Architectural pattern for native android applications

Sergio Martins Fernandes[1], Paulo Jose Mendes[2]

[1]Universidade Salvador (UNIFACS) Bahia – Brazil

sergio.fernandes@unifacs.br, pmendes.ti@gmail.com

Abstract. *Android platform is the absolute leader on the market of mobile devices, and the developers' community has been analyzing different approaches for producing good applications, for the growing number of potential users. Some community members point out the benefits of platform-independent development, and others don't give up the advantages of the native APIs. An issue frequently debated among the adepts of the native method is the difficulties to apply MVC-based patterns to native applications. In this paper, the authors analyze some acknowledged architectural patterns, including one specially conceived for Android, along with characteristics and non-functional requirements belonging to this environment, and propose an architectural pattern adapted for native Android applications.*

1. INTRODUCTION

Android operating system has been dominating the market of mobile devices since 2012. From then on, its advantage over Apple's iOS – its most relevant competitor – has been consistently increasing [3]. Its high acceptance is also noticeable in the numbers revealed by Google. There are hundreds of millions of mobile devices in more than 190 countries running Android, and more than one million new devices are activated every day [4].

The operating system maintained by Google is no longer related only to smart phones and computer tablets. Currently, Android environment is being extended to products of different segments, such as smart TVs, wearable devices, and even automobiles. As things become increasingly intelligent and connected to networks – specialized companies, and skilled developers may spot a multitude of possibilities and opportunities on the market of applications for Android platform.

Google also provides the Android Software Development Kit, the official set of tools for creating native applications. It comprises essential and supportive apparatus for the development processes – building, debugging, testing, and running in a variety of emulated virtual devices. The SDK also contains a complete Java API for programming, including a framework and libraries that give support for creating graphic user interfaces, handling user gestures, managing application life cycle, persisting data, accessing device sensors and hardware, connecting to networks, and so forth.

There are alternative approaches to develop applications that may run in Android OS. They all rely on third-party software – but differs both in the adopted technologies and in the final output. One of them draws on tools, such as NativeScript and Xamarin, which compile a unique source code base, written in its API language, into different native applications for diverse mobile operating systems.

The other method, also known as "Hybrid Development", is based on web

standards, requiring a hybrid framework responsible for bundling source files written in HTML, CSS and JavaScript – along with platform-specific components provided by the framework. For that reason, the resulting software is called “hybrid”. It consists of one or more web pages, which are platform-independent to be rendered either by the operating system's engine or by a proper Web View bundled together with the web code. This front-end makes use of native elements, which in turn can communicate with the devices' hardware, and to access several low-level functionalities [5].

Those “develop-once-deploy-everywhere” solutions might obviously seem more attractive when considering the reduced effort, cost and time, required to release applications for the major mobile operating systems. However, there are some important issues to be observed. Relying upon third-party software may be concerning as they have limited possibilities and demand plugins that can become deprecated and abandoned [1].

Moreover, each platform has its own “look and feel” – that is, its own visual identity and interaction logic to which their users become accustomed. Applications must benefit from the host system's look and feel, following its provided guidelines, to be user-friendly and harmonious with the system and its native applications [6].

A disadvantage of hybrid applications is the overhead caused by the additional web layer necessary to render the non-native code. In this scenario, the responsiveness is fairly compromised, being perceptible to the user how the web interface doesn't flow smoothly and doesn't reacts promptly to gestures [6].

Developers face a fierce competition on the market of mobile applications, where many of them are made available, promising to meet the same expectations. Users are inclined to choose the one that will offer not only the desired utilities, but also a pleasant experience [6, 7].

In general, natively developed apps have many advantages to consider. They have full access to all APIs provided by the system, with no intermediates, bringing better performance, stability and also being more secure. The graphic interface is built with native widgets that implement the proper User Experience principles of the platform, displaying consistency with the system and its native programs [6, 7].

Although the diverse sorts of Android-powered products, the operating system is still more frequently present in devices with relative low computational resources and limited power source – smart phones and tablet computers. A typical mobile application must be able to manage these restrictions and adhere to inherent non-functional requirements and qualities, for example: maintainability, efficient usage of device resources, responsiveness, connectivity, stability, usability, and security [2].

1.1. Motivation

Schmidt (2006) points out some complexities brought by modern middleware platforms, such as JEE, .NET and CORBA. These tools were designed to raise the programming abstraction level, which would shield developers from recurrent computational domain issues – such as transactions, distributed resources management, fault tolerance, event notification, security, and so on [8].

Although these frameworks have supported developers and teams over the years, sparing them from repetitive and complex strategies implementations, their usage

spawned some vexing side effects. They tools evolve faster than the general-purpose languages they make use of, hence a huge amount of classes, methods, and intricate dependencies require considerable effort to effectively work with – mainly when optimization is necessary. Even the means created to relieve those raised difficulties – such as annotations and configuration files coded in markup language – have caused semantic gaps between the design intent and the structured code, that doesn't clearly express both domain semantics and design intent [8].

Android development shares in considerable extent the issues referred by Schmidt. When developing native applications, developers make use of a complex and sophisticated API that covers managing the life cycle of application components, configuration of behaviors and interfaces objects through markup languages files, parallel tasks, connectivity, hardware and sensors handling, among others.

The official documentation for android development published on internet concerns scads of user interface and user experience issues – which, of course, is very important. Technical information of classes, methods, hardware sensors handling, and examples of API usage are also demonstrated in quite isolated contexts. Likewise, there are articles and tips presenting best practices for efficient memory management, stability, security and so on – but no relevant knowledge of architectural application design [1].

Android application framework and SDK are handy from perspectives such as assets management, internationalization, building view components, among others.

However, it's utterly negligent in enforcing an architecture which could guide and support the process of developing and maintaining native applications. This absence, both in documentation and in framework, has given rise to plenty of discussion in forums about well-established architectural patterns used for structuring Android applications, such as MVC and its adapted variations [1].

1.2. Objective

This article's main goal is to propose a variant of a known architectural pattern for developing Android native applications. First it is crucial to identify non-functional requirements and desirable qualities relevant for this platform, which must base the efforts to design a generic architecture. Likewise, another important sub-objective is evaluating architectural patterns for Android native development that may already exist.

The proposed architectural pattern must arrange the elements of an Android application accordingly to the general non-functional requirements desirable qualities present in mobile environment. This proposal makes use of diagrams to support the description of the proposed architecture pattern.

1.3. Justification

The description of the proposed architectural pattern will support the community of developers in structuring Android native applications, addressing its non-functional requirements. Furthermore, it will also have a role as reference for those who might come to work on mechanisms that provide automation, partial or complete, of architecture parts and their interactions.

2. Architecture and Architectural Patterns

Conceptually, Software Architecture is the depiction of the building-blocks of a software system. It describes how subsystems and their components are organized and how they are associated with each other aiming to meet functional and non-functional properties of the system [9].

Architectural patterns are documented representations of predefined high-level subdivisions of software systems, which also specify their responsibilities and relationships. Each of these patterns intends to support the implementation of global properties required by the application to be built, and may be found categorized into groups according to the problems it proposes to address. Most frequently, software systems must adhere to more requirements than a single architectural pattern is intends to address, so, in these common cases, suitable architectural patterns must be combined to ultimately set up a complete architecture [9].

It is worth to point out that an architectural pattern is not a complete software system architecture. It should be understood as an architectural framework which provides the rules and guidelines for all the consecutive stages of the software development. Thus, the definition of an architectural pattern is the first step to conceive the architecture of the product itself [9].

The architectural patterns, which intend to support the development of systems that requires wide interaction with human-beings, can be gathered into a specific category dedicated to interactive systems. Although they are structurally different, their common purpose is to decouple the user interface from the business logic, giving the possibility to adapt and evolve them separately [9].

Some of the most prominent interactive architectural patterns are the Model-View-Controller (MVC) and its variants, such as Model-View-Presenter (MVP), and the Martin Fowler's patterns – Supervising Controller and Passive View, which in turn are derived from MVP [12].

2.1. Model-View-Controller

Model-View-Controller (MVC) is the main architectural pattern for interactive systems. It first appeared within the SmallTalk community, conceived by Trygve Reenskaug, in the end of the 1970's [10]. The pattern states that an application must be split into three components – Model, View and Controller – collaborating with each other. Over the years, MVC has been adapted to distinct environments, and its implementation has been varying accordingly.

In general, the pattern is described as follows: The View correspond to the system's frontier with which the user interacts. It's also contains the presentation logic used to display the information sent directly from the Model through the Observable design-pattern implementation. The Controller cooperates with the View handling events resulting from user actions – e.g.: typing and clicking –, receiving input data and passing it to the Model. The Model contains the data state to be presented by the View, and the business logic operations [10, 11].

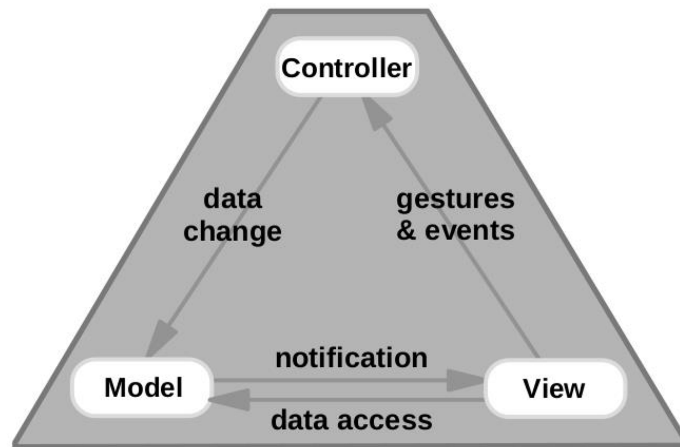

Figure 1. Model-View-Controller [15]

2.2. Model-View-Presenter (MVP)

This architectural pattern is based on MVC, changing some responsibilities and roles of its components. The Model is decomposed into three parts – Commands, Selections and Model –, each one with well-defined functions. Commands comprises the business logic and operations for processing data; Selections concern the abstractions for specifying different subsets of the Model's data; and the Model is simply the data itself [15].

The interactive part of the application lies under the responsibility of View and Presenter – the former is the corresponding of the MVC Controller –, which are tightly coupled. View becomes merely the representation of Model's data on the interface, and the interaction events are now handled by a new element placed between the View and the Presenter, the Interactor. Presenter assumes the duty of creating all the MVP components, and orchestrating their interaction, just like the traditional main function or event loop [15].

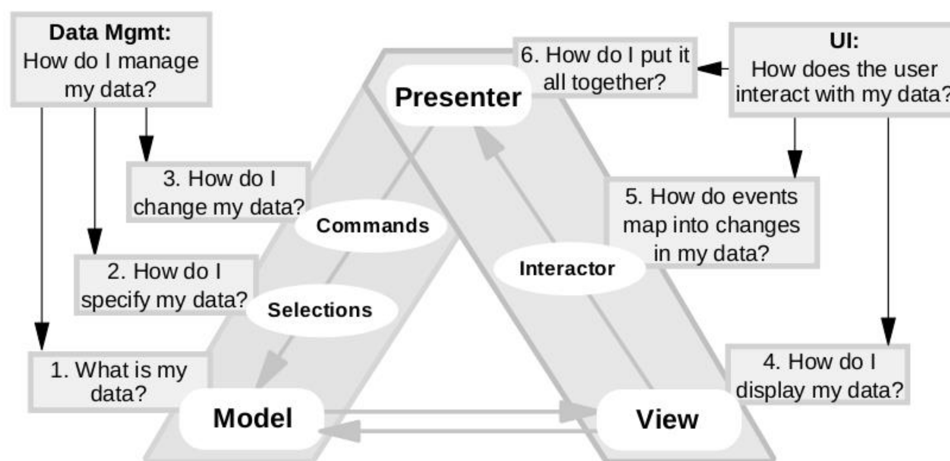

Figure 2. Model-View-Presenter [15]

2.3. Fowler's Patterns

Fowler (2006) discusses two distinct implementations of MVP, the original one by Mike Potel and the model of Dolphin Smalltalk framework. He points out divergences between them, and suggests that the MVP pattern should be split into two others, which he calls Supervising Controller and Passive View [12].

The Supervising Controller pattern declares the Controller pretty much like the MVP Presenter – closely coupled to the View, but without the subdivisions proposed by Taligent's MVP [15]. Here, the Controller/Presenter not merely handles user inputs, it goes further and keeps complex presentation logic to manipulate the View, which in turn has only very simple behavior. The Controller/Presenter monitors changes in the View through the Observer design pattern, and may access the Model to retrieve necessary data for its operations. The synchronization between View and Model is achieved with the Observer pattern as well, following the same principle in MVC and MVP [13].

In Passive View, the interface elements become even more humble, having absolute minimum behavior, and no logic at all. View and Model synchronization via Observer pattern can no longer be adopted, since the View doesn't have the necessary logic to handle notifications from the Model. The Controller assumes the entire presentation logic, and so becomes responsible for manipulating the View whenever user interactions or changes on the Model occur [14]

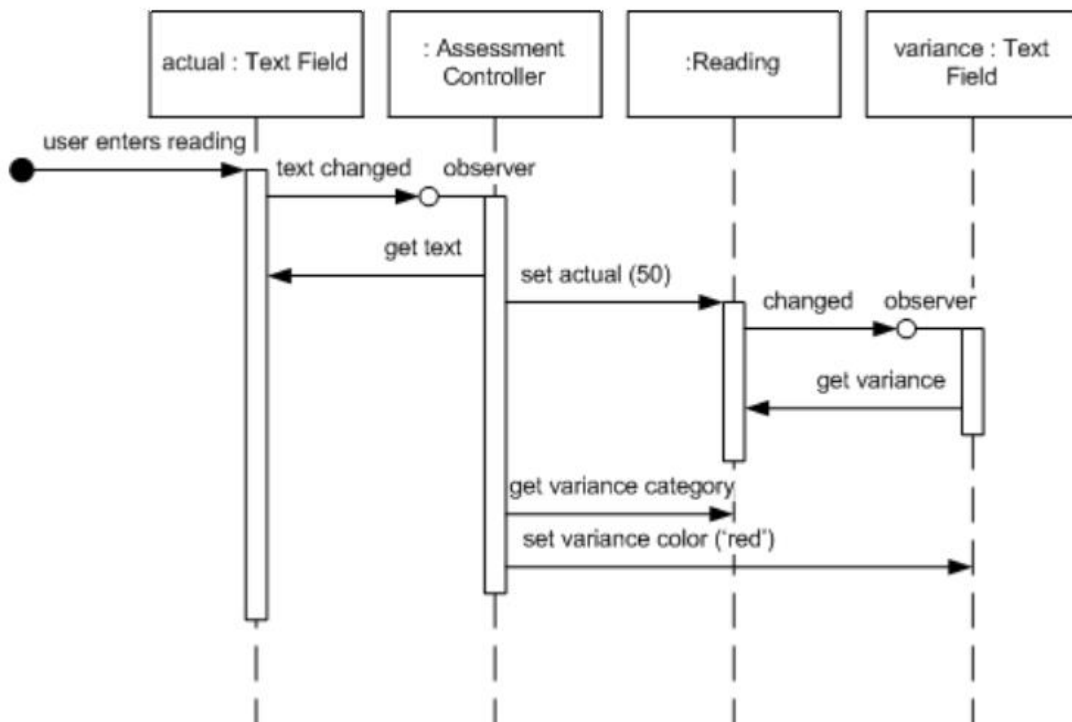

Figure 3. Sequence Diagram representing the Supervising Controller [13]

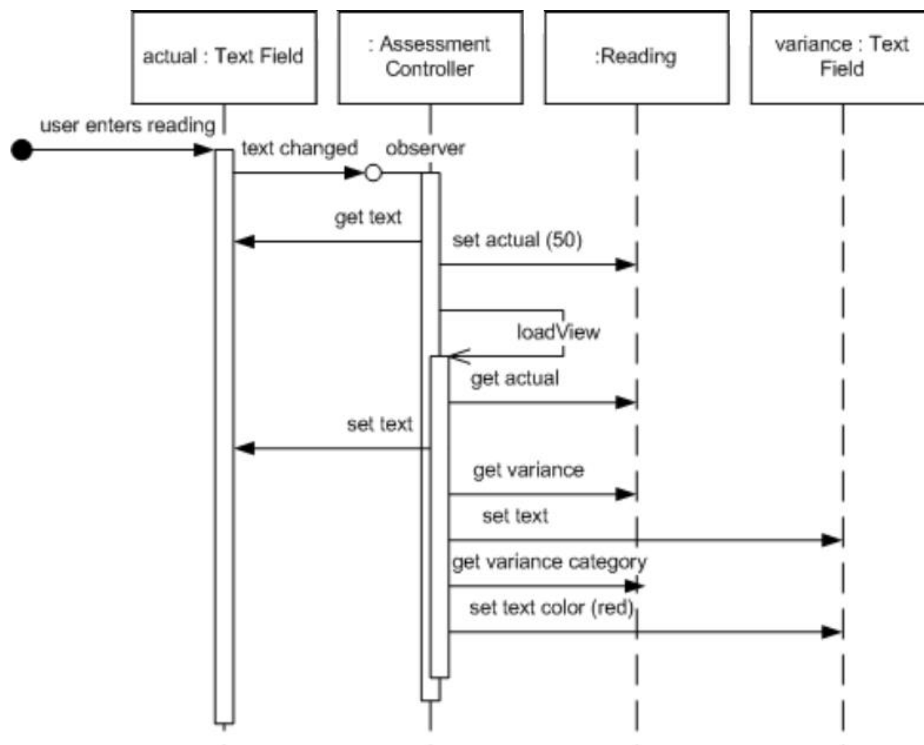

Figure 4. Sequence Diagram representing the Passive View pattern [14]

2.4. Layers

The Layers architectural pattern doesn't specifically belong to the family of Interactive Systems. In fact, it is a more generic pattern that may be applied to different classes of software systems – including operating systems, virtual machines, games and information systems – as well as being combined with other patterns [9].

It states that an application must be decomposed into subgroups – or layers – of software components, according to their level of abstraction, from the highest level at the top, to the lowest at the bottom. Each layer provides services for the one immediately above, as it occurs, for example, in the communication protocols. There's no specification determining the level of granularity of the layers and its components, which depends on the implemented design strategy [9].

In principle, the elements can only communicate with others that are in the same level, or with those in the adjacent layers. But this rule can be suppressed for the sake of performance and simplification. The Relaxed Layered System is a less restrictive variant of the Layers pattern [9].

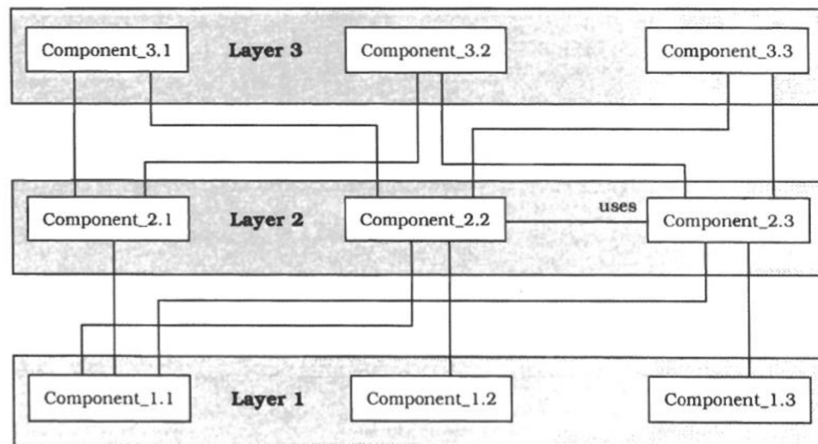

Figure 4. Layer Pattern [9]

3. Architectural Patterns for Android Native Applications

One characteristic of any Android application – which deeply affects its design – is the necessary presence of a component called Activity. When an application is started by the user, the Activity set up as the application entry-point is instantiated, occupying the whole screen area of the device. It is not properly a View object, but it is responsible for instantiating View objects from its corresponding XML configuration file – which depicts all the widgets and other View components embodied in the Activity.

Only one Activity can be activated at a time, representing the current application context. In fact, the Activity class is a heir of the Context class. Some sorts of native components depend on an explicit reference of the active context to be created. It is, for example, the case of classes that manage device connectivity and access local database.

An Activity also implements a chain of life cycle callback methods to be invoked by the operating system during the application execution. These methods lead the Activity to specific states, as shown in figure 5, and are called when events occur in the system. These events include device rotation, incoming phone calls, user sending application to background, application not responding, resources shortage, among others.

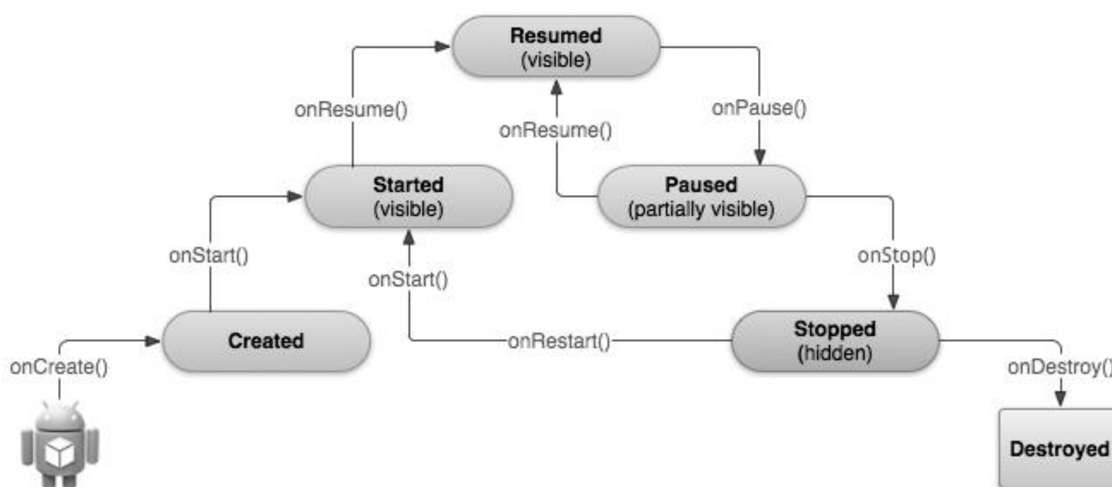

Figure 5. The main methods and states of an Activity life cycle [18]

One of the simplest scenario that serves to illustrate the importance of the proper Activity life cycle management is the device rotation. When user changes the device's orientation, the operating system destroys the current Activity – along with all its objects – and then recreates it according to the current position, horizontal or vertical. Before the Activity destruction, the system calls the methods onPause, onStop, and onDestroy, sequentially. In one of these methods, the developers have the chance to implement critical actions such as persisting the state of objects on secondary memory, to be retrieved when Activity is being recreated, during the execution of onCreate or onStart.

The community of developers are used to follow instructions presented in Google's official documentation and in specialist books. They focus on specific APIs usage examples, showing the View being controlled entirely by the Activity similarly to the pair View-Controller in Passive View pattern. This practice makes the Activity weighty, inducing to a loss of cohesion, as it accumulates too many responsibilities: application life cycle management, input events handling and presentation logic.

3.1. Requirements for Android Applications

Dealing with the inevitable existence of the Activity, including managing its life cycle, is the first non-functional requirement one may notice. Other concerns mentioned by Wasserman (2010) are relevant for the success of any mobile application, regardless of the operating system: performance (efficient usage of device resources, responsiveness, scalability), reliability (robustness, connectivity, stability), quality (usability, installability), and security [2].

The Android native development promotes these qualities, once it has available usability guidelines, tools and libraries optimized for the Android platform. It's important that developers make sense of them. For example, there are native classes such as AsyncTask that easily separate the execution of potentially expensive tasks from the main thread, to prevent the application from freezing and eventually being terminated by the operating system [22]. It's also worth to highlight the Firebase Cloud Messaging, a resource offered by Google that abstracts the implementation of reliable

connections through the internet, using device resources efficiently [21].

The architectural design decisions are not impacted by such components, as they are very specific and provide APIs that may preferably be encapsulated. On the other hand, Sokolova, Lemercier and Garcia (2013) bring up non-functional requirements that are affected and can be improved by a suitable architecture: extensibility, and maintainability [1].

The ISO/IEC/IEEE 24765:2010 comprehends the concept of both software maintainability and extensibility. Maintainability is defined as “the ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment” [19]. While extensibility, or extendibility, is “the ease with which a system or component can be modified to increase its storage or functional capacity.” [19]. Here, extensibility is one of the notions covered by maintainability.

To achieve a high level of maintainability, it's important to organize an application following the principles of low complexity, low coupling and high cohesion. Systems are easier to maintain when they are well structured, with clearly defined responsibilities assigned to its subsystems and components [20].

3.2. Analysis of Android Passive MVC

A MVC-based pattern is introduced in [1], intending to adapt the Activity into a widely known architectural model to create a solution easily understandable by developers coming from other systems.

Android Passive MVC reveals itself as a good effort to keep clear the responsibility assignment of the components. According to this pattern, the Activity becomes the fourth element in MVC, responsible for creating the View, its corresponding Controller, and a link for direct communication between them. The idea is to move input event handling from the Activity to the Controller, making the Activity lightweight.

Some issues about this configuration might be raised. The first is that the View and Controller, could not be replaced so easily, as their life cycle is bound to the Activity's states and methods, that cannot be delegated. Therefore, the Activity is necessarily coupled to the View, requiring code refactoring in an eventual View substitution.

The need of the Controller's presence is questionable in this case. The multiple View classes of Android native APIs are extremely rich, and already have plenty of methods that implements mapping and handling user gestures. These features don't need to be managed by other component. Developers may easily implement their own View classes in Java, extending widget classes and overriding the methods associated to pertinent events. The derived classes are perfectly available to be referenced in the Activity's XML configuration file.

The remaining of the pattern is the Domain Model shown in figure 6. A business facade component provides an API for Domain Model services used by the Controller. The article omits how the Model would be created and whether the Activity would manage it's life cycle or not. There is only vague reference to a relationship between Model and Controller, without more information.

Furthermore, this facade object represented in the pattern might not be necessary. In mobile applications, one single View object typically calls web services and local storage for quite specific operations that don't need any business logic, such as data fetching for input validations, autocomplete, among others. If the business service facade tries to provide all those specific operations, it may end up too complex and with low cohesion.

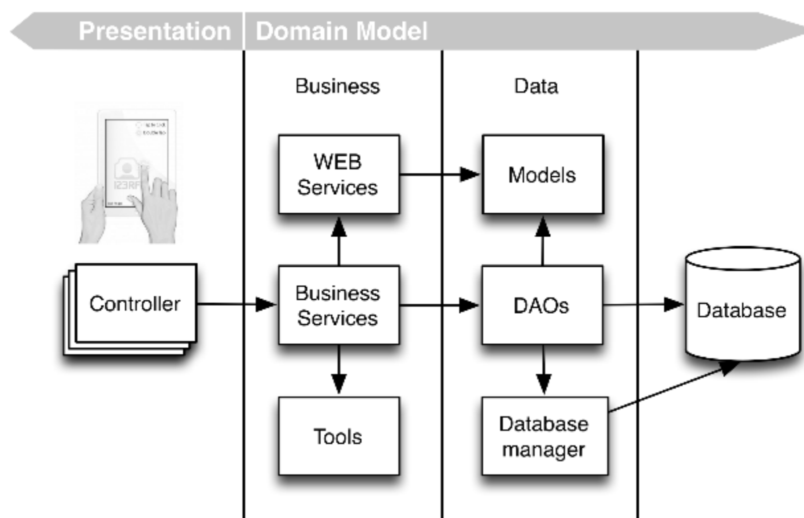

Figure 6. Android Passive MVC Domain Model [1]

3.3. Android Layered Application

Removing the Controller and delegating its function to the View is a rupture with the MVC-based patterns. All the MVC family have the figure of the Controller, even though its responsibilities and name may vary. So, the solution introduced by this article considers alternative architectural patterns to be adapted to the Android application characteristics.

Android Layered Application is based on Relaxed Layered System. This pattern is relevant due to its flexibility that makes it applicable to diverse sorts of software systems, and also because of its wide popularity among developers of other platforms. This proposed pattern is a structure of layers placed on top of each other according to their abstraction level, as demonstrated in figure 7.

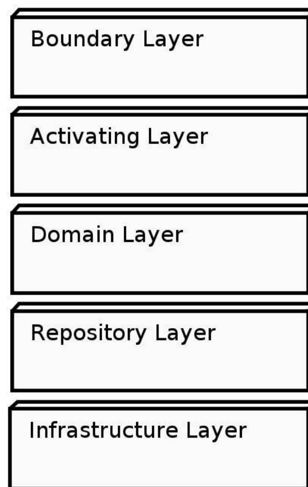

Figure 7. Overview of Android Layered Application

The top layer contains all the View components. It is named Boundary because that is where the interaction with the user occurs. Both presentation logic and event reactions implemented by child classes of native widgets resides in the Boundary.

Underneath the Boundary is placed the Activating layer, named after "Activating Components". This is the category of Android components – in which the Activity is the most fundamental element – responsible for starting an application context and manage its life cycle. The Activating layer is the application entry-point, and creates all the others layers when started.

The Domain layer, in the middle of the layers stack, was inspired by the corresponding Model of Android Passive MVC [1]. But in this case, it consists of Domain Models elements that concerns only data model and business logic methods. Even so, they may rely on web services and data persistence services offered by lower layers.

The Repository layer provides the abstractions for fetching data, locally and remotely. Data synchronization between device and the remote server, which can be tricky, must be executed within this layer. Repository API should implement static methods, since it must not keep neither data, nor connection objects (database or server) in primary memory – due to the device's limited resources and the fast access to flash memory. Every resource used is created and discarded in method scope.

The layer at the stack bottom, called Infrastructure layer, deals with the implementation of persistence strategies. It must be accessible exclusively by the Repository, providing interfaces that abstract the Infrastructure elements and configurations. Schema and queries for database, interfaces for manipulation storage files, specific libraries, data and methods needed for communicating to servers, they are all encapsulated in the Infrastructure layer. In this manner, new libraries to be released in upcoming Android API versions may be implemented without impacting the layers that makes use of the Repository.

For packaging the source code, Android Layered Application follows the recommendation of [16] to employ the method Package By Feature instead the Package

By Layer. The former method produces packages with higher cohesion and modularity. Other desired outcome is the easier code navigation, which increases the maintainability. The figure 8 depicts the suggested package organization.

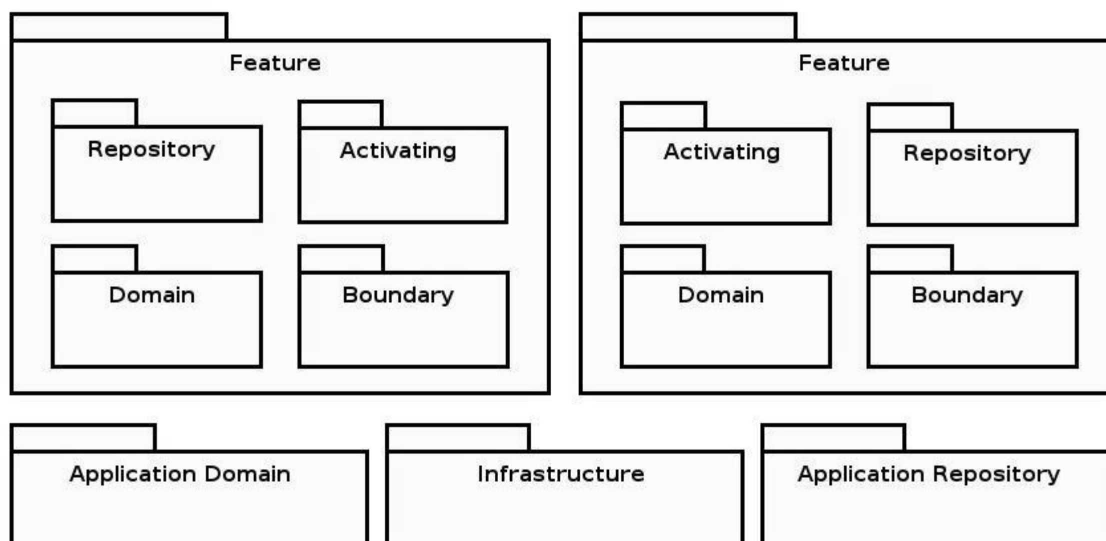

Figure 8. Example of Packages Organization

The highest-level packages represent the application features. Each of them contain sub-packages corresponding to the layers it makes use of. Obligatorily, an Activity must be present to represent the screen, which in turn correspond to the Activating layer of a particular feature. The View classes stay in the Boundary's sub-package, and the same logic applies to Domain and Repository concerns.

Some concerns inherent to, or used by, two or more features may be placed in packages at the same hierarchy level of the feature's. It is the case of the Infrastructure layer, that is abstracted by the Repository components of each feature. Likewise, Repository and Domain packages may also appear at the application level, for the cases in which their elements are shared between more than one feature.

The creation and communication between layers, is similar to the Relaxed Layered System pattern, except for the Infrastructure, as mentioned before. It is illustrated in figure 9 and 10.

The Boundary communicates with the Activating layer when the Activity instantiates the View objects and when their states need to be saved and recovered according to the application life cycle.

When necessary, the Activity may also intermediate the communication between two or more Boundary objects via implemented listeners.

The Boundary creates its Domain collaborators during initiation, and access them directly. In the same way, the Boundary knows and interact with Repository with no mediator. Domain business methods may also need to call the Repository, therefore, the relationship between these two layers must exist.

The Activating Layer and the Model don't know each other, and for the purpose

of application life cycle management, the Boundary is the responsible for providing primitive data corresponding to the Model objects state.

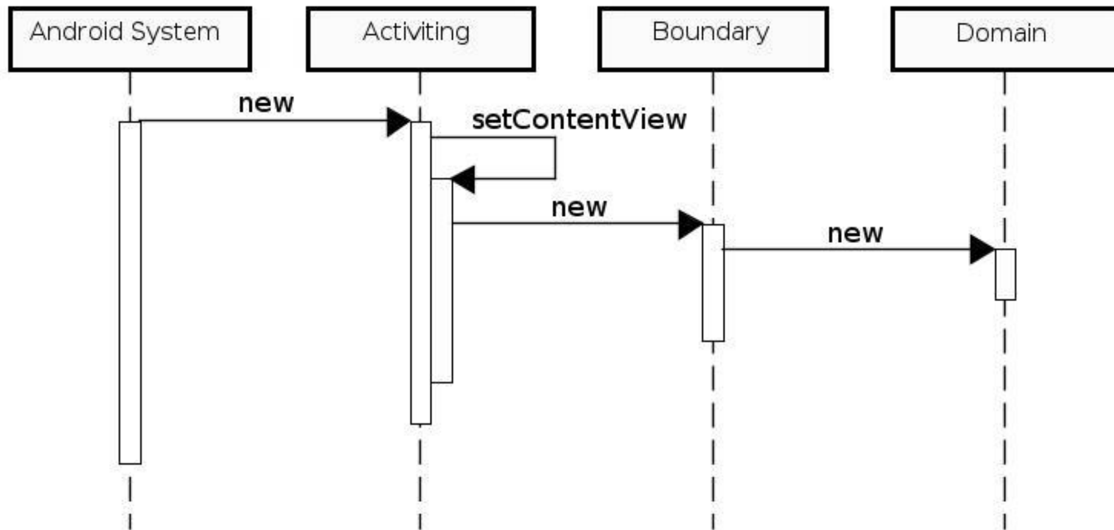

Figure 9. Layers Components creation

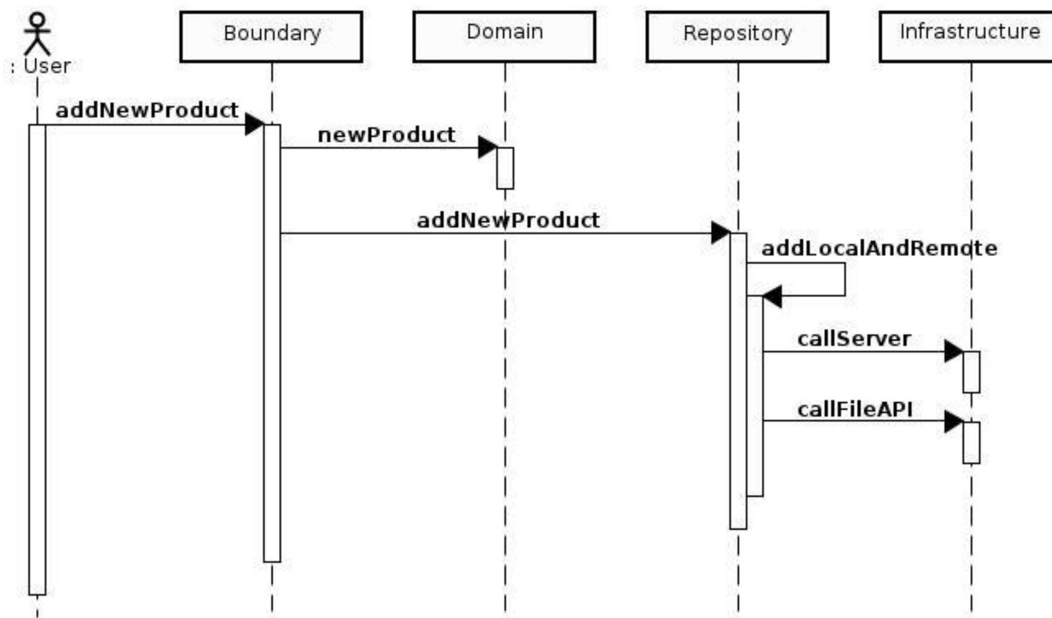

Figure 10. Example of interaction through layers

4. Conclusion and Future Works

In this article, the author has analyzed some of the most prominent architectural patterns in order to find a suitable solution for Android application development, considering the identified non-functional requirements of the platform. Another work addressing the same issue has also been observed, which contributed substantially with the understanding of many concepts related to Android development, and with the conception of the final result.

The next step is to investigate acknowledged methods and parameters that may be used for evaluating the proposed solution, and develop a medium-sized application – that might include other native components – following the principles of Android Layered Application pattern.

References

[1] K. Sokolova, M. Lemercier, and L. Garcia (2013) "Android Passive MVC: a Novel Architecture Model for the Android Application Development", in Patterns 2013, IARIA, Ed., 2013.

[2] Wasserman I., Anthony. 2010. "Software Engineering Issues for Mobile Application Development",http://repository.cmu.edu/cgi/viewcontent.cgi?article=1040&context=silicon_valley, November.

[3] Areppim: information, pure and simple. Mobile OS (Operating System) Percent Market Share – World (As of June 2015), http://stats.areppim.com/stats/stats_mobiosxtime.htm, November.

[4] Android Developer. Android, the world's most popular mobile platform, https://developer.android.com/about/index.html , November 2016.

[5] Georgiev, M., Jana, S. and Shmatikov, V. (2014) "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks", https://www.cs.cornell.edu/~shmat/shmat_ndss14nofrak.pdf , November.

[6] Heitkotter, H., Hanschke, S., and Majchrzak, T. A. (2013) "Evaluating Cross-Platform Development Approaches for Mobile Applications", http://link.springer.com/chapter/10.1007%2F978-3-642-36608-6_8 , November.

[7] Shahbudin, F. E. and Chua, F. (2013) "Design Patterns for Developing High Efficiency Mobile Application", http://dx.doi.org/10.4172/2165-7866.1000122 , November.

[8] Schmidt, D. C. (2006) "Model Driven Engineering", http://www.fing.edu.uy/inco/grupos/coal/uploads/Main/mdepaper.pdf , November.

[9] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) "Pattern-Oriented Software Architecture Volume 1: A System of Patterns", Wiley.

[10] Reenskaug, T. (1979) "THING-MODEL-VIEW-EDITOR, an Example from a planningsystem", https://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf , November.

[11] Reenskaug, T. (1979) "MODELS–VIEWS–CONTROLLERS", http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf , November 2016.

[12] Fowler, M. (2006) "GUI Architectures", http://martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp , November.

[13] Fowler, M. (2006) "Supervising Controller", http://martinfowler.com/eaaDev/SupervisingPresenter.html , November.

[14] Fowler, M. (2006) "Passive View", http://martinfowler.com/eaaDev/PassiveScreen.html , November.

[15] Potel, M. (1996) "MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java", http://www.wildcrest.com/Potel/Portfolio/mvp.pdf , November.

[16] Collected Java Practices (2003) "Package by feature, not layers", http://www.javapractices.com/topic/TopicAction.do?Id=205 , November.

[17] T. Ihme and P. Abrahamsson (2005) "The Use of Architectural Patterns in the Agile Software Development of Mobile Applications",

[18] Android Official Documentation, https://developer.android.com/ , November.

[19] ISO/IEC/IEEE 24765:2010(E), "Systems and software engineering – Vocabulary", https://pascal.computer.org/sev_display/24765-2010.pdf , November.

[20] Lindvall, M., Tesoriero Tvedt, R. and Costa, P. (2003) "An Empirically-Based Process for Software Architecture Evaluation", http://ai2-s2-pdfs.s3.amazonaws.com/74c7/ad68e2012e815350079b89567e2e3b4c1f5b.pdf , November.

[21] Firebase Cloud Messaging, https://firebase.google.com/docs/ , November.

[22] Android Developers Reference, https://developer.android.com/reference/packages.html , November.