Unveiling the Architecture and Design of Android Applications
An Exploratory Study

Edmilson Campos[1,2], Uirá Kulesza[1], Roberta Coelho[1], Rodrigo Bonifácio[3] and Lucas Mariano[1,2]
[1]*Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, Natal-RN, Brazil*
[2]*Federal Institute of Education, Science and Technology of Rio Grande do Norte, Natal-RN, Brazil*
[3]*Computer Science Department, University of Brasilia, Brasilia, Brazil*

Keywords: Mobile Applications, Android's Applications and Design Pattern.

Abstract: This work presents an exploratory study whose goal was to investigate the architectural characteristics of Android's applications. We selected twelve popular and open-source applications available on the official Android's store for analysing. Then, we applied techniques of the reverse engineering to each target application in order to investigate three main aspects: (i) architecture of each application; use of the (ii) design patterns; and (iii) expecting handling policies. Support tools were used in order to identify dependencies between architectural components implemented in each target application, and to graphically present those dependencies. Then, based on this analysing, we present a qualitative analysis carried out on the extracted architectures. One of the outcomes consistently detected during this study was an overview of the main architectural choices that have been adopted by Android developers, resulting on formulation of a preliminary conceptual architecture for Android applications.

1 INTRODUCTION

Over the past few years, there was a rapid growth in the usage and demand of mobile applications development. According to recent data survey from the Worldwide Quarterly Mobile Phone Tracker (IDC, 2014) more than 1 billion smartphones were sold worldwide in 2013 (Caputo, 2014). The survey also showed that 78.6% of such devices used the Android as operating system, which represents 793.6 million units sold in that year, an increase of 58.7% compared to sales of the same system in 2012.

In a similar rate, there is an increasing demand for mobile applications (usually called as apps) to address the needs of such new users. The number of applications available for download only on Google Play Store – the Android official store – has grown to over 1 million applications in the first semester of 2014 (Caputo, 2014). However, despite the growing number of applications developed daily, there is a lack of studies about the architecture and design strategies adopted on such mobile applications. Recent research works (Ruiz et al., 2012; Mojica et al., 2013; Linares-Vásquez et al., 2014; Linares-Vásquez et al., 2013; Bavota et al., 2014) focused on implementation characteristics of such applications. For instance, Ruiz et al., (2012) and Mojica et al., (2013) analysed the degree of code reuse in such applications; Linares-Vásquez et al., (2014) studied the source code idioms associated with the energy consumption of such applications; and other research works (Linares-Vásquez et al., 2013; Bavota et al., 2014) reported studies analysing the fault and change-proneness of APIs used by Android applications. However, the existing research works do not explored how Android mobile applications have been designed and whether it is possible to derive architecture styles or guidelines that can be reused for other applications form the same domain.

In this context, this paper presents an exploratory study that investigates how Android applications have been designed and implemented, and in particular, we investigate the decisions related to the implementation of the exception handling concern. Our study analyses twelve open-source applications (each of which contains on average 1/2 million downloads in the Play Store) from different categories. To identify which architectural and design patterns have been adopted in the target applications and how the exception handling concern have been implemented, we carried out a static analysis of the applications' source code using both JDepend (Clark,

Campos E., Kulesza U., Coelho R., Bonifácio R. and Mariano L..
Unveiling the Architecture and Design of Android Applications - An Exploratory Study.
DOI: 10.5220/0005398902010211
In *Proceedings of the 17th International Conference on Enterprise Information Systems* (ICEIS-2015), pages 201-211
ISBN: 978-989-758-097-0

2012) and Graphviz (Ellson et al., 2002) tools.

The remainder of this paper is organized as follows. Section 2 explains the study settings. Section 3 presents the results of the exploratory study realized, and Section 4 summarizes them. Section 5 discuss existing related research work. Finally, Section 6 presents the conclusions and possible future works.

2 STUDY DESIGN

This section details the methodology used in the exploratory study. Our study involved the analysis of open-source applications developed for the Android platform, for analysing architectural and design patterns that have been adopted by their developers.

2.1 Study Aim and Research Questions

This exploratory study aims at analysing open-source Android applications (hereafter target systems) to identify the architectural patterns and solutions, which have been adopted by developers. In particular, the exploratory study was developed in order to answer the following research questions (RQs):

RQ1. Which architectural patterns have been adopted in the Android applications, and how they have implemented the components of their architecture?

RQ2. Which design patterns have been adopted and implemented in the Android applications?

RQ3. How is the exception handling concern implemented in the Android applications?

2.2 Selection of the Applications

One of the requirements for the selection of the target systems for our study is that they must be available on an open source repository. The selection of such applications has been performed manually by random sampling of the major applications available at the official Android store which were also: (i) available as an open source project, (ii) a popular application, with a minimum of 1/2 million of downloads at the official store application; and/or (iii) had professional character, or as official applications developed enterprise solutions.

Table 1 presents the selected applications for this study, showing their names, corresponding categories, and the number of installations of each one. All data were extracted directly from the official store of the Android platform.

Table 1: Analysed applications.

#	Category	App	N. Downloads
01	Books and references	Wikipedia	10 ~ 50 MM
02		iFixit	0,5 ~ 1 MM
03		FBReader	10 ~ 50 MM
04	Communication	Firefox	50 ~ 100 MM
05		ZapZap	1 ~5 MM
06		ConnectBot	1 ~ 5 MM
07	Game	Frozen Bubble	1 ~ 5 MM
08		OpenSudoku	1 ~ 5 MM
09		Freeciv	0,1 ~ 0,5 MM
10	General	Wordpress	1 ~ 5 MM
11		c:geo	1 ~ 5 MM
12		My Tracks	10 ~ 50 MM

2.3 Analysis Procedures

In order to answer our research questions, we have conducted an architecture reconstruction of the selected applications using the JDepend (Clark, 2012) and Graphviz (Ellson et al., 2002) tools. JDepend was used to identify dependencies between architectural components implemented in each selected application, while Graphviz was used to show graphically those dependencies. After that, we have read and investigated the source code of the applications to study how specific components, design patterns, and the exception handling policies have been implemented.

The analysis of architecture, design and code of the Android applications were guided by the systematic identification of specific issues, which are listed below:

(i) Architecture of Applications (RQ1):

- Identification and analysis of the architecture style (MVC, Layers, etc.) adopted by application;
- Identification and analysis of how the main architecture components interact and are implemented.

(ii) Design Patterns (RQ2):

- Identify and analysis of the official solution or specific to Android platform;
- Identify and analysis of which (and with what purpose) design patterns are implemented.

(iii) Exception Handling (RQ3):

- Identify which one policy exception handling has been adopted by such applications.

3 MAIN STUDY RESULTS

This section presents and discusses the main results of the study. They are presented according to the

criteria of analysis adopted in our study.

3.1 Architecture of the Android Applications (RQ1)

This section details the obtained results during the analysis of the architecture of Android mobile applications.

3.1.1 Adoption of the MVC in Android

The Model-View-Controller (MVC) (Gamma et al., 1994) pattern organises a given software application into three interconnected parts that separate internal representations of information (model) from the ways that information is presented to (view) or accepted (controller) from the user. MVC is one of most popular software architectural pattern. Nevertheless, although Android applications do not need necessarily be based on the MVC, in our study we observed that most of the selected applications adopt the MVC pattern by structuring their architecture using the pattern components. We also analysed existing dependencies between the MVC components from the reverse engineering accomplished using the JDepend (Clark, 2012) and Graphviz (Ellson et al., 2002) tools. During this analysis, we identified that most of existing applications follow the MVC pattern, but we also found some architectural violations to the pattern. Next we present and discuss such results.

a) Adoption of the MVC Pattern

Table 2: Implementation of the MVC Components.

App	MVC Components						
	Model			View		Controller	
	Entity	Service	Data	XML	Java	Activity	Fragment
#01	--	--	--	--	--	--	--
#02	X	X	--	X	X	X	X
#03	X	X	X	X	X	X	X
#04	X	X	X	X	X	X	X
#05	X	X	X	X	X	X	X
#06	--	X	--	X	X	X	--
#07	X	X	--	X	--	X	--
#08	--	--	X	X	--	X	--
#09	--	--	--	X	--	X	--
#10	X	--	X	X	X	X	X
#11	X	--	X	X	--	X	X
#12	X	X	X	X	--	X	X

Table 2 shows that the MVC pattern was adopted by most of the investigated applications, although we found some variations in the way that the pattern components were implemented. Since XML files are used to implement the graphical user interface in Android applications, most of the applications tend to have at least the view component clearly defined and separated from the controller classes. In our study, we only found some Android applications, where the model and controller components were not clearly modularized.

Figure 1 presents an overview of the **WordPress** (#10) architecture, which illustrates a full adoption of the MVC pattern. It shows an XML file responsible for creating a link in a post that implements the view component. The visual elements (i.e. *listings* and *buttons*) of this XML file are captured by a controller class (`EditLinkActivity`). This class calls another controller class (`EditPostContentFragment`), which interacts with the model (`Post` class). Finally, this model class notifies changes for the view, communicating with the Java view class that customizes the XML file. Next we discuss how each MVC component has been implemented for the investigated applications.

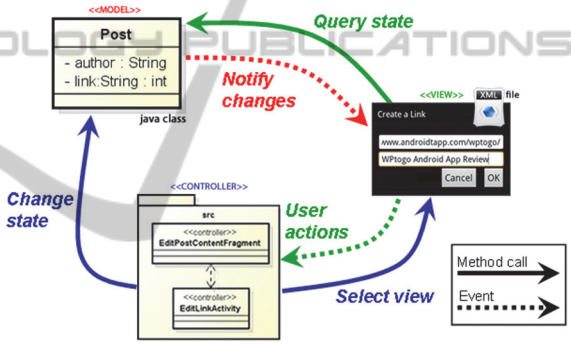


Figure 1: MVC structure of the WordPress app.

Model Implementation. In our study, we found that the Model component was implemented using one or more of the following modules - Entity, Service and Data. The Entity module is responsible to implement the domain classes. The Service contributes to expose the services provided by the model component. Finally, the Data classes provides implementation for data storage of the domain classes. As we can see in Table 2, most of applications have implemented the Entity or the Service modules. The only exception is the **OpenSudoku** (#08), which implemented only the data classes in the model component.

View Implementation. In the Android applications, the view component is usually implemented by XML files. These view components contain Android API default visual elements (e.g. *button*, *listview*, etc.) that are accessed by Java controller classes, which are responsible for manipulating model classes and updating the XML views. This XML implementation was adopted by almost all application (see Table 2),

except the **Wikipedia** (#01), that not adopted MVC pattern (see topic about non-adoption). Developers can also customize their own visual elements creating custom views (Java classes that extend the View class), which can be more robust and reusable. Tabl shows, for example, that the **Firefox** (#04), **ZapZap** (#05), and other four applications customized Java classes that inherits from the View class, in order to support the creation of their own view components. On the other hand, **WordPress** (#10) mainly used this alternative of customized view implementation view, with almost no XML based codification.

Controller Implementation. Finally, the Controller component presents the more uniform implementation among others. It is developed using activities and/or fragments classes from the Android platform. All investigated applications (see Table 2) implemented activities classes, while fragments classes were only implemented by application that adopted the Multi-Panel Layout pattern (see section about Android specific pattern). Both activities and fragment classes access elements of the XML files to provide data to the graphical user interface with which users can interact when running the application. Because XML element (View) cannot directly access classes of the model component, it is need to find a new way to recover/transfer data between the view and the model components. Thus, the controller classes are as bridge between these components. Figure 2 shows a code fragment of **iFixit** (#02) architecture that exemplifies how this communication occurs. It shows the structure of file "guide_create.xml" with its visual elements. In the example, the `GuideCreateActivity` controller class accesses a *listview* element of the interface passing its id, by class `R`, for method `findViewById()`. In Android, the data provided in the visual elements (*buttons*, *textfields*, etc.) specified in the XML files of the view component can be recovered through the activity and fragment classes of the controller component using the class `R` (resource). For example, Figure 2 shows also the `GuideCreateActivity` controller class accessing the `Guide` model class to recover/transfer data for the view component.

b) Non-adoption of the MVC Pattern

Despite most applications have adopted the MVC pattern, not all architectures of the applications adhere to the MVC pattern, like **Wikipedia** (#01) and **Freeciv** (#09).

Wikipedia (#01) was the only target applications that did not implement any of these types of classes or any model component. It has adopted a specific framework – PhoneGab (PhoneGab, 2014) – for building its graphical user interface (GUI) and enable the code generation of the view component. It was also the only selected application that used web artefacts (HTML and CSS) to implement its GUI, instead of using XML files.

Freeciv (#09) is an untypical case where the application structure consists of a set of monolithic classes. These classes group almost all project code in a few classes, thus this application does not present a well-modularized architecture.

c) Violations of the MVC Pattern

During our analysis of the dependency graphs of the MVC components generated by JDepend and Graphviz tools, we also found some violations to the MVC pattern. Although most of these violations do not represent a clear threat to the evolution of the architecture of the existing applications, they need to be monitored in order to avoid future maintenance difficulties. Next we present and discuss some of them.

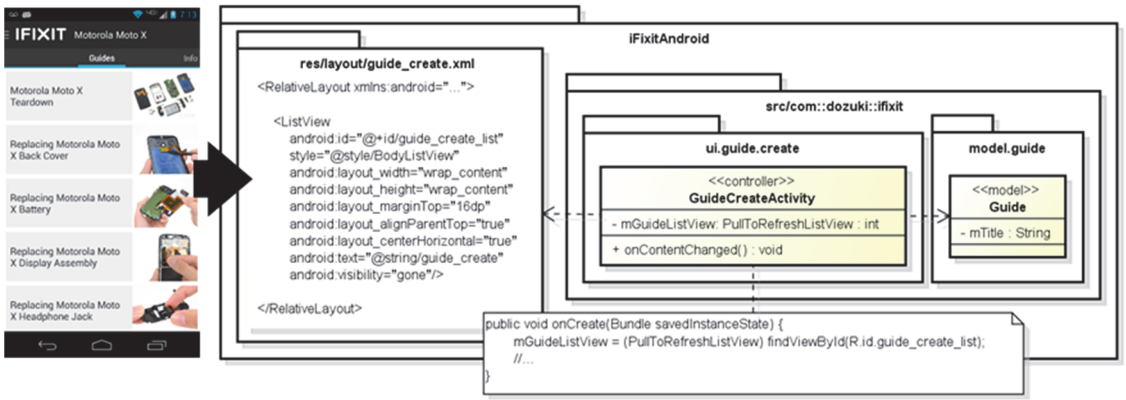


Figure 2: Fragment of the iFixit architecture.

Model-to-View Violation. The first found scenario was a violation of the independence of the model component, where model classes have direct calls to classes of the View component. Figure 3 illustrates an example of this violation in the context of the **c:geo** (#11) application. It shows a code fragment that implements an example of a model class (`CGeoMap`) in application. The sample illustrates a typical violation between model and view components. The `CGeoMap` class (model) imports the `R` class in order to recover a visual element (view) aiming to update directly an information on the view.

```
package cgeo.geocaching.maps;
//..
import cgeo.geocaching.R;
public class CGeoMap extends AbstractMap{
 private String mapTitle;
 private void setTitle (String title) {
 final TextView titleview =
 ButterKnife.findById(activity,
 R.id.actionbar_title);
 if (titleview != null) {
 titleview.setText(title);
 }
 //...
 }
 //...
}
```

Figure 3: Model-to-View violation example.

Model-to-Controller Violation. Similar to the previous kind of violation, this second case implies the rupture of independence of the model component, since this component is dependent of the controller component. We have identified this kind of violation is some existing service classes of the model components from the **WordPress** (#10) and **ZapZap** (#05) applications. Figure 4 shows an example of this Model-to-Controller violation of the **ZapZap** (#05) application. Code fragment presents a Model class (`ScreenReceiver`) that implements a service referencing a Controller class (`ApplicationLoader`) in order to set the application screen. It is a violation due to Model class is referencing Controller class.

3.1.2 Data Access in Android

Most applications need to have some kind of data persistence. Android provides the data storage options following (Android, s.d.): (i) Shared Preferences — commonly used to save private value pairs of primitive data types; (ii) Internal Storage — when data are saved directly on the device's internal storage, using cache files; (iii) External Storage — store public data on the shared external storage; (iv) SQLite Databases — a self-contained database, compact, with native support in Android, which does not require special configuration or installation; and (v) Network Connection — data are stored on the web and the communication with the remote database often occurs through web services or remote address for the server.

```
package org.telegram.messenger;
//...
import org.telegram.ui.ApplicationLoader;

public class ScreenReceiver
    extends BroadcastReceiver {
 @Override
 public void onReceive(
    Context context, Intent intent) {
     if (intent.getAction()
 .equals(Intent.ACTION_SCREEN_OFF)){
 //...
 ApplicationLoader
     .isScreenOn = false;
 } else if (intent.getAction()
 .equals(Intent.ACTION_SCREEN_ON)){
 //...
 ApplicationLoader
     .isScreenOn = true;
 }
 }

}
//...
```

Figure 4: Model-to-Controller violation example.

The choice for a these solution depends of the specific needs of each application/device, such as performance, memory availability and practicality of the implementation. In our study, we analysed the storage options adopted in each application selected in order to identify common purpose between application that have adopted the same type of strategy. Table 3 shows a summary of the adoption these persistence alternatives analysed.

`SharedPreferences` was a strategy more adopted between the investigated applications. This was already somewhat expected because it is mainly used to save some app configuration. Moreover, this data storage strategy is very simple and allows saving only primitive data that are persistent across user sessions. `Cache` and `FileBackupHelper` are type of internal storage which is saved in files with access (by default) only to own application and that are removed when the user uninstalls the application. Cache file was specially, for example **Firefox** (#04), **ZapZap** (#05) and **c:geo** (#11) applications used to store other types of data that could not be saved with primitive variables, such as historic data. On the other hand, instances of `FileBackupHelper` class were used to realize backup of databases on own device internal storage.

Table 3: Implementation of the persistence type.

App	SQLite database		Shared preferences	Cache Files	FileBack-upHelper	Other strategy
	helper class	create method				
#01	-	-	X	X	-	Javascript persistence
#02	X	-	X	-	-	-
#03	-	X	-	X	-	-
#04	X	-	X	X	-	-
#05	-	-	X	X	-	-
#06	X	-	X	-	X	-
#07	-	-	X	-	-	-
#08	X	-	X	-	-	-
#09	-	-	-	-	-	Dropbox
#10	X	-	X	X	-	-
#11	-	X	X	X	X	-
#12	X	-	X	X	X	Google Drive

Moreover, we identified several applications using the SQLite, but varied the way to implement it. Some applications implemented using a helper class, such as **WordPress** (#10) **ConnectBoot** (#06) and **OpenSudoku** (#08), which is responsible mainly the version database control. Other applications, such as **FBReader** (#03) also used the SQLite library to store data in its projects, however without use the `SQLiteOpenHelper` class to create its SQLite database. Instead of this, they used the method `openOrCreateDatabase`. This method allows developer to create a database passing a name or path and application context as parameter directly in code. This non-official implementation is simpler that using the class `SQLiteOpenHelper`, however less recommended when the app realizes changes constantly in its databases.

Other solutions different from the previous presented were found on **Wikipedia** (#01), **Freeciv** (#09) and **My Tracks** (#12). In the first, Wikipedia, the persistence is realized in the web layer, directly in the Javascript files, since that this app was implemented using the framework PhoneGab (PhoneGab, 2014). The last applications (#09, #12) developers used the DropBox and Google Drive API, respectively, to synchronize data in files stored in cloud servers.

3.1.3 Package Structure

We also analysed the package organization of each selected application in order to identify whether the separation into architectural components as reflected in the package structure. This also brings positive influence to the maintenance and evolution of the applications. In addition, we were also interested in identifying packages/layers common to all applications investigated, aiming to extract and idealize a default structure for Android applications.

After this analysis, we observed that many applications grouped their classes based on the MVC components, creating a separated package for each architectural component (model, view or controller). On the other hand, some applications have grouped their classes according to the business concern. In this case, different classes of the same architectural component (e.g. model classes) are spread over multiple and different packages, according to any particular business criteria of the application.

Table 4: Package Structure of Android applications.

App	MVC components		Specific package for DAO layer	Main packs.
	single packg.	separated		
#01	--	--	No	`org.wikepedia` (single pack.)
#02	X	--	No	`ui`, `util`, `model`
#03	--	X	No	`core.view`, `view`, `model`, `util`, `network`
#04	--	X	Yes	`firefox`
#05	X	--	Yes	`objects`, `ui`, `SQLite`, `messenger`, and `PhoneFormat`
#06	X	--	No	`bean`, `util`, `service`, and `transport`
#07	N/a	N/a	No	`frozenbubble` (single pack.)
#08	N/a	N/a	Yes	`game`, `command`, `utils`, `gui`, and `db`
#09	N/a	N/a	No	`main`
#10	X	--	Yes	models, dataset, ui, util, xmlrpc
#11	--	X	Yes	activity, maps, ui, conector, utils
#12	X	--	Yes	Content, util, fragments, io, maps

A common characteristic of the package structure of most applications was the usage of a package "`util`" for utilities classes and methods. This kind of package was found in all the selected applications and has been used for different purposes. For example, app **OpenSudoku** (#06) has used the `util` package to organize the version code, on the other hand the app **ZapZap** (#05) defines the phone formatters in this package. Overall, the util package contains the helpers, parsers, formatters, classes with specific applications rules, among other auxiliary and utility class in general considering the analysed applications. Table 4 presents a summary of the analysis of the package structure realized, focusing on analysis of packages that implement the MVC components and the DAO layer.

3.2 Design Patterns (RQ2)

Our study also examined the use of specific design patterns for the Android platform (Android, s.d.), and traditional design patterns (Gamma et al., 1994). Next subsections detail the results for this pattern analysis.

3.2.1 Specific for the Android Platform

The Android Developer's Official Guide (Android, s.d.) provides some patterns to manage variabilities in the platform, such as different languages, screens (resolution and size), and platform's versions (Android, s.d.). These deployment patterns handle common problems in Android development and propose solutions whose implementation allows adaptations. Our study found that most of these solutions are actually implemented, albeit partially, by the developers of the analysed applications. Next we present some of the patterns identified in the study.

a) Multiple-View Layout

The Android's Guide proposes a design pattern known as *Multiple-View Layout* (Android, s.d.) to deal with different screen on devices. The pattern provides the implementation of XML files fragmented that are programmatically composed (at run time) to form the layout of the application. Each fragment can be reused and combined to assemble a composed adjusted view, confirming orientation (horizontal or vertical) and size of device screen. Figure 5 illustrates an example with different composite views from the implementation of this pattern in the **WordPress** (#10) application.

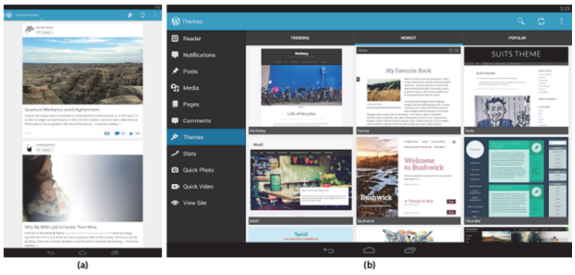


Figure 5: Example of the orientation type: (a) vertical and (b) horizontal.

The main roles in this pattern are:

- `Fragment` (`ArticleFragment`): Fragment represents a part of the screen that to be composed to create different views for the user. It defines its own lifecycle, receives its own input events, and allows add or remove it while the activity is running. It must extend the class of Android's API or similar class (when using external or compatibility API);
- `Manager` (`FragmentManager`): Responsible for performing an operation such as adding or removing a fragment. You must instantiate a `FragmentTransaction`, which provides APIs to add, remove, replace and perform other operations.

The **iFixit** application (#02) used an external library, known as Sherlock API, in order to implement fragments. This API enables, among other things, the execution of fragments on devices with pre-3.0 versions of Android, since Android API provides native support only for appliances from this version. However, there is also an official Android's Compatibility API, known as Support Library, which launched later. This API was used in 33,4% of the studded applications (see Table 5). In terms of implementation, the main change in each one of these classes occurs only in inherited classes for creating fragments.

Table 5: Implementation of fragments in apps.

	Apps	N. apps (%)
Android Native API	#10	1 app (8,3%)
Android Support Library	#04, #05, #11, #12	4 apps (33,4%)
Sherklock API	#02	1 app (8,3%)
Do not use fragments	all others	6 apps (50%)

Figure 6 presents a code fragment which implemented the Multi-Panel Layout pattern in the **ZapZap** (#05). In this example, if instead of using the native Android API to implement the pattern it had been used the Sherlock API, the class `BaseFragment` would be changed by the `SherLockFragment` class. presents the amount of applications that implemented fragments with information about the adopted API. Only half of the applications adopted this pattern. The other apps do not use any alternative implementation. We also found that the gain with the adoption of this pattern is directly related to the treatment of the variability of multiple screens and indirectly to reuse screens, which are composed to form different views.

b) Content Provider

Content providers (Android, s.d.) are roles of another official Android pattern, responsible for managing access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content provider is the design pattern that connects data in one process with code running in another process. In our study, this pattern was identified in the following applications: **Firefox** (#04), **WordPress** (#10) and **My Tracks** (#12).

WordPress (#10) uses a content provider for managing access to its data repository. This is an Android recommendation (Android, s.d.). A content provider might be implemented as one or more classes in an Android application, along with elements in the manifest file. In the example, the `StatsContentProvider` class extends a `ContentProvider` class, which connect your provider

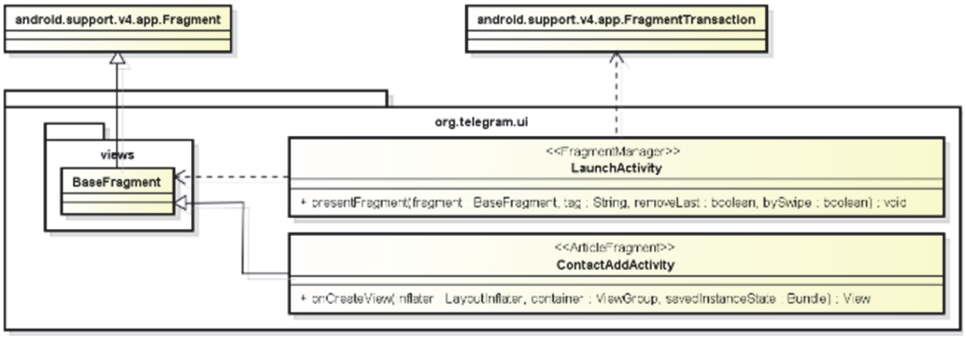


Figure 6: Multi-Panel Layouts on the ZapZap.

and other applications. Although this content provider is used to make data available to other applications, you may of course have activities in your app that allow the user to query and modify the data managed by a content provider.

Other official solutions have been adopted, such as the use of strings and resources to manage different languages and platform versions. These solutions was adopted by all target applications that need to deal with such kind of variabilities, behaving more like a common way to implement this type of problem than a design pattern.

3.2.2 Traditional Design Patterns

Our study also aims to identify the implementation of other patterns of traditional designs that have been adopted in the development of Android applications. In this context, we identified the use of some platform independent patterns, such as *Provider Model* (Howard, 2004), *Factory Method* (Gamma et al., 1994), and *Adapter* (Gamma et al., 1994).

Table 6: Traditional Design Patterns.

Patterns	Apps	Purpose
Provider Model	#02	Provide a common interface for setting of search autosuggestion feature
	#10	Provide a common interface for records in the application of statistical table
	#11	Provide common interfaces for setting plug-ins used to capture maps
	#12	Provide common interfaces for configuration of the search engine
Factory	#02	Implement a factory of connection (data and network)
	#03	
	#05	
	#06	Factory of transport protocol
	#08	Factory for importing using XML Parser
	#11	Factory of interfaces of maps
	#12	Factory to manage sensors
Adapter	#02	Extend the package "`android.widget`" through of the interfaces `BaseAdapter` and `ArrayAdapter` to customize the visual components of the app, such as its `widget` and `ListView`
	#04	
	#05	
	#11	
	#12	Deal with different APIs

Table 6 presents an overview of these patterns with the respective applications that have adopted them and purpose.

Other patterns identified: Singleton (#02, #07), Facade (#07), Strategy (#07, #12), Command (#02, #08), Publish-Subscribe (#02), and Observer (#08, #12).

3.3 Exception Handling (RQ3)

Another aspect investigated in our study was how the exception handling concern was implemented for such existing applications. We have identified and investigated manually: (i) the amount of handlers (try/catch) and signallers (throws) present in the code of each application; (ii) which components more commonly thrown and handle exceptions; and (iii) what is the main type of exception handling performed inside the exceptions catches.

Table 7: Among of handlers and signallers.

	handlers (try/catch)					signallers (throws)			
	% by component			∑		% by component			∑
	M	**V**	**C**	**try**	**catch**	**M**	**V**	**C**	**throws**
#07	57%	0%	43%	7	6	0%	0%	0%	0
#09	50%	0%	50%	12	17	100%	0%	0%	1
#01	0%	0%	100%	8	9	10%	0%	90%	1
#08	27%	0%	73%	22	21	10%	0%	90%	10
#02	83%	0%	17%	63	64	99%	0%	1%	85
#10	54%	14%	32%	159	182	85%	2%	14%	59
#03	80%	0%	20%	305	286	83%	0%	17%	374
#11	59%	10%	32%	304	284	83%	17%	0%	41
#05	52%	10%	38%	389	390	93%	5%	2%	61
#06	69%	0%	31%	80	81	98%	0%	2%	44
	53%	**3%**	**44%**			**66%**	**2%**	**22%**	

Table 7 presents the amount of handlers and signallers identified, as well as the percentage of occurrence by component in each application analysed. Based on these results, we can see more predominance of exceptions handlers and signallers in controller and model components, with the latter component being mainly used to throw exceptions (66%) and the controller component for handling

them (53%). The low number of exception handling in the view layer is justified by the use of XML files without access to the Java classes that implement this component.

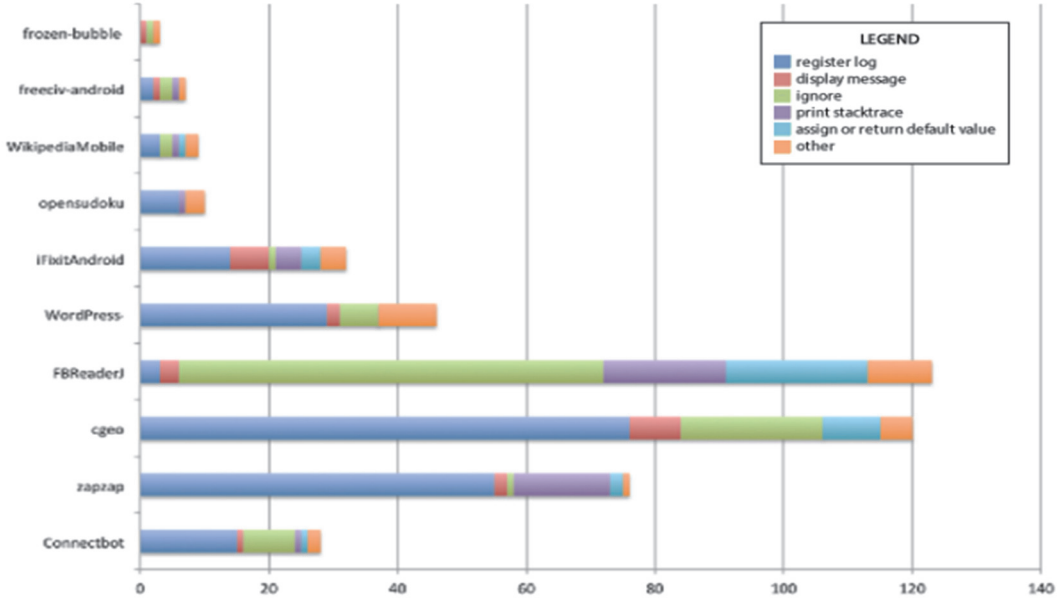
Moreover, our study also analysed the specific strategy used for handling the exceptions of the applications. The following strategies were found in our analysis: (i) registration of the exception in log file; (ii) display of error or warning message to the user; (iii) printing stack trace; (iv) no action (ignore the treatment - only capture); (v) returns default value (blank, null or false); (vi) among others. Graphic 1 bellow presents a summary of this analysis. As we can see, there is a great number of exceptions that are only logged (blue color) or are not adequately manipulated being ignored (green color). Only a few exceptions of each application are really displayed to the user (red color), which can mean that most of mobile application errors are not exposed to them.

4 DISCUSSIONS AND LESSONS LEARNED

In this section, we present and discuss some preliminary lessons learned from our exploratory study.

Architecture of Applications (RQ1). In our study, we have found that most of the analysed Android applications are structured by following the MVC architectural pattern. However, there are many applications where the relationships between the MVC components do not follow the traditional ones. This usually happens due to the implementation views on Android platform using XML files, which prevents access to Java classes from the View component of the MVC, thereby requiring reverse order communication between the components in relation to this. There are also cases where we found architectural violations in the relationships of the MVC components, which can impair the adequate evolution and maintenance of such Android applications. As a result, our study has revealed that even open-source and popular Android applications are not adequately using the MVC architectural pattern when structuring their classes, which requires: (i) the definition of specific implementation strategies for the MVC in Android platform; and (ii) the usage of automated supported tools (e.g., static analysis) to detect violations of existing applications.

Data Persistence (RQ1). Regarding the data component, we have observed that most of studied applications define data access classes, but they adopt different strategies for the implementation of such component. Based on our results, we can infer that: (a) shared preferences was a strategy more adopted to storage short data, such as configuration information. (b) internal storage was implemented using cache and backup files preferably in order to store private values with larger size, such as navigation historic. (c) SQLite has been used as main database for several of investigated applications, but varies its implementation. The recommend implementation by Android, which provides the implementation of the



Graphic 1: Exception Handling Strategies.

helper class, was also more adopted by applications with constantly evolving their databases.

Design Patterns (RQ2). Our study made possible to identify a significant number of design patterns that are used in the Android applications. The Adapter pattern, for example, was commonly used to adapt existing Android classes to create customized views. On the other hand, regarding the Android specific patterns, we have noticed that there are many of them that are not being used by existing applications, although they are recommended as best practices by the Android community. Our results also lead to the conclusion that many Android patterns were used in order to implement known variabilities, such as languages, devices and screen size. Moreover, it also varies the implementation them, with some of them using external API.

Exception Handling (RQ3). Our study also identified that existing exception handling policies of Android apps are absent or very simple – with only the logging of the thrown exception. Only a few applications provide an adequate exception handling by displaying, for example, explicit messages for the user. Our findings also show that there is no pattern on the way the exceptions are thrown and handled by the MVC components. The study data revealed, for example, that the model and controller components are both responsible to throw and handle exceptions for most of the investigated applications.

Figure 7 presents a diagram with the resulting conceptual architecture of our study. It illustrates the common MVC architecture adopted for most applications by indicating the explicit Android technologies used in each of the components, including the data objects. In addition, it gives an overview of the design patterns used in different components. Finally, it indicates which specific components of the architecture are usually responsible to throw and handle exceptions during the execution of the Android applications.

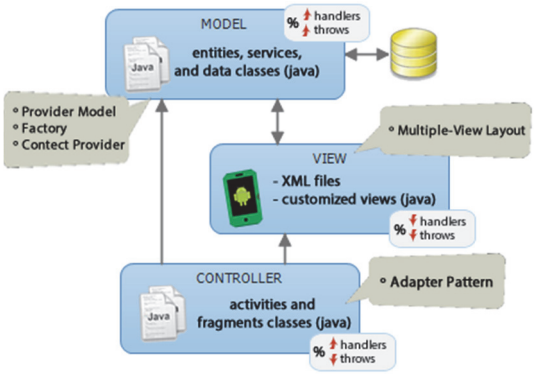


Figure 7: Conceptual Android Architectural.

5 RELATED WORK

This section presents some research works related to ours. Andreou et al., (2005) performed a pioneer study on design of the mobile applications. This work studied current design methodologies and proposed an approach for designing and developing mobile commerce (m-commerce) services and applications. The proposed process focuses on user requirements and needs as well as on constraints associated with current mobile and wireless technology. While this work proposed a set of engineering phases to guide mobile engineers in m-commerce development, our work explores the state of practice of a set of mobile applications from different categories available in Android official store. In another pioneer work about mobile applications, Nilsson (2009) performed a study which investigated the usage of design patterns for mobile applications. However, his work focused on patterns for user interface (UI), aiming at creating a design guideline to aid developing more user-friendly applications on mobile devices in general. This work did not restrict the analysis to Android applications. In a more recent work, Neil (2014) presented a more extensive catalogue of UI patterns for mobile devices. In our work, we investigated the adoption of design patterns in mobile applications, and not UI patterns as the works described before.

Other research works (Ruiz et al., 2012; Mojica et al., 2013; Linares-Vásquez et al., 2014; Linares-Vásquez et al., 2013; Bavota et al., 2014) also focused on the analysis of characteristics of Android applications. Linares-Vásquez et al., (2014) investigate the usage of patterns in Android applications focusing on API for energy-greedy, with the purpose of understanding particular instances of API calls and API usage patterns that cause (unusually) high energy consumption. Ruiz et al., (2012) and Mojica et al., (2013) mined application downloaded directly Play Store in order to analyse software reuse in the Android mobile application market along two dimensions: (a) reuse by inheritance, and (b) class reuse. On other hand, Bavota et al., (2014) and Linares-Vásquez et al., (2014) also analysed Android applications, but in order to study how the fault and change-proneness of APIs used by free Android applications relates to applications' lack of success, estimated from user ratings. As in our work, those works investigated a limited set of applications, although the studies cannot be generalized they provide interesting insights about the state of practice on mobile development. The purpose of our study differs from the previously mentioned studies, since it focused on

identification of architectural characteristics, such as design pattern adoption and exception handling structuring.

6 CONCLUSIONS

This paper presented the results of an exploratory study which aimed at identifying common architectural characteristics of a set of twelve real Android applications. During this study, we could observe that Android development does not need necessarily be based on the MVC or any other model. The choice of architecture model in Android development should be a consequence of the developer experience. One of the main contributions of this study was to perform a qualitative analysis on the extracted architectures. The results of the analysis were divided into three main groups discussions: (i) architectural analysis; (ii) use of platform independent design patterns or design patterns specific for the Android platform; and (iii) exception handling policy adopted. The architectural analysis shown that MVC pattern was adopted by most applications, even partially. Furthermore, we observed the adoption of known design patterns and we identified that more components handlers and thrown exceptions. Finally, based on analyses of these results, we created a conceptual architectural diagram that synthesizes the main findings of our study. The diagram is based on the MVC architecture indicating the main classes' relationship, patterns and exception handling strategies adopted by each component.

As possible future work, we intend to extend the studies on Android applications, refining the search engines and selection of applications and automating the analysis. Furthermore, we intend to deepen our study about solutions adopted by Android platform developers in order to investigate also types of variability in this application scenario.

REFERENCES

Andreou, A. S. et al., 2005. Key issues for the design and development of mobile commerce services and applications. *Journal International Journal of Mobile Communications,* December, 3(3), pp. 303-323.

Android, s.d. *Design Patterns.* [Online] Available at: https://developer.android.com/design/patterns/

Android, s.d. *The Developer's Guide.* [Online] Available at: https://developer.android.com/guide/

Bavota, G. et al., 2014. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering (TSE).*

Caputo, V., 2014. *Android e iPhone foram 93,8% dos aparelhos vendidos em 2013.* [Online] Available at: http://exame.abril.com.br/tecnologia/noticias/android-e-iphone-foram-93-8-dos-aparelhos-vendidos-em-2013 [Acesso em 29 Julho 2014].

Clark, M., 2012. *JDepedn homepage,* s.l.: s.n.

Ellson, J. et al., 2002. *Graphviz— Open Source Graph Drawing Tools.* Vienna, Springer Berlin Heidelberg, pp. 483-484.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* New York: Addison-Wesley Professional.

Howard, R., 2004. *Provider Model Design Pattern and Specification, Part 1.* [Online] Available at: http://msdn.microsoft.com/en-us/library/ms972319.aspx [Acesso em 29 Julho 2014].

IDC Worldwide Mobile Phone Tracker, 2014. *Worldwide Smartphone Market Grows 28.6% Year Over Year in the First Quarter of 2014,* Framingham: s.n.

Linares-Vásquez, M. et al., 2014. *Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study.* Hyderabad, s.n., pp. 2-11.

Linares-Vásquez, M. et al., 2013. *API Change and Fault Proneness: A Threat to the Success of Android Apps.* Saint Petersburg, s.n., pp. 477-487.

Mojica, I. J. et al., 2013. A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEE Software,* 31(2), pp. 78-86.

Neil, T., 2014. *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps.* 2ª ed. Sebastopol: O'Reilly Media.

Nilsson, E. G., 2009. Design patterns for user interface for mobile applications. *Designing, modelling and implementing interactive systems,* December, Volume 40, pp. 1318-1328.

PhoneGab, 2014. [Online] Available at: http:// phonegap.com/

Ruiz, I. J. M., Nagappan, M., Adams, B. & Hassan, A. E., 2012. *Understanding reuse in the Android Market.* Passau, IEEE, pp. 113-122.