International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

Volume 2 Issue 1 January 2013

International Manuscript ID : ISSN22308849-V2I1M1-012013

A STUDY PAPER ON ANDROID UI

Prof. Rajkumar A. Soni,
Asst. Professor,
MCA Department,
L. C. Institute of Technology, Bhandu,
Gujarat, India

Abstract
Android is an integrated open platform for mobile devices provided by Google Inc. It includes operating system, middleware and some key applications. It has also an excellent development and debugging environment. The main objective of this study is to explore how to design the user interfaces of handheld device based on Android. This paper focuses on the basic layout architecture for the user interface in an activity contained in an android application. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts. An android application can also create View and ViewGroup objects programmatically.

Keywords: **Android, Handheld Device, Layout, User Interfaces, Widget.**

1. Introduction

As and when time elapses, the generation is moving towards hand-held devices. As they are becoming increasingly powerful and diverse, the developers and designers gain the attention of the users by providing an attractive user interface with significant characteristics. Currently, users not only expect that handheld device has strong applications, but also a friendly user interface. Products which possess good UI design will achieve better user's demand and more profits. Thus this paper is aimed at exploring the designing of the user interfaces of handheld device based on Android platform. This paper discusses the design procedure through the coding method and also by using an XML. Design of user interface of handheld device is implemented on the basis of the requirement of an application. It covers the basic elements that make up a screen, how to define a screen in XML and load it in the code, and various other tasks users need to handle for user interface.

International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

Volume 2 Issue 1 January 2013

International Manuscript ID : ISSN22308849-V2I1M1-012013

2. UI Architecture

This hierarchy tree can be as simple or complex as you need it to be, and you can build it up using Android's set of predefined widgets and layouts, or with custom Views that you create yourself. As depicted in the figure 1.if we want to co-relate the hierarchy then we can see it from the base class as android.view.View. This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.).

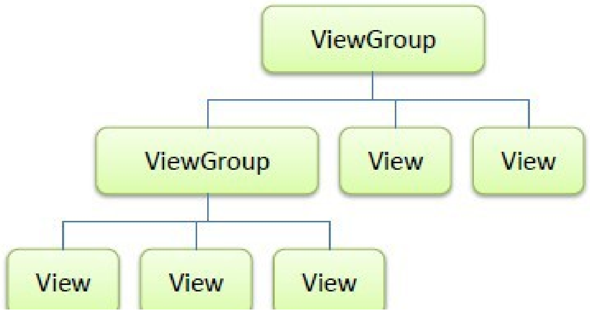

Fig 1. UI-Architecture

The ViewGroup subclass is the base class for layouts, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties. The view group is the base class for layouts and views containers.

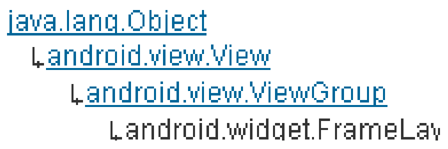

Fig 2. Class Hierarchy in context of UI-Architecture

We are having android.Widget.FrameLayout as the last in hierarchy which is designed to block out an area on the screen to display a single item. Generally, FrameLayout should be used to hold a single child view, because it can be difficult to organize child views in a way that's scalable to different screen sizes without the children overlapping each other. You can, however, add multiple children to a FrameLayout and control their position within the FrameLayout by assigning gravity to each child, using the android:layout_gravity attribute. Child views are drawn in a stack, with the most recently added child on top. The size of the FrameLayout is the size of its largest child (plus padding), visible or not (if the FrameLayout's parent permits). In order to attach the view hierarchy tree to the screen for rendering, your Activity must call the setContentView() method and pass a reference to the root node object. The Android system receives this reference and uses it to invalidate, measure, and draw the tree. The root node of the hierarchy requests that its child nodes draw themselves — in turn, each view group node is responsible for calling upon each of its own child views to draw them selves. The children may request a size and location within the parent, but the parent object has the final decision on where how big each child can be. Android parses the elements of your layout in-order (from the top of the hierarchy tree), instantiating the Views and adding them to their parent(s). Because these are drawn in-order, if there are elements that overlap positions, the last one to be drawn will lie on top of others previously drawn to that space.

3. Understanding UI

In this section, we will talk about the various elements that make up the UI of an Android application. We will discuss the various layouts available in Android to position the various widgets on the screen.

3.1 Android Screen UI Components

An Activity displays the user interface of an android application, which may contain widgets like Button, TextView, EditText, etc. Typically, you define your UI using an XML file (for example, the main.xml file located in the res/layout folder), which may look like this:

```
<?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”
```

International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

Volume 2 Issue 1 January 2013

International Manuscript ID : ISSN22308849-V2I1M1-012013

```
/>
</LinearLayout>
```

During runtime, we load the XML UI in the onCreate() event handler in our Activity class, using the setContentView() method of the Activity class:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
}
```

During compilation time, each element in the XML file is compiled into its equivalent Android GUI class, with attributes represented by methods. The Android system then creates the UI of the Activity when it is loaded. While it is always easier to build our UI using an XML file, there are times where we need to build our UI dynamically during runtime (for example, when writing games). Hence, it is also possible to create your UI entirely using code.

3.2 Widgets

- TextView is used in place of label which describe just a piece of a static text.
- EditText is having text entering and editing capability, it is generally used to enter the text inside it.
- Button is used to pass an action or say an event.
- CheckBox having check/uncheck capability basically used for selection.
- RadioButton contain a round selection which is used in a group so that a user can not select more than one option at the same time.
- RadioGroup is having more than one RadioButton when they are not allowed to get selected at the same time.
- ToggleButton can be used in an event that describes on/off functionality.
- ImageView is used to contain an image in it.
- ImageButton is a button having a look of a particular selected image.
- Spinner is a view that displays one child at a time and lets the user pick among them.
- Gallery is a view that shows items in a center-locked, horizontally scrolling list.
- AutoCompleteTextView is an editable text view that shows completion suggestions automatically while the user is typing. The list of suggestions is displayed in a drop down menu from which the

user can choose an item to replace the content of the edit box with.

- ProgressBar is a visual indicator of progress in some operation. It displays a bar to the user representing how far the operation has progressed; the application can change the amount of progress as it moves forward.
- GridView shows items in two-dimensional scrolling grid. The items in the grid come from the ListAdapter associated with this view.
- AnalogClock is a widget display an analogical clock with two hands for hours and minutes.
- DigitalClock is a widget display a digital clock. It implements separate views for hours/minutes/seconds
- DatePicker is a widget for selecting a date. The date can be selected by a year, month, and day spinners or a CalendarView. The set of spinners and the calendar view are automatically synchronized. The client can customize whether only the spinners, or only the calendar view, or both to be displayed. Also the minimal and maximal date from which dates to be selected can be customized.
- TimePicker is a view for selecting the time of day, in either 24 hour or AM/PM mode. The hour, each minute digit, and AM/PM (if applicable) can be controlled by vertical spinners. The hour can be entered by keyboard input. Entering in two digit hours can be accomplished by hitting two digits within a timeout of about a second.
- ListView is a view that shows items in a vertically scrolling list. The items come from the ListAdapter associated with this view.
- RatingBar is an extension of SeekBar and ProgressBar that shows a rating in stars. The user can touch/drag or use arrow keys to set the rating when using the default size RatingBar. When using a RatingBar that supports user interaction, placing widgets to the left or right of the RatingBar is discouraged. The number of stars set will be shown when the layout width is set to wrap content.

3.2 View and ViewGroup

An Activity contains Views and ViewGroups. A View is a widget that has an appearance on screen. Examples of widgets are Button, TextView, EditText, etc. A View derives from the base class android.view.View. One or more Views can be grouped together into a ViewGroup. A ViewGroup (which is by itself is a special type of View) provides the layout in which you can order the appearance and sequence of views. Examples of Viewgroups are LinearLayout, FrameLayout, etc. A ViewGroup derives from the base class android.view.ViewGroup. Each View and ViewGroup has a set of common attributes, some of which are shown below.

International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

Volume 2 Issue 1 January 2013

International Manuscript ID : ISSN22308849-V2I1M1-012013

Table 1: Common Attributes of View and ViewGroup

Attribute	***Description***
layout_width	Specifies the width of the View or ViewGroup
layout_height	Specifies the height of the View or ViewGroup
layout_marginTop	Specifies extra space on the top side of the View or ViewGroup
layout_marginBotto m	Specifies extra space on the bottom side of the View or ViewGroup
layout_marginLeft	Specifies extra space on the left side of the View or ViewGroup
layout_marginRight	Specifies extra space on the right side of the View or ViewGroup
layout_gravity	Specifies how child Views are positioned
layout_weight	Specifies how much of the extra space in the layout to be allocated to the View
layout_x	Specifies the x-coordinate of the View or ViewGroup
layout_y	Specifies the y-coordinate of the View or ViewGroup

Some of these attributes are only applicable when a View is in certain specific ViewGroup(s). For example, the layout_weight and layout_gravity attributes are only applicable if a View is either in a LinearLayout or TableLayout. Fill_parent constant indicates that it fills up with the entire width of its parent, while wrap_content constant means that it sizes itself with the contents contained within it. For e.g. If you do not wish to have the <TextView> to occupy the entire row, you can set its layout_width attribute to wrap_content.

Android supports the following ViewGroups:

- LinearLayout
- AbsoluteLayout
- TableLayout
- RelativeLayout
- FrameLayout
- ScrollView

It is recommended to nest different types of layouts to create the dynamic UI. The LinearLayout arranges views in a single column or single row. Child views can either be arranged vertically or horizontally. The default orientation of LinearLayout is set to horizontal. To change its orientation to vertical, we can change the orientation attribute to vertical also as depicted below.

```
<LinearLayout
        android:layout_width=”fill_parent”
        android:layout_height=”fill_parent”
        android:orientation=”vertical”
        xmlns:android=”http://schemas.android.com/apk/res/android”
>
```

The AbsoluteLayout lets you specify the exact location of its children using android:layout_x and android:layout_y attributes. Ideally AbsoluteLayout is used when we need to reposition our views when there is a change in the screen rotation.

```
<?xml version=”1.0” encoding=”utf-8”?>
<AbsoluteLayout
        android:layout_width=”fill_parent”
        android:layout_height=”fill_parent”
        xmlns:android=”http://schemas.android.com/apk/res/android”
>
<Button
        android:layout_width=”188px”
        android:layout_height=”wrap_content”
        android:text=”Button”
        android:layout_x=”126px”
        android:layout_y=”361px”
/>
</AbsoluteLayout>
```

The TableLayout groups views into rows and columns. We use the <TableRow> element to designate a row in the table. Each row can contain one or more views. Each view you place within a row forms a cell. The width for each column is determined by the largest width of each cell in that column.

```
<?xml version=”1.0” encoding=”utf-8”?>
<TableLayout
        xmlns:android=”http://schemas.android.com/apk/res/android”
        android:layout_height=”fill_parent”
        android:layout_width=”fill_parent”
```

International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

Volume 2 Issue 1 January 2013

International Manuscript ID : ISSN22308849-V2I1M1-012013

```
        android:background=”#000044”>
<TableRow>
<TextView
        android:text=”User Name:”
        android:width =”120px”
/>
<EditText
        android:id=”@+id/txtUserName”
        android:width=”200px” />
</TableRow>
</TableLayout>
```

The RelativeLayout lets you specify how child views are positioned relative to each other. Each view embedded within the RelativeLayout have attributes that allow them to align with another view. The attributes are as below. The value for each of these attributes is the ID for the view that you are referencing.

- layout_alignParentTop
- layout_alignParentLeft
- layout_alignLeft
- layout_alignRight
- layout_below
- layout_centerHorizontal

```
<?xml version=”1.0” encoding=”utf-8”?>
<RelativeLayout
        android:id=”@+id/Rlayout”
        android:layout_width=”fill_parent”
        android:layout_height=”fill_parent”
        xmlns:android=”http://schemas.android.com/apk/res/android”
>
<EditText
        android:id=”@+id/txtComments”
        android:layout_width=”fill_parent”
        android:layout_height=”170px”
        android:textSize=”18sp”
        android:layout_alignLeft=”@+id/lblComments”
        android:layout_below=”@+id/lblComments”
        android:layout_centerHorizontal=”true”
/>
<Button
        android:id=”@+id/btnSave”
        android:layout_width=”125px”
        android:layout_height=”wrap_content”
        android:text=”Save”
        android:layout_below=”@+id/txtComment”
        android:layout_alignRight=”@+id/txtComment”
/>
</RelativeLayout>
```

The FrameLayout is a placeholder on screen that we can use to display a single view. Views that we add to a FrameLayout is always anchored to the top left of the layout. We can add multiple views to a FrameLayout, but each will stack on top of the previous one. In the below example we must have an image named androidlogo.png under res/drawable folder.

```
<?xml version=”1.0” encoding=”utf-8”?>
<AbsoluteLayout
        android:id=”@+id/widget68”
        android:layout_width=”fill_parent”
        android:layout_height=”fill_parent”
        xmlns:android=”http://schemas.android.com/apk/res/android”
>
<FrameLayout
        android:layout_width=”wrap_content”
        android:layout_height=”wrap_content”
        android:layout_x=”40px”
        android:layout_y=”35px”
>
<ImageView
        android:src = “@drawable/androidlogo”
        android:layout_width=”wrap_content”
        android:layout_height="wrap_content"
/>
</FrameLayout>
</AbsoluteLayout>
```

A ScrollView is a special type of FrameLayout in that it allows users to scroll through a list of views that occupy more space than the physical display. The ScrollView can contain only one child view or ViewGroup, which normally is a LinearLayout. One recommendation is there for not to use a ListView together with the ScrollView. The ListView is designed for showing a list of related information and is optimized for dealing with large lists.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
        android:id="@+id/widget54"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android"
>
<LinearLayout
        android:layout_width="310px"
        android:layout_height="wrap_content"
        android:orientation="vertical"
>
<Button
        android:id="@+id/button1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Button 1"
/>
<EditText
        android:id="@+id/txt"
        android:layout_width="fill_parent"
        android:layout_height="300px"
/>
</LinearLayout>
```

```
</ScrollView>
```

4. Dynamic UI Creation using XML Inflation

[8] Basically there are two ways to create a User Interface in Android, either through XML or by creating the UI Programmatically. In this section we are going to see that how we can mix the two and use it for building dynamic User Interfaces. This kind of example can be explored when the application contains more UI elements to appear on the same screen and an action of the user achieved through one of the UI elements we have added.

We can achieve it in two ways:

- The dynamic part of the UI can be created programmatically. However it is not a very good way to mix UI and code. So, we can
- Define the dynamic UI too as an XML and use XML inflation to include it into the existing UI. We will see how to do the 2nd way, which probably is a good practice too.

Assume I have a very simple linear layout. In that I want to include a button. I can do it as part of the main XML itself. However, assume that this button is supposed to be reused in many activities and hence I have defined it as a separate XML.

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”
        android:id=”@+id/layout1”
>
</LinearLayout>
```

buttons.xml located under res/layout folder

```
<?xml version=”1.0” encoding=”utf-8”?>
<Button
        xmlns:android=”http://schemas.android.com/apk/res/android”
        android:id="@+id/button_small_left"
        style="?android:attr/buttonStyleSmall"
        android:text="Press to close"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
/>
```

Activity's onCreate(…) method of the InflateView class

```
public void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        final LayoutInflater inflater = (LayoutInflater)
        getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        Button b =(Button)inflater.inflate(R.layout.buttons,null);
        lLayout =(LinearLayout)findViewById(R.id.layout1);
        lLayout.addView(b);
}
```

First line indicating a call to Activity's onCreate() method by passing the previous state as a parameter of Bundle type. Second line is setting the current view's layout as main.xml. Third line is used to instantiate layout XML file into its corresponding View objects. getSystemService(String) method is called to retrieve a standard LayoutInflater instance that is already hooked up to the current context and correctly configured for the device we are running on. We are getting a handle to the LayoutInflater through the getSystemService(String) method. This inflater has a method inflate to which we pass the buttons.xml by passing the parameter R.layout.buttons. Then, we try to append this button to the LinearLayout that already exists and is set as the view in line 2 setContentView(R.layout.main). So we get a handle to the LinearLayout lLayout and add the new button to it in the last line. This one is the simplest way to inflate an XML and append it to an existing view. Here is the piece of code which demonstrates the dynamicity.

```
b.setOnClickListener(new OnClickListener()
{
public void onClick(View v)
{
// getChildAt() method returns the view at the specified //position in the group.
        if (lLayout.getChildAt(2) == null)
        {
```

```
        TextView tv = (TextView)inflater.inflate(R.layout.text, null);
        lLayout.addView(tv);
        }
}
});
```

On the click of this dynamically added button, we are showing how we can add more to the UI dynamically through inflation. Assume, on the click of the button, we want to show some new text. This TextView is defined in another XML called text.xml which is also in the res/layout folder. So, we are inflating from this XML and appending it to the LinearLayout view. So, a lot can be achieved for dynamic UI through inflation.

5. Model View Controller Concept

[7] Android GUI is single-threaded, event-driven and built on a library of nest-able components. The Android UI framework is organized around the common Model-View-Controller pattern.

5.1 The Model

The model represents data or data container. You can see it as a database of pictures on your device. Say, any user wants to hear an audio file, he clicks play button and it triggers an event in your app, now the app will get data from data store or database and as per input and creates data to be sent back to the user.

5.2 The View

The View is the portion of the application responsible for rendering the display, sending audio to speakers, generating tactile feedback, and so on. For example the view in a hypothetical audio player might contain a component that shows the album cover for the currently playing tune. User will always interact with this layer.

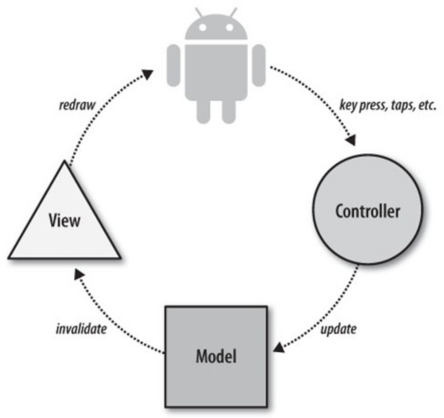

Fig 3. Model-View-Controller Concept

5.2 The Controller

The Controller is the portion of an application that responds to external actions: a keystroke, a screen tap, an incoming call, etc. It is implemented as an event queue. On User’s action, the control is passed over to controller and this will take care of all logic that needs to be done and prepare Model that need to be sent to view layer.

References

[1] Android Developers official website, http://developer.android.com/guide/topics/ui/index.html

[2] A visual interface editor for Android, http://www.droiddraw.org/

[3] Lauren Darcey and Shane Conder, “Android Wireless Application Development”, Pearson Education, 2nd ed. (2011)

[4] Reto Meier, “Professional Android 2 Application Development”, Wiley India Pvt. Ltd. (2011)

[5] Mark L Murphy, “Beginning Android”, Wiley India Pvt Ltd. (2009)

[6] Sayed Y Hashimi and Satya Komatineni, “Pro Android”, Wiley India Pvt Ltd. (2009)

[7] Rick Rogers and John Lombardo , “Android Application Development”, O'Reilly Media, Inc, 1st ed (2009)

[8] Technical Blog of Sai Geetha dedicated to Android, http://saigeethamn.blogspot.com/

[9] Jason Morris, “Android User Interface Development, Beginner’s guide”, PACKT publishing Ltd.