D’après “Modeling Android Applications” de Angel Roman, and Bruce Trask (MDE Systems, Inc.)

Android Background:

Google and T-Mobile released the revolutionary G1 mobile phone in 2008. Since then, other device manufacturers and service providers have committed to or have released android devices. Google and the Open Handset Alliance provide a powerful SDK and Application Framework for developing Android mobile applications.
The Android Application Framework provides an infrastructure that facilitates the re-use and creation of software components targeted at mobile consumer devices. Among the framework's capabilities are facilities for the application developer to select whether to reuse existing UI components - Maps, Contact lists, Message Lists, etc. - or create their own. In addition, applications are encouraged to delegate common actions - selecting an item from a contact list or viewing a location on map - to pre-defined components.

As powerful as this framework is, the developer is tasked to mold their application by tediously editing several XML files and writing glue code. These tasks have very little to do with the application logic. Domain specific tools and languages can provide a higher level of expressive power to handle the creation of Android applications enabling the user to concentrate on the application logic. Eclipse's modeling frameworks and tools provided by EMF, GEF and GMF are perfectly suited to provide these domain specific tools and languages.

Android Domain Overview:

As an illustrative example, this paper focuses on a significant subset of the Android application framework. We will focus on the following elements:

- *Activities*
- *Views*
- *Content Providers*
- *Services*
- *Intents and Intent Filter*
- *Broadcast Receivers*
- *Permissions*

The UI of an Android application is defined by one or more activities, each activity represents a single user interface screen. An activity uses one or more Views (widgets) to display information and interact with the user. Analog Clocks, Buttons, Checkboxes, DatePickers and Text are some example views provided by the Android framework.

Content Providers, as the name implies, provide access to application data to other android applications. An android application can gain access to the list of contacts or SMS received by interacting with the appropriate content provider. An android application can provide custom content and hence custom content providers to grant access to other applications.

Services are useful when a background task is required. The typical example is the audio player. Most likely the user would expect the ability to listen to music while browsing the internet or reading e-mail. Since android grants control of the screen to one application activity at any given moment, services come to the rescue. By implementing the audio player as a service, music can remain playing in the background while another activity is presented to the user. The audio service can be controlled via an activity that binds to the audio service. The implementation of audio playing as a service enables other applications to provide custom front ends to the audio functionality.

Android applications interact with each other by reacting to, or emitting Intents. Intents have the role of message containers. It typically contains the details of a desired action to execute. As an example, an application can dial a number by creating an intent with an URI of “tel: phone_number” and action of DIAL. To display the location on a map, an application does not need to re-implement th entire mapping application. It would simply emit an intent with an URI of “geo:latitude,longitude” and action of VIEW. Through some concatenation of miracles, the android framework notifies the built-in mapping application with the location Intent.

An android application specifies which intents are of interest by defining Intent Filters. The mapping application defines an intent filter that matches the URI scheme of “geo” and an action of VIEW. Using Intents and Intent Filters, applications have the ability to interact with each other while remaining loosely coupled.
When an application desires to be notified of an event, it may do so by registering broad cast receivers with the

proper intent filters. As an example, by registering a broadcast receiver with an intent filter with an action value of android.provider.Telephony.SMS_RECEIVED, an application can react to incoming messages.

Security is of high priority for android applications. Before an application can read the contacts list, access the internet, or performs an action which would result in access to sensitive information or service charges, permission must be granted by the user. By using Permissions an application can request access to GPS information, Network State, Camera, Calendar, etc.

Software Artifacts:

Having an understanding of the domain, we can take a look at what is required to implement two simple android applications. The first application will be an SMS Printer application which simply listen for incoming SMS messages and prints the content to the standard output stream. Although simple in concept, this application will use several android application elements.

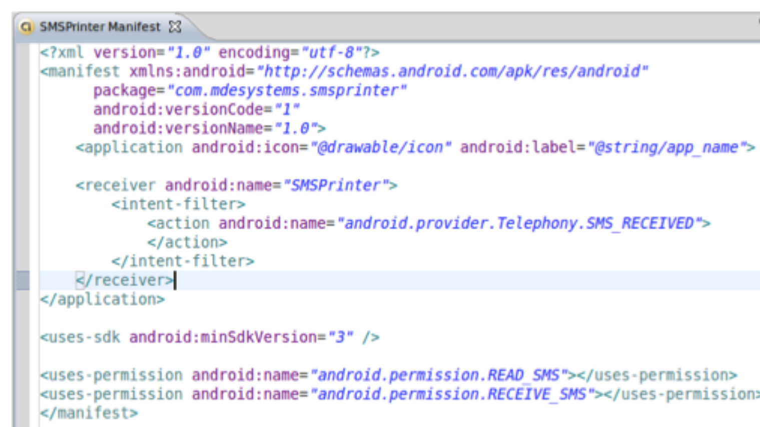
One possible solution to this application would consist of one Broadcast Receiver, an intent filter that allows SMS_RECEIVED intents to pass through, and two permissions: READ_SMS and RECEIVE_SMS. These permissions and action values are described in the android developers documentation. The Broadcast Receiver is implemented as a Java class that extends from Broadcast Receiver. The intent filters and permissions are contained within the applications manifest file. The manifest file is an xml file which describes the different android elements contained by an application.

Illustration-1 portrays the SMSPrinter BroadcastReceiver class.

```
SMSPrinter.java
package com.mdesystems.smsprinter;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.telephony.gsm.SmsMessage;

public class SMSPrinter extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Object[] objs = (Object[]) intent.getExtras().get("pdus");
        SmsMessage sms = SmsMessage.createFromPdu((byte[]) objs[0]);
        System.out.println("From: " + sms.getDisplayOriginatingAddress());
        System.out.println("Contents: " + sms.getDisplayMessageBody());
    }
}
```

Figure 1: Broadcast Receiver - SMS Printert

Illustration-2 portrays the android manifest file. intent filter and associated with the SMS Printer Broadcast Receiver.



```
SMSPrinter Manifest
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.mdesystems.smsprinter"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">

    <receiver android:name="SMSPrinter">
        <intent-filter>
            <action android:name="android.provider.Telephony.SMS_RECEIVED">
            </action>
        </intent-filter>
    </receiver>
</application>

<uses-sdk android:minSdkVersion="3" />

<uses-permission android:name="android.permission.READ_SMS"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_SMS"></uses-permission>
</manifest>
```

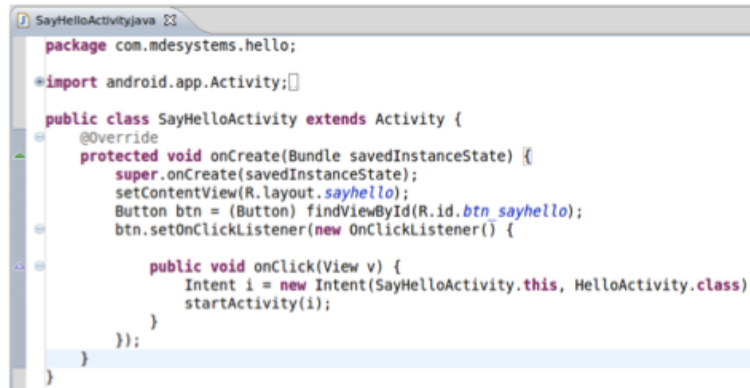
Figure 2: SMSPrinter XML Manifest file

As trivial as this example may seem, it requires error prone human based string matching, tedious xml input, and boiler plate java code for a simple task. The actual task specific SLOC count is 4 lines of code – the code inside

the onReceive method of SMSPrinter. The rest of code can be considered framework completion code which is rote and typically has very little to do with the actual solution. The ratio of boiler plate code to creative code is 5:1. Even though this is a trivial example, larger systems tend to exhibit a 4:1 ratio.

Our second example, will make use of other android domain elements. The application will be a GUI version of HelloWorld. The goal of the application is to display a button which will cause the hello world message to be displayed on a separate screen. Knowing that android allows only one UI screen at a time, and that each UI screen is implemented as an Activity. We can safely assume that we will make use of two Activities.

Illustration 3 represents the code for the activity containing the button prompting the user to "Say Hello." It consists of a Java class which extends the Activity class. The the onCreate() method displays a Button and registers a button listener. The button listener's onClick() method emits an intent whose purpose is to display the activity with the Hello message.



```
SayHelloActivity.java
package com.mdesystems.hello;

import android.app.Activity;

public class SayHelloActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sayhello);
        Button btn = (Button) findViewById(R.id.btn_sayhello);
        btn.setOnClickListener(new OnClickListener() {

            public void onClick(View v) {
                Intent i = new Intent(SayHelloActivity.this, HelloActivity.class);
                startActivity(i);
            }
        });
    }
}
```

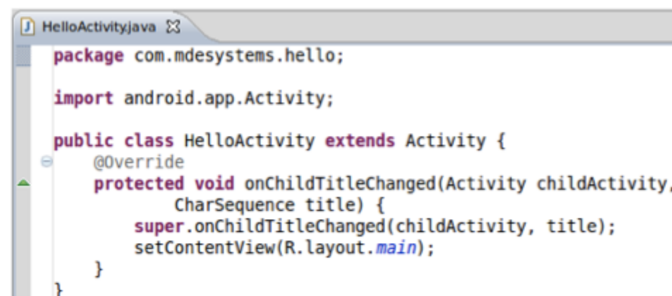
Figure 3: SayHelloActivity

Illustration 4 represents the xml description of the layout for Activity "Say Hello." It is a simple layout with a single Button labeled "Say Hello."

```
sayhello.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Say Hello"
        android:id="@+id/btn_sayhello">
    </Button>
</LinearLayout>
```

Figure 4: XML Layout for SayHelloActivity

Illustration 5 represents the Activity that displays the actual "Hello World" message.



```
HelloActivity.java
package com.mdesystems.hello;

import android.app.Activity;

public class HelloActivity extends Activity {
    @Override
    protected void onChildTitleChanged(Activity childActivity,
            CharSequence title) {
        super.onChildTitleChanged(childActivity, title);
        setContentView(R.layout.main);
    }
}
```

Figure 5: Hello Activity

Illustration 6 represents the xml layout for the screen displaying the "Hello World" message.

```
main.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    android:layout_gravity="center"/>
</LinearLayout>
```

Figure 6: XML Layout for HelloActivity

Illustration 7 portrays the runtime view of both activities, the button, and the message displayed.

Figure 7: Runtime view of Say Hello and Hello Activities

As simple as these applications are, they both required the creation of several domain artifacts. The majority of the code created is considered boiler plate code. Extrapolating these findings to bigger, non-trivial applications, it is easy to imagine the significant amount of effort expensed on the creation of boiler plate code.