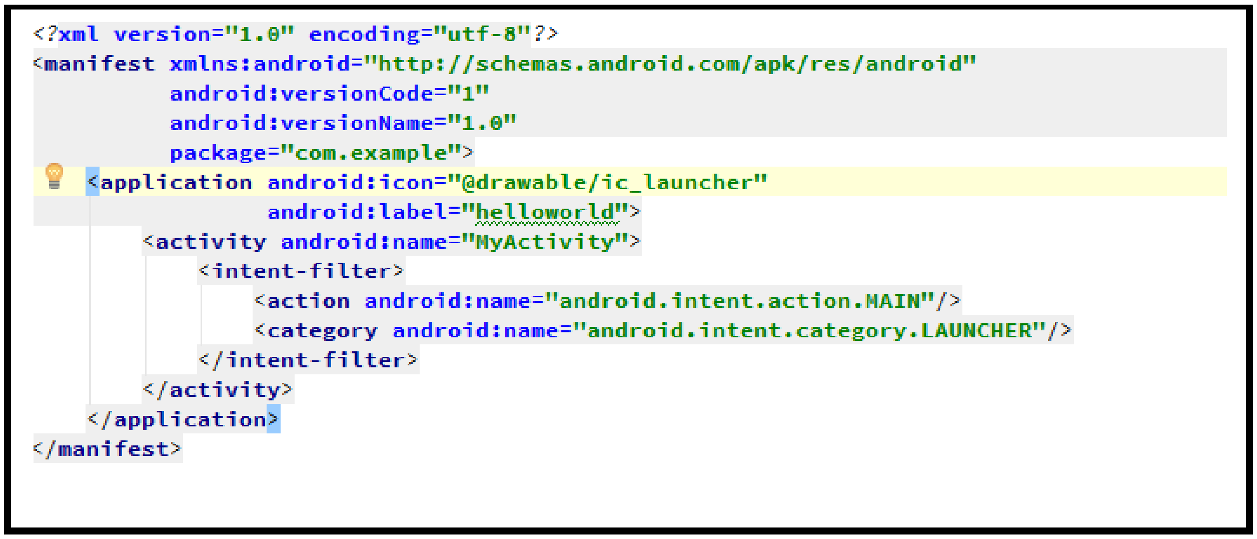
The following sections describe some of the more interesting or challenging parts of EasyAndroid's implementation, as well as the fundamentals. Some sections give more background on the decision-making process to implement that particular part of EasyAndroid in that way, while others simply give an in-depth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

5.0: 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 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 new 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.



```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          android:versionCode="1"
          android:versionName="1.0"
          package="com.example">
    <application android:icon="@drawable/ic_launcher"
                 android:label="helloworld">
        <activity android:name="MyActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

AndroidManifest.xml: A Default AndroidManifest File

5.1: 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 describes the contents of 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.

5.1.0: 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 – 130 to be exact – an even smaller number that are commonly used, and an even smaller number than that that the 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.

5.2.1: 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 crashes the app. 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 simply 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.

Since the rest of the exact details of the contents of an AndroidManifest are unimportant and uninteresting, 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.

5.2: "Global" versus "Local" Commands

*Note: all commands are technically all-lowercase and have no punctuation; in the following sections, commands may appear otherwise (i.e. capitalized, or with punctuation) for grammar purposes.

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's contents 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 eventually an apk file; 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.

5.3: Global Commands

5.3.0: Open (File), Save (File)

The open and save file feature is more supplementary in nature because users could do everything in one session, but still convenient, useful, and valuable. This feature allows 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 allows this method to 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.

5.3.1: 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.

5.3.2: 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.

5.3.3: Build

Running “build” on the project is actually composed of several steps; the compilation itself is 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 tool runs the Ant build script, which is generated by the Android tool. 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.

5.3.4: 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.

5.3.5: 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 a background thread 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”).

5.3.6: 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.

5.3.7: 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, and 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 external Android project.

“debugfunctions” gives the full list of all of the user’s custom functions – “custom functions” are defined in detail later – 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 custom imports – explained in detail later – 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 example 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.

5.4: Local Commands

5.4.0: 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."

5.4.1: 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.

5.4.2: 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 replacement of 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.”

5.4.3: Main Class

The “mainclass” command changes the main class of the app to the value of the first argument, where the main class of an app is simply another name for the main 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 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.”

5.4.4: 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.

5.4.5: 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 app will search for the old main class, which no longer exists.

5.4.6: 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 how a more complex function might 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 parts of each pair 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.

5.4.6.0: 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 typical users 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.

5.4.7: Custom Import

As a complement to the "customfunction" command, the "customimport" command allows users 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.

5.4.8: Widget Commands: Create [Widget Type], Up, Down, Remove

5.4.8.0: (Create) [Widget Type]

Because the commands to add widgets are fundamentally the same, it is easier to first define them generally, and then 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. I could have also had a naming convention like, “button1,” “text1,” and so on, but prioritized other things (implementation would be *slightly* more complicated – I would have to keep track how many of each type of widget there are).

TextView is the Android name for the widget which simply displays text. In addition to the basic properties, 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 typical function is 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, 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. New for the Button widget, however, there is also the action property, which is the Java code that runs whenever the button is clicked. Note that the value of the action property is the *Java code* that runs – 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:

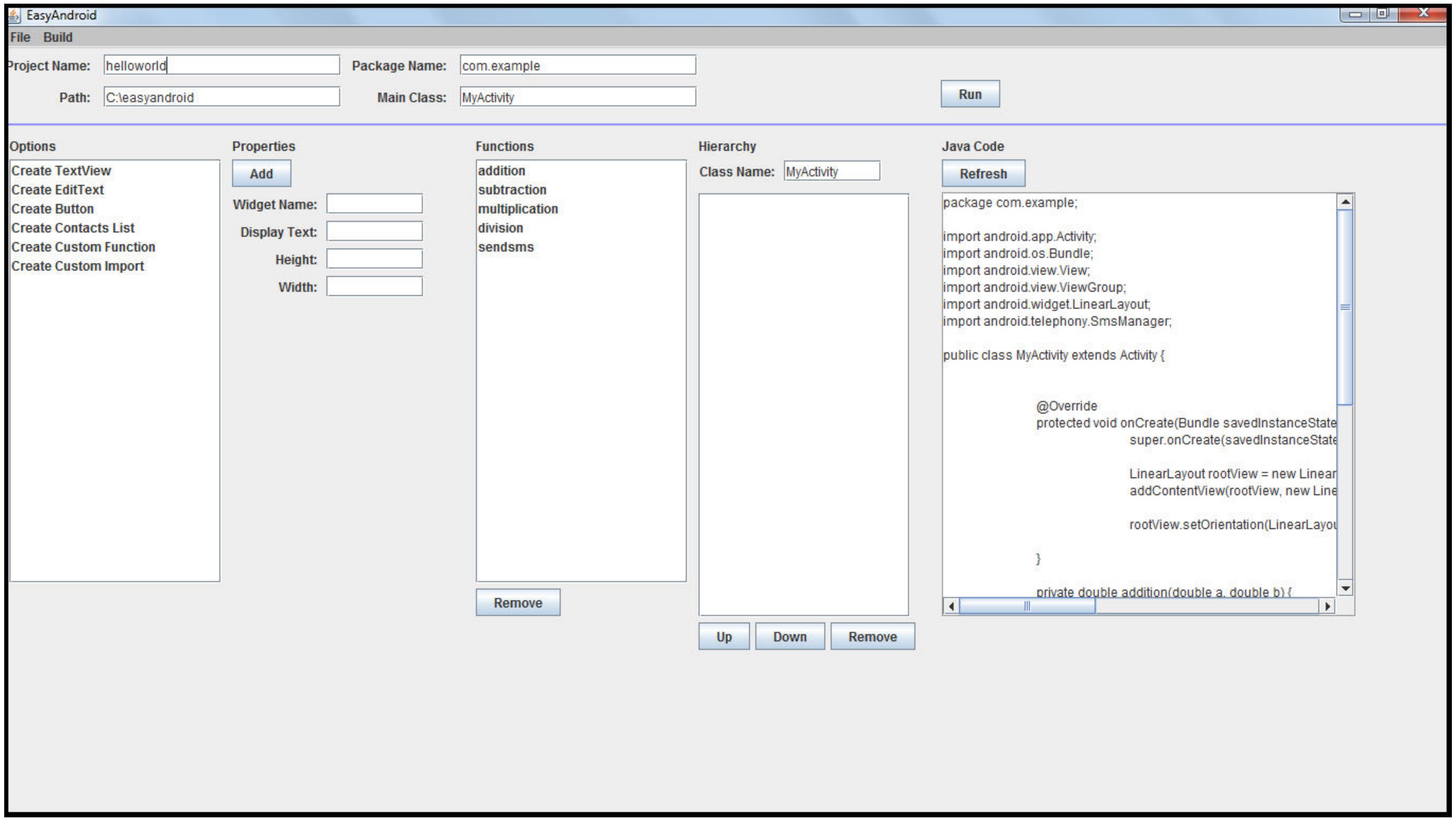
```
contactslist -divider ||| -hasNumber no
```

5.4.8.1: 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.

5.5: Graphical User Interface



Project Name: helloworld
Package Name: com.example
Path: C:\easyandroid
Main Class: MyActivity

Options:
Create TextView
Create EditText
Create Button
Create Contacts List
Create Custom Function
Create Custom Import

Properties:
Widget Name:
Display Text:
Height:
Width:

Functions:
addition
subtraction
multiplication
division
sendsms

Hierarchy:
Class Name: MyActivity

```
package com.example;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.telephony.SmsManager;

public class MyActivity extends Activity {


            @Override
            protected void onCreate(Bundle savedInstanceState
                        super.onCreate(savedInstanceState

                        LinearLayout rootView = new Linear
                        addContentView(rootView, new Line

                        rootView.setOrientation(LinearLayou

            }

            private double addition(double a, double b) {
```

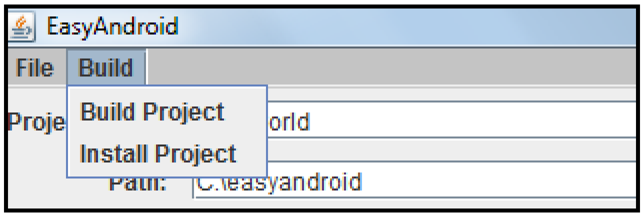
EasyAndroid: Graphical User Interface

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, the nature of my target audience made a GUI imperative, or else EasyAndroid would simply be unusable. 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, though much of the structure carried over.

5.5.0: Design and Functionality

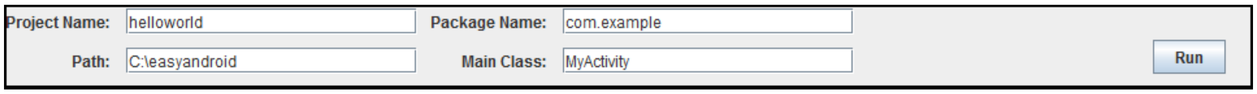
In designing the GUI, I first laid out what the essential pieces were. Among these pieces were the project properties, a list of widgets the user could add, a panel for the definition and creation of a widget, a list of the current available functions, a list of the currently existing widgets, and lastly,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

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



Menu bar with "Build" items shown

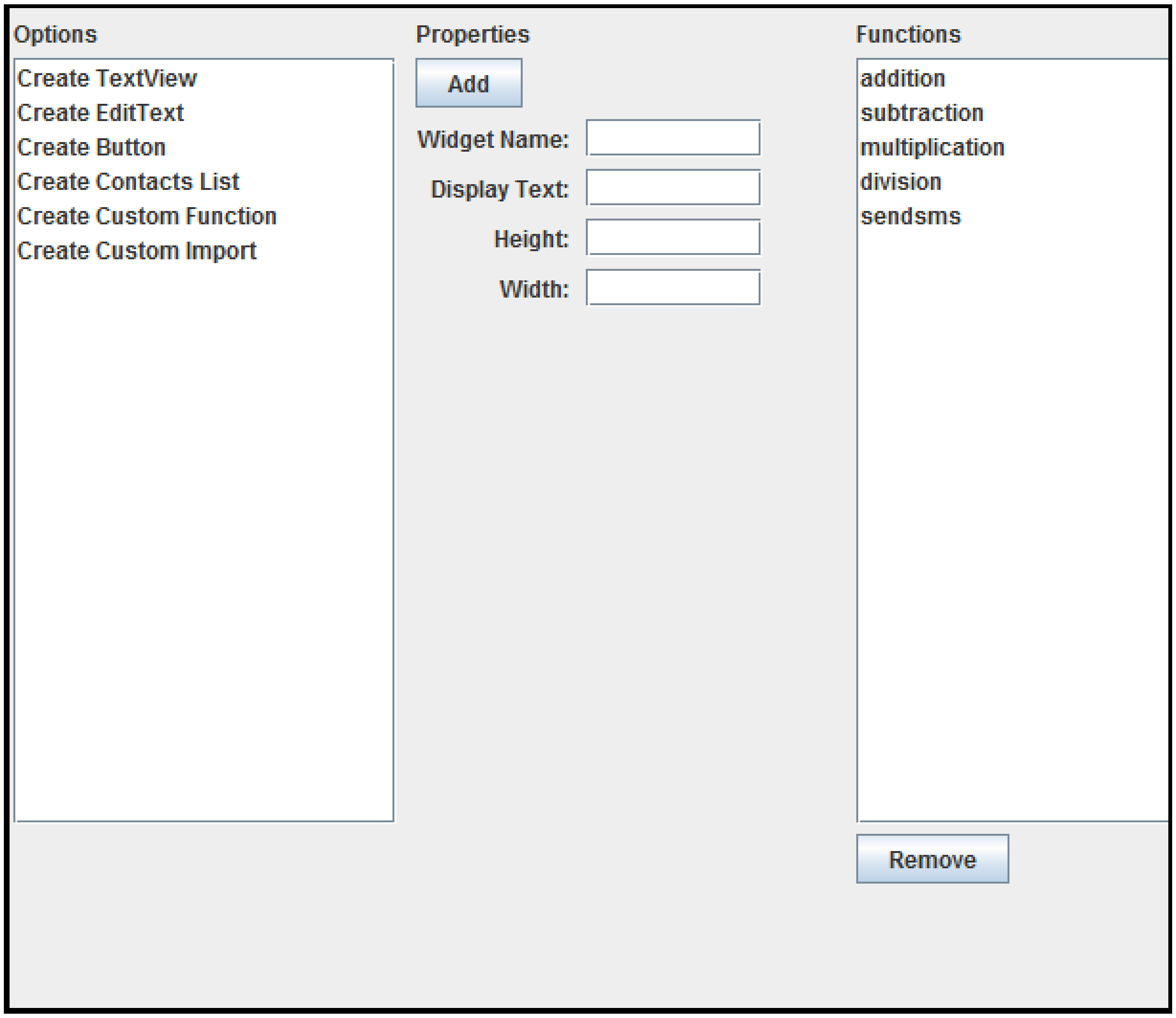
The top bar menu is intended to contain all of the "global" commands described in the previous section. Those include commands such as save file, open file, exit, build, install, and eventually help. A submenu "File" contains the "Save," "Open," "Reset," and "Exit" commands, and a second submenu "Build" contains the "Build Project" and "Install Project" commands. As the tool is developed further, more submenus and more items in the submenus would of course be added. 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."



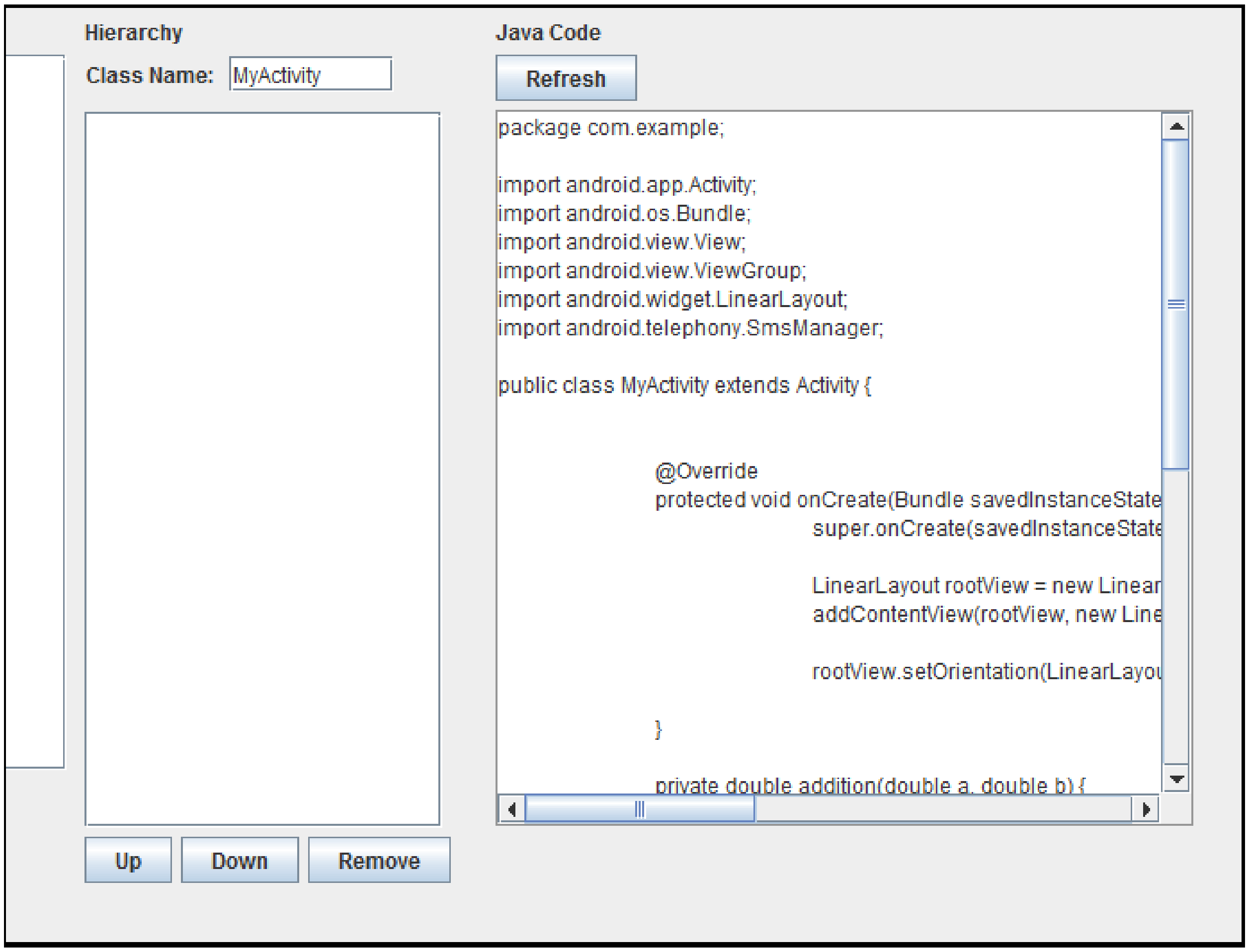
Project Name: helloworld
Package Name: com.example
Path: C:\easyandroid
Main Class: MyActivity

The header section

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. New to Version 2, however, is the "Run" button that appears in this section – "Run" is the all-in-one create-build-install shortcut feature that should be quite noticeable to the user.



The left half of the "app ingredients" section



```
package com.example;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.telephony.SmsManager;

public class MyActivity extends Activity {


		@Override
		protected void onCreate(Bundle savedInstanceState
				super.onCreate(savedInstanceState

				LinearLayout rootView = new Linear
				addContentView(rootView, new Line

				rootView.setOrientation(LinearLayou

		}

		private double addition(double a, double b) {
```

The right half of the "app ingredients" section

The last section, where the majority of content is, is the "app ingredients" section. This is the most complicated section of the three, itself further divided into (currently) five sections, or "fragments." From left to right, the fragments are: options, properties, functions, hierarchy, preview. Excluding the functions fragment, all of these are related to each other and manipulating one potentially changes others. More specifically, the properties fragment 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 fragment. Lastly, the preview fragment is a synthesis of all of the content from the project, so any persisted change in any of the fragments will change the preview. A more in-depth explanation of each of the five fragments follows.

The options fragment is what it suggests: it contains the available options for widgets that could be added to the Android app. Currently, these options are, “Create Button,” “Create TextView,” “Create EditText,” “Create Contacts List,” “Create Custom Function,” and “Create Custom Import.” The last two, “Create Custom Function” and “Create Custom Import” are outliers, but because they follow the same structure in which they have properties that need specification, before being added to the project, they fit well as “options,” though technically not widgets that will physically appear in the app. Additionally, only these last two options do not have default properties that replace empty input. This is because if the user were to attempt to create a custom function without defining all the properties, such as having an empty body, 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 for custom functions (functions can have no parameters, so I allow that), and the value of the import package for custom imports. The rest of the options, however, are simply widgets that the user has the ability to add to his or her Android app. As mentioned earlier, because each option has a different set of properties that can or need to be defined, choosing a new option triggers an update to the properties fragment if necessary.

Second, the properties fragment is where all of the properties for a particular widget (or option) appear. The 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 properties fragment – that is, which name-value pairs appear – varies depending on which option is selected from the options fragment. Lastly, above the list of properties, is an “Add” button, which takes the widget defined in the fragment and commits it to the underlying structure, if possible. Upon a successful addition, the properties are cleared, and the hierarchy and preview fragment are updated.

The third fragment is the functions fragment, which lists all of the defined functions in the project available to the user. These are all essentially helper functions, which are generally meant to be used in the “action” part of widgets’ properties. At the moment, this fragment 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 existing functions via the “Remove” button, for situations in which a function is defined incorrectly. When adding functions, the user must define and add the function through the options and properties fragment.

The fourth fragment is the hierarchy fragment, 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. Also, at the top of the fragment is a field for the current Activity or class name, which is by default “MyActivity” – changes to the Activity class name can be made here. At the bottom of the fragment are three buttons: “Up,” “Down,” and “Remove,” which allow the user to manipulate the placement (or existence) of the widgets. Changes made by these buttons are automatically reflected in the preview fragment. In the future, this fragment would most likely be replaced by a drag-and-drop graphical layout designer. Because of the difficulty of implementing this, however, the current design simply lists the widgets in vertical order and allows the user to manipulate their vertical placement.

Finally, the fifth fragment is the preview fragment, which shows the user a preview of the Java code associated with the project. Because users have the ability to "edit" the code (changes are not saved), the fragment 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.

5.5.1: 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 – on learning the Java Swing API, and figuring out which classes to use and how to use those classes.

Before continuing, it is worth describing the different Java Swing layout managers that I used. Firstly, similar to Android, Java Swing uses the concept of "containers" – each of which is attached to a "layout," or formally, a "layout manager" – which composes 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 minimize the *depth* of the hierarchy tree, and instead, define everything in terms of the root layout manager, intuitively maximizing performance. 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, which meant awkward spacing between fragments, and other fragments requiring more space than was given to them, which made those fragments cramped.

Consequently, I switched to SpringLayout, which allows me to position the different containers in the GUI *in relation* to each other. SpringLayout proved to be much easier to work with in general. As an example, the container for the header section is positioned vertically above the container for the app ingredients. Lastly, I also use BoxLayout, which lists its components as a single row or a single column – BoxLayout is thus useful for extremely simple containers that simply list objects. So for instance, the five ingredients fragments that appear in a horizontal row are arranged in a BoxLayout.

The main Java class file that does all the work 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 class sets up and populates the default JFrame with the various containers and also initializes the CommandLineObject (referred to as CMD in the future), which manages all of the actual project infrastructure and data.

Implementing the JMenuBar (the menu bar) was probably the most straightforward part of the GUI, especially because my use of 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 will 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. Essentially, each of the menu items is tied to an ActionListener that sends the appropriate command – typically the menu item's namesake – to CMD. The only exception is the "Build Project" menu item, which, before telling CMD to "build," technically first updates the global project properties and runs "create." All of this together results in the whole menu bar for EasyAndroid.

The header properties section below the menu bar is also quite simple in implementation, though using GridBagLayout initially made it more complicated than it needed to be. As previously mentioned, the header section contains four project properties, which function as name-value pairs (using a label and text field combination). After initialization, the properties are 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). Lastly, a "Run" button is added to the right of those properties, as of Version 2.

Finally, the app ingredients is, to no surprise, the most complicated of the three sections. Specifically, the app ingredients is a container of five items, which themselves are containers of more items. As mentioned above, this outside container utilizes a horizontally-aligned BoxLayout in order to simply display the inner containers in a row. While this could be managed through SpringLayout, using BoxLayout is more natural – when adding a new fragment, for example, the positioning of the fragment would automatically be taken care of. Other than that, the outside container is quite simple, aligned below the container for the header properties.

Furthermore, I define several helper functions to reduce repetitive code in dealing with the ingredients fragments since they can all be managed similarly. In particular, I define a method "generatePanel," which takes an ArrayList of JComponents as an argument and returns a container for the Arraylist of elements aligned vertically via SpringLayout. 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 ingredients fragments use this static helper function to convert the list of components for that particular fragment into one container, which then gets inserted to the outside container.

Each fragment also has a “refresh” helper function associated with it. The only exception is the options fragment, which never changes and thus never needs to be refreshed. The rest of the fragments, however, are dynamically updated based on the user’s actions and input, and every time the user changes the project, the appropriate fragments need to be updated to reflect those changes. These refresh helper functions are all defined in the same way: first, the outdated fragments are removed from the outside container. A replacement fragment is then generated based on the current project state and added, and finally, the GUI itself is refreshed to reflect the changes. These refresh helper functions are used whenever the UI needs to be updated.

Now I address the implementation of each fragment individually. Firstly, in implementing the options fragments, I considered a few different options – a drop-down menu, a column of buttons – before deciding to simply use a list (JList) of Strings. Ultimately, the list implementation was easiest to use, and most scalable as well. Since the options themselves never change and are not project-specific, I define them statically (hard-code). I use the ListSelectionListener (what happens when an item is clicked) to keep track of which item is currently selected, which is important for the other fragments. Finally, this is all wrapped in a scroll container to make it scrollable. The fragment also has a title, which is simply a JLabel that displays the title text (“Options”), making the total component count two.

Secondly, the properties fragment is probably the most dissimilar and most complicated of the ingredients fragments. Below its title is the “Add” button, which pulls the values of the text fields from the user interface, converts them to command line input format, and sends the formatted command to CMD. Below the “Add” button is the main part of this fragment: the properties themselves, which are generated dynamically based on which item from the options fragment is selected. For each property, a combination of label and text field (or text area for more verbose properties) is added to the fragment. The title, the "Add" button, and the list of properties make up the three components of the fragment.

Third, the functions fragment is almost identical to the options fragment in implementation; it also uses a JList to display the content. The main difference is that while the options fragment's JList is populated from a static String array, the functions fragment is dynamically populated based on CMD's list of custom functions, which is retrieved via a helper function. Currently, this fragment does nothing, but, again, a future version would include a text display of some sort that offers guidance for the user on how to use functions – once it is implemented, the list's listener would trigger changes to that text display. The functions fragment also includes the aforementioned "Remove" JButton – which removes the selected function from CMD and then updates the GUI – and the title, summing to three total components.

Next, the hierarchy fragment is extremely similar to the functions fragment in implementation. Aside from the "Class Name" component – which is persisted upon build – the list of currently existing widgets and the buttons are shown in an analogous way to those of the functions fragments. The "Up," "Down," and "Remove" buttons – contained in a horizontally-aligned BoxLayout – simply serve as an interface to CMD, which updates the project appropriately depending on the button. The fragment itself thus contains four child components: the title, the "Class Name" item, the list of widgets, and the item containing the buttons – these components, as with all the fragments, are aligned using the helper function.

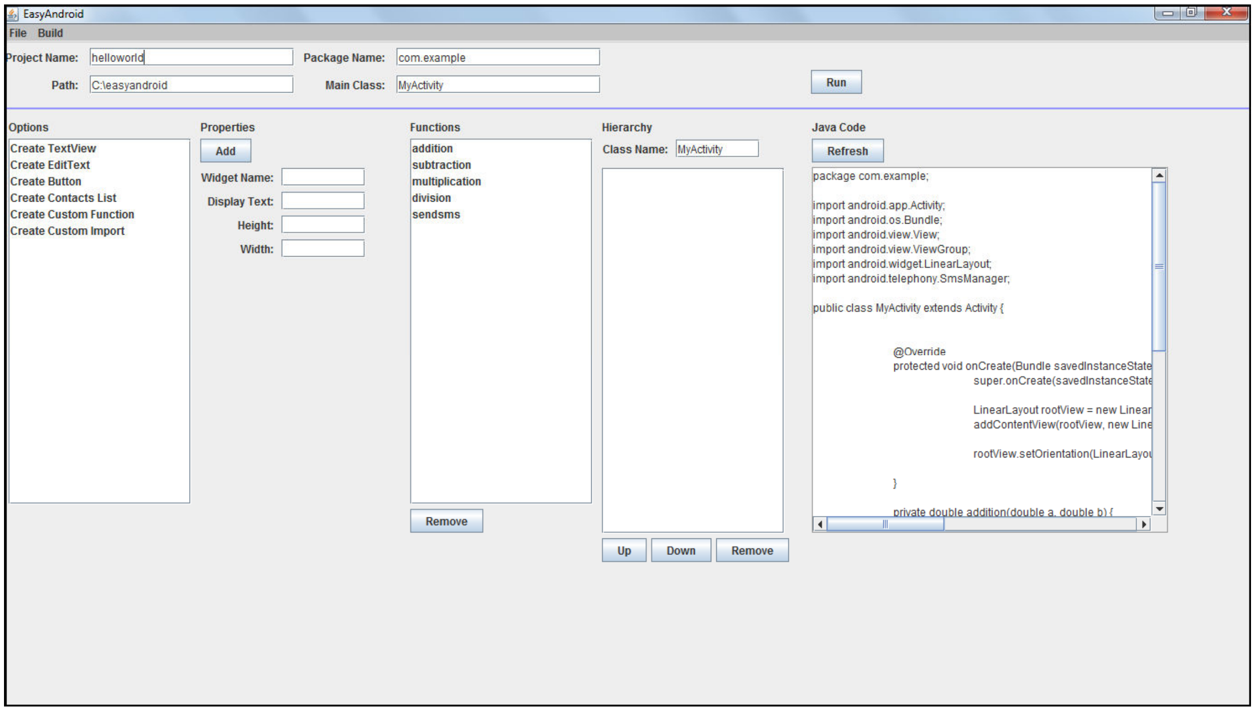
Last is the preview fragment that shows a preview of the Java code that makes up the project. The most important and most noticeable part of the fragment is the actual code preview window itself, which is a scrollable text area, pre-populated by CMD's Java code. Because the Java code does not have a 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 fragment: the title, the "Refresh" button, and the preview text area.

That is the complete implementation of EasyAndroid's Java Swing GUI. While it took quite a bit of time and effort to reach the solution that is currently being used, the key steps 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 – switching from GridBagLayout to using exclusively BoxLayout and SpringLayout simplified implementation considerably. Though the GUI is lacking in many regards, it still does the job, allowing users to create their own simple app.

6: Demo

In this section, I show a demonstration of the steps that would be taken in EasyAndroid to build a full app that has a Button, which sends an SMS text, pulling the body of the text message dynamically via a text field (EditText). The app will also have a TextView that simply says "Hello!" Additionally, I give the app the name "EasyText" and set "com.easyandroid" as the package name.

6.0: Project Properties

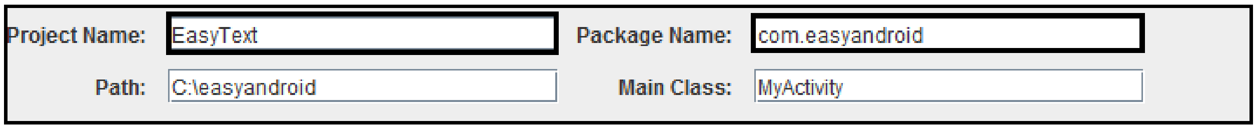


Project Name: helloworld
Package Name: com.example
Path: C:\easyandroid
Main Class: MyActivity
Widget Name:
Display Text:
Height:
Width:
Class Name: MyActivity

```
package com.example;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.telephony.SmsManager;

public class MyActivity extends Activity {


		@Override
		protected void onCreate(Bundle savedInstanceState
				super.onCreate(savedInstanceState

				LinearLayout rootView = new Linear
				addContentView(rootView, new Line

				rootView.setOrientation(LinearLayou

		}

		private double addition(double a, double b) {
```

EasyAndroid: The Opening Screen

The first thing we do is change the project properties. We change the default properties to "EasyText" for Project Name and "com.easyandroid" for Package Name.

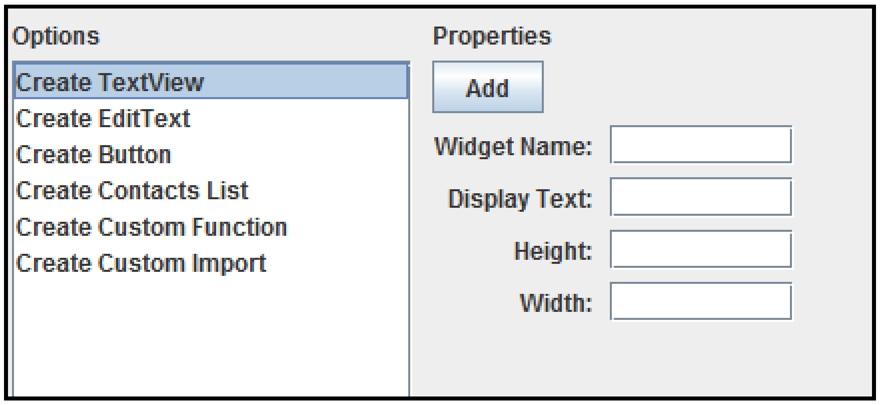


Project Name:	EasyText	Package Name:	com.easyandroid
Path:	C:\easyandroid	Main Class:	MyActivity

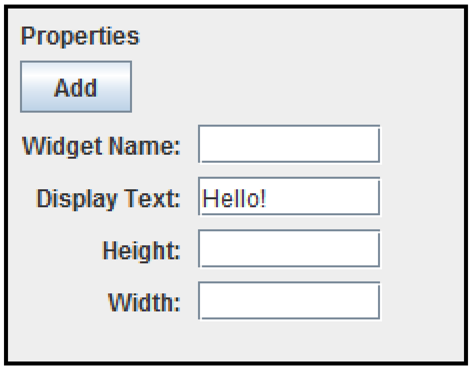
Thus, the global project properties have been successfully changed.

6.1: Say "Hello!"

Next, we add a TextView widget with the text property, "Hello!" Since it does not affect the app, we leave the other properties empty (default).

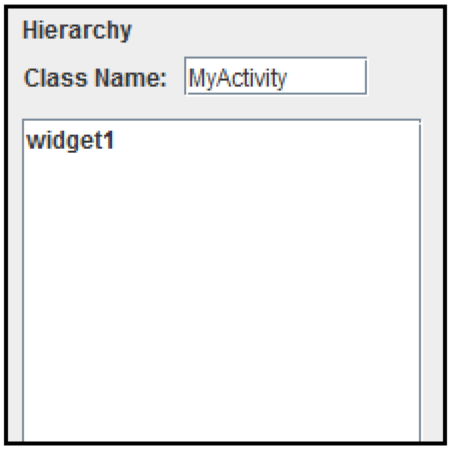


Selecting the "Create TextView" option generates the corresponding properties fragment.



Widget Name:	
Display Text:	Hello!
Height:	
Width:	

Clicking the "Add" button adds the widget to the project.

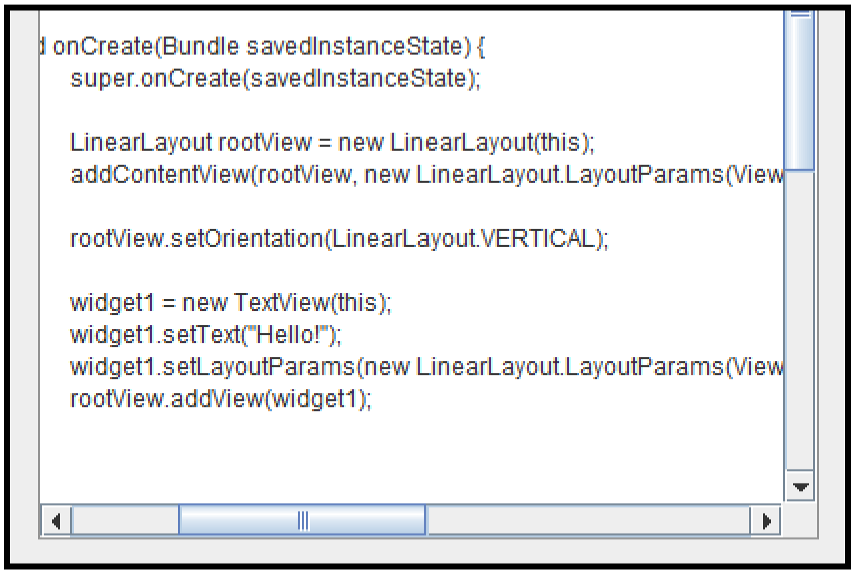


Hierarchy

Class Name: MyActivity

widget1

Because I left the “Widget Name” property empty, it defaults to “widget1.”

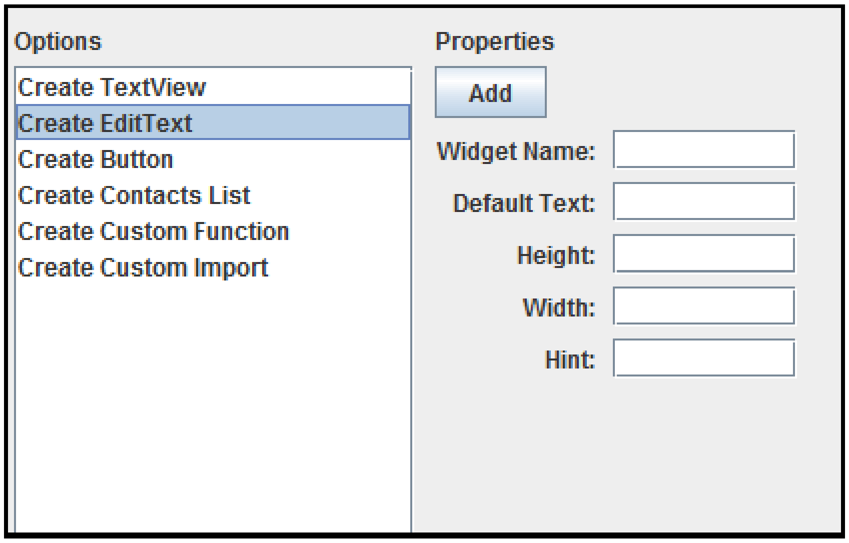


```
d onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    LinearLayout rootView = new LinearLayout(this);
    addContentView(rootView, new LinearLayout.LayoutParams(View

    rootView.setOrientation(LinearLayout.VERTICAL);

    widget1 = new TextView(this);
    widget1.setText("Hello!");
    widget1.setLayoutParams(new LinearLayout.LayoutParams(View
    rootView.addView(widget1);
```

Here’s the corresponding java code that is shown in the preview fragment. That is all it takes say “Hello!”

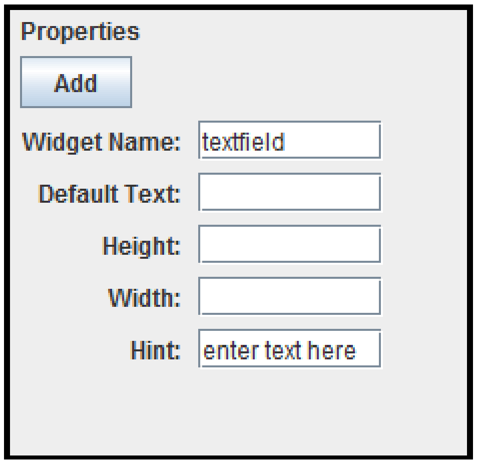
6.2: Add the EditText

The next step is to add the EditText, in which users (of the app) will input their text message body.



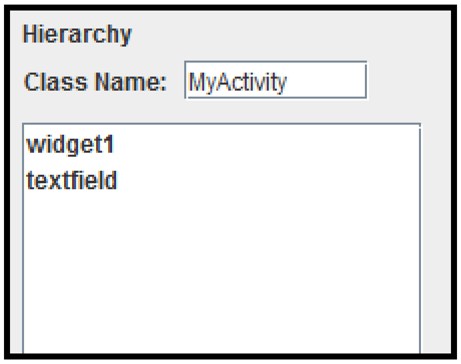
Widget Name:	
Default Text:	
Height:	
Width:	
Hint:	

Selecting the "Create EditText" option generates the appropriate properties fragment.



Widget Name:	textfield
Default Text:	
Height:	
Width:	
Hint:	enter text here

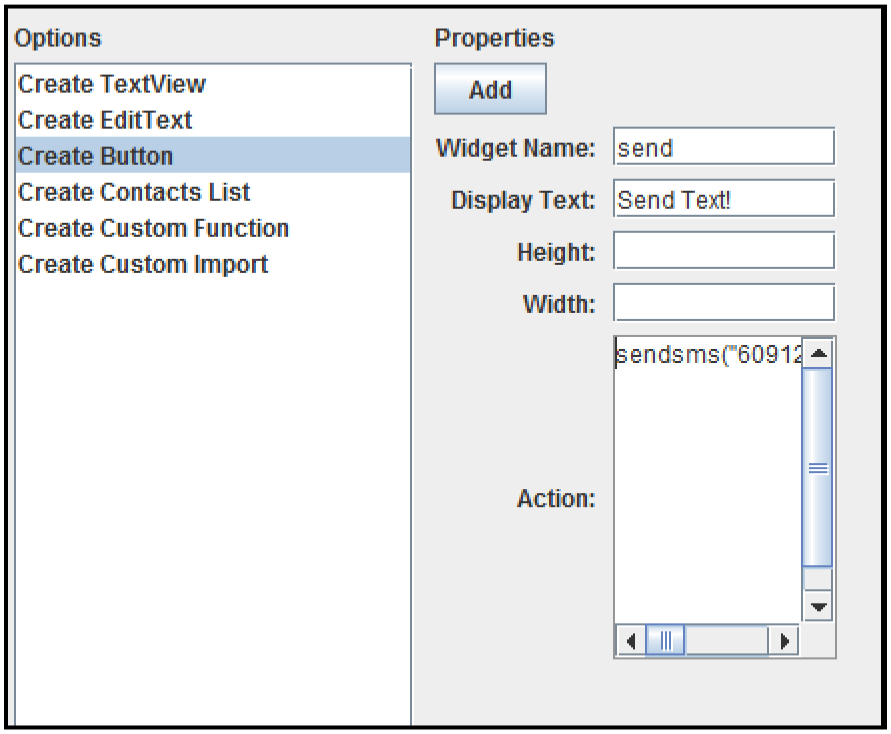
I fill in the properties for the EditText, naming it "textfield" for familiarity's sake, and having a hint of "enter text here." I do not specify the "Default Text" because I want the text field to start off blank.



Clicking “Add” adds the widget to the hierarchy fragment, and the “textfield” widget joins “widget1.” Thus, I am done adding my EditText.

6.3: Add the send-SMS Button

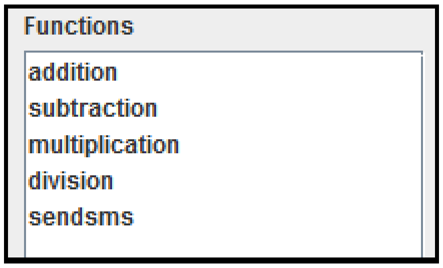
Now it is time to add the “send” button, which actually sends the text message.



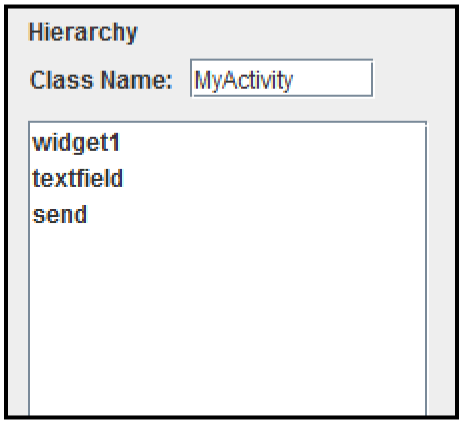
Widget Name: send
Display Text: Send Text!
Height:
Width:
Action: sendsms("60912

First things first – I want a button, so I choose the "Create Button" option. Naturally, it populates the properties fragment with the properties that a button can have; I fill in those properties as shown above. The full text (Java code) of the "Action" property is:

```
sendsms("6091231234", textfield.getText().toString());
```



"sendsms" is, of course, one of the default functions, which is displayed in the functions fragment (shown above; the "Remove" button is cropped out). Aside from that, it is obvious that a regular user would not know how to use "get" the text from an EditText widget. That step would, of course, require some guidance or display text to give hints to the user.



Class Name: MyActivity

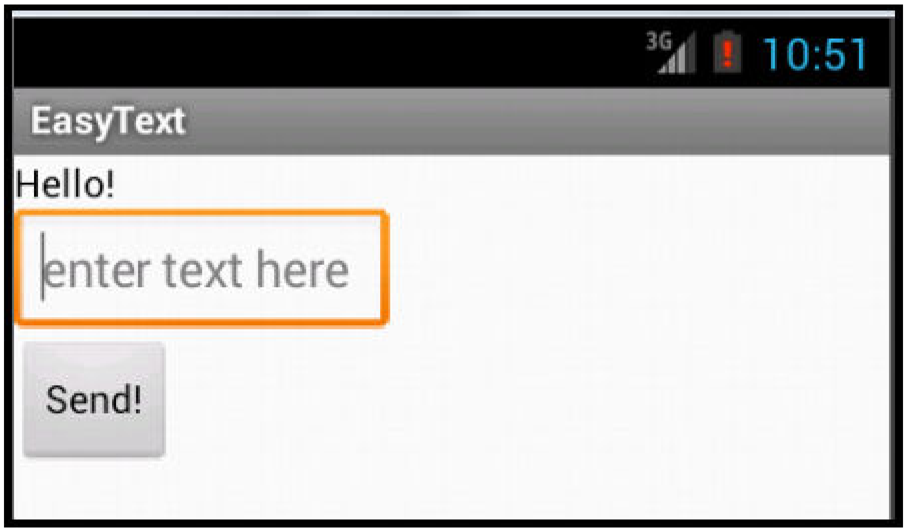
widget1
textfield
send

After that, I add the "send" widget to the list of the app's widgets, thereby completing development of the app.

6.4: Run

The last step is easiest. Click the "Run" button in the top right of the app.

Assuming there is an emulator or Android device connected to the computer, this builds the app and installs it onto the phone automatically. All that's left for the user to do is go into their phone and open the app up. Here's what the app looks like in an Android emulator (slightly modified for black-and-white printer):



Warning! Don't actually click the "Send!" button. It WILL send a text.

7: 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.

7.0: 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 creativy. 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.).

7.1: The Problem of Setup Costs Remains

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, reduces 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, and 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.

7.2: 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.

7.3: 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 then 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.

7.4: 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.

7.5: "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.

7.6: Windows OS only

It should be noted that the current implementation of EasyAndroid only works on Windows. Expanding it to work for Mac OSX would be easy, but I saw no need for it. The only difference is how the file paths are generated.

8: Challenges

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

8.0: 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.

8.1: Keeping the Design Abstract and Flexible

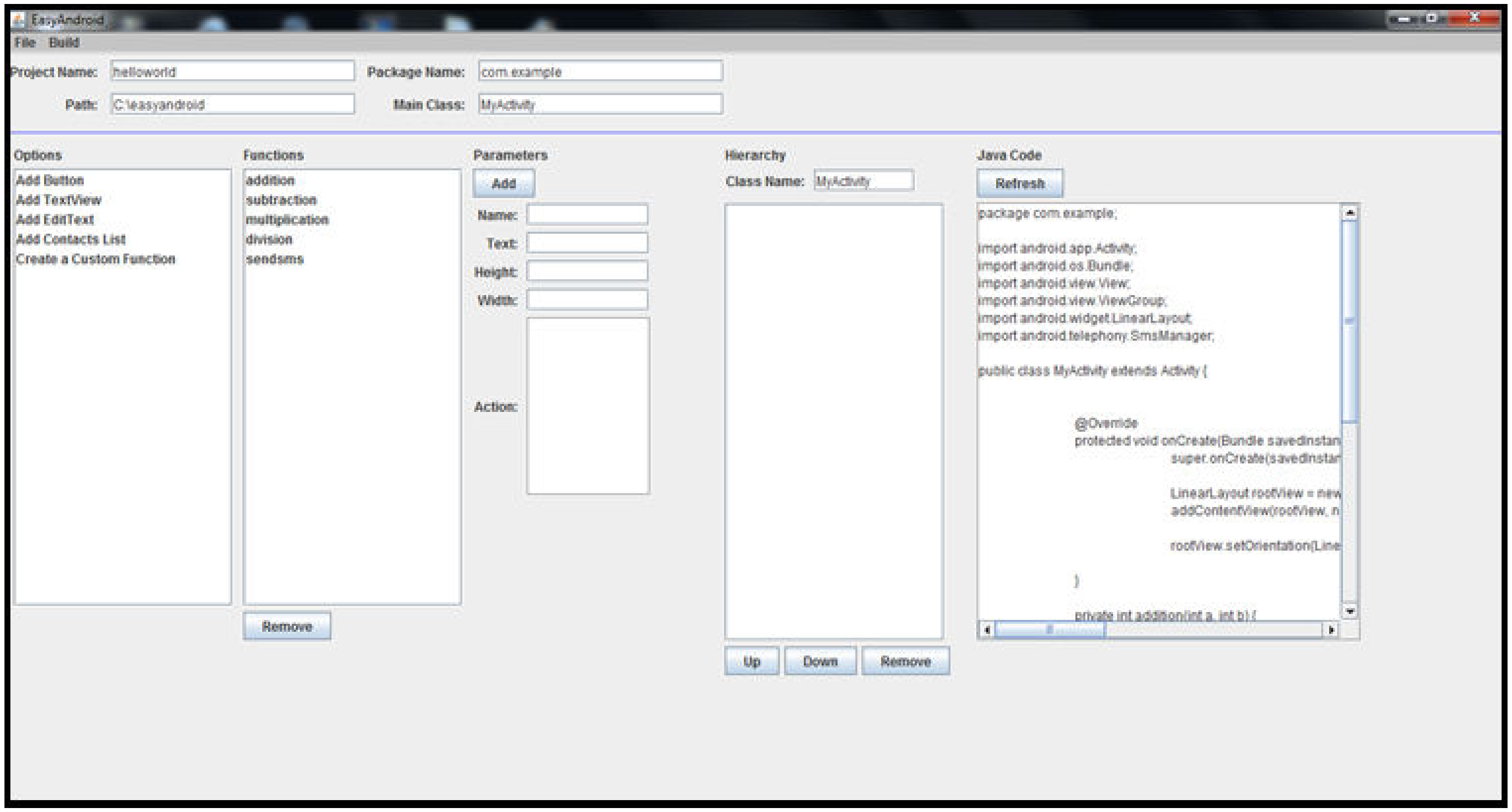
One of the challenges that existed from the beginning and remained throughout the development process was simply establishing the design for the tool. As mentioned before, there was a constant struggle to balance between keeping the tool's structure as abstract and flexible as possible, while not wasting 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, so much of the "wasted" or "lost" time was a necessary and unpreventable step in the process of refining EasyAndroid.

8.2: Focus

As mentioned briefly in the last section, the direction and vision for EasyAndroid changed several times over the course of its development. 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 is 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.

9: Evaluation



Project Name: helloworld
Package Name: com.example
Path: C:\easyandroid
Main Class: MyActivity
Name:
Text:
Height:
Width:
Action:
Class Name: MyActivity

```
package com.example;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.telephony.SmsManager;

public class MyActivity extends Activity {


        @Override
        protected void onCreate(Bundle savedInstan
                super.onCreate(savedInstan

                LinearLayout rootView = new
                addContentView(rootView, n

                rootView.setOrientation(Line

        }

        private int addition(int a, int b) {
```

Version 1 of EasyAndroid

9.0: 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.

9.0.0: 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 to confirm 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. This app is actually almost the same as the app that was built in the "Demo" section earlier.

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).

9.0.1: 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

9.0.2: 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 expected 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.

9.1: 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” fragment and the “Properties” fragment, I rearranged the fragments from “Options”-“Functions”-“ Properties” to “Options”-“ Properties”-“Functions.” Though the change is somewhat minor, I expect it to clarify the role of the “Options” fragment and the “Properties” fragment’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). On a related note, I also renamed the “Parameters” fragment as “Properties” to reduce confusion over the property “parameters” for the custom function option, and the “parameters” of each widget. Hopefully, these small changes give the user a better sense of exactly what those properties and features do.

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.

10: 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, and also makes it easier to portray what the user wants to build as their app. For example, “position these 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, would be completely automated. While there are situations in which 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 on, which 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.

11: 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.

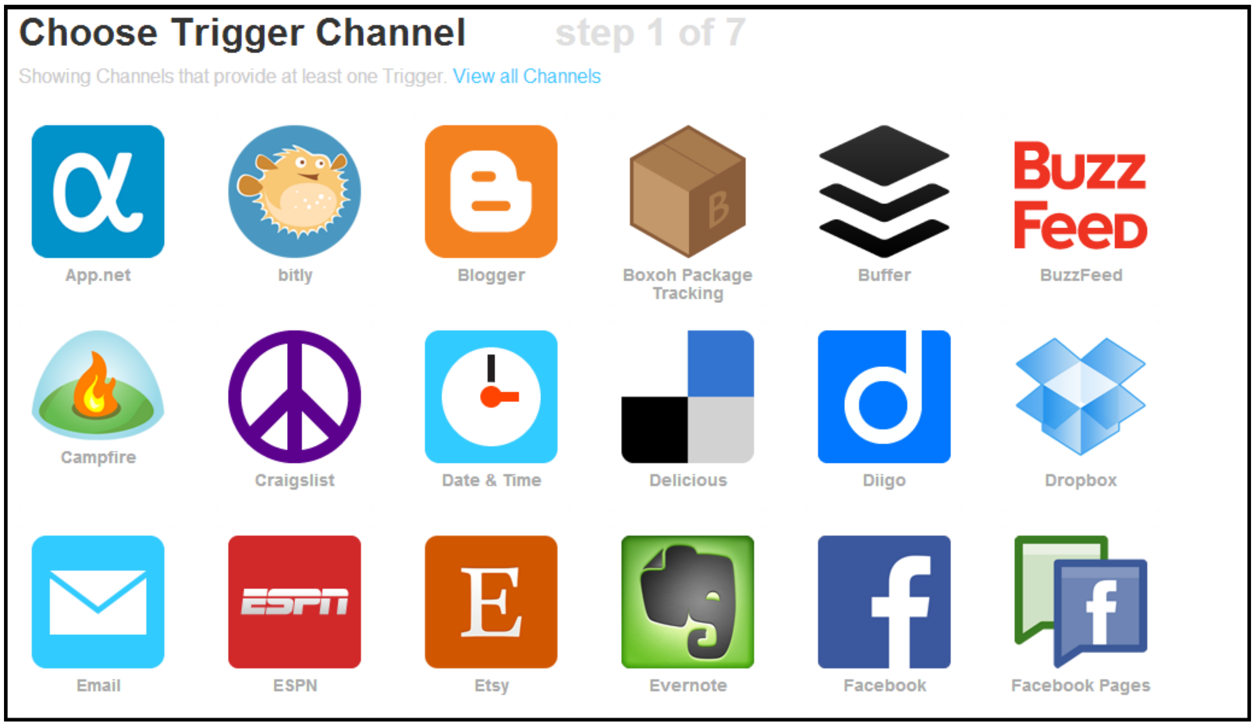
Another key lesson is that I should have emphasized research on related products and tools early on. I assumed that the four or so related tools that I discovered quickly would be enough and would background EasyAndroid the best, but then I discovered Andromo – the closest product to EasyAndroid – months 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.

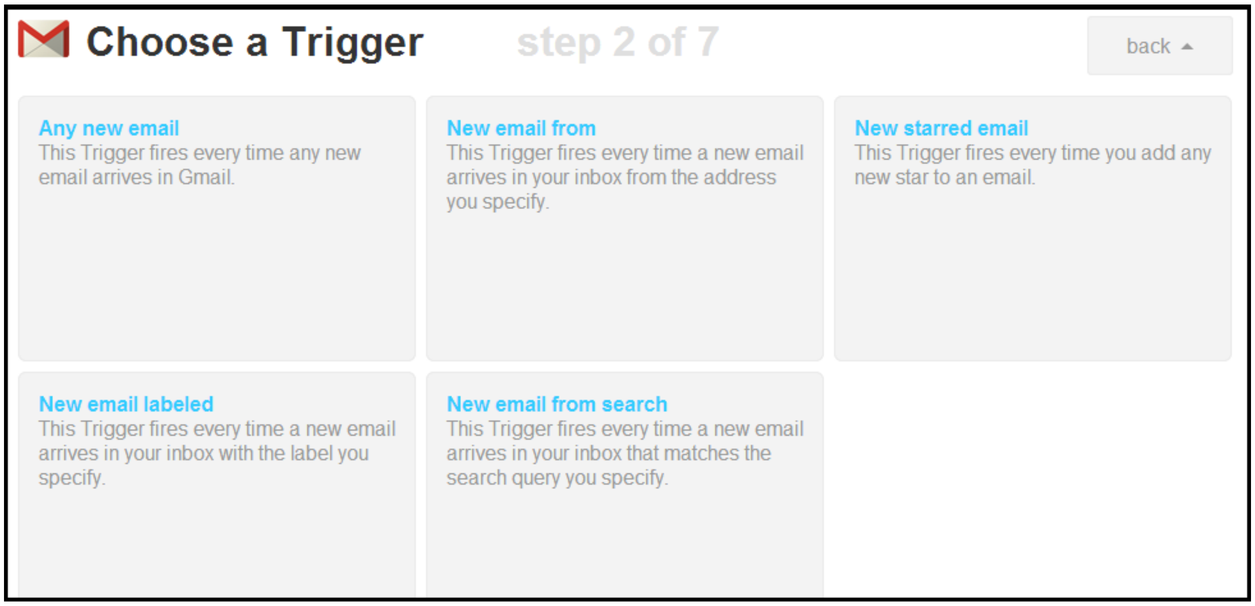
I also learned that you can never account for everything. Even spending as much time as I did on design and planning, there were still issues that I missed, which forced me to backtrack and change EasyAndroid. For example, in specifying custom functions, I originally created a new BufferedReader to capture 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 failing to account for potential issues manifested themselves in similar ways, further 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 – and 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."

Appendix 1: If This Then That





Choose a Trigger step 2 of 7

back

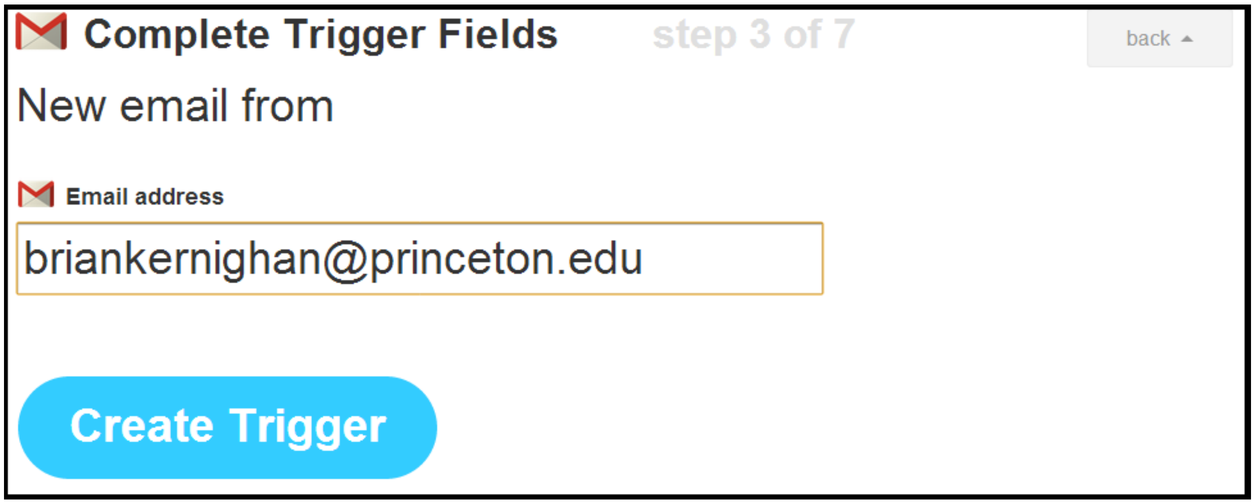
Any new email
This Trigger fires every time any new email arrives in Gmail.

New email from
This Trigger fires every time a new email arrives in your inbox from the address you specify.

New starred email
This Trigger fires every time you add any new star to an email.

New email labeled
This Trigger fires every time a new email arrives in your inbox with the label you specify.

New email from search
This Trigger fires every time a new email arrives in your inbox that matches the search query you specify.



Complete Trigger Fields step 3 of 7

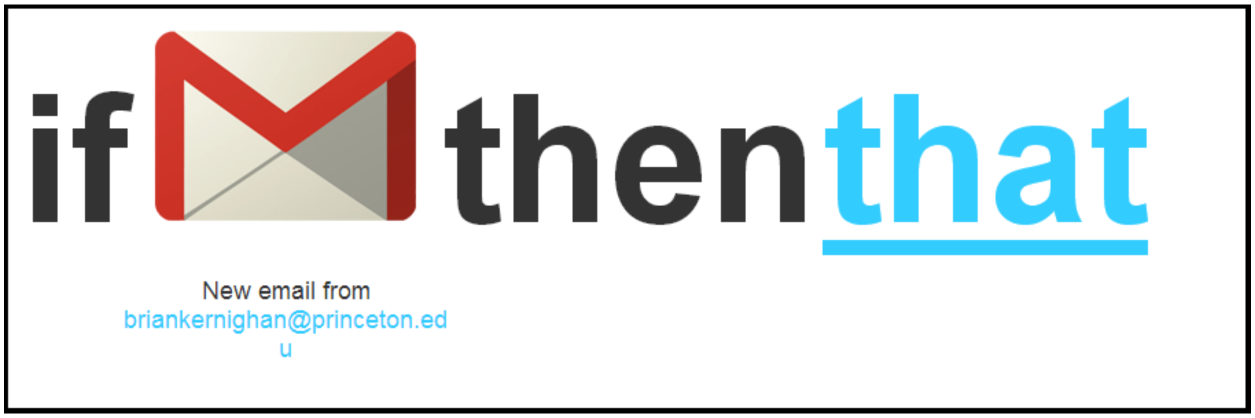
back

New email from

Email address

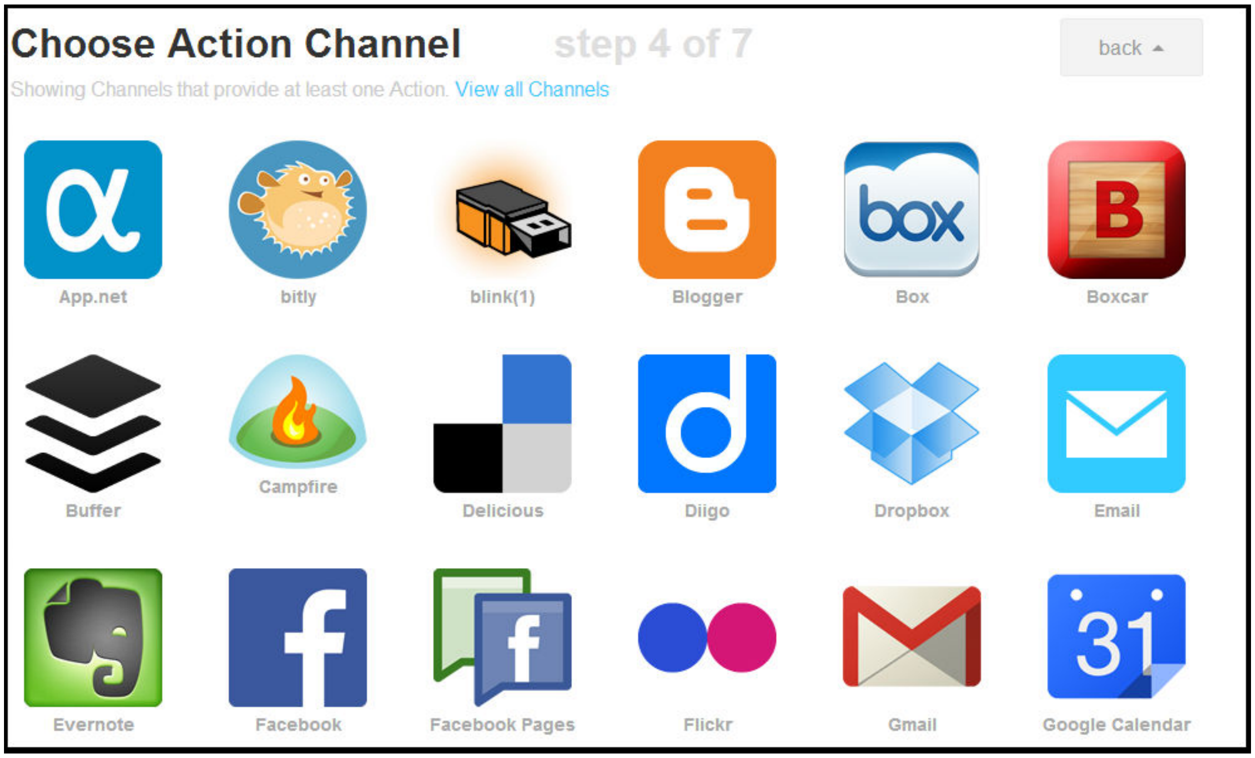
briankernighan@princeton.edu

Create Trigger



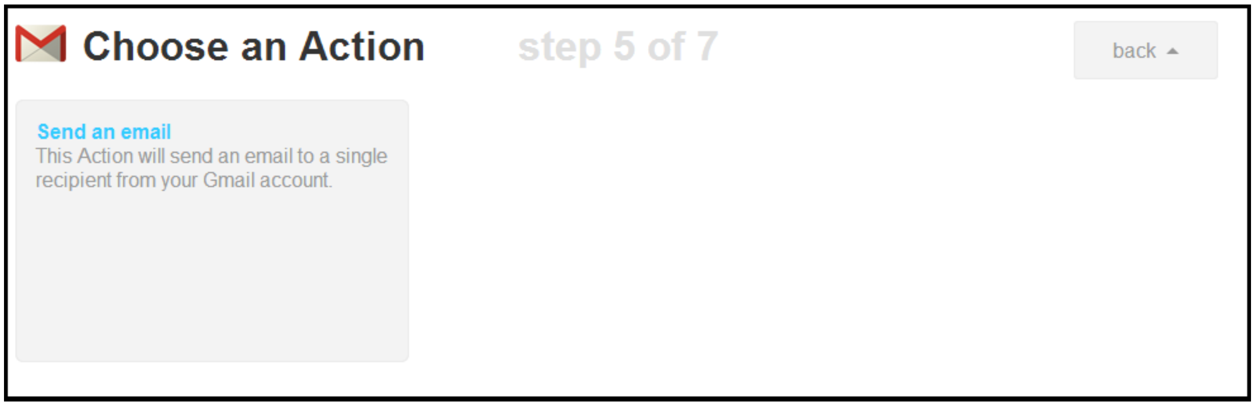
if then that

New email from
briankernighan@princeton.ed
u



Choose Action Channel step 4 of 7 back

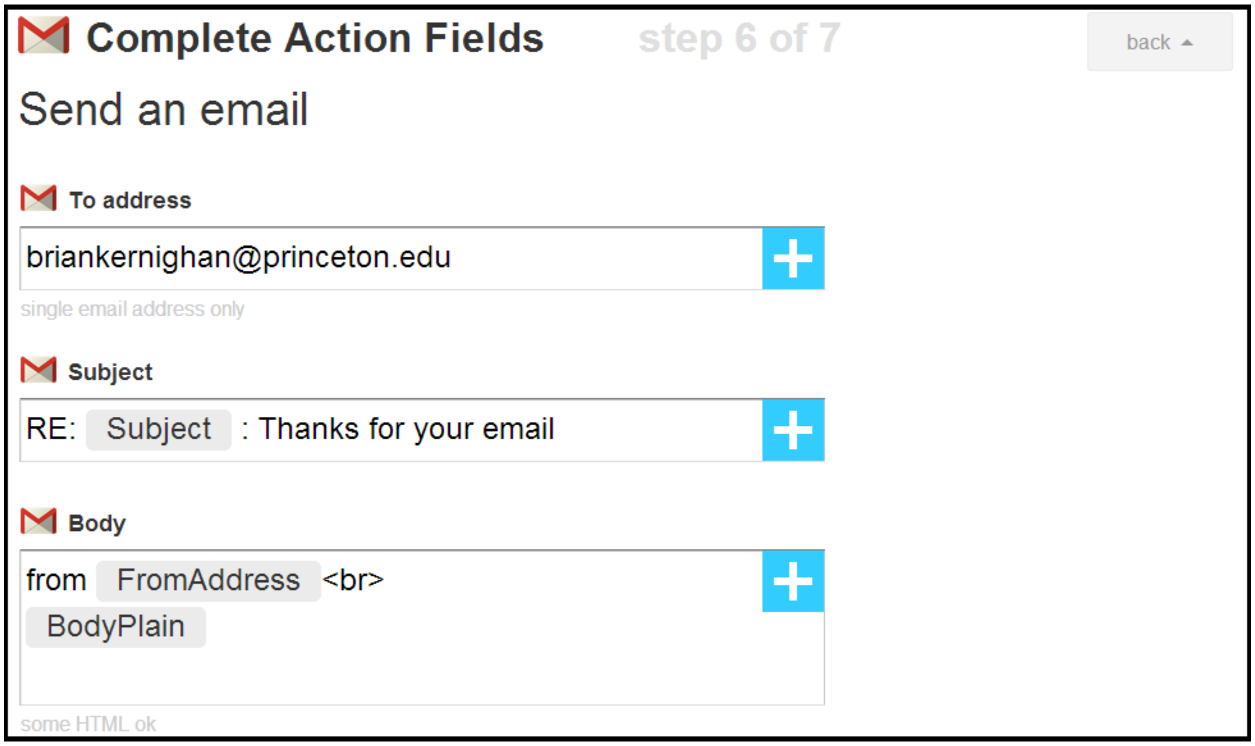
Showing Channels that provide at least one Action. View all Channels



Choose an Action step 5 of 7 back

Send an email

This Action will send an email to a single recipient from your Gmail account.



Complete Action Fields step 6 of 7

back

Send an email

To address

briankernighan@princeton.edu

single email address only

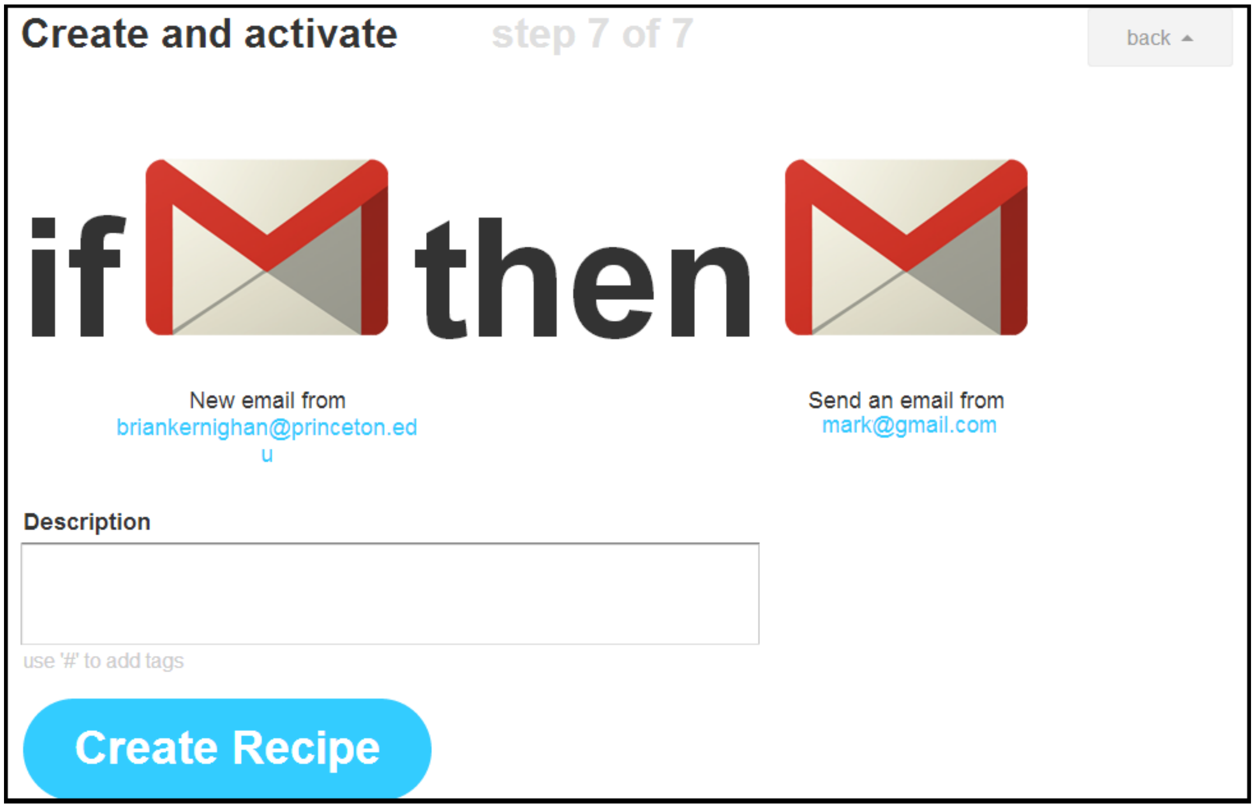
Subject

RE: Subject : Thanks for your email

Body

from FromAddress
BodyPlain

some HTML ok



Create and activate step 7 of 7

back

if then

New email from briankernighan@princeton.edu

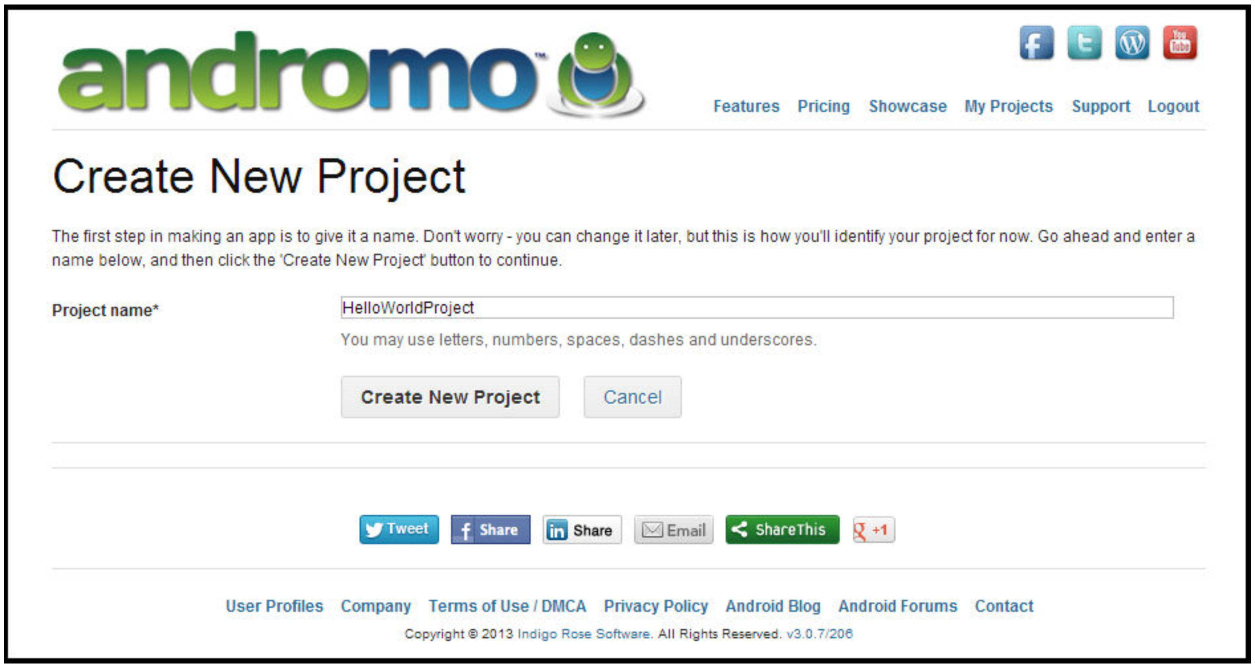
Send an email from mark@gmail.com

Description

use '#' to add tags

Create Recipe

Appendix 2: Andromo



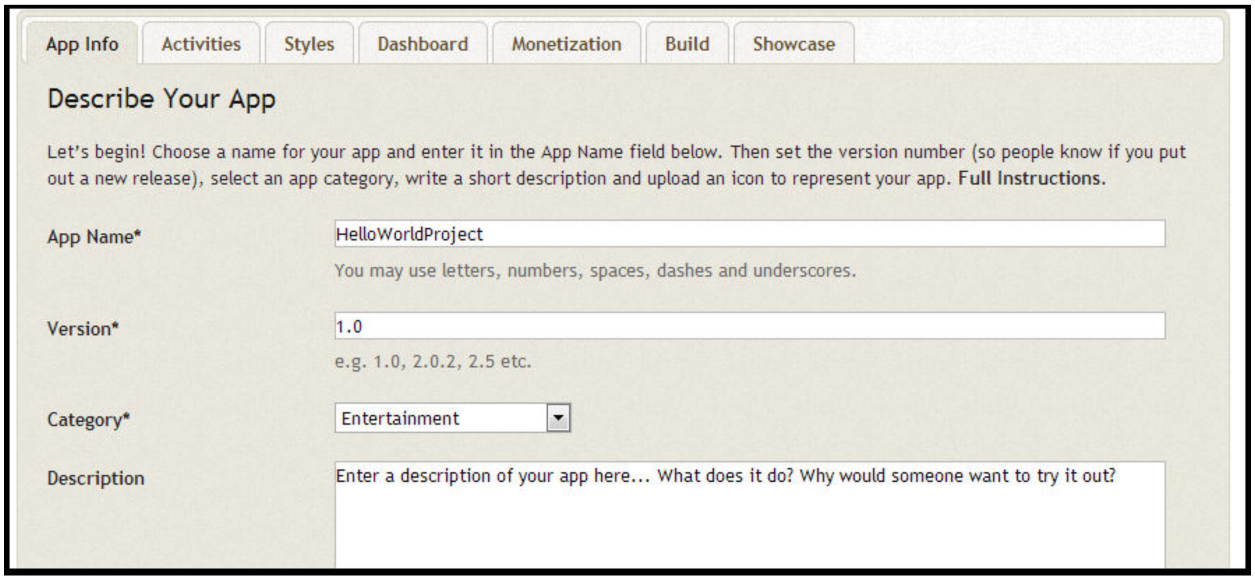
Create New Project

The first step in making an app is to give it a name. Don't worry - you can change it later, but this is how you'll identify your project for now. Go ahead and enter a name below, and then click the 'Create New Project' button to continue.

Project name*: HelloWorldProject

You may use letters, numbers, spaces, dashes and underscores.

Copyright © 2013 Indigo Rose Software. All Rights Reserved. v3.0.7/206



Describe Your App

Let's begin! Choose a name for your app and enter it in the App Name field below. Then set the version number (so people know if you put out a new release), select an app category, write a short description and upload an icon to represent your app. **Full Instructions.**

App Name*: HelloWorldProject

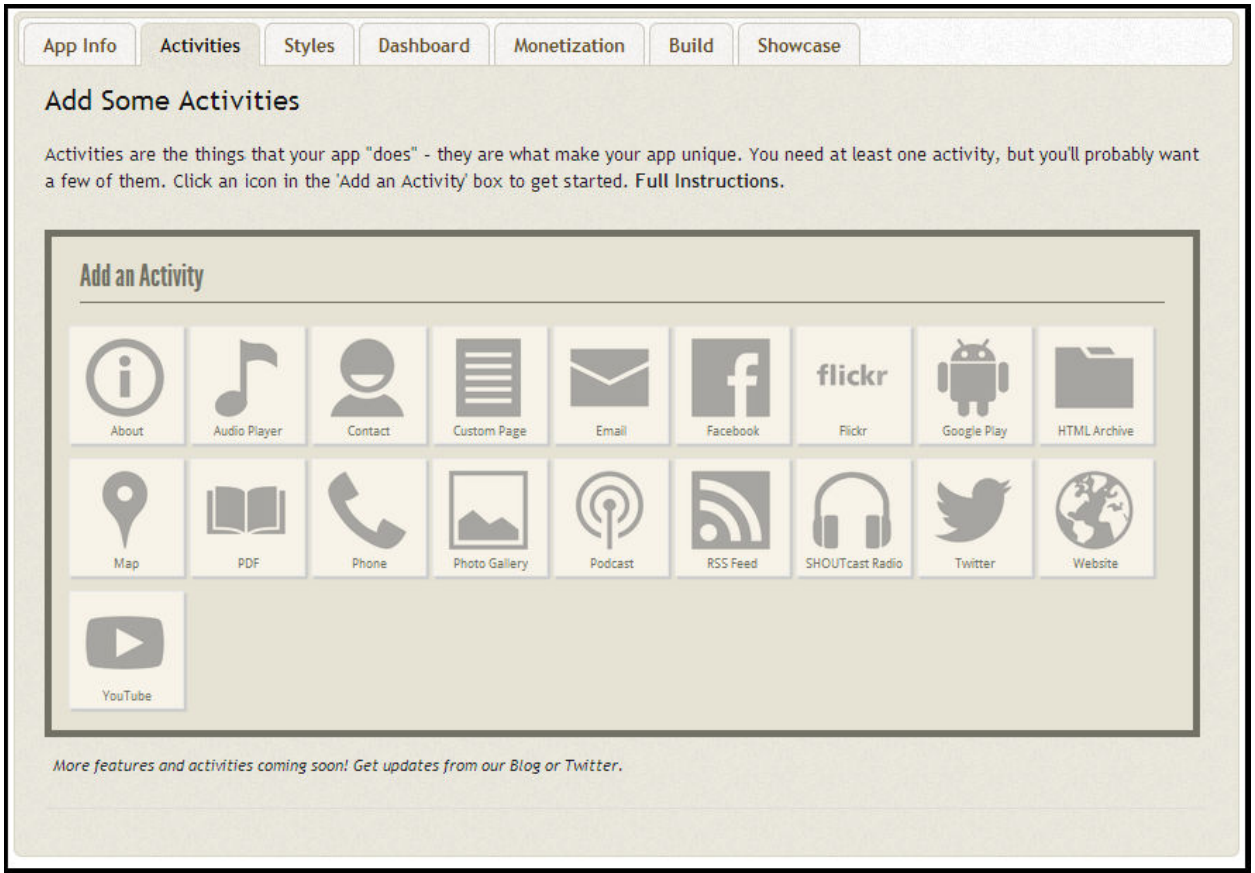
You may use letters, numbers, spaces, dashes and underscores.

Version*: 1.0

e.g. 1.0, 2.0.2, 2.5 etc.

Category*: Entertainment

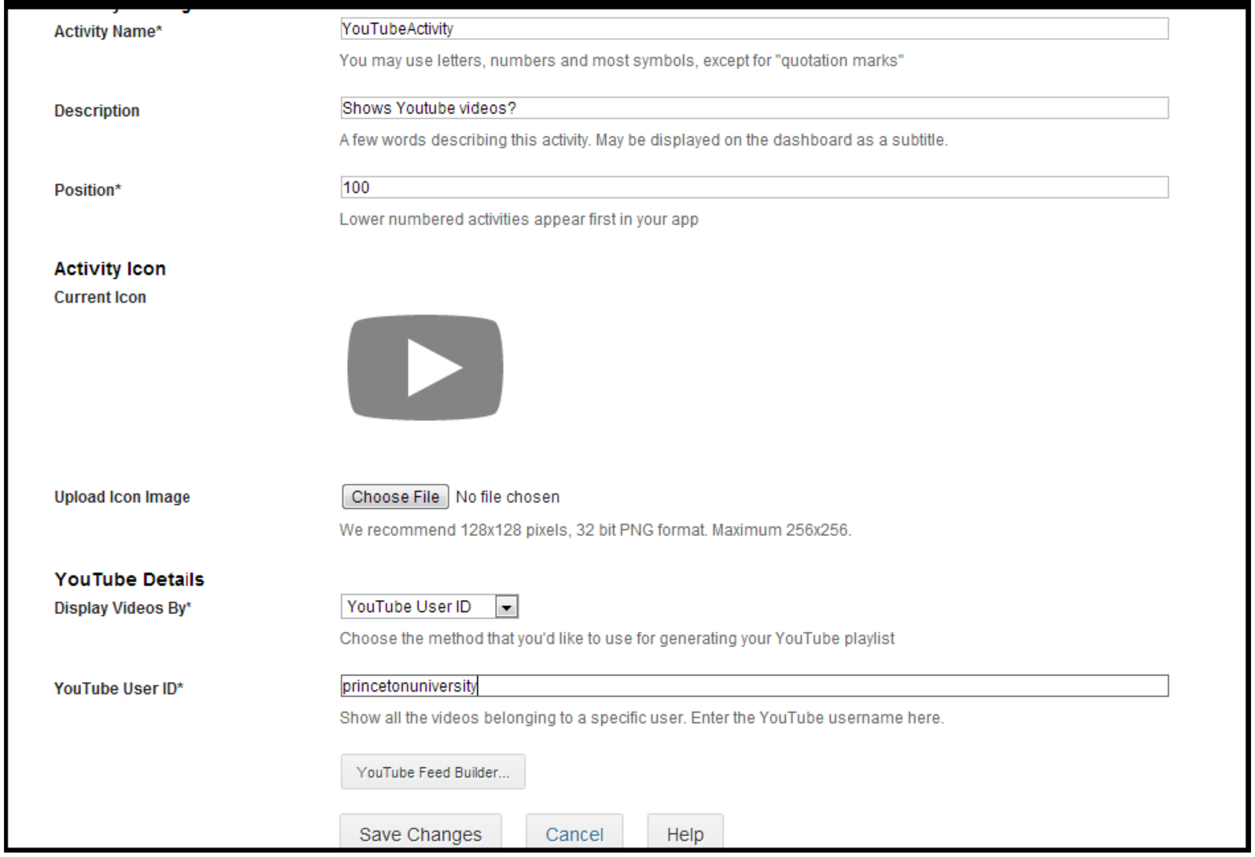
Description: Enter a description of your app here... What does it do? Why would someone want to try it out?



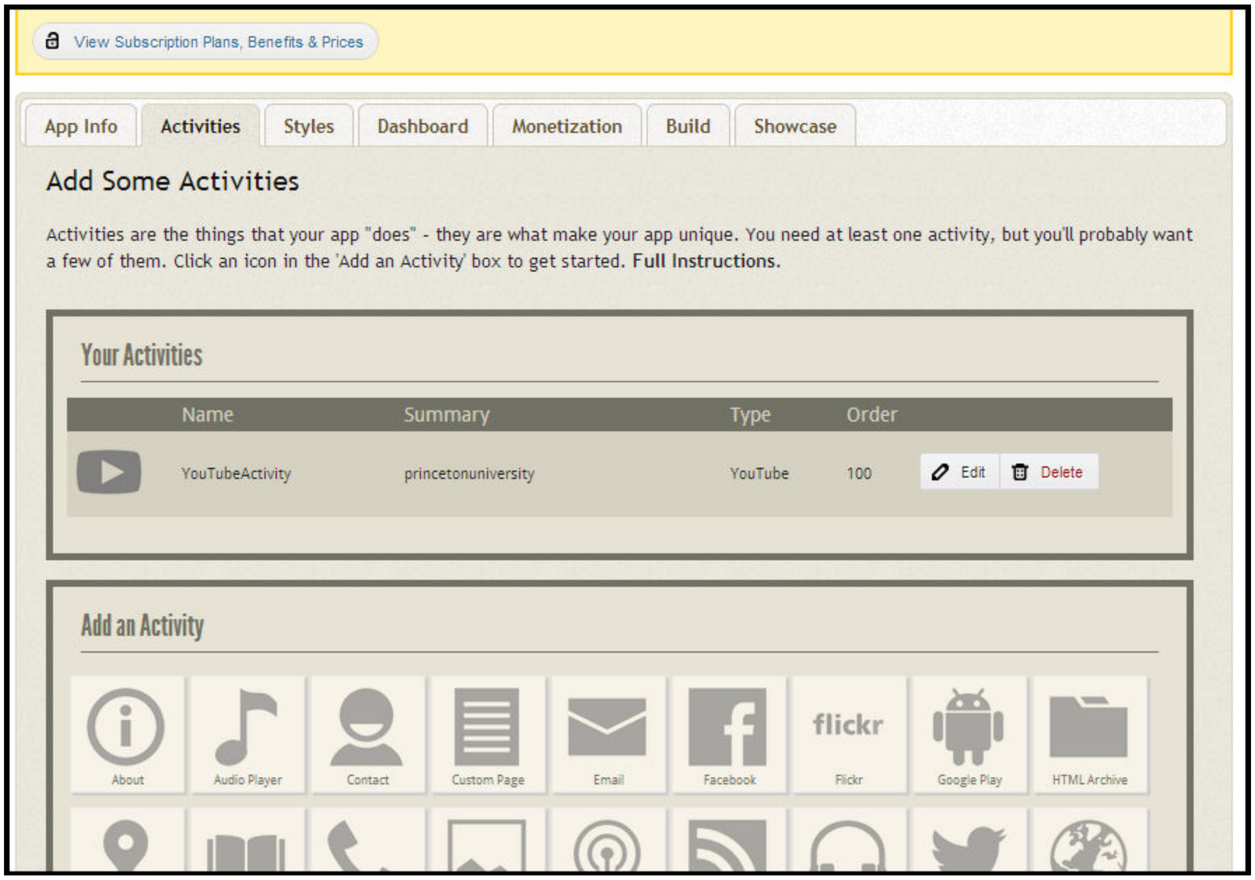
Add Some Activities

Activities are the things that your app "does" - they are what make your app unique. You need at least one activity, but you'll probably want a few of them. Click an icon in the 'Add an Activity' box to get started. **Full Instructions.**

More features and activities coming soon! Get updates from our Blog or Twitter.



Activity Name*	YouTubeActivity You may use letters, numbers and most symbols, except for "quotation marks"
Description	Shows Youtube videos? A few words describing this activity. May be displayed on the dashboard as a subtitle.
Position*	100 Lower numbered activities appear first in your app
Activity Icon **Current Icon**	
Upload Icon Image	Choose File No file chosen We recommend 128x128 pixels, 32 bit PNG format. Maximum 256x256.
YouTube Details **Display Videos By***	YouTube User ID Choose the method that you'd like to use for generating your YouTube playlist
YouTube User ID*	princetonuniversity Show all the videos belonging to a specific user. Enter the YouTube username here.



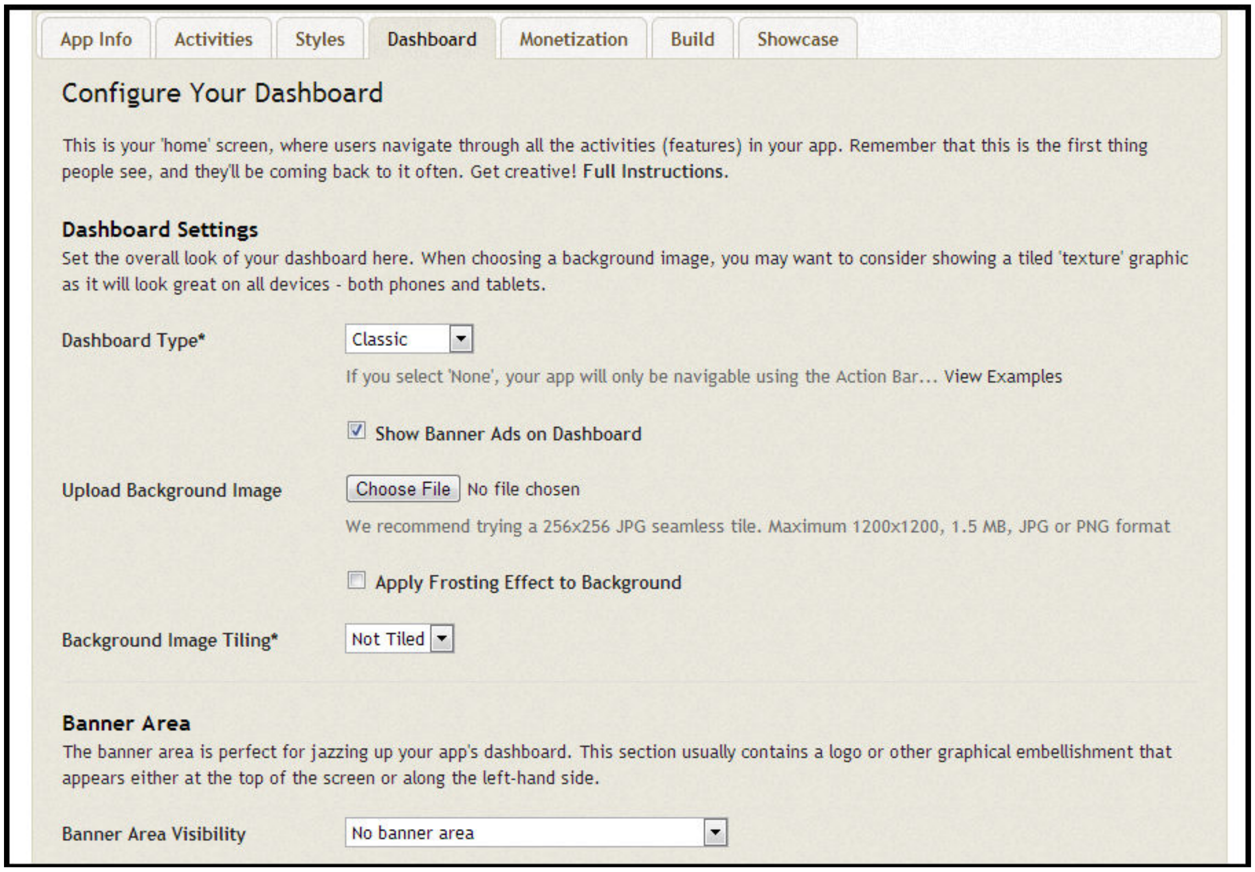
Add Some Activities

Activities are the things that your app "does" - they are what make your app unique. You need at least one activity, but you'll probably want a few of them. Click an icon in the 'Add an Activity' box to get started. **Full Instructions.**

Your Activities

	Name	Summary	Type	Order	
	YouTubeActivity	princetonuniversity	YouTube	100	Edit Delete

Add an Activity



Configure Your Dashboard

This is your 'home' screen, where users navigate through all the activities (features) in your app. Remember that this is the first thing people see, and they'll be coming back to it often. Get creative! **Full Instructions.**

Dashboard Settings

Set the overall look of your dashboard here. When choosing a background image, you may want to consider showing a tiled 'texture' graphic as it will look great on all devices - both phones and tablets.

Dashboard Type* Classic

If you select 'None', your app will only be navigable using the Action Bar... View Examples

[✓] Show Banner Ads on Dashboard

Upload Background Image Choose File No file chosen

We recommend trying a 256x256 JPG seamless tile. Maximum 1200x1200, 1.5 MB, JPG or PNG format

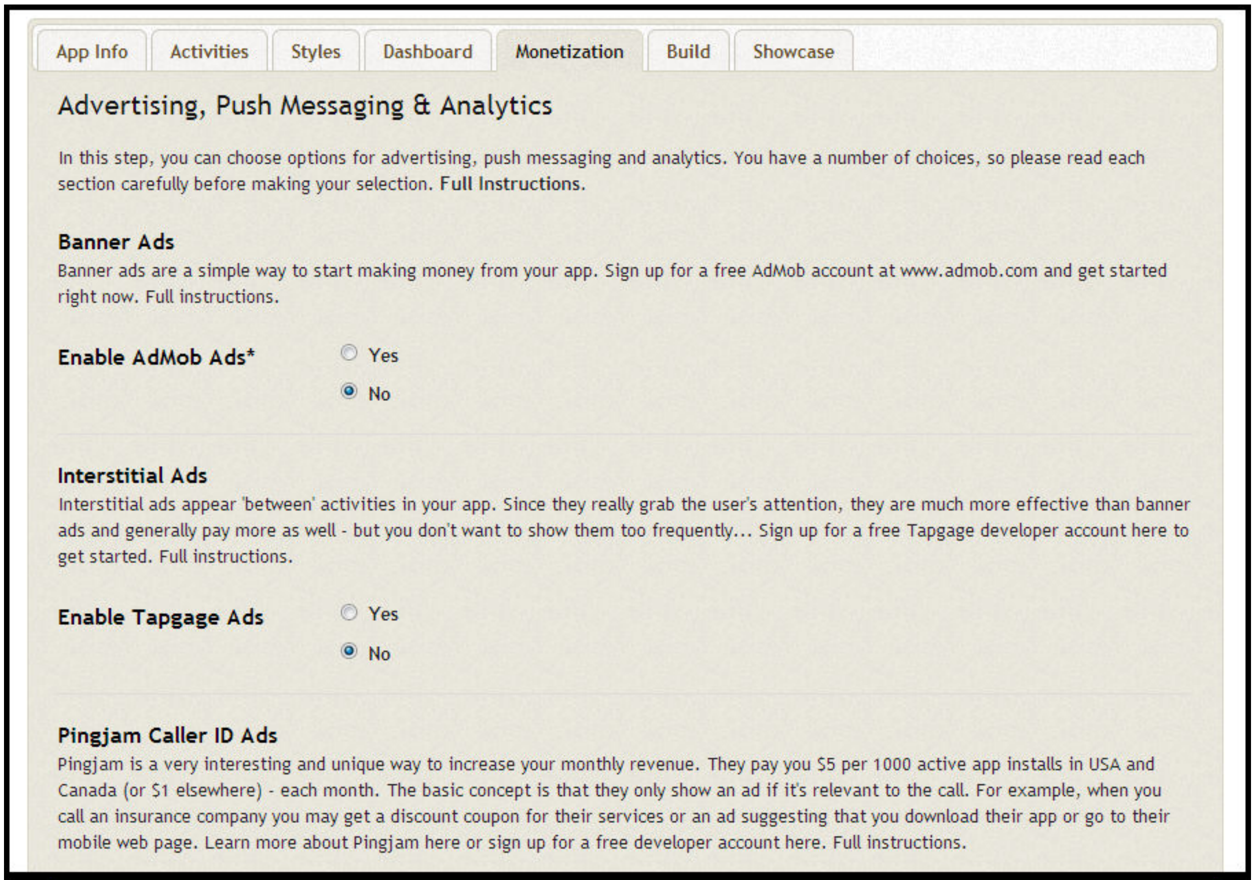
[ ] Apply Frosting Effect to Background

Background Image Tiling* Not Tiled

Banner Area

The banner area is perfect for jazzing up your app's dashboard. This section usually contains a logo or other graphical embellishment that appears either at the top of the screen or along the left-hand side.

Banner Area Visibility No banner area



Advertising, Push Messaging & Analytics

In this step, you can choose options for advertising, push messaging and analytics. You have a number of choices, so please read each section carefully before making your selection. **Full Instructions.**

Banner Ads

Banner ads are a simple way to start making money from your app. Sign up for a free AdMob account at www.admob.com and get started right now. Full instructions.

Enable AdMob Ads* Yes [✓ No]

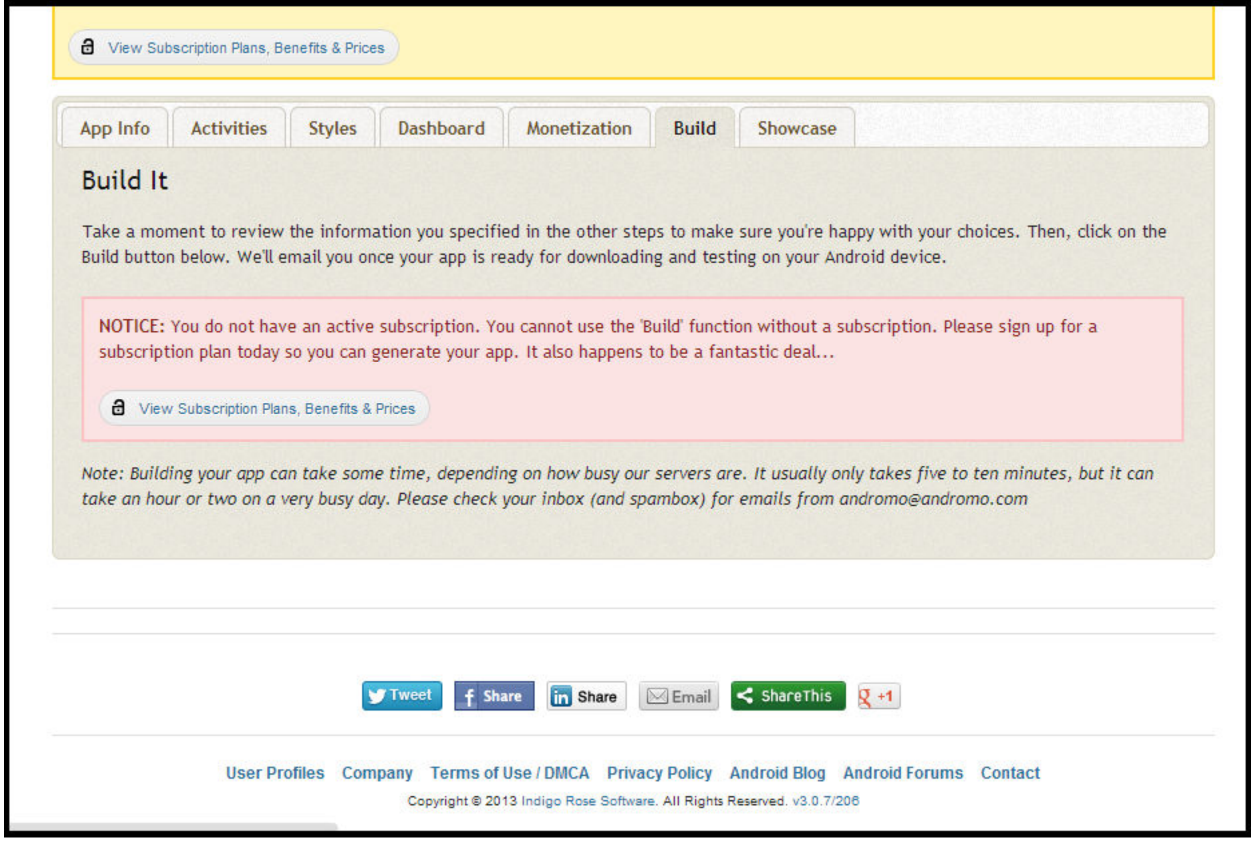
Interstitial Ads

Interstitial ads appear 'between' activities in your app. Since they really grab the user's attention, they are much more effective than banner ads and generally pay more as well - but you don't want to show them too frequently... Sign up for a free Tapgage developer account here to get started. Full instructions.

Enable Tapgage Ads Yes [✓ No]

Pingjam Caller ID Ads

Pingjam is a very interesting and unique way to increase your monthly revenue. They pay you $5 per 1000 active app installs in USA and Canada (or $1 elsewhere) - each month. The basic concept is that they only show an ad if it's relevant to the call. For example, when you call an insurance company you may get a discount coupon for their services or an ad suggesting that you download their app or go to their mobile web page. Learn more about Pingjam here or sign up for a free developer account here. Full instructions.



Build It

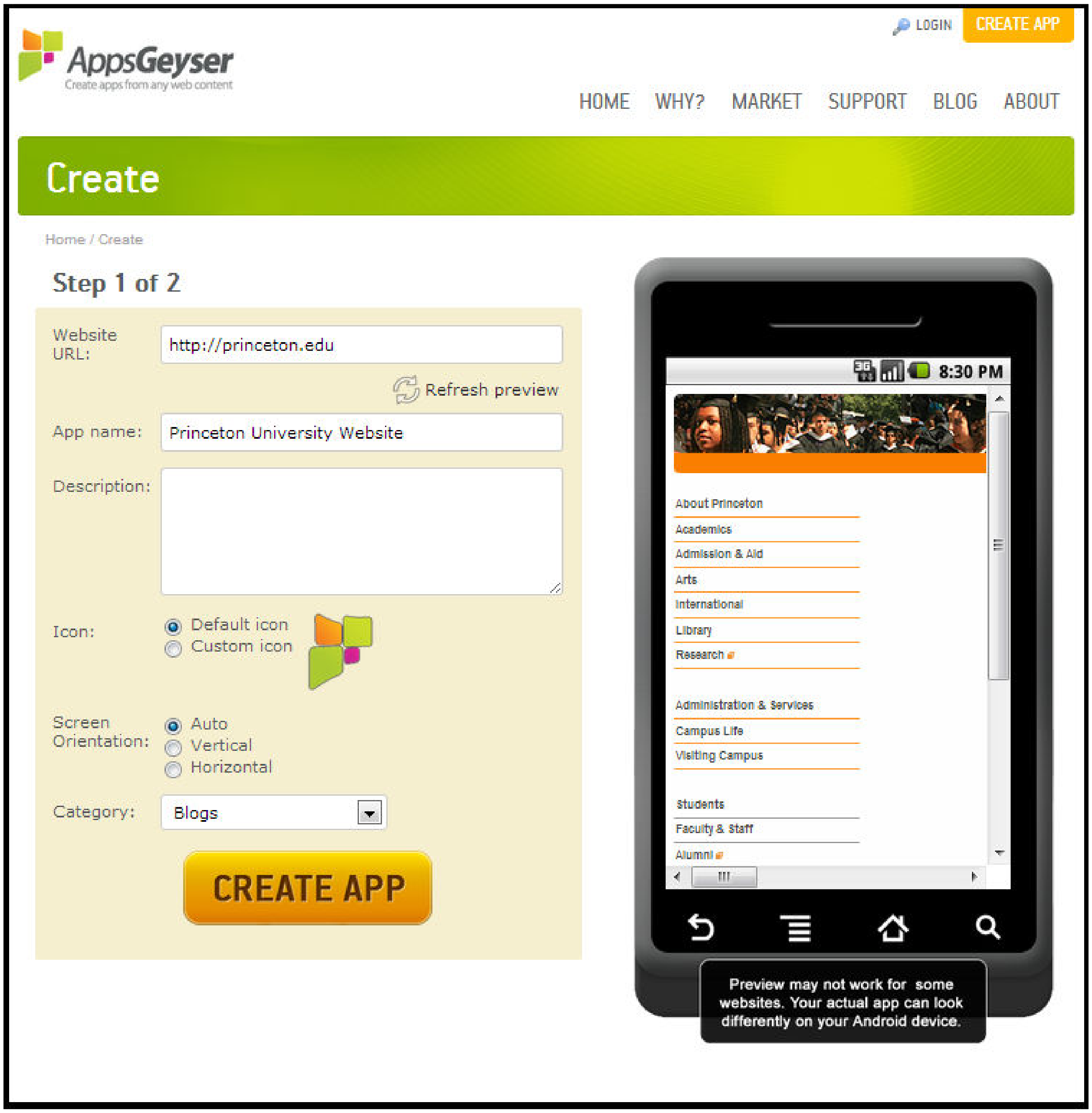
Take a moment to review the information you specified in the other steps to make sure you're happy with your choices. Then, click on the Build button below. We'll email you once your app is ready for downloading and testing on your Android device.

NOTICE: You do not have an active subscription. You cannot use the 'Build' function without a subscription. Please sign up for a subscription plan today so you can generate your app. It also happens to be a fantastic deal...

Note: Building your app can take some time, depending on how busy our servers are. It usually only takes five to ten minutes, but it can take an hour or two on a very busy day. Please check your inbox (and spambox) for emails from andromo@andromo.com

Copyright © 2013 Indigo Rose Software. All Rights Reserved. v3.0.7/206

Appendix 3: AppsGeyser



Step 1 of 2

Website URL: http://princeton.edu

App name: Princeton University Website

Description:

Icon: [✓ Default icon] Custom icon

Screen Orientation: [✓ Auto] Vertical Horizontal

Category: Blogs

Preview may not work for some websites. Your actual app can look differently on your Android device.