An Exploratory Study of MVC-based Architectural Patterns in Android Apps

Aymen Daoudi
Université du Québec à Montréal
Montréal, Québec, Canada
daoudi.aymen@courrier.uqam.ca

Ghizlane ElBoussaidi
École de technologie supérieure
Montréal, Québec, Canada
ghizlane.elboussaidi@etsmtl.ca

Naouel Moha
Université du Québec à Montréal
Montréal, Québec, Canada
moha.naouel@uqam.ca

Sègla Kpodjedo
École de technologie supérieure
Montréal, Québec, Canada
segla.kpodjedo@etsmtl.ca

ABSTRACT

Mobile app development now represents a significant part of the software industry, with Android being the largest ecosystem. Android development comes with its own design practices and templates (layouts, activities, etc.). Developers also use different established architectural patterns for designing interactive software such as MVC, MVP and MVVM. They implement these patterns based on their understanding and experience. Thus, the choice and the implementation of such patterns varies from a developer to another. To the best of our knowledge, there is no work that provides a comprehensive view of the use of these patterns in mobile apps. Moreover, there is no clear understanding of which pattern to use and what is the trend for designing mobile apps using such patterns. In this paper, we propose an automatic approach to identify which MVC-based architectural pattern (MVC, MVP and MVVM) is used predominantly in a given app. For this purpose, we defined each of these patterns through a number of heuristics according to the pattern's potential implementations within the Android framework. We conducted an empirical study on a large set of mobile apps downloaded from the Google Play Store. We found, not surprisingly, a dominance of the popular MVC pattern, a rare use of MVP while MVVM is almost unused and a significant number of apps do not follow any pattern. The empirical study also enabled us to analyse the use of these patterns by domain, size and last-update date of the apps. We observed that MVC has been the most used pattern over the past years and it continues to gain popularity, and that small-size apps are mostly the ones that do not use any pattern.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; **Design patterns**; *Object oriented languages*; Frameworks; • **Human-centered computing** → *Smartphones*;

KEYWORDS

Software architecture, Architectural patterns, Design patterns, Mobile development, MVC, MVP, MVVM

ACM Reference Format:
Aymen Daoudi, Ghizlane ElBoussaidi, Naouel Moha, and Sègla Kpodjedo. 2019. An Exploratory Study of MVC-based Architectural Patterns in Android Apps. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3297280.3297447

1 INTRODUCTION

Mobile apps are without doubt a growing market. The vast majority of these apps operate under Android, which has established itself as the predominant mobile platform with a market share of 88%[1]. Mobile devices usually come with more constraints, in terms of memory, battery, computational power, competition for resources, etc. To cope with these, the Android framework proposes a number of mechanisms and architectural components that put a greater emphasis on well-structured design patterns, as shown by the empirical study performed by Bagheri *et al.* [4]. Android developers have to compose with these mechanisms and decide on how to use them to meet their app's requirements.

The impact of these choices on the quality of an app cannot be overstated. In particular, with user interactivity being so central in almost any app, well established patterns at the presentation layer such as the Model View Controller (MVC) and its variants such as Model View Presenter (MVP) and Model View ViewModel (MVVM) become especially relevant. Although these patterns are widely used and discussed within the Android developers community, there is no consensus around their use or on which one is best suited for Android apps [9, 27, 29, 35]. In fact, there are no available data or studies that may be used to decide which MVC-based pattern is appropriate for which kind of apps. Moreover, each of these patterns may be implemented in different ways within the Android framework.

In this context, there is a need for studies that analyse which and how these MVC-based patterns are implemented within Android apps. These data are essential to provide an overview of design

[1]https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

trends in mobile apps and to specifically develop a better understanding of the relationship between the characteristics of Android apps and MVC-based patterns. Recent works on mobile apps include [21, 23–25, 27, 30, 32] and [4]. However, these works focus either on the possible implementations of such patterns [23, 25], manual identification and comparison [27], architecture recovery [4] or the automatic identification of anti-patterns and code smells [21, 24, 30, 32]. To the best of our knowledge, there is no work that supports the automatic detection of presentation layer architectural patterns such as MVC, MVP and MVVM in Android apps. Thus, this work aims to answer the following research questions:

- **RQ1:** What is the frequency of usage of the MVC-based patterns in Android apps?
- **RQ2:** Which kinds of Android apps use the MVC-based patterns?

To answer these questions, we propose an approach, called Rimaz, to identify the dominant MVC-based pattern in an Android app. The approach is based on heuristics that consider the possible implementations of MVC variants in Android. Using Rimaz, we conducted a study on 5,480 apps from Google Play Store to get a sense of the Android landscape relatively to the use of MVC variants. Therefore, our contributions are essentially two-fold: 1) an heuristics-based approach to identify Android MVC variants and 2) an empirical study to answer questions pertaining to the use of such patterns in Android apps.

The rest of the paper is organized as follows. Section 2 presents the related work as well as some key concepts about Android and the MVC-based patterns. Section 3 presents Rimaz, our heuristics-based approach. The evaluation of our approach is provided in Sections 4 and 5. Section 6 concludes the paper and discusses future work.

2 BACKGROUND AND RELATED WORK

This section briefly presents the Android framework and its key components. It then describes the three architectural patterns (MVC, MVP and MVVM) and their implementations in Android.

2.1 Android Framework

Android app development is organised around key building blocks, which understanding is required for developers and designers. These app components —such as Activity, Service, BroadcastReceiver, and ContentProvider— serve as entry points for an Android app. *Activities* are key components that define screens for user interaction, while the other three are faceless components that deal with background tasks (*Services*), notifications (*Broadcast receivers*) and access to shared data (*Content providers*). In addition to Activities, there are a few lower level components, attached to an Activity, such as Fragments and Dialogs that help for user interaction. At an even lower level are UI objects such as `Button` and `EditText`.

All these components help in the realisation of an Android app. The concept of Application is explicitly defined in Android with a top-level Application class that is declared in a manifest file (AndroidManifest.xml), with information pertaining to its visible activities, services, content providers and broadcast receivers. The application source code is bundled with data and resources, compiled using respective language compilers and the Android SDK. This results in an archive file called an APK (Android Application Package), which contains all the necessary information to install and run the application [16].

The Application classes as well as Activities, Fragments and Dialogs are the main components that contribute to the behavior and the life cycle of mobile apps with user interfaces. As this work targets the study of patterns of the presentation layer, we will mainly focus on Android apps that use these components.

2.2 MVC, MVP and MVVM

The development of maintainable and high-quality interactive software requires a clear separation between the underlying data, its presentation and manipulation. The MVC-based architectural design patterns are thus widely used in such type of software [6]. There are however different perspectives leading to different MVC-based patterns.

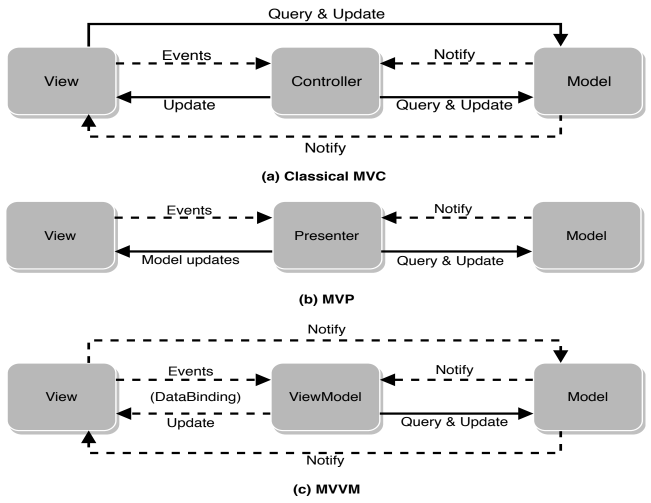


Figure 1: Diagrams presenting MVC, MVP and MVVM

2.2.1 *Model View Controller.*

Proposed in 1979, MVC aims at separating the business logic from the presentation layer [31] [8]. It defines three main layers. First, the **Model**: basically a set of entities that represent knowledge, information and a set of rules that handle updates and access to these information [31]. To ensure loose coupling with the other layers, the Model should not know about the other components but can send them notifications about the state of its objects ([6] [7]). Second, the **View**, which is in charge of providing a visual representation of the Model. The View shows data that it retrieves from the Model by sending queries, and can change its state through messages [31] [8]. Third, the **Controller**, which is the main entry point of the application, manipulates the View and translates the interaction of the user to the model [31]. Furthermore, the Controller communicates directly with the Model and must be able to change its state. [6]. Figure 1 (a) shows the three components of the original MVC architecture and describes the connections between them.

Since its introduction, MVC has evolved along with the apparition of new development frameworks. Many adaptations were brought to the pattern to fit in the targeted framework, resulting in new variants.

2.2.2 *Model View Presenter*.

MVP (Figure 1 (b)) is a variant of the MVC architecture, introduced in the early 90s. It brings more decoupling between the domain layer and the user interface.

Compared to MVC, the Presenter in MVP is more isolated - than the Controller in MVC - from the underlying system and should know nothing about the system [5]. The Model has essentially the same responsibilities whereas the View handles actions that manipulate components of the system or the user interface and only delegates to the Presenter actions that manipulate the model entities.

If the Presenter needs to make visual changes to the user interface or make interactions with the underlying operating system, it should call a method of the View that implements this need [12].

2.2.3 *Model View ViewModel*.

MVVM is an evolution of the MVP pattern [22]. As shown in Figure 1 (c), it aims to realise more decoupling between the user interface and the middle layer represented by the ViewModel. The communication between these two is done exclusively through two-way data-binding notifications.

The View, generally written in declarative languages, usually XML variants, represents the user interface. It is as passive as possible; declaring only widgets and user interface objects, and delegating all the logic to its backing ViewModel, from which it retrieves any kind of updates [28]. The ViewModel is simply a Model of the View. It defines and implements the user input events, transforms the Model data and makes it ready to be displayed by the View. Importantly, the ViewModel should be completely ignorant about its corresponding View. No reference to a View should be declared nor used within the ViewModel. The ViewModel should be ready at anytime to be used with any other View entity without breaking code [28]. The role of **Model** entities remains unchanged compared to the two other patterns.

2.3 Implementation of the MVC-based patterns in Android apps

MVC is the first pattern adopted in the Android world. Yet its implementation in Android apps is subject to debate. At the center of this is the interpretation of the Controller [35] [9]. It is stated to be the entry point and the layer handling the user interactions with the view and translating them to the data layer.

Some developers consider the Activity as the main component that plays the role of the Controller, for being: i) the main entry points of the application and ii) the most natural place to declare input events.[2] Other developers believe that Activities, Fragments and Dialogs should be considered as part of the View, or at least helpers of the View components, while Controller components should be separate classes. Both choices are backed by strong arguments, well used in real world and should be taken into account by a detection approach.

MVP (Model View Presenter), on the other hand, is straightforward to implement on Android. Following the standard definition, MVP keeps the declaration of the events in the View, and delegates the work that they perform to the Presenter, which should delegate the manipulation of platform components and functionalities to the View. To achieve this, the View has a field referencing the Presenter and vice versa. In the Android world, and contrarily to MVC, Activities, Fragments and Dialogs are considered as parts of the View. Figures 2 and 3 simply illustrate the relation between an Activity and its corresponding Presenter.

```
Class ConcretePresenter extends AbstractPresenter
{
        AbstractView view;
        public ConcretePresenter (AbstractView concreteView)
        {
                ⋮
                view = concreteView;
                ⋮
        }
        Public presenterMethod()
        {
                ⋮
                View.viewMethod();
                ⋮
        }
}
```

Figure 2: Relation between a Presenter class and an Activity

```
Class Activity extends AbstractView, IEventListener
{
        AbstractPresenter presenter;
        public onCreated()
        {
                ⋮
                presenter = new ConcretePresenter(this);
                ⋮
                UIElement.SetEventListener(this);
                ⋮
        }
        Public onEventListenerAction()
        {
                presenter.presenterMethod()
        }
        Public viewMethod()
        {
                //manages UI
        }
}
```

Figure 3: Relation between Activity and a Presenter class

MVVM, came late to the Android world. However, since the introduction of new Architecture components by Google in 2017, it has taken a bigger profile and is now a recommended design for Android apps [18]. Android now simplifies the use of MVVM, through types such as the *ViewModel* [20] and the *LiveData* [19] types. It remains that Google started supporting fully advanced data-binding only in 2016 [15], so developers previously had to use third party libraries to implement MVVM. The ViewModel is represented by classes that play the role of a data-binding context for each View component. Binding View components to ViewModel classes, generates intermediate classes that define all binding operations and prepare them for runtime [15]. Events are bound from layout files to the corresponding ViewModel classes as methods, they will be later redefined in the data-binding generated classes using the traditional Listener pattern [15].

[2]In fact, even if the developer declares listener-based events elsewhere (which is possible), some sort of events, called event-handlers, can only be declared inside these components.

2.4 Related work

Recently, some works targeted the manual analysis of MVC-based patterns in mobile apps while only one treated their architecture recovery. In fact, there is no work relating specifically to the automatic detection of patterns in mobile apps, except for anti-patterns and code smells. We report in the following the related works.

2.4.1 *Design and MVC-based patterns in mobile apps.*

Few recent works deal with design patterns including MVC-based ones in mobile apps. A recent Master's thesis [27] presents a manual analysis and some empirical data to verify whether MVP and MVVM architectures are better than MVC in Android apps from a quality perspective. Another recent book [23] presents how to implement MVVM pattern on the Xamarin framework. Similarly, La *et al.* [25] presented how to design a balanced MVC architecture for service-based mobile apps. Other similar works studied design patterns and proposed their own adapted patterns for mobile apps [34], [33].

Although all these works provide interesting contributions to design and MVC-based patterns in mobile apps, none of them proposed automatic approaches for their identification. The closest works relate to the automatic detection of anti-patterns and code smells in mobile apps.

2.4.2 *Automatic detection of design patterns in object oriented software.*

Antoniol *et al.* [2] were the first to present an approach based on an UML representation of the source code and the architecture of the program, that is used to extract metrics to detect the presence of the patterns. Fontana and Zanoni [11] proposed a plugin called MARPLE for the Eclipse IDE. This tool allows the detection of object oriented (OO) design patterns by extracting metrics from the source code and using a representation based on the abstract syntax tree of the application.

Such studies are useful for the detection of traditional design patterns in OO applications in general, but cannot be used to identify presentation layer architectural patterns in Android apps. These latter require metrics and heuristics that are tightly coupled with the platform's architecture and components.

2.4.3 *Automatic detection of anti-patterns and code smells in mobile apps.*

There are few relevant works on the automatic identification of anti-patterns and code smells in mobile apps. Hecht *et al.* [21] proposed an automatic tool, called Paprika, which allows the detection of both common object-oriented and Android-specific code smells by analyzing the binaries of Android apps. Similarly, Palomba *et al.* [30] proposed a tool, called aDoctor, that is able to identify 15 Android-specific code smells from the catalogue of *Reimann et al.* [32] using metric-based detection rules. Kessentini *et al.* [24] proposed an alternative approach for detecting Android smells using a multi-objective genetic programming algorithm.

2.4.4 *Architecture Recovery in Mobile Apps.*

Bagheri *et al.* [4] conducted the first and only in-depth analysis of Android's app architecture. They identified several drivers that motivated the adoption of architectural principles in mobile software, specifically Android. They reverse-engineered the architecture of hundreds of apps and mined five different types of architectural styles including message-based explicit-invocation, message-based implicit- invocation, publish-subscribe, shared state, and distributed object styles. They found many improper usages of architectural constructs and styles in the reverse-engineered architecture of popular apps.

In our work, we specifically focused on the presentation layer, and in particular on MVC-based architectural patterns.

3 IDENTIFYING MVC-BASED PATTERNS IN ANDROID APPS

3.1 Overview of the Proposed Approach

Our tooled approach, called Rimaz, aims to identify the MVC-based architectural patterns in Android apps. Rimaz relies on heuristics that describe the different potential implementations of MVC-based patterns in Android apps. Using these heuristics, Rimaz identifies the dominant MVC-based pattern. We specifically focus on the MVC, MVP and MVVM patterns. Written in Java, Rimaz relies on the Soot framework to analyze Dalvik bytecode contained within apk files [36]. Soot converts the Dalvik bytecode into intermediate representations that are easier to understand and analyse.

Figure 4 depicts the whole process used by Rimaz to identify these MVC-based patterns in an Android app. Steps 1 to 3 are preprocessing steps that aim at retrieving a number of information required by our detection algorithm. In the step 4 we apply the heuristics-based detection algorithm on the app. We describe each of these steps in details in the following subsections.

3.2 Step 1. Filtering Application Code

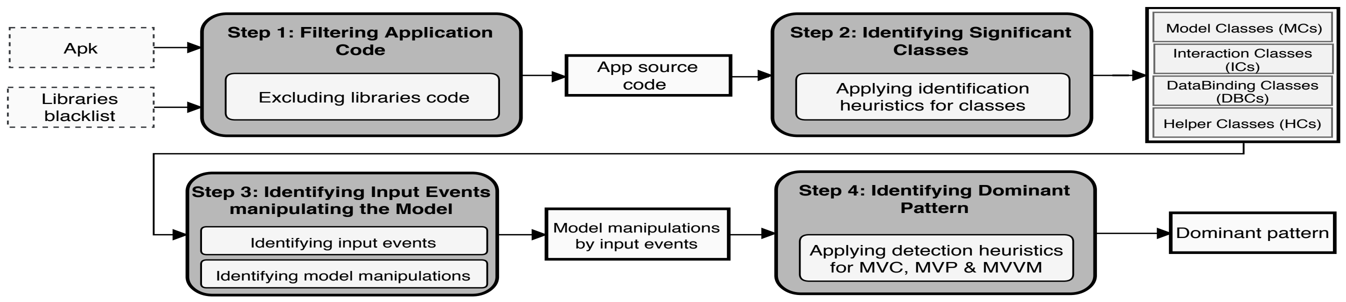
This step aims at differentiating the code of the app from the code belonging to used libraries. This is a common challenge encountered when analyzing Android apps since APKs contain both codes. Running analysis tools on the entire packaged code source will certainly produce incorrect results, and can affect the execution time of the analysis [26]. To fix this problem, some approaches use the package name declared within the manifest file as root folder of the app; they filter out all source code that does not belong to the package having that name. This approach has two limitations:

(1) Sometimes, developers change the package name and use a completely different one in the manifest file.
(2) Some Android projects may have many root folders; using only the one declared within the manifest file would exclude parts of the app's code.

Until now, the best workaround solution for overcoming this problem is to use a *blacklist* containing the names of most used libraries' packages. Li *et al.* [26] have conducted a study on a large set of Android apps and built a blacklist with more than 1,353 package names of the most used libraries. Since this list has not been updated since 2015, we updated it by adding another list of 1,176 package names of the widely used libraries that we manually collected from community websites such as [37] and [3].

3.3 Step 2. Identifying Significant Classes

As explained in the background, the MVC-based patterns aim at decoupling the user interface from the entities representing the model. To determine which of these patterns is used in an app, we need to identify the classes playing particular roles in the application. In particular, we need to distinguish between: classes through which the user interacts with the app, classes representing



Apk
Libraries blacklist
Step 1: Filtering Application Code
Excluding libraries code
App source code
Step 2: Identifying Significant Classes
Applying identification heuristics for classes
Model Classes (MCs)
Interaction Classes (ICs)
DataBinding Classes (DBCs)
Helper Classes (HCs)
Step 3: Identifying Input Events manipulating the Model
Identifying input events
Identifying model manipulations
Model manipulations by input events
Step 4: Identifying Dominant Pattern
Applying detection heuristics for MVC, MVP & MVVM
Dominant pattern

Figure 4: Overview of the Rimaz approach to detect dominant MVC-based patterns in Android apps

the entities of the model, classes responsible for data-binding, and other remaining classes.

3.3.1 ***Identifying Interaction Classes.*** In general, in the context of an Android app, classes that support user interactions are Activities, Fragments, Dialogs and Application classes. We call these classes *Interaction Classes* (**IC**s). The heuristic to identify these classes in a given app is as follows:

- An Activity, Fragment, Dialog or Application class is any class that has at least one super-type whose qualified name ends with `.app.Activity`, `.app.Fragment`, `.app.Dialog` or `.app.Application` respectively.

These super-types are usually all part of the `android.app.*` package. However, when developers target multiple API versions, they use super-types provided by support libraries [17]. For instance, a Fragment class originally extends `android.app.Fragment`, but it may also extend the support class `android.support.v4.app.Fragment`. Generally, a class provided by support libraries has a fully qualified name that starts with `android.support.vX` where `X` is the number of the supported version.

3.3.2 ***Identifying Model Classes.*** *Model classes* (**MC**s) are classes describing the state of the system, which may be persistent or not. In an Android app, these classes can be implemented as simple *Bean classes* or built using *ORMs*, *serialisation* or *SQL data base libraries.* To cover the most common implementations of MCs, we first listed from [3, 37] a set of up-to 50 of the most used libraries of ORMs, SQLDBs and serialisation libraries. Then we applied simple heuristics that check if these libraries were used to identify MCs.

3.3.3 ***Identifying DataBinding Classes.*** Data-binding eases synchronisation and enables more decoupling between the UI and the business logic. MVVM relies on data-binding to decouple the View and the ViewModel. When using data-binding capabilities in Android, binding classes are automatically generated at compile time and executed during runtime to perform the desired bindings; we call these classes *DataBinding classes* (**DBC**s). We use the following heuristic to identify these classes in an Android app:

- A DBC is any class extending `android.databinding.ViewDataBinding`.

3.3.4 ***Identifying Helper Classes.*** We call *Helper classes* (**HC**s) all classes that are neither ICs nor MCs nor DBCs. Here, the concept of HCs does not relate to the classical object-oriented programming meaning of a helper class. In our context, HCs are usually classes used by ICs and do not play any significant role in the particular used architectural pattern (*i.e.* MVC, MVP or MVVM).

3.4 Step 3. Identifying Input Events Manipulating the Model

Besides recognising significant classes in an Android app, we need to analyse the interactions between these classes to be able to identify which of MVC, MVP or MVVM is applied. To do so, we need to identify the events that are triggered when the user interacts with the app and how these events are propagated to MCs. Input events are a key parameter in identifying which architectural pattern is used in Android apps. Specifically, the declaration/triggering location of input events as well as the way they access and manipulate MCs definitely determine the applied pattern.

3.4.1 ***Identifying input events.*** Google, in the Android documentation, calls events *Input Events* (**IE**s). We distinguish between:

a. **Listener-based input events**: which are triggered by UIElements[3] and implemented using the listener pattern [14]. To refer to these events, we will use the abbreviation **LIE**s (i.e. Listener-based Input Event) in the following.
b. **Event handlers**: these are methods belonging to classes extending/implementing certain types (classes/interfaces) of the Android framework. The overridden methods are invoked by Android when specific events are triggered [14]. They can be:
   - Life cycle event handlers : such as Activiy's `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, `onDestroy` and `onRestart` methods;
   - Non life cycle event handlers : for example, the `onKeyDown` method is called when the user presses one of the phone volume buttons.

To identify LIEs, we rely on the characteristics of the listener pattern, which is a specific implementation of the Observer pattern [13], which includes: **1)** A contract, represented by an interface or an abstract class, corresponding to the listener and containing one or several methods that are invoked when the event is fired; and **2)** The UIElement through which the event is triggered, and which must have a field representing the listener (or a collection of listeners and methods) to set the listener (or to add one to the collection of listeners).

[3]The Android convention is to call UI objects (such as Button, EditText etc.) *Views* since they are all sub-types of `android.view.View`. This term can be confused with the term View used in architectural patterns. To avoid any confusion, we use the term *UIElement* for all UI objects [14].

To identify event handlers, we collected, from the Android documentation, a list of all the potential super-types of classes containing event handlers. This list includes, in addition to ICs and UIElements, types such as `Application.ActivityLifecycleCallbacks`, `KeyEvent.Callback`, `Window.Callback`, `LayoutInflater.Factory`, and `View.OnCreateContextMenuListener`. We consider as an Event handler any method that overrides/implements a method defined by one of the types of this list.

3.4.2 ***Identifying model manipulations by input events.*** We focus on input events that manipulate MCs because it is the way these events are propagated to MCs that differentiates between MVC, MVP and MVVM. An input event can manipulate a given MC by reading information from its static fields or from instance fields of the MC's instances. This is generally done through calling *getter* methods. The input event can also write to the MC by performing an initialisation or calling a *setter* method to assign values to its instance or static fields. Model manipulations are identified in our approach using a number of heuristics.

Depending on the applied architectural pattern, input events may directly or indirectly manipulate MCs. Therefore we distinguish and identify the following situations:

- **A direct manipulation**: In this case the manipulation of the MC is performed in the body of the execution methods of an event. Here we call *execution method* any method that is invoked when the event is fired in case of a LIE, or any overridden method in case of an event handler. For example, for the click event, the `OnClickListener` listener is used, and its execution method is the method `onClick()`.
- **Via sibling methods manipulation**: this represents an indirect manipulation of the MC through the call of a sibling method (*i.e.* a method belonging to the same class).
- **Via another class manipulation**: this represents an indirect manipulation of the MC by calling methods of other classes (*e.g.,* HCs or DBCs).

3.4.3 ***Classifying model manipulations according to the definition location of LIEs .*** Depending on the class in which input events are defined – we call it definition location – and if these events directly or indirectly manipulate MCs, we can determine which of the MVC, MVP and MVVM patterns was applied in a given app. For instance, when most of the manipulations of MCs are initiated by LIEs defined in DBCs, we suspect that MVVM was applied in the app under analysis. Therefore we need to classify each model manipulation according to the definition location (*i.e.* ICs, HCs or DBCs) of the related input event. Note that we specifically target LIEs. In fact event handlers are defined in interaction classes (ICs) as explained in Section 3.4.1; their definition location is enforced by the Android framework and is independent of the applied MVC-based pattern. Therefore we classify model manipulations initiated by LIEs into two sets :

(1) Model manipulations initiated by LIEs that were declared in `Activity`, `Fragment` or `Dialog` classes (We exclude `Application` classes from ICs in this part).
(2) Model manipulations initiated by LIEs that were declared in HCs or DBCs.

3.5 Step 4. Identifying Dominant Pattern

Developers tend to be flexible about their implementations of patterns, sometimes violating or combining them. Our algorithm is interested in revealing the dominant pattern, which is probably the conscious choice of the app designers.

In the RIMAZ approach, when there are no Model classes, there is no point in looking for MVC variants. When we do have Model classes, our algorithm tries to determine the dominant kind of model manipulations. The activity diagram given in Figure 5 summarises the different heuristics for the detection of the MVC-based patterns.

A first key question, from which we derived our heuristics ([Hx]), is the following: *Where are most model manipulations initiated by LIEs defined ?*

3.5.1 ***Case 1: Most model manipulations initiated by LIEs originated from ICs.*** The detection can be narrowed to MVC or MVP. A second key question then becomes: *Do most model manipulations occur in ICs – either directly or through sibling methods?* In fact, after confirming that most of manipulations are initiated by LIEs that originiated from ICs, we consider at this step all manipulations of MCs, including those initiated by event handlers. **[H1] If yes, it means the developer considered ICs as being Controllers: we have MVC as the dominant pattern.** In this case, the app implements MVC. The View is represented by layout files and any custom UIElement, while the Controller consists of Activities, Fragments and Dialogs.

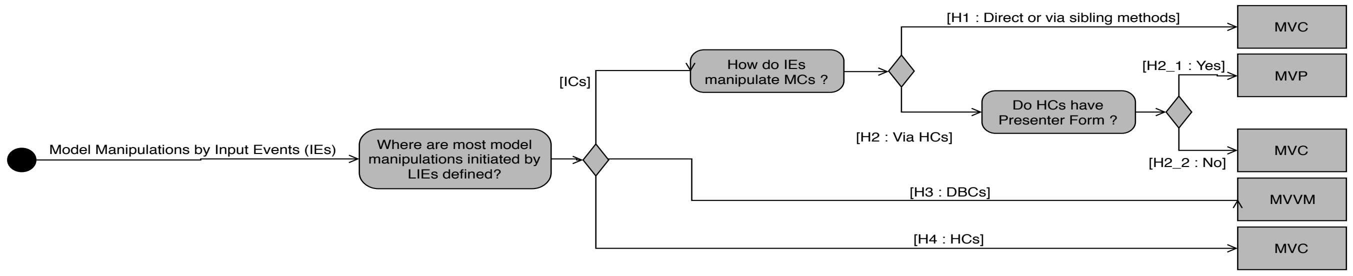
[H2] Otherwise, this means the ICs delegate the manipulation of model entities to HCs: [H2$_1$] if these HCs have a *presenter form*, we have an MVP. In this case, the View consists of the Activities, Fragments, Dialogs, their corresponding layout files, as well as any custom UIElement; **[H2$_2$] otherwise it is MVC.** The HCs along with the Activities, Fragments and Dialogs are considered as Controllers, while the View is represented by the layout files and any custom UIElement.

3.5.2 ***Case 2: Most model manipulations initiated by LIEs did not originate from ICs.*** **[H3] If the LIEs are mostly defined in DBCs, this means that events were defined using Android data-binding techniques and this most probably indicates the use of the MVVM pattern**. To confirm that it is an MVVM, we check that: 1) the DBC has a private field whose type is of one of the HCs, 2) this HC is also used to define a field in an IC, and 3) this IC also defines a field having the same type as the type of the DBC. In this case, the ViewModel corresponds to the used HC and the View consists of the Activities, Fragments, Dialogs and any layout file.

[H4] If the LIEs are mostly defined in HCs, this is an MVC where the HCs are considered as Controller classes. In this case, the Controller consists of both HCs and ICs while the View consists of layout files and any custom UIElement.

4 EXPERIMENTAL SETUP

The goal of our experiments was to elicit the trends in terms of using the MVC, MVP and MVVM patterns within the mobile community. Specifically, our experiments aim at addressing the following research questions:



Model Manipulations by Input Events (IEs)
Where are most model manipulations initiated by LIEs defined?
[ICs]
How do IEs manipulate MCs ?
[H1 : Direct or via sibling methods]
MVC
[H2 : Via HCs]
Do HCs have Presenter Form ?
[H2_1 : Yes]
MVP
[H2_2 : No]
MVC
[H3 : DBCs]
MVVM
[H4 : HCs]
MVC

Figure 5: Activity diagram summarising the MVC-based patterns detection algorithm

- **RQ1**: What is the frequency of usage of the MVC-based patterns in Android apps? The goal of this question is two-fold : 1) find out which of the MVC, MVP and MVVM patterns is mostly used; and 2) know how the usage frequency evolved over time.
- **RQ2**: Which kinds of Android apps use the MVC-based patterns? Here, we want to find out if the usage of these patterns correlate with some properties of mobile apps.

To answer these questions, we conducted an empirical study by applying our approach on thousands of apps from *Google Play Store*. Prior to this empirical experiment, we conducted a preliminary study on a set of open-source mobile apps to evaluate the correctness of our heuristics-based detection algorithm. In the remaining of this section, we describe briefly our preliminary evaluation (Subsection 4.1) and we present the design of the empirical study (Subsection 4.2). The results of our empirical study are discussed in Section 5.

4.1 Preliminary evaluation

To the best of our knowledge, there is no existing study (or benchmark) on the detection of the MVC-based patterns in Android apps. Manual inspection of an app source code appeared thus as the best way to validate the results of our tooled approach Rimaz. Therefore, before conducting a large scale study on apps taken from the *Google Play Store*, with unlikely access to the apps' source code, we sought to get a measure of our success rate in classifying apps according to their MVC pattern choice. To do so, we randomly selected 100 applications from *F-Droid* [10], a well-established repository for free open-source apps, and analysed them with our tool Rimaz. After which, we asked three experts in the related field to manually inspect the source code of each app and annotate it with the architectural pattern that he/she could detect including MVC, MVP, MVVM and NONE (For apps with no patern used). After that, we considered the majority vote between the experts annotations and compared it to the result returned by our tool Rimaz.

Rimaz detected instances of the three studied patterns in the 100 apps; MVC was the most popular (73%), MVP was rarely used (16%), MVVM was only detected once, and some applications did not use any of the patterns (10%). The manual analysis of these apps revealed an accuracy of 90%, with a precision of 85.87%, a recall of 90.71% and a value of F-Measure of 88% [4]. Interestingly, the only instance of MVVM was a "showcase" app for illustrating the application of MVVM within the Android framework. Ten apps were wrongly classified. Two apps, that were actually mostly MVP, were mis-classified as MVC because of third party libraries that were not filtered out. Seven apps were detected as implementing MVP although they were implementing MVC. This was the result of a cross-reference between the View (embodied by Activity classes) and the Controller (corresponding to some helper classes), which is not the typical way of implementing MVC. One app was classified as an app not using any pattern although it used MVC, this was due to the fact that the app didn't use any of the common libraries for implementing the MCs.

[4]Precision and recall were calculated by computing the average of the precision and the recall of all classes (MVC, MVP, MVVM and NONE)

4.2 Setup of the empirical study

Figure 6 gives an overview of the experimentation process. For our dataset, we used the *AndroZoo* repository [1] that provides a comprehensive weekly updated list of Android apps from many different markets. We randomly selected from this list **5,480** apps that are present on the Google Play Store and for which we could get the metadata. To answer our research questions, we use the following metadata:

- Size: We use two factors to measure the size of apps including the number of classes and the number of lines of code. We only show the distribution of apps by number of lines of code for space constraints (Figure 7).
- Category: The apps are classified by category in the *Google Play Store*. Theses categories give some hints on the domain of an app. Figure 8 shows the number of apps by category.
- Last update year: The apps in our dataset were randomly selected. Thus they were probably created and updated at different times. To accomplish our analysis we do not rely on the creation date of the apps; but on their last update date. This is due to the fact that an app could have been created without using any of the MVC-based patterns and evolved into an MVC-based app. Figure 9 shows the number of apps by their last update year.

5 RESULTS AND DISCUSSION

This section reports the results of our empirical study, responds to each of our research questions and discusses the threats to validity of our study.

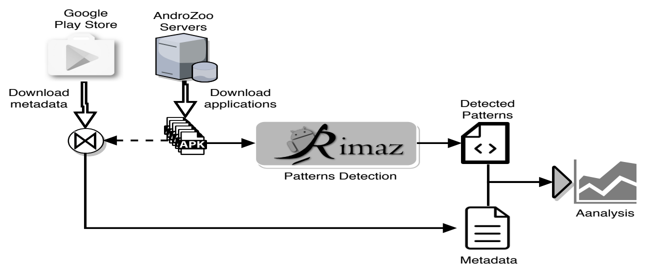


Figure 6: Experimentation process of the Rimaz approach

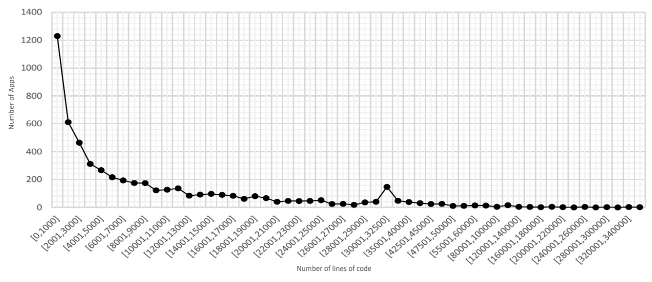


Figure 7: Apps distribution by number of LOC

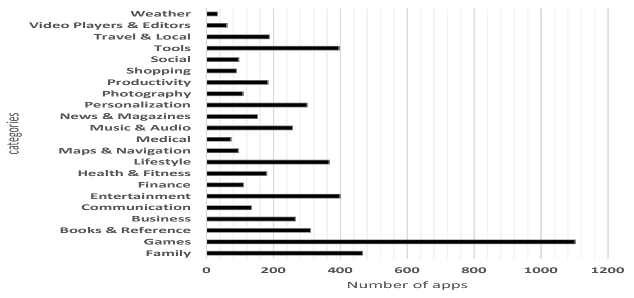


Figure 8: Apps distribution by category

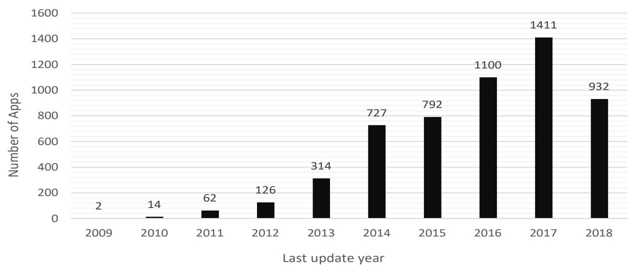


Figure 9: Apps distribution by last update year

5.1 RQ1: What is the frequency of usage of the MVC-based patterns in Android apps?

We analysed the **5,480** apps of our dataset using Rimaz. The results of the detection algorithm are shown in Figure 10. 57% of the analysed apps used MVC while 8% of them used MVP and 35% did not use any of the MVC, MVP or MVVM pattern. Therefore MVC is the most used pattern by Android developers. This can be explained by three factors: 1) MVC was historically the first pattern adopted in the Android community; 2) a very large number of mobile apps are very simple with very few screens, and MVC offers enough separation of concerns to support these apps; and 3) developers tend to declare most of the input events in Activities which at the same time contain a number of UIElements and this naturally creates more coupling than what is suggested by MVP or MVVM.

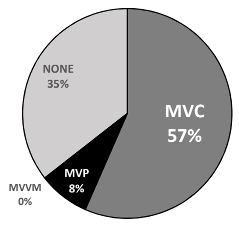


Figure 10: Distribution of the MVC-based patterns in dataset

Despite the fact that MVP and MVVM provide more decoupling and separation of concerns, very few apps use MVP (8% in our dataset) and there is no occurrence of MVVM. This can be explained by: 1) the small size of the mobile apps, the relationship between the size and the applied MVC-based pattern is analysed in the following subsection (5.2); and 2) the complexity that these two patterns may involve for non-experienced developers. The absence of MVVM however, is no surprise, as the mechanisms to ease the support for data-binding were added very recently to the Android framework (i.e. the Android Data Binding Library).

RQ1- Observation 1: MVC is the most used pattern by Android developers.

We also analysed how the usage frequency of the MVC, MVP and MVVM patterns evolved over time. For this purpose, we use the last update date of the apps. As explained above, knowing that an app could have been created without using any of the MVC-based patterns and evolved into an MVC-based app, we cannot speculate on the relationship between the creation dates and the application of the patterns. Figure 11 shows the evolution of the percentage of apps applying MVC, MVP or none of them over the years.

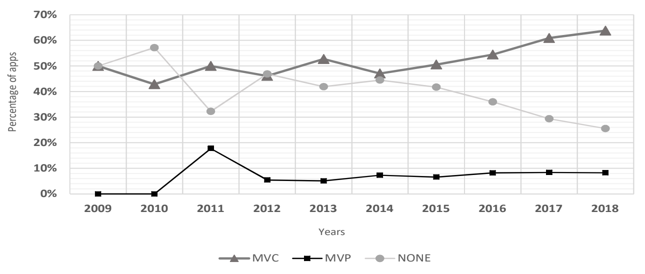


Figure 11: Evolution of the adoption of MVC-based patterns

These results show that Android developers started applying MVP around the year of 2010. Given all the recent debates around

MVC and MVP, we expected Android developers to be more interested in MVP and to use it more in the recent years. However, the results show that the overall increase of the usage of MVC during the last four years is more significant than the one of MVP. In fact MVP continues to be used but in a small portion of Android apps (less than 10%). This would suggest that the debate among developers' communities on the use of alternative patterns to MVC, such as MVP and MVVM, did not translate in increased use of those alternatives. Additionally, the percentage of apps that do not use any of the patterns has dropped significantly (from around 40% to about 25%) for the apps that were updated in the last four years, and this decline was in favor of an increase in the use of MVC.

RQ1- Observation 2: MVC has been the most used pattern over the past years and it continues to gain popularity.

5.2 RQ2: Which kinds of Android apps use the MVC-based patterns?

To get insights on factors that may affect the choice of a given pattern, we analysed the relationships between the usage of MVC-based patterns and the kinds of apps that were part of our dataset. Figure 12 shows the distribution by category of the occurrences of MVC and MVP that were found in our dataset. Interestingly enough, more than 50% of the apps use a pattern for each category. At first glance, these results do not indicate any correlation between the usage or not of a pattern and the category of an app.

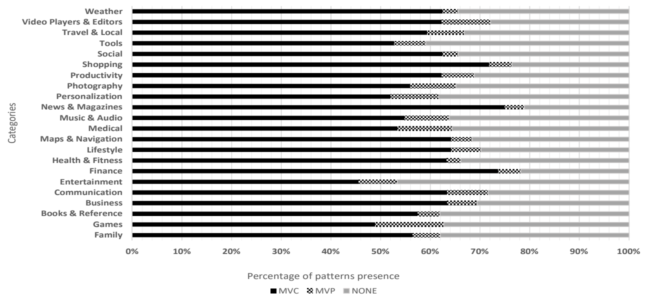


Figure 12: Distribution of MVC-based patterns by category

The categories *Finance*, *Shopping* and *News & Magazines* have the highest percentages of apps that use a pattern and these apps mostly use MVC. These categories generally include professional apps such as France24 (5 million downloads), Daily Expenses 3 (with 1 million downloads) and OLX - Jual Beli Online (10 million downloads). These three apps, for instance, were all implemented using MVC. The categories *Entertainment* and *Tools* have the highest number of apps that do not apply neither MVC nor MVP. We looked into the apps of these categories to see why they are not using any pattern. We found out that around 50% of the apps in these two categories have a number of classes that are less than 35 and number of lines of code less than 1600. Some examples of apps that do not use any pattern are DroidCam Wireless Webcam (size: 30 classes, 1260 lines of code) and Real Horn Sounds (28 classes, 477 lines of code).

Regarding MVP, it was used in the *Games* category more than any other category. This suggests that these apps require a clear separation of the layers View, Control and Model.

RQ2- Observation 1: No apparent relationship between the choice of MVC or MVP and the category of the app.

Our previous analysis suggests that there may be a relationship between the usage of patterns and the size of apps. Figures 13 shows the distribution of the MVC-based patterns by the number of lines of code of the analysed apps. We can see clearly that more than 50% of apps having their number of lines of code less than 2000 (their number of classes under 50), do not use any pattern. In general, for bigger sizes, the number of apps with no pattern decreases significantly.

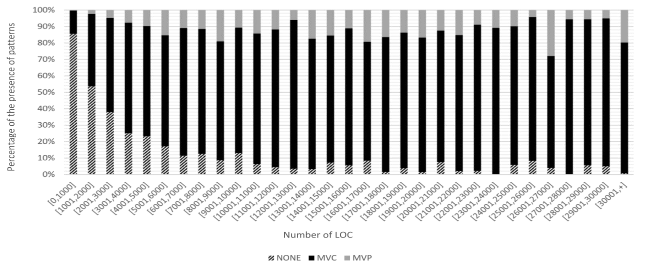


Figure 13: Patterns by number of lines of code

RQ2- Observation 2 : Small-sized applications are the least implementing any of the studied patterns.

5.3 Threats to Validity

In this section, we discuss the threats to validity of our study based on the guidelines provided by Wohlin *et al.* [38].

Internal validity threats concern the causal relationship between the treatment and the outcome. In this study, these threats can be caused by an incorrect detection of the patterns with RIMAZ, which may be due to the analysis of the code that belongs to a used library. This is why we used a blacklist of the most used libraries. We also updated this list by an additional list based on community recommendations. This list will be continuously updated to enhance the precision of RIMAZ. Moreover, we first evaluated manually our approach on a representative number of apps and applied it on a large number of apps. Finally, we tried to be careful when interpreting the results of our analysis. We consider the general trend known inside the Android developers community.

External validity threats concern the possibility to generalize our findings. We cannot claim that our findings are generalizable to the entire Android app ecosystem. We nonetheless randomly selected a large number of apps which can reasonably be expected as representative.

Reliability validity threats concern the possibility of replicating this study. We tried to provide all necessary information to replicate our analysis as well as our tool RIMAZ and the list of the analyzed apps from https://github.com/TheRimaz/Rimaz.

Finally, the *conclusion validity* threats refer to whether the conclusions reached in a study are correct. We did not make conclusions that cannot be validated with the presented results.

6 CONCLUSION AND FUTURE WORK

Mobile app development, especially for Android, is an appealing option for more and more software developers. User interactivity is a key element for most apps and as such, developer choices as to how to organise the basic interaction-centric components provided by the Android framework are particularly important. Interactive design patterns such as MVC, MVP, and MVVM are thus especially relevant for Android apps.

In this paper, we propose an approach to identify which of these MVC-based patterns is dominant in an Android app. The approach is based on heuristics that describe the different potential implementations of MVC-based patterns in Android apps. We validated the approach on a sample of open-source apps from F-Droid, then conducted a large scale study on 5,480 apps downloaded from Google Play Store. Our study shows that MVC is, with 57%, by far the most used pattern and seems to actually be growing in popularity in the Android community. MVP is less popular (around 8% of our dataset) and a significant percentage of the apps seemed to have no MVC-based pattern. Although present in the initial sample from F-Droid, MVVM was not detected in any of our dataset. Our study also revealed that there is no relationship between the usage or not of a pattern and the category of an app. Finally most of small-sized apps do not use any pattern.

Overall, our study is the first of its kind to provide some insights on Android developers' choice with respect to the use of MVC-based patterns. As future work, we plan to study how well these patterns are respected and well implemented, by detecting the most practiced violations by the community.

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. ACM, 468–471.

[2] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. 1998. Design pattern recovery in object-oriented software. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*. IEEE, 153–160.

[3] Roy Aritra. Accessed on 2018-02-01. *Ultimate Android Reference*. https://github.com/aritraroy/UltimateAndroidReference/blob/master/README.md

[4] Hamid Bagheri, Joshua Garcia, Alireza Sadeghi, Sam Malek, and Nenad Medvidovic. 2016. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software* 119 (2016), 31–44. https://doi.org/10.1016/j.jss.2016.05.039

[5] Andy Bower and Blair McGlashan. 2000. Twisting the triad: The evolution of the Dolphin Smalltalk MVP application framework. *Tutorial Paper for ESUG 2000* (2000), 1–7.

[6] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. 2007. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. 639 pages. https://doi.org/10.1007/s13398-014-0173-7.2 arXiv:arXiv:1011.1669v3

[7] John Deacon. 2005. Model-view-controller (mvc) architecture. *Computer Systems Development* Mvc (2005), 1–6. https://techsimplified2.com/Uploads/Agendas/October28,2011.pdf

[8] Tamal Dey. 2011. A Comparative Analysis on Modeling and Implementing with MVC Architecture. Mvc (2011), 44–49. https://doi.org/10.1.1.472.1591

[9] Maxwell Eric. Accessed on 2017-10-28. *The MVC, MVP, and MVVM Smackdown*. https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

[10] F-Droid website. Accessed on 2018-04-14. *F-Droid*. https://f-droid.org/en/about/

[11] Francesca Arcelli Fontana and Marco Zanoni. 2011. A tool for design pattern detection and software architecture reconstruction. *Information Sciences* 181, 7 (2011), 1306–1324. https://doi.org/10.1016/j.ins.2010.12.002

[12] Martinn Fowler. 2006 (Accessed on 2018-04-17). *GUI Architectures*. https://www.martinfowler.com/eaaDev/uiArchs.html

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 395 pages. https://doi.org/10.1007/3-540-48249-0_10

[14] Google's official Android documentation. Accessed on 2018-02-01. *Input Events*. https://developer.android.com/guide/topics/ui/ui-events.html

[15] Google's official Android documentation. Accessed on 2018-03-01. *Data Binding Library*. https://developer.android.com/topic/libraries/data-binding/index.html

[16] Google's official Android documentation. Accessed on 2018-04-12. *Application Fundamentals*. https://developer.android.com/guide/components/fundamentals.html

[17] Google's official Android documentation. Accessed on 2018-04-14. *Support Library*. https://developer.android.com/topic/libraries/support-library/index.html

[18] Google's official Android documentation. Accessed on 2018-04-21. *Guide to App Architecture*. https://developer.android.com/topic/libraries/architecture/guide

[19] Google's official Android documentation. Accessed on 2018-04-21. *Live Data*. https://developer.android.com/topic/libraries/architecture/livedata.html

[20] Google's official Android documentation. Accessed on 2018-04-21. *View Model*. https://developer.android.com/topic/libraries/architecture/viewmodel.html

[21] Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the Software Quality of Android Applications along their Evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 12.

[22] Gossman John. Accessed on 2018-04-18. *Introduction to Model-View-ViewModel pattern for building WPF apps*. https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/

[23] Paul Johnson. [n. d.]. *Using MVVM Light with your Xamarin Apps*. Apress. 200 pages. https://doi.org/10.1007/978-1-4842-2475-5

[24] Marouane Kessentini and Ali Ouni. 2017. Detecting Android smells using multi-objective genetic programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 122–132.

[25] Hyun Jung La and Soo Dong Kim. 2010. Balanced MVC Architecture for Developing Service-Based Mobile Applications. In *IEEE 7th International Conference on E-Business Engineering*. 292–299. https://doi.org/10.1109/ICEBE.2010.70

[26] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. An Investigation into the Use of Common Libraries in Android Apps. (2015). https://doi.org/10.1109/SANER.2016.52 arXiv:1511.06554

[27] Tian Lou. 2016. *A Comparison of Android Native App Architecture âĂŞ MVC, MVP and MVVM*. Master's thesis. Aalto University, Finland.

[28] Microsoft's MSDN. Accessed on 2018-04-18. *Implementing the Model-View-ViewModel Pattern*. https://msdn.microsoft.com/en-us/library/ff798384.aspx

[29] Joshua Musselwhite. Accessed on 2018-03-01. *Android Architecture*. http://www.therealjoshua.com/2011/11/android-architecture-part-1-intro/

[30] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Lightweight detection of Android-specific code smells: The aDoctor project. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017*. IEEE Computer Society, 487–491. https://doi.org/10.1109/SANER.2017.7884659

[31] T. Reenskaug. 1979. THING-MODEL-VIEW-EDITOR: an Example from a Planning System. *Xerox PARC Technical Note (May 1979)* (1979).

[32] Jan Reimann, Martin Brylski, and Uwe AÃ§mann. 2014. A Tool-Supported Quality Smell Catalogue For Android Developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung âĂŞ MMSM 2014*.

[33] Fadilah Ezlina Shahbudin and Fang-Fang Chua. 2013. Design Patterns for Developing High Efficiency Mobile Application. *Journal of Information Technology Software Engineering* 3, 3 (2013), 667–684. https://doi.org/10.4172/2165-7866.1000122

[34] Karina Sokolova and Marc Lemercier. 2014. Towards High Quality Mobile Applications: Android Passive MVC Architecture. 7 (06 2014), 123–138.

[35] StackOverflow thread. Accessed on 2018-03-01. *MVC pattern on Android*. https://stackoverflow.com/questions/2925054/mvc-pattern-on-android

[36] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[37] Bauer Vladislav. Accessed on 2018-02-01. *Android Arsenal*. https://android-arsenal.com/

[38] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.