International Journal of Enhanced Research in Management & Computer Applications
ISSN: 2319-7471, Vol. 5 Issue 3, March-2016, Impact Factor: 1.544

Productivity and Efficiency Gains in Mobile Layouts Creation A case study with Android

Paulo Alves[1], Porfírio Lopes[2], Sara Paiva[3]

[1,2]Instituto Politécnico de Viana do Castelo, Portugal

[3]Instituto Politécnico de Viana do Castelo e Centro Algoritmi, Portugal

ABSTRACT

Mobile devices are currently present in everyone´s life and apps development has grown exponentially. The problem in developing for mobile devices is the existence of smartphones with multiple dimensions and resolutions. Users expect apps to adapt to their smartphone and provide an optimized interface for each device and/or orientation. Sometimes, depending on the user interface, we prefer to rotate the phone so we better visualize information. However, apps have to support the orientation change and be optimized for them. Sometimes, this demands programmers to develop two different layouts what makes them spend almost twice the time. In this paper, we present a desktop application and two Android Studio plugins that aim to help programmers to quickly generate a layout for portrait and landscape orientations based on a simple input or to automatic recognize and create of missing orientation files layout. We also present in this paper the design rules created and that allow the creation and transformation of layouts orientation. In the end of this paper we present the evaluation of the solution with examples of portrait and landscape layouts automatically generated by this solution.

KEYWORDS

Automatic interface generation; Android mobile layouts; multiple orientation support; interface design rules.

1. INTRODUCTION

We currently assist to a growing number of companies that are creating mobile version of their apps, which is normally being accompanied by the increase in mobile devices usage and their computing power. Android application development is continuously growing and like a desktop or web application, mobile apps need an interface. Some application only use portrait or landscape but most of the times they use both.

Android application development is continuously growing. In the fourth quarter of 2014, the Android operating system (OS) held 76,6% market share (IDC 2015) and the number of applications in the Play Store in July 2014 that was approximately 1.300.000 (Statista 2015).

With this rate of importance in the market, it is normal to assist to a big number of apps made by companies and also freelance developers. Android layout design has not an easy start considering that Android devices are manufactured by hundreds of different companies and can have several dimensions and resolutions. Programmers expect apps to adapt to all those devices and present an optimized interface for each device. Sometimes, depending on the user interface, we prefer to rotate the phone so we better visualize information. However, apps have to support the orientation change and be optimized for them. This demands programmers to develop two different layouts what makes them spend twice the time. While in other OSs such as Windows or iPhone, the Interface Development Editors (IDE) – respectively Visual Studio and XCode - can support a graphical layout construction, Android Studio – the most recent IDE for Android development – does not entirely support it, in a simple manner at least. Despite allowing more flexibility and better results than its predecessor Eclipse, developers often have to edit XML code directly to produce the final user interface (UI) (Hu & Zhang 2014).

This paper describes a methodology, first introduced in (Alves et al. 2015) to automatically create Android layouts which allows programmers to increase their level of efficiency and gains when coding. The methodology was applied in a desktop application and two Android Studio Plugins. The desktop application (and a similar Android Studio plugin) allows the definition of an interface layout in a simple way automatically creating portrait layout, landscape layout and the java class files in a specified location, ready to be used in Android Studio. The desktop application also allows Another feature is that the programmers to import templates with a predefined layout. The second Android plugin searches and checks the existent layouts of a given project and generates the missing orientation layout files.

Both desktop application and plugins, and the transformations made between the two types of orientation, are supported by self defined design rules that we also explain in this paper.

In the next section we introduce the problematic of applications with orientation support and also the structure of an android project. We refer some existent contributions and fundament the decisions made for the proposed solution. In section 3 we refer to the solution implementation, explaining in detail the purpose of the desktop application and plugins, the usage workflow, design rules defined for the transformations and the user interface. Section 4 presents the evaluation of the developed solutions and Section 5 presents conclusions and future work.

2. PROBLEM ANALYSIS AND LITERATURE REVIEW

The importance of a custom and optimized mobile layout depending on the device orientation is clear. It allows to maximize the space usage creating a better user experience. A typical example is provided in (GoogleDeveloper 2015a) which refers to a master/detail application. In this case, in a portrait orientation in a smartphone, a list is shown and when the user clicks an item, a new activity appears with its details. If in landscape orientation or tablet, the screen can be divided in two showing the list in the left size and on the right side the details, allowing a quicker visualization. As the master/detail view is a frequent layout, Android Studio already includes a wizard that allows the creation of this behavior without additional work for the programmer. But there are innumerous other situations where the programmer has to develop two independent layouts for both orientations which significantly increases the application time delivery.

Android project folder is mainly composed of a *java* folder (for functionality implementation) and a *res* folder, which is dedicated to all project resources, such as icons, images, menus, layouts, styles, etc. Inside the *res* folder there is always a *layout* folder that holds the layout of the application and inside it there are xml files for each defined layout. If the programmer wants, he can create new folders at the same level of the *layout* folder to define a layout for a specific size, density, orientation or aspect ratio. The supported sizes are *small*, *normal*, *large* and *xlarge*. The supported densities are: *ldpi* (low-density screens of ~120dpi), *mdpi* (medium density screens of ~160dpi – baseline density), *hdpi* (high-density screens of ~240dpi), *xhdpi* (extra-high-density screens of ~320dpi), *xxhdpi* (extra-extra-high-density screens of ~480dpi), *xxxhdpi* (extra-extra-extra-high-density screens of ~640dpi), *nodpi* (all densities) and *tvdpi* (screens somewhere between mdpi and hdpi of approximately 213dpi). The supported orientation are portrait and landscape. Finally, the supported aspect ratio are *long* (screens that have a significantly taller or wider aspect ratio - when in portrait or landscape orientation, respectively - than the baseline screen configuration and *notlong* (screens that have an aspect ratio that is similar to the baseline screen configuration) (GoogleDeveloper 2015b). For example, the file *res/layout/my_layout.xml* is the layout for normal screen size and the file *res/layout-xlarge-land/my_layout.xml* is the layout for extra-large in landscape orientation.

Considering the variety of options, every software that can contribute to automatically generate code is extremely important (Wen-zhen et al. 2014) and became a hot topic (Hu & Zhang 2014). One of such contributions was provided by Hu and Zhang which propose a code generation method based on model rule to improve the development efficiency of an application and reduce the development cycle duration. The proposed solution aims to automatically generate code from a series of design rules and models, disregarding however the device orientation issue. Another relevant study is presented in (Sahami Shirazi et al. 2013) where the authors analyzed 400 apps from Google Play regarding several aspects such as multilingual and high pixel density support and also the type of used layout and controls. 67% of the analyzed apps used the LinearLayout container followed by the Relative Layout with 24%. Regarding controls, the most used ones are the TextView (36%), ImageView (16%) and the Button (9%).

We analyzed current top applications such as Instagram, Pinterest, 9gag, LinkedIn and Facebook and concluded their layout could be created, in its majority, using LinearLayout containers and TextView, ImageView, EditText and Button controls. With this knowledge, we defined as our purpose to build a solution that helps programmers to create a first version of the layout of an application, for portrait and landscape, saving them time.

3. IMPLEMENTATION

As afore mentioned, this paper has two main contributions: a desktop application and two Android Studio plugins. One of the plugins have a similar behavior to the desktop application, so we start to describing both in the next section. The other plugin is explained in the last section.

3.1 Layouts creation - Desktop and Plugin

The first contribution this work provides is a desktop application and a very similar Android Studio plugin (for those that want to work inside Android Studio and not in a different application). Both provide a simple interface to specify the intended layout and automatically create portrait and landscape layouts along with the java class files, ready to be used in Android Studio. In the desktop application only, the user can also import templates to the project. The main goal is to provide the user with a simple interface to specify the controls he wants in the interface and generate both portrait and landscape version with a button click. After specifying the intended layout, and in order to create the two orientation files for the layout, two operations are performed:

- Creation of an intermediary file based on the widgets the user inserted
- Creation of portrait and landscape XML layouts based on design rules and the intermediary file.

3.1.1 Usage Workflow

In the Desktop Application, the programmer starts by selecting the project he wants to create automatic layouts in. Then, and as shown in Figure 1, he can start creating automatic layouts that will be validated before generating portrait and landscape layouts and also the correspondent JAVA activities. The programmer can also choose a template that has a pre-defined layout which accelerates his work.

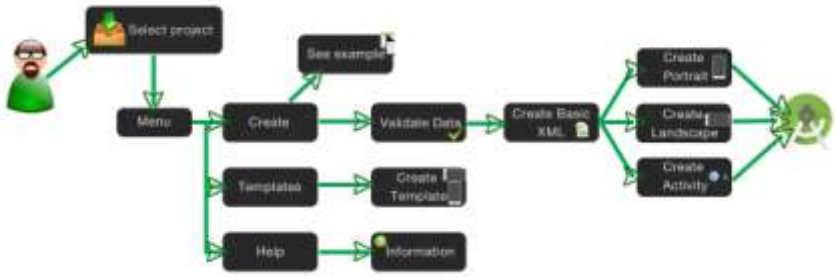

Figure 1. Functionalities of the desktop application

The plugin for creating layouts is very similar but in this case the programmer is using the same tool and not a different one which can be important in terms of efficiency. Android Studio will automatically detect which project is in use and, after the layout is specified, all files regarding layouts and JAVA activities are automatically created in the right folders and the project is refreshed. The functionalities are expressed in Figure 2.

Figure 2. Functionalities of the plugin for layouts creation

3.1.2 Intermediary file creation

The intermediary file is created based on the controls the user specified for the interface and has the XML Schema Definition (XSD), represented on Table 1. The intermediary file has a XML Element named *widget* for each control and inside it two other XML elements: one that specifies the type of control (ImageView, TextView, EditText, Button, RadioButton, CheckBox or Switch) and other that specifies the group. The group value will be zero for controls that do not share the horizontal line with other controls. When a line has several controls they will all share the same group id. Different lines will have different group ids.

Table 1. XSD of the intermediary file

```
<xs:schema attributeFormDefault=”unqualified”
elementFormDefault=”qualified”
xmlns:xs=”http://www.w3.org/2001/XMLSchema”>
  <xs:element name=”Layout”>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=”Widget” maxOccurs=”unbounded”
minOccurs=”0”>
          <xs:complexType>
            <xs:sequence>
              <xs:element type=”xs:string” name=”Type”/>
              <xs:element type=”xs:byte” name=”Group”/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The algorithm followed to create the intermediary file is represented in Figure 3. The first step after reading the input from the textbox is to validate the user input that mainly consists on checking if any unsupported control was inserted. If the input is valid, each line is iterated and a different sub-process is applied for lines with one single control and for lines with more than one control. If only one control exists, the control code is transformed to uppercase (only for standardization) and after identifying the control (ImageView, TextView, EditText, Button, RadioButton, CheckBox or Switch), a new Widget tag is added to the intermediary XML file with tag *Type* equals to the control and the tag *Group* equals to 0 (which means there is no other control in that line). If the line has more than one control, the line is split and for each character a new Widget tag is added to the intermediary XML file with tag *Type* equals to the control and the tag *Group* equals to 1 (which means there is another control in the line).

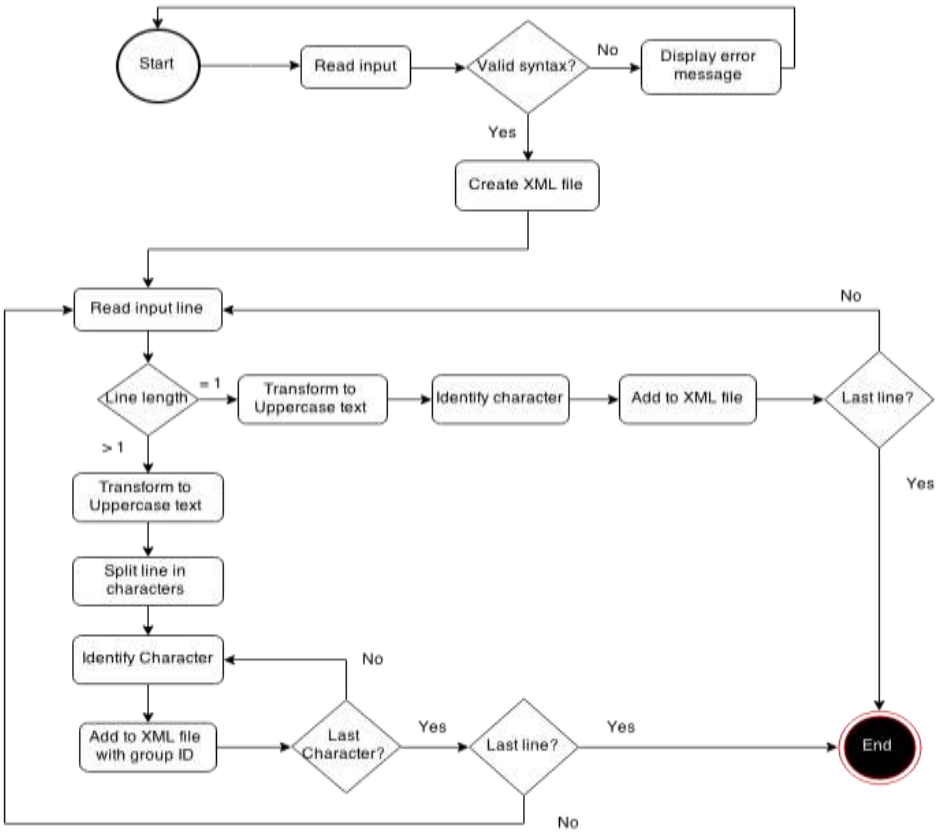

Figure 3. Flowchart for the intermediary XML file creation from the user input

3.1.3 Design rules

After the intermediary file is created, a set of design rules is executed to transform the intermediary file in Android layout files, which will be created in the chosen *res* folder and Android Studio is automatically refreshed so the programmer can continue to edit the layout file, namely specifying the images source, the text style, margins, alignment, among others.
Two distinct set of design rules were defined: one to create portrait layout from the intermediary file (represented in the flowchart on the left in Figure 4) and another to create the landscape layout from the same intermediary file (represented in the flowchart on the right in Figure 4).

The portrait generation flowchart starts by creating a XML file and inserting a linear layout with vertical orientation (VLL). The intermediary file is opened and an iteration is made through all existent widgets. If the group id tag of the widget is zero it means it is the only control in that line so the control identified by the type tag is added directly to VLL. We increment the id counter so each control added to the XML layout file has a different identifier, which is required by Android. If the group id of the widget is greater than zero, it means the control will be inside a horizontal linear layout (HLL) with the id equal to the group id of the widget. So we start by verifying if that HLL exists (it will exist to all widget of the group except to the first one). If it does, we simply add the control to it. If it doesn't we first create the HLL inside the VLL, and then add the control to the HLL. In both cases, we also increment the id counter.
The landscape generation flowchart starts by creating a XML file and a horizontal layout with two vertical layouts inside. The purpose is to use all the horizontal space and distribute the controls through both vertical layouts. In order to decide if we put the group in the left or right layout, we start by counting the number of groups (*ng*) in the intermediary file. For each widget read, we verify if it is part of a group or not. If it isn´t, we verify the number of this group and compare to *ng*. If it is equal or less than *ng/2*, the control is added to the left vertical layout; otherwise it is added to the right vertical layout. The purpose of this reasoning is to have the same number of groups in the right and in the left if the number of groups is pair. If it is odd the left vertical layout has one more group than the right vertical layout. After adding the control to the appropriate vertical layout we increment the counter and process the next widget. In case the widget read is part of a group, the reasoning is similar to the portrait layout creation described above, where we verify if the horizontal linear layout is already created before adding the control. Whenever a new horizontal layout needs to be created the same test is performed to verify if the creation is on the left or in the right side.

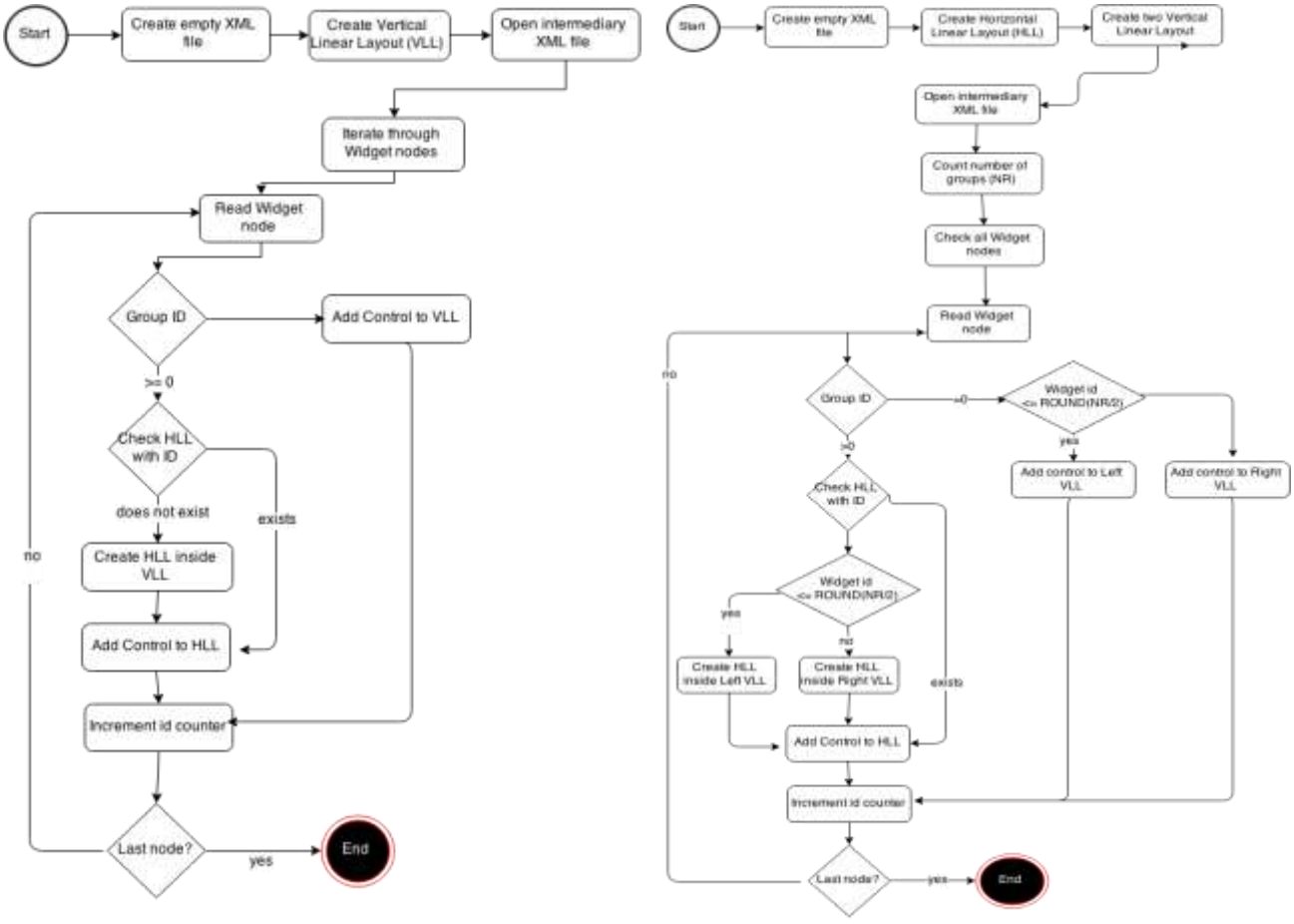

Figure 4. Flowchart for portrait (on the left) and landscape (on the right) creation

3.1.4 User interfaces

The above described design rules were applied in the desktop application and the layouts creation plugin. Next we present the main user interfaces of both desktop and plugin.

Desktop application

The main interface of the desktop application (shown in the left side of Figure 5) shows the supported controls which currently are ImageViews, TextViews, EditTexts, Buttons, RadioButtons, CheckBoxes and Switches (new ones can be added with minimum effort). In the left area the user types the code of the control he wants to insert (I for ImageView and so on) and insert a line break when he wants that same effect on the portrait interface. In the example provided below the user wants an interface that has an image (the icon launcher is used as default), below the image two TextViews (side by side), below them two EditTexts, below them a Spinner and below a Button.

The right side of Figure 5 shows the layout and JAVA files already generated in Android Studio project structure, namely *ActivityOne.JAVA, activity_activityone.xml* (portrait and landscape versions). It is also possible to notice the correct association being made in the *onCreate* method between the JAVA file and layout file.

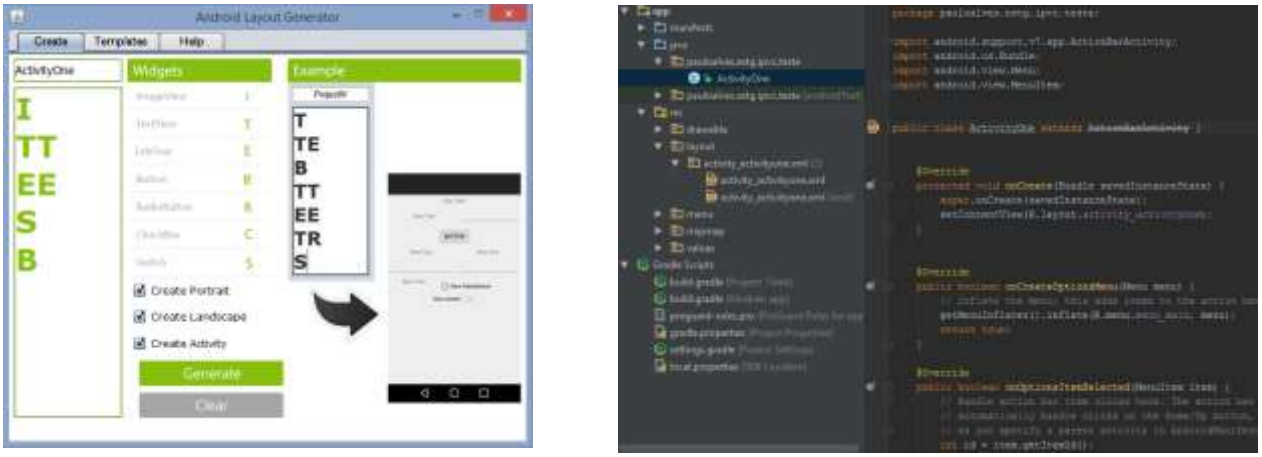

Figure 5. Desktop application user interfaces

Figure 6 shows the layouts generated from the above example.

Figure 6. Generated layouts example

Another feature of the desktop application is to provide templates (
Figure 7) so programmers can choose one that is suitable for a given situation and save time. Each template gives the programmer a starting point for their app without the need to input the widgets. The first template creates a group of widgets, mainly to be used for tests, for example, testing a new method. The second is similar to some popular apps in which there are posts and the users of that app can vote and comment on that same post. The third has some similarities to some social networks. Note that none of the images displayed are imported, instead is shown the default android icon with specific dimensions. To import the templates to the project the desktop application creates new layouts and copies

the template XML code to them. After that it creates a new activity with the name Temp1, Temp2 or Temp3 according to the template selected.

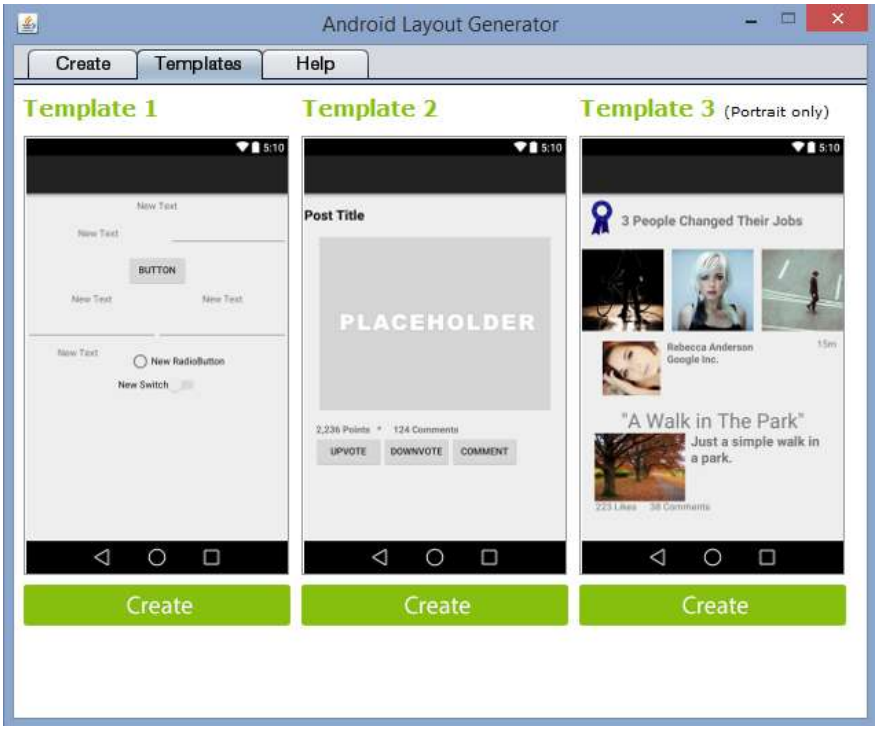

Figure 7. Desktop Aplication Templates

Plugin

The plugin has a similar behavior to the desktop application but the templates are not included. Figure 8 shows in the left side an image of the interface that allows to enter the widgets the programmer wants in the final layout. The center image shows the correspondent portrait layout and the right image the landscape version of the layout.

Figure 8. Plugin user interface and examples

3.3 Synchronization Layouts Plugin

The synchronization layouts plugin has a slightly different purpose from the desktop application and the other plugin already introduced. The synchronization plugin is supposed to help programmers that have already defined a portrait

version of the layout (graphically or using XML inside Android Studio) to automatically generate its landscape version or vice-versa. With a simple click in the plugin button, three situations can happen:

- If by any chance there are no layout files at all, an error message will be displayed informing the user that nothing can be executed. In this case it is advised to use the first plugin or the desktop application to create a new layout.
- If a layout directory exists, the plugin checks each file and verifies if the correspondent landscape or portrait exists.
- If it doesn´t, design rules are applied to transform portrait into landscape and vice-versa.

3.3.1 Workflows

The main purpose of this plugin (functionalities indicated in Figure 9) is to automatically detect which layouts exists in the current project and for each one verify if the other orientation exists. If it doesn´t it creates the correspondent folder and created the layout file that is missing, based on a set of design rules.

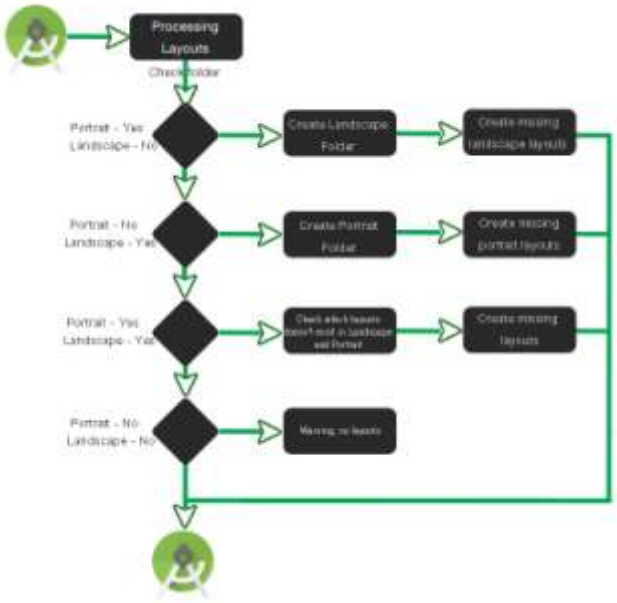

Figure 9. Structure for Synchronization Layouts Plugin

3.3.1 Design rules

In order to convert from either portrait to landscape or from landscape to portrait, a set of design rules were defined, different from the ones defined for the desktop and the first plugin.
In order to convert a layout from portrait to landscape, the plugin starts by counting all the widgets inside the portrait layout. Linear Layouts or any other type of layout counts as one. After having obtained the number of widgets (*nw*) we verify if that number is even. If it is, its divided by two (*nw/2*); if is odd, we increment 1 to *nw* and then divide it by two (*(nw+1)/2*). The value obtained will be referenced as (*hw*).
Next, an empty XML that will represent the landscape version is created. To that file we add a root element (of type Linear Layout Horizontal) and inside it we add two Vertical Linear Layouts (Left and Right). Next, we iterate through the widgets in the portrait version. Until we reach *hw*, widgets are added to the Left Vertical Linear Layout. The remaining widgets are added to the Right Linear Layout.
In the end of this process, the landscape version is created.
The conversion from landscape to portrait assumes the existence of Left and Right containers. We first iterate through widgets of the Left container and add them to portrait file and then we add the ones from the Right Container.

4. EVALUATION

4. 1 Layouts creation example

Based on the input presented in Figure 5, the first step of the desktop application generation process creates an intermediary file with the content presented in the left side of Figure 10. The first line of the input is an image and therefore the tag Type of the first widget has the text Image. The group is zero because the image is the only control in that line. Next there are two TextView's belonging to group with id equals to one (this number is sequential). The third

line of the input is an EdiText which has also group 0 in the intermediary file. Finally, there are three buttons in the same line with group id equals to 2. Figure 10 also presents the portrait and landscape layouts generated from the same intermediary file, applying the design rules described above. We can notice the four vertical groups of controls in the portrait layout, where some of them have only one control (groups 1 and 3) and other have several controls (groups 2 and 4). In landscape mode, the same controls are presented differently. As there are four groups, there are two in each vertical layout. The left vertical layout has two horizontal linear layouts for the first two groups and the right vertical layout has two other horizontal linear layouts for the other groups.

```
<?xml version="1.0" encoding="UTF-
8"?>
<Layout>
<Widget><Type>Image</Type>
        <Group>0</Group></Widget
>
<Widget><Type>TextView</Type>
        <Group>1</Group></Widget
>
<Widget><Type>TextView</Type>
        <Group>1</Group></Widget
>
<Widget><Type>EditText</Type>
        <Group>0</Group></Widget
>
<Widget><Type>Button</Type>
        <Group>2</Group></Widget
>
<Widget><Type>Button</Type>
        <Group>2</Group></Widget
>
<Widget><Type>Button</Type>
        <Group>2</Group></Widget
>
</Layout>
```

Figure 10. Intermediary file contents with the correspondent portrait and landscape layouts generated

Another generation example is shown in Figure 11. In this case we demonstrate how the tool is versatile and can be used to generate several types of layouts including those similar to well known applications such as LinkedIn or 9gag. The example shows the layout firstly created by the desktop application and after edition in Android Studio mostly replacing images and changing the text content and style.

The first layout represents a first draft built using the desktop application, simply specifying the input and with minimum effort for the programmer. The second layout shows how simply changing the images and texts a different look can be achieved, in this case to build an application similar to LinkedIn. The third and fourth layouts are an example of building layouts for an application such as 9gag, first with the draft from the desktop application and then after editing it in Android Studio accordingly to the intended purpose.

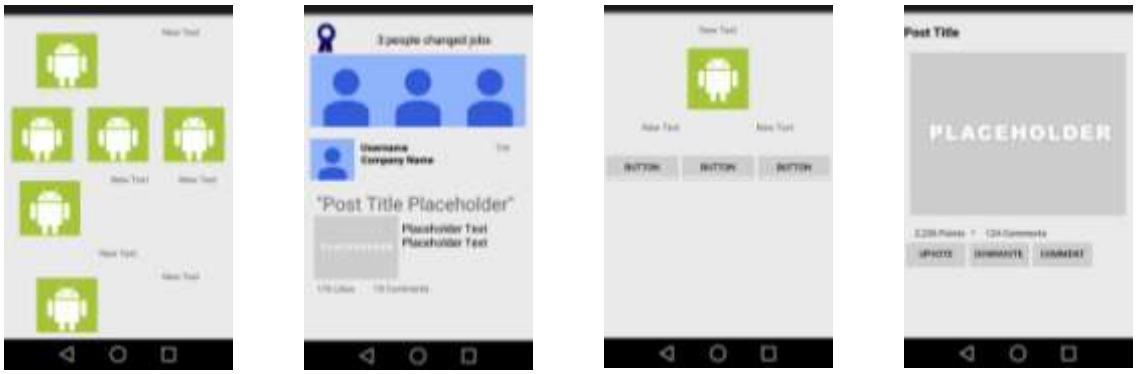

Figure 11. Creation of layout of LinkedIn and 9gag

Regarding the execution of the plugin, the results are pretty much the same shown for the desktop application as the design rules are the same. The layouts shown in Figure 10 can be provided as an example. When executing the plugin in an Android Studio project only with the portrait layout defined, the intermediary file representation is created with the shown content and the landscape layout created as also shown. The process is valid and similar for an Android Studio project with only the landscape layout defined.

4. 2 Layouts creation times

To measure the time a layout takes to be constructed is not an easy task as it depends on quite a few factors. Still, we conducted two tests, with two different layouts, where a group of 8 people built layouts and correspondent activities using 1) the desktop app and 2) manually. We then compared the average times needed to finish each construction.

Test #1
The portrait and landscape layouts to be created for conducting test number 1, as well as correspondent activities, are as illustrated in Figure 12.

Figure 12. Layouts to perform test #1

The gathered times are represented in Table 2 and show a significative reduction time in layouts construction when using the proposed solutions introduced in this work.

Table 2. Measured times for test #1

Method	Time Necessary	
	Portrait	*Landscape*
Using the Desktop Application / Plugin	2m 21s	2m 56s
Manually	5m 53s	3m 32s
Reduction time with proposed solutions	60%	23%

Test #2
The portrait and landscape layouts to be created to perform test number two, as well as correspondent activities, are as illustrated in
Figure 13.

Figure 13. Layouts to perform test 2

The gathered times are represented in Table 3 and also show a significative reduction time in layouts construction when using the proposed solutions introduced in this work.

Table 3. Measured times for test #2

Method	Time Necessary	
	Portrait	*Landscape*
Using the Desktop Application / Plugin	0m 46s	1m 37s
Manually	0m 54s	2m 33s
Reduction time with proposed solutions	15%	41%

5. FUTURE WORK AND CONCLUSIONS

In this paper we present a solution for minimizing the effort and time spent on creating Android layouts with support for multiple orientation. Creating apps with this support is an important feature as allows final users to have a better experience taking the most out of the space of the device depending on the orientation being used. As Android Studio layout creation tool is more time consuming than for example Visual Studio or XCode, software to help in the automatically generation becomes necessary.

The developed solutions include a desktop application that allows specifying the controls for the interface and with a simple button click create two layouts, one for portrait and other for landscape, as well as necessary JAVA code and classes. The desktop application also allows to import pre-defined layout templates for the programmer to be more efficient in this stage of the development process.

Two Android Studio plugins were also developed. The first one is very similar to the desktop application (without the templates component for now) and its main purpose it to allow the programmer to use solely one IDE. Both solutions – desktop application and plugin – generate layout files based on design layout rules. For the portrait layout we basically use one vertical horizontal layout with several horizontal linear layouts inside where each one of these layouts have a group of controls. For the landscape layout we use a horizontal linear layout with two vertical linear layout inside splitting the groups of controls for the two vertical linear layouts so we use all the space of this orientation.

The other plugin aims to verify the existent layouts in a given project and generate the missing ones, regarding orientation.

The layouts support the following controls at the moment: ImageView, TextView, EditText, Button, RadioButton, CheckBox or Switch. We chose these containers and controls after analyzing top applications such as LinkedIn, Instagram or 9gag that are possible to build with the mentioned widgets.

In this paper we also presented some examples of applications the developed solutions can produce and how they can help programmers to be quicker in the first draft of layout development.

For future work we will incorporate the notion of priority of a control so a programmer can specify the most relevant controls of the layout. This information can be important so in landscape version we do not handle each group of controls the same way, giving more emphasis to those with higher priority. We will also consider the use of relative layouts and definitely incorporate more widgets.

REFERENCES

[1]. Alves, P., Lopes, P. & Paiva, S., 2015. Automatic Layout Code Generation for Android Applications with Multiple Orientation Support. In *IADIS International Conference Interfaces and Human Computer Interaction*. Las Palmas, Gran Canaria.

[2]. GoogleDeveloper, 2015a. Fragments. Available at: http://developer.android.com/guide/components/fragments.html.

[3]. GoogleDeveloper, 2015b. Supporting Multiple Screens. Available at: http://developer.android.com/guide/practices/screens_support.html.

[4]. Hu, W. & Zhang, K., 2014. Research and Implementation of Android Embedded Code Generation Method based on Rule Model. *International Journal of Multimedia & Ubiquitous Engineering*, 9(11), pp.273–282. Available at: http://www.sersc.org/journals/IJMUE/vol9_no11_2014/27.pdf [Accessed April 28, 2015].

[5]. IDC, 2015. Smartphone OS Market Share, Q4 2014. Available at: http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[6]. Sahami Shirazi, A. et al., 2013. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. *5th ACM SIGCHI Symposium on Engineering interactive computing systems*, pp.275–284. Available at: http://dl.acm.org/citation.cfm?doid=2494603.2480308.
[7]. Statista, 2015. Number of apps available in leading app stores as of July 2014. Available at: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.
[8]. Wen-zhen, C., Wei-feng, B. & Ya-wei, Z., 2014. Porting LCD Drive Based on Android OS. *Instrumentation Technology*, 14(16).