Towards High Quality Mobile Applications: Android Passive MVC Architecture

Karina Sokolova*†, Marc Lemercier*

*University of Technology of Troyes
Troyes, France
{karina.sokolova, marc.lemercier}@utt.fr

Ludovic Garcia†

†EUTECH SSII
La Chapelle Saint Luc, France
{k.sokolova, l.garcia}@eutech-ssii.com

***Abstract*—Nowadays, the demand for mobile application development is high. To be competitive, a mobile application should be cost-effective and should be of good quality. The architecture choice is important to ensure the quality of the application over time and to reduce development time. Two main leaders are very represented on the mobile market: Apple (iOS) and Google (Android). The iOS development is based on the Model-View-Controller design pattern and is well structured. The Android system does not require any model: the architecture choice and the application quality highly depends on the developer experience. Heterogeneous solutions slow down the developer, while the one known design pattern could not only boost development time, but improve the maintainability, extensibility and performance of the application. In this work, we investigate widely used architectural design patterns and propose a unified architecture model adapted to Android development. We provide implementation examples and test the efficiency of the proposed architecture by implementing it on real applications.**

Keywords–Smart mobile devices (smartphones, tablets); design patterns; Model-View-Controller; Android architecture model; Fragments; Android passive MVC.

I. INTRODUCTION

This paper is an extended version of the conference proceedings [1].

The mobile market has grown rapidly in recent years. Many enterprises feel the need to be present on mobile markets and propose their services with mobile applications. Compared to computer programs, mobile applications often have limited functionalities, shorter shelf life and lower price. New applications should be developed fast to be cost-effective and updated often to keep users interested. The quality of the application should not be neglected, as mobile users are very pernickety and competition is stiff. Architecture choice remains important for mobile applications to ensure quality: mobile applications, as well as other systems, could be complex and evolve over time.

The demand for smartphone application development is high especially for the two market leaders: Apple (iOS) and Google (Android). Cross-platform solutions, such as PhoneGap, Rhodes Rhomobile and Titanium Appcelerator reduce development time, as one application is developed for several platforms [2], but have limited possibilities – often requiring native plug-ins. Cross-platform solutions also add complexity to the native code (e.g., web layer) that decreases the performance of the application. The support of non-native solutions could be abandoned. Moreover, the cross-platform solution forces having the same user interface for all platforms, while users of different platforms have different habits from native elements. The final application interface that is not common to the platform could be rejected by the user. Native solutions enable use of all the platformÕs options with better performance and lighter code enabling the creation of an application adapted to the platform, therefore developers often choose native software development kits (SDK).

The iOS SDK imposes the Model-View-Controller (MVC) design pattern for the iOS application development [3]. Android requires no particular architecture [4] – developers choose a suitable architecture for their applications that is especially difficult for less experienced developers. Complex applications that do not follow any architecture can end as a 'big ball of mud' code: incomprehensible and unmaintainable [5]. Suitable architecture can improve three non-functional requirements of software structural quality: extensibility, maintainability and performance. A defined architecture could additionally reduce the complexity of the code, simplify the documentation and facilitate collaboration work [6].

Android development books and tutorials are mostly focused on Android SDK technical details and user interface design. Only a few works have been dedicated to the Android application architecture, while the Android community identifies an architecture as an important part of successful system design and development. Developers open many discussions about suitable Android architecture on forums, blogs and groups.

In this work, we provide an overview of some widely used architectural patterns and propose an MVC-based architecture particularly adapted to the Android system. Android Passive MVC simplifies the development work giving the guidelines and solutions for common Android tasks enabling the creation of less complex, high-performance, extendable and maintainable applications.

We provide the detailed pattern description with possible implementations. We introduce several usage scenarios and propose an example of a social networking mobile application 'Tweetle' developed with our pattern. We also discuss the applicability of other presented patterns on Android development, special cases that are relevant to and the difference with

Android Passive MVC implementation.

We evaluate Android Passive MVC regarding the maintainability, extensibility and reusability with scenario-based software architecture evaluation method using the two implementations of 'Tweetle'. We also compare two implementations of 'TaskProjectManager' Android application made for a client by an experienced developer: Android Passive MVC implementation and an old implementation having no defined architecture. We conduct an experiment of long time pattern usage for real Android applications development: two developers applied Android Passive MVC for 10 months on their everyday Android projects and gave their feedback.

The remainder of the paper is organised as follows: the second section presents architectural patterns used in software development. Section 3 presents briefly the architecture used in iOS application development. Section 4 presents the Android SDK and existing difficulties in adapting one known architecture to Android. In Section 5, we propose a design pattern adapted to the Android environment - Android Passive MVC. Section 6 provides some typical cases that may arise while developing an Android application and the corresponding Android Passive MVC implementation. Section 7 describes a concrete example of a social networking application implemented using Android Passive MVC. In Section 8, we go further and provide an architecture for the core of an application. Section 9 evaluates the Android Passive MVC. Sections 10 and 11 discuss the applicability of other architecture presented in Section 2. Section 12 presents works related to mobile applications architecture and Section 13 concludes this work and presents some perspectives.

II. Architectural Design Patterns

We present five architectural design patterns in historical sequence. These patterns are widely used in desktop and web applications development. If mobile development assimilates similar design, developers moving from other systems could take advantage of their knowledge. Different components and existing variants of models are included in the description.

A. Model-View-Controller (MVC)

Presented in 1978, Model-View-Controller is the oldest design pattern and has been successfully applied for many systems since its creation [7][8][9].

The goal of this model is to separate business logic from presentation logic. The business logic modifications should not affect the presentation logic and vice versa [7]. MVC consists of three main components: *Model*, *View* and *Controller*. The *Model* represents data to be displayed on the screen. More generally, *Model* is a Domain model that contains the business logic, data to be manipulated and data access objects. The *View* is a visual component on the screen, such as a button. The *Controller* handles events from user actions and communicates with the *Model*. The *Controller* also communicates with the *View* directly if the *Model* does not need to be changed (e.g., scrolling action). The *View* and the *Controller* depend on the *Model*, but the *Model* is completely independent. The design pattern states that all *Views* should have a single *Controller*, but one *Controller* can be shared by several *Views*.

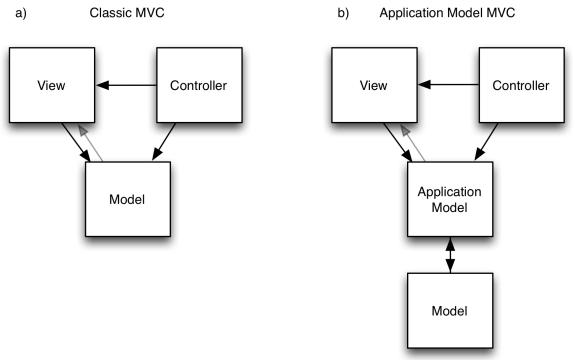

Figure 1. a) Classic MVC, b) Application Model MVC

MVC model has three varieties: Classic MVC, Passive Model MVC and Application Model MVC (AM-MVC). The scheme of Classic MVC and Application Model MVC is shown in Figure 1. The Classic MVC is shown on the left (a) and the AM-MVC is shown on the right (b). The scheme of Passive Model MVC (c) is shown in Figure 2.

In Classic MVC and Passive Model MVC, *Controller* handles events and communicates directly with a *Model* that is indicated by a black arrow. On the Classic MVC the *Model* processes data and notifies the *View*. The *View* handles messages from the *Model* and updates the screen using the data received from the *Model*. This behaviour is implemented using the Observer pattern (grey arrow in Figure 1). Conversely, the communication between the *Model* and the *View* in Passive Model MVC is done exclusively via the *Controller*. The *Model* notifies *Controller* which then notifies *View* and finally the *View* makes changes on the screen [10].

The AM-MVC is an improved Classic MVC with an additional component. The *Application Model* component was added for the presentation logic (e.g., change the screen colour if the value is greater than 4) that was often added to View or Controller previously and makes a bridge between the *Model* and the *View-Controller* couples.

B. Presentation-Abstraction-Control (PAC)

The PAC architecture was introduced in 1987 [11]. This architecture aims to improve the modularity of the system that is limited with MVC. PAC propose to decompose the system functionalities into hierarchically organised cooperating agents

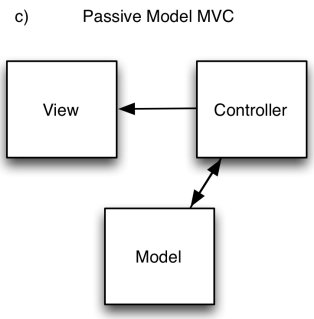

Figure 2. Passive Model MVC

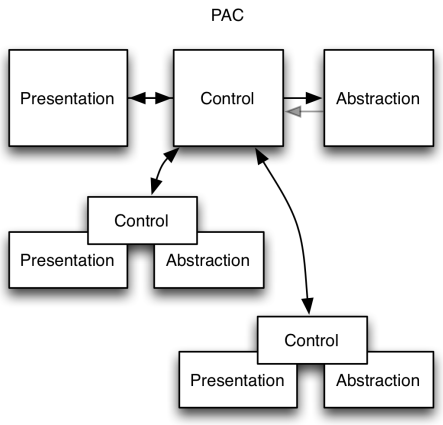

Figure 3. PAC architecture

each responsible for a particular task. Each agent manages the part of the user interface and maintains its data and state. Some agents could also exist without any particular user interface but coordinating other agents. The system can be extended by additional agents, the modification of one agent should not affect other agents.

Each agent of the PAC system consists of three components: *Presentation*, *Abstraction* and *Control*. *Presentation* component contains the presentation logic. *Abstraction* component contains the functionality of the agent and the data it maintains. *Control* component links the *Presentation* and the *Control* acting as an adapter and allows communication between agents. One can see that PAC agent is organised as Passive MVC with the difference that the user events are intercepted by the *Presentation* component [12]. Figure 3 depicts the architecture.

Agents are organised in the hierarchy where lower level agents depend on their parents. High-level agents contains the core functionalities, manage the database and main interface. Low-level agents maintain particular functionalities, particular interfaces, the information about the interface and expose actions to the user. Lower level agents could, for example, manage different sensors. Intermediate-level agents combine, maintain and coordinate low-level agents.

The actions intercepted by the low-level agents can be redirected to the upper agents to access their functionalities, the outgoing events such as an error event is also transferred to the particular 'error manager' agent via parental *Control* components. The changes in high-level agent's data are also transferred to collaborators agents.

This architecture allows a very modular system to be made with communicating agents but the system can become very complex with the fast growing number of agents. The organisation or communication between agents could also become complex.

C. Model-View-Presenter (MVP)

The Model-View-Presenter was introduced in 1996 as an MVC adaptation for the modern needs of event-driven systems [13]. The model consists of three components: *Model*, *View* and *Presenter*. In this model, the *View* represents a full screen

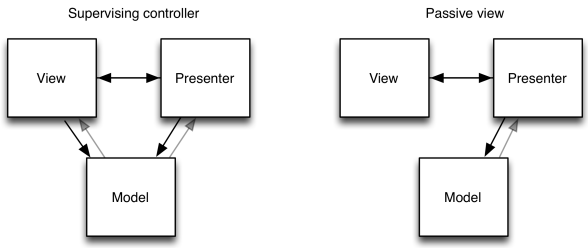

Figure 4. Supervising controller and Passive view

and it handles events from the user actions. The *Presenter* is responsible of the presentation logic. The *Model* is a Domain model.

There are two types of MVP: Supervising controller and Passive view. Both models are shown in Figure 4. The Supervising controller uses the Observer pattern for the communication between *Model* and *View*. The *View* can interact directly with the *Model* to save the data if there is no change to be made on the screen. Otherwise, the communication between the *View* and the *Model* is made via the *Presenter*. Interaction between *View* and *Model* of the Passive View MVP is done exclusively via *Presenter* [13].

D. Hierarchical-Model-View-Controller (HMVC)

The Hierarchical-Model-View-Controller was first introduced in 2000 and is similar to PAC architecture. HMVC is presented as a Classic MVC adaptation for Java programming [14]. This model takes into account the hierarchical nature of Java graphical interface components: the main window frame contains panes that contain components. The authors propose to create layered architecture for the screen with Classic MVC triads for each layer communicating with each other by *Controllers*. The HMVC model is shown in Figure 5.

Thereby the child *Controller* intercepts methods from its *View*. If a *View* of the upper hierarchy (parent *View*) needs to be changed, the child component informs the parent *Controller*, which makes the changes. The communication between layers is made exclusively via *Controllers*. Unlike PAC, the *Controllers* of HMVC have direct access to the *Model* and to core components without interacting with the high-level triad.

E. Model-View-ViewModel (MVVM)

Model-View-ViewModel is another model to separate the presentation and business logic. The *ViewModel* is a linking component between *View* and *Model*. This design pattern is mainly used in Microsoft systems [15]. The realization of this

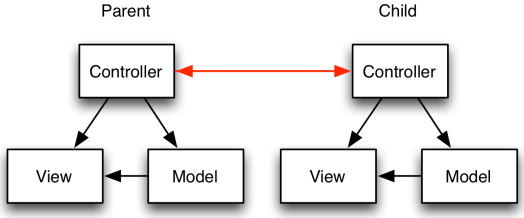

Figure 5. Hierarchical-Model-View-Controller

model is done with binding between components [16]. The binding is not supported in Android by default but could be implemented using the very recent Android-binding framework. As stated in [17], a good basic model should not use any additional framework and should be easily implemented with original components, therefore this model is not dealt with in the paper.

III. iOS APPLICATION DEVELOPMENT

The iOS mobile development has already adopted an architecture. We want to take advantage of iOS experience and knowledge in making the Android architecture. In this section we present the main principles of the architecture used in iOS development.

iOS is a OS X based system adapted to mobile devices. iOS developers use specific language called Objective C to create mobile applications.

The base architecture for mobile iOS application is an adapted Passive MVC. Like the original Passive MVC, the iOS architecture is based on three components: View, Model and Controller. Models and Views are independent and communicate with each other only via Controllers. The communication between Controllers and Model is organised via an Observer-Observable pattern.

Views and Models are highly reusable. Multiple Views are already provided by Apple: SplitView, TableView, ImageView, PageView, CellView, WebView, MapView, TextView, ButtonView, etc. Controllers are less reusable, they link Views with Models, set up the Views (contain presentation logic), and intercept actions made on View to call methods from the Model. Many controllers are already predefined in iOS: ViewController, SplitViewController, TableViewsController, etc.

A main controller for each screen or a group of screens exists in iOS applications. For example TabBar represents a menu and there are as many screens as tabs in this menu. All screens are managed by the same controller - TabBarController. Each screen can embed other Views that can have a corresponding Controller or can be managed by the parent Controller.

One can see the logic of iOS applications is similar to Android applications; knowledge of iOS architecture is helpful to adapt an Android architecture.

IV. ANDROID APPLICATION DEVELOPMENT

A. Background

Android is a Linux-based open source operation system designed for mobile devices. Android was first presented by Google in 2007 and in spite of huge competition from Apple has been the leading smartphone platform since 2010. Google continues to work on the system systematically integrating new features and correcting bugs. Many manufacturers of smartphones and tablets adopted this open-source solution; the National Security Agency (NSA) and National Aeronautics and Space Administration (NASA) also choose Android for their projects.

Android applications are mainly written in Java using the Android SDK [18]. The code is compiled to be executed on the Dalvic virtual machine on a smartphone. Additionally, developers can use the Native Development Kit (NDK) to add a C or C++ written code referred to as native. NDK allows more advanced features and better performance, however, the complexity of the code increases with the quantity of native code [19] – Google suggested minimizing the use of this kit.

Four principal components of Android SDK are used in Android application development: Activity, Service, Content provider and Broadcast receiver. Developers use predefined extendable classes to implement those components.

Activity is a main mandatory component of Android applications created when the application is opened. The simplest Android application can contain the only class implementing the Activity. Activity is also the entry point to the application: to start the application the system must launch the Activity component. Applications can make the Activity public to share the functionality it proposes.

Many Activities can exist in the application but only one is active at a time. The Activities history is saved: the system automatically maintains the stack of Activities and opens the previous Activity with its last state when the button 'back' is pressed. The oldest Activities are deleted from the stack for other memory usage.

The Service works on the background of an application permitting an execution of long tasks (e.g., file download) without freezing the screen. When the application is closed, unlike Activity, the work of the Service is not interrupted. Services can communicate directly with the Activity it is attached to.

The Content provider component gives access to the local data stored in SQLite databases. Content provider is aimed to be used for the data sharing between applications but can also be used internally.

The Broadcast receiver is a messaging system that enables communication inside the application and between multiple Android applications installed on the phone.

In 2010, Google introduces a new component into the Android systems called Fragment. Fragment is a new extendable class available in the Android SDK. Visible interface elements can now be controlled by Fragments instead of Activity, which permits the elaboration of more flexible interfaces. Therefore, part of the interface can be changed by treplacing one Fragment with another Fragment. Each Fragment is attached to the Activity and maintains access to the Activity.

Fragments main intention was to simplify the adaptability of an application between smartphones and tablets where two screens on a smartphone can become a single screen on a tablet due to the size difference. Fragments increase the modularity of the Android applications.

An exhaustive description of Android development environment and modules can be found in [20].

B. Experience

Activity causes major difficulties in implementing the known architecture: is it a *View*, a *Controller*, a *Presenter* or

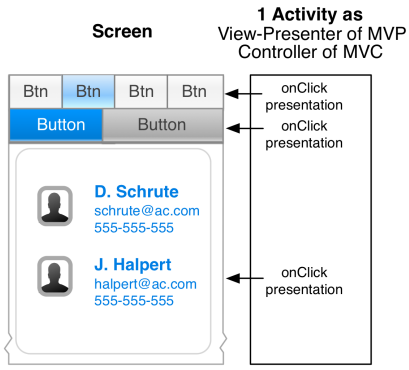

Figure 6. Activity as 'View-Presenter' of MVP or a 'Controller' of MVC

none of them?

One can observe that Android SDK already integrates many simple *Views* such as Button, TextView, ImageView, EditText and also more complex *Views* such as ListView, AdapterView, etc. One can also find several *Controllers* such as ViewFlipper, ViewSwitcher, etc. *Views* can be combined together on the screen by layout.xml and even embeds other *Views*, defining the appearance.

The most common way to develop Android application is to create one Activity per screen. Naturally, Activity initialize *Views* and intercepts actions made on *Views* by the user (methods corresponding to actions could be directly defined in layout.xml, Activity should implement the defined methods). Presentation logic of the full screen and a communication with the core of an application is often situated in the Activity making it very heavy and complex [21]. Thereby Activity managing actions and the presentation logic of the full screen behaves as a *View-Presenter* couple of MVP or a big *Controller* of MVC. The simple schema is shown in Figure 6.

We also found examples where a single Activity manages an entire application: all possible actions of an application and the full presentation logic is managed by only one class - Activity.

The *View-Presenter* or thick *Controller* implementation leads to multiple problems: reutilization, maintenance, extensibility, code clarity, team development and even performance. Parts of code integrated into single Activity cannot be reused, methods can only be copied to another Activity making the redundant code. Any additional *View* and action complicates the Activity. A modification of one action repeating on several screens requires modification in all related Activities (assuming one Activity per screen). Activity can contain the implementation of very different actions non related to each other, this can make the Activity very complex, unreadable and incomprehensible. Activity is kept in memory while the application is running, thereby a very big Activity affects performance. Finally, the modification in the user interface and application logic can lead to the need of full redevelopment of all Activities.

Some developers improve the architecture placing the Activity as a MVP *View* and put the presentation logic to the *Presenter* component. It makes Activity lighter as it manages only actions available on one screen, but reutilization and maintenance problems remain the same as explained above. The simple schema is shown in Figure 7.

Another MVC implementation place Activity instead of *View* and creates the Controller separately. Activity cannot be implemented as a *View* due to the particularity of the component, but Activity can initiate and regroup all *Views* on the screen. Thereby we obtain very thin Activity and thick Controller handling all screen events and managing the presentation logic. The simple schema is shown on Figure 8.

Even if *Controllers* could be reused by other Activities the full object is needed to reuse methods related to one View from the previous screen; the structure of application becomes unclear due to the reutilization. Problems of extensibility and maintenance persist.

This solution works for simple applications where one Activity represents one visual block, while Activity usually manages several *Views*: main screen, menu, dialogue box, lists, forms, etc. In complex visual applications *Controllers* becomes heavy.

Assuming the Activity cannot be a *View*, as *Views* are already available and extensible on Android, few developers replace the MVP *Presenter* with Activity. The simple schema is shown in Figure 9.

This solution makes *View* intercept event of all visual components available in the screen; presentation logic moves to Activity, but similar problems appear: reusability, extensibility, code clarity, etc. Presentation logic cannot be reused, but should be copied to another Activity if needed. The complexity of a single *View* increases with the number of events. This is suitable only for very simple applications with very simple screens.

The appearance of Fragments could have solved the architecture ambiguity, but Google proposes new components without suitable documentation about the utilisation of Fragments, thereby creating new ambiguity instead of solving the problem. Now developers ask themselves in what cases they should use the Fragments and not the simple Activity, what component should handle actions and presentation logic, where to place the Fragment management code, etc. We find previously explained MVC/MVP solutions, where the component that is not implemented as Activity becomes a Fragment (e.g., MVP implementation where Presenter is implemented as Activity and View is implemented as Fragment).

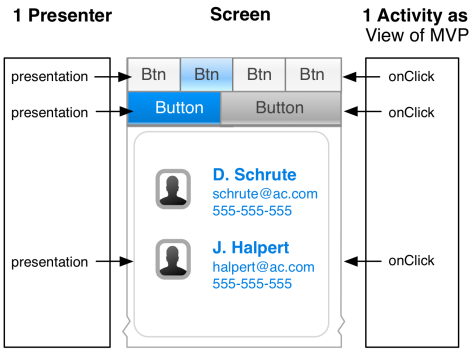

Figure 7. Activity as 'View' of MVP with additional Presenter component

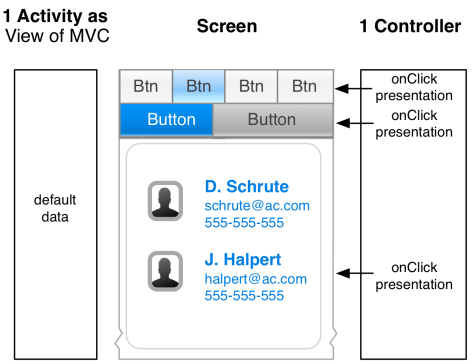

Figure 8. Activity as 'View' of MVC with additional Controller component.

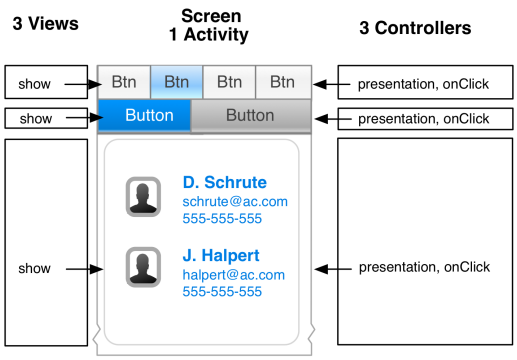

Figure 10. Activity as an intermediate component between Views and Controllers.

Nowadays more and more developers use Fragments, often as *Controllers* of MVC, but questions about presentation logic, communication between components, the actual purpose of components and its logic remains unanswered.

The full code organisation needs to be clarified: what existing component should be used and for what purpose, what type of code can be placed in those components and when and for what purpose should additional components be created? We find many applications where the part of core logic of an application is placed in the Activity or in the *Controller/Presenter* making them even more complex. Developers are often unsure about the decomposition of an application to Activities and Fragments and have problems in core organisation.

We did not find any Android application example developed using HMVC or PAC architectures. The implementations of MVVM requires additional libraries, therefore we do not take them into account.

V. Android Passive MVC : Presentation

Even if MVC and MVP architectures seem suitable for Android developments they are not intuitive to implement. The main defined problem is an Activity component that is hardly reusable. We aim to define a new architecture that can be easily implemented with Android-specific components, such as Activity and Fragments. The implementation of the model should improve the application and code quality: reduce the complexity of an application, clarify the code and improve extensibility. The coupling between components should be weak to avoid the modification of other components if one is modified. Modules should be reusable [17][22]. A mobile phone has a limited memory, therefore the creation of unnecessary objects should be avoided. Objects remaining in the memory should be lightweight [19]. Modification in user interface or in navigation logic should involve the minimum modification of the application.

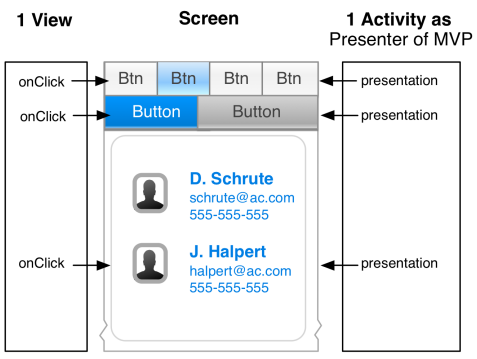

Figure 9. Activity as 'Presenter' of MVP with additional View component.

In this section we present in details our proposition: the architecture for Android application development we named Android Passive MVC.

We have decided to base our architecture on the MVC model, as MVC is well-known and widely used in desktop and web systems as well as in iOS mobile development. Developers coming from other systems would be able to easily appropriate the Android development architecture.

Activity is an inevitable component of the Android application. Previous experience of the Android community shows Activity does not fit well on the MVC model, while it seems to be well adapted to developers' needs. Many View components are already available on Android but Activity cannot be a Controller or a Model. From the previously described development experiences one can see that the screen cannot be represented entirely by one or two components. It becomes trivial that the screen should be decomposed into many logical parts and each part should have the related components. For the new architecture we decided to create MVC triads around Activity making the Activity the fourth component.

We can think of Activity as a main screen (parent) controller in HMVC model. The simple schema is shown in Figure 10.

An observer-observable pattern is relevant for multi-screen systems but only one screen is active at a time in Android applications. This pattern implies keeping in memory Views and Models that appear heavy for the mobile environment, therefore we chose the Passive Model MVC as a basis for our architecture.

In our model, Activity becomes an intermediate component between the Views and the Controllers. The Activity represents a screen controller or, in some cases, a main controller for a group of logically conjoint screens.

Controllers take the event handling responsibility and the presentation logic making the Activity lightweight. Controllers are also lightweight because one Activity can interact with many small reusable Controllers. Controller handles events and presentation logic only for a small number of views logically linked together. Controllers should not contain any code related to the core application functionalities.

The Views are the interface components, such as a form, a menu or a list of elements. View components contain methods that allow the setting or obtaining of data from the user interface on Controller demand, the setting of event listeners on visual components and the modification of visual components (set errors, change colours, etc.). Views are created if necessary extending Android predefined Views, otherwise the Android predefined Views fit to the architecture. Views are independent and do not communicate. Views should not contain any application logic or data.

The Model in our architecture is a Domain Model containing the application core logic and data. The simple scheme of the Android Passive MVC architecture with all components is shown in Figure 11.

The starting Activity creates a link between a View and a corresponding Controller to make them communicate directly. Controller set up the View it is responsible of: visual presentation and the data. The Controller handles events from the user action (e.g., button click), calls necessary methods from the Model and then updates the View on Model response.

Simple hierarchy of Activity and Controllers depending on this Activity will be suitable for many simple applications, although Android interface is a modular interface similar to Java. We propose to organise View-Controller couples in Hierarchy as HMVC and PAC architectures. The actions of interface modules controlling another interface module will be organised as parent-child controllers.

We define two type of controller: Mediate Controller and Coordinating Controllers. We borrowed names from iOS architecture also having two types of controllers.

Coordinating Controller is a simple Controller for an independent part of the screen coordinating the presentation logic and events of its Views. This Controller can call the Model, modify its View visualisation, show dialogues, call Activities but does not exchange Controllers. The Coordinating Controllers do not have any child controllers. Coordinating Controllers are very reusable and make an application very modular. Coordinating controllers can be perceived as low-level PAC agents.

Mediate Controller often corresponds to the part of the interface modifying or exchanging Coordinating Controllers (part of the interface). Menus in the interface would often

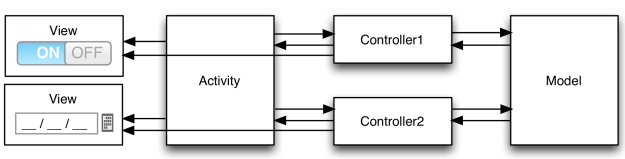

Figure 11. Android Passive MVC

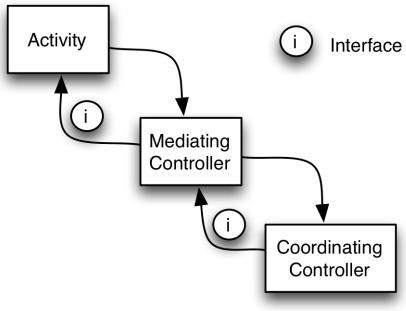

Figure 12. Communication between Controllers

correspond to the Mediate Controller. Mediate controllers can also initialise child Mediate Controllers (e.g., for a submenu). Activity Mediate Controller manages Activity replacement. Mediate controllers are similar to the Intermediate-level PAC agents with the difference that they have direct access to the Domain Model (application core).

Mediate Controllers are not very reusable as they need all their children to function, although Mediate Controllers show the presentation logic of the application; the logic of the interface can be modified by updating or changing the Mediator controller.

To keep components loosely coupled it is recommended to ensure communication between Controllers and Activity via interfaces. The communication schema is shown in Figure 12.

Android Passive MVC makes Activity lightweight by moving all event handlers and presentation logic to Controllers and interface management to Views. Views and Controllers created on demand avoid unnecessary objects, saving memory. Android predefined View fits the model and new developer Views are reusable in future applications. Coordinating controllers are very reusable and makes the application very modular. Mediate Controllers are less reusable but enable easy modification of the logic of the application only by modifying the Mediate Controllers.

Developers can easily modify or remove application components by only updating or deleting the corresponding View-Controller couple. Application can be extended with View-Controller couples. The Model is independent from the View, the Controller and the Activity. The user interface could be replaced without any impact on Model, therefore the maintainability of the application is high.

VI. Android Passive MVC: Implementation

This section presents some examples of Android Passive MVC implementation. We introduce more details and special cases of architecture usage. Controllers of AP-MVC can be implemented with simple Java classes or with the Android Fragment component.

Both implementations are suitable for the new manually created Activities. Some predefined Activities, especially from third-party libraries, will possibly not fit the implementation.

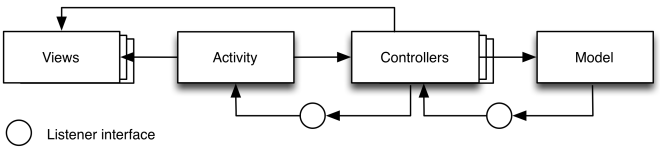

Figure 13. Android Passive MVC implementation

A. Java classes implementation

Controllers can be implemented as simple Java classes, the same as Views. Controller should be linked to the Activity, therefore the Activity should implement a Controller listener interface and pass it to the Controller to establish the communication. The components communicating via listeners are loosely coupled and the communication of Android components via interfaces is presented in [23].

As the Activity would initialise the Controller, it can communicate with Controller directly, but the communication via interface is preferable. Activity should also retrieve the View and pass this View to the Controller to establish a direct communication. We propose to establish the communication between the Controller and the Model via listeners (interfaces).

Figure 13 shows the Android Passive MVC implementation diagram. Listeners increase the performance of the application and create a weak coupling between components that improve maintainability.

For the example showing the implementation without Fragments we created a login screen with a classic login form to enter the login and password; if the login is successful the user goes to the welcome page, otherwise an error message appears.

The example contains two Activities: Login Activity managing the login page and Welcome Activity for the welcome page. The login form is managed by Login View and Login Controller. Login Activity implements the LoginControllerListener interface to be able to receive calls from the Login Controller. The schema is shown in Figure 14.

Login View contains methods for obtaining login and password (getters), methods to set button listener and methods to set errors. Login Controller handles events from the login form implementing the onClickListener; while the button is pressed, Controller calls the model that launches simple verifications. If login is successful, Controller opens a welcome screen. If login fails Controller sets up an error message.

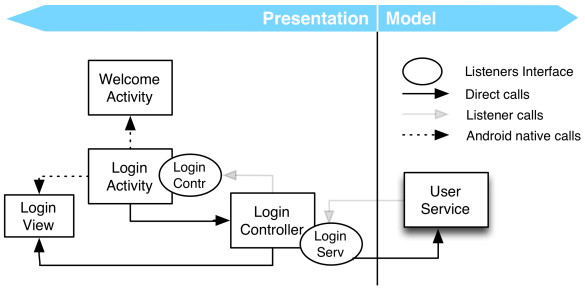

Figure 14. Login implementation example

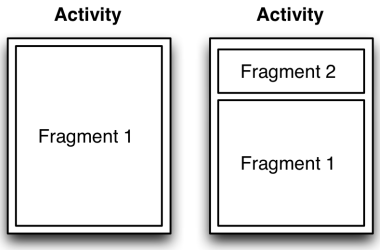

Figure 15. AP-MVC impose the creation of Fragment event if the only one is currently used within Activity

B. Fragments implementation

Fragments is an Activity-like component that can represent and control a part of the interface. Fragments can be used to implement Controllers in Android Passive MVC. Since the introduction of Fragments, Google insists on the high usage and integration of Fragments into Android applications and deprecates Activity-based functionalities.

Fragments propose multiple advantages in Android Controller implementation versus simple Java classes:

- Fragments are native Android components automatically linked to the Activity via layout.xml having the native possibility to communicate with the Activity.
- Fragments have their life cycle linked to the Activity.
- Fragments are automatically linked to Views via layout.xml and can retrieve Views to communicate directly.
- Android integrates the Fragment manager: Fragments can be easily replaced, deleted or added to the Activity.
- Activity has access to all attached Fragments.
- Fragments integrate the back button gesture: option of saving the Fragment with its state in the back stack and retrieving it on back button press. We can also choose to retrieve the existing Fragment with its state or create a new Fragment with the default state.
- Fragments can manage other Fragments.

A Fragment is created for each piece of an interface having an action or several logically linked actions. We propose to distribute all actions between Fragments and do not add user actions directly to the Activity. Even for only one simple form (e.g., login form) the Android Passive MVC imposes the use of the Controller (Fragment) along with the Activity. This makes the application more modular and improves maintainability, the same independent Fragment can be easily reused in the future. Figure 15 shows a single Activity with a single Fragment and a single Activity with two independent fragments.

Fragment is linked to corresponding Views via the layout.xml. Fragment should not retrieve other Views available in the Activity to stay independent.

Fragment can play the role of Mediate Controllers and manage other Fragments or change Activity. One Fragment cannot exchange itself with another Fragment therefore it

needs a parent Fragment (Mediate Controller) to perform the transaction. Figure 16 illustrates an example.

Android gives the option of adding a Fragment that is not directly linked to the interface, permitting the creation of Mediate Controllers without visual components. In some cases, actions from different screens and different Activities can be combined in one single Mediate Controller if those screens are logically linked. For example, a form can have several pages (screens) with 'next' button or 'go to first page' button; the appearance of the screen changes on click event interception. In this case, rather than adding an action to each fragment separately, making them dependent, the developer should create a Mediate Controller combining those actions in one place. Figure 17 illustrates this example. Therefore, in the case of user interface reorganisation (e.g., add new screen in the middle of the chain) only the Mediate Controller needs to be modified. The same should be done for a bundle of dependent fragments within a single Activity.

Fragment initialises itself with default data or the data recovered from the bundle (Android mechanism to pass the data between Activities), therefore Fragments stay maximally independent from other Fragments. Some Fragments can be initialised by an Activity or parent Fragment (Mediate Controller) to increase reusability. The possible communication between Fragments and Activities is shown on Figure 18 with two Mediate Controllers and one (the upper right) Coordinating Controller. Fragments should rarely have a callback to parent Fragments but if necessary the callback can be implemented with interfaces.

Developer should avoid high hierarchy between Fragments within a single Activity as parent Fragment is linked to the child Fragments. Activities make components more independent and simplifies the Fragment management.

In some cases, Fragments depend on each other (cannot be a parent-child, but should initialise each other), we observe the circular dependency between Fragments. Figure 19 shows an example: a list of folders and a file path to the parent folder. By clicking on the folder in the file path, the folder list should be updated; by clicking on the folder in the list, the file path should be updated. Another example is a mobile tablet with a large screen that contains the statistics data represented in different Views: tables or graphs. Changes in any of the Views should affect all other Views.

This is also a typical case where the Classic MVC is very pertinent where the Observer-Observable pattern can be used

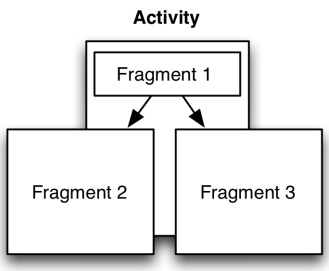

Figure 16. Mediate Fragment corresponding to the possible menu that exchange 2 Fragments depending on intercepted action

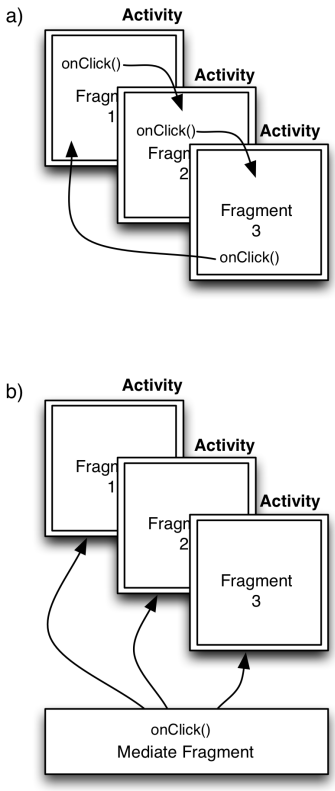

Figure 17. A chain of dependent Fragments or Activities. a) Direct calls make dependent Fragments b) Mediate Controller makes components independent

instead of Mediate Controller: several Views represent the data using the same Model, Controllers can modify the Model and all Views should be updated. Although Mediate Controller keeps components more independent.

It is possible in Android to retrieve one Fragment from another Fragment and to call the initialisation method. Although this makes very tight coupled components. A better way is to make those Fragments communicate via listeners implemented by a parent component: a Fragment playing the role of Mediating Controller.

VII. Concrete Application with Android Passive MVC: 'Tweetle'

To illustrate the implementation mechanism we take an example of a Twitter client (microblogging social network) with three buttons (main menu); one screen has an additional

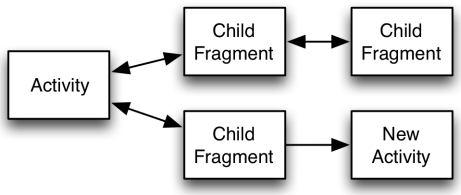

Figure 18. Communication between Fragments and Activity

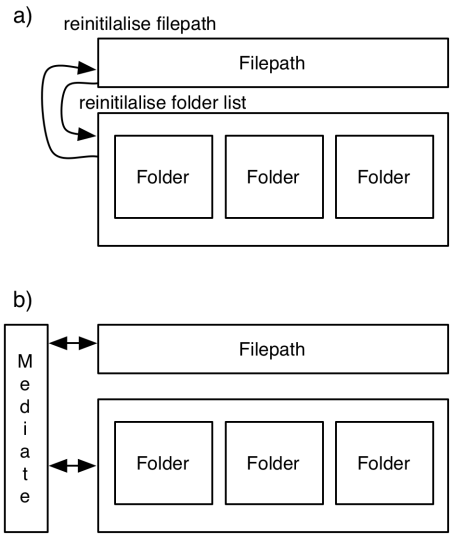

Figure 19. Circular dependency between Fragments a) Direct calls, dependent Fragments b) Mediator Controller makes Fragments independent

submenu. The user can see the Twitter timeline, send tweets with and visualise the list of tweets of his followers and followees. The bar with the copyright button showing the application authorÕs name appears permanently on the up of the screen. bar with the copyright button showing author's name of the application appears permanently on the up of the screen. The interface of 'Tweetle' is depicted in Figure 20.

One can see that all three screens are logically linked together by the main menu; one screen is divided into two logically linked parts by the submenu. Main actions are clicks on the main menu and clicks on the submenu. Additionally, by clicking on any of the list, a user can retweet the message. Finally, the button sending the tweet is presented on the last screen.

One can notice that we need two Mediate Controllers for the main menu and a submenu and at least two Coordinating Controllers for the copyright bar and the list of tweets.

Application can be implemented in two ways. We present

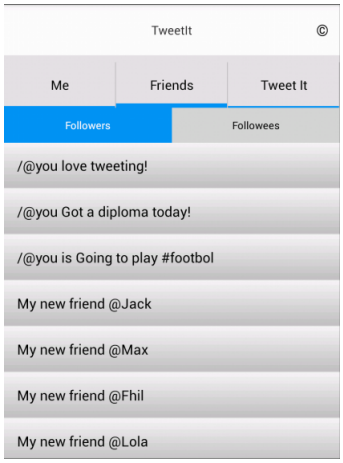

Figure 20. 'Tweetle' application user interface with followers/followees messages active tab

both implementations and discuss advantages and disadvantages of each.

1) First implementation: each different screen interface is manages by a separate Activity. This possibility is similar to "before fragments appears" implementation solutions. In our example, the main menu imposes three Activities with three buttons. One can also suppose to create one Activity by submenu or to keep the single Activity for both submenu actions as we did in our implementation example as modifications on the screen are minimal.

The screen can be decomposed into four Fragments: mostly repetitive elements on the screen. Fragment should only contain the presentation logic and actions that are logically linked together. Different actions like "retweet" and "onMenuPressed" need different Fragments.

The bar containing the copyright button corresponds to a copyright Coordinating Controller (Fragment). This controller is highly reusable even for different applications of the same developer. The bar and the button can be personalised with an layout.xml, but the Controller containing copyright action calling the dialog or a new Activity can be reused exactly in the same way in another application.

The second Fragment is a main menu Mediating Controller. This controller will change the screen (Activity) depending on the button pressed. Controller also manages the presentation of the main menu: active and non-active buttons.

The third Fragment is a list Fragment: retweet Coordinating Controller. We can reuse the same Fragment for all lists as the user action is the same for all lists of the application and there is no presentation logic.

The fourth Fragment corresponds to the submenu Mediating Controller and manages the changes in the data of the list (reinitialise the data or change the Fragment) and the presentation logic of active and non-active buttons.

Last Fragment corresponds to the form permitting to send the tweet - tweet Coordinating Controller.

Activity plays the role of an initialiser of child Controllers or a main Mediating Controller. Mediating Controllers can also initialise themselves using the data from the bundle to be more independent. Copyright Fragment is attached only by the layout.xml and do not need any additional initialisation. Activity initialise the main menu: call the Fragment method to set up active button and event listeners. Activity also initialises the list of tweets of the first screen: Activity as a main Controller can call the Model to retrieve the data and to set it to the list Coordinating Controller. List Coordinating Controller (Fragment) can retrieve the data itself either. For the Followers/Followees screen the submenu Mediate Controller (Fragment) with its default state is attached automatically to the Activity. Submenu Fragment initialises the list of tweets. Initialisation calls are depicted in Figure 21.

2) Second implementation: create an Activity for a group of logically linked screen. In our example we only have one Activity. Fragments remains the same: one submenu Fragment, one list Fragment and 'send message form' Fragment.

Menu actions can be implemented either in the Activity or a Mediator Controller (Fragment). We suggest to keep the

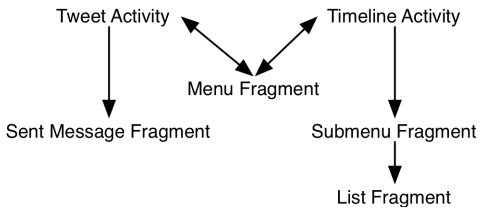

Figure 21. Activity by screen initialisation calls

main menu independent in the Fragment instead of adding it to the Activity directly to enforce maintainability. Main menu Fragment manages clicks on buttons, apply visual modifications on buttons and exchanges other visible Fragments. The submenu Fragment have an child list fragment to manage: the information shown by the list depends on the action made on the submenu. Initialisation call schema is depicted in Figure 22.

3) Both implementations: are very similar but have advantages and disadvantages.

The first implementation is easy to set up and keeps the structure clear. Activities are nearly empty thereby the only active fragments take place in memory, the number of fragments is also limited and easy to manage. 'Back' button is manages automatically. Android integrates a bundle mechanism allowing information to pass between Activities; Fragments could initialise themselves retrieving the information from the bundle while the Activity is changed. Otherwise, this implementation is only suitable for lightweight interfaces as all Fragments are reinitialised for each screen. The time response increases significantly if heavy images appear on the interface.

The second implementation has a clear structure but could be trickier to manage. This solution permits to reinitialise only necessary fragments, therefore can be used with more heavy static images, for example with the background image. Although, developer should assure to keep in memory only visible fragments. A large number of Fragments managed by a single Activity can be complicated and heavy if all Fragments are kept in memory. Fragment should be manually added to the back stack to manage the 'back' button. The second implementation is also useful for Activities aimed at being shared and at returning messages to other applications: this type of functionality should be implemented within a single Activity that another application could call for result.

VIII. Android Domain Model

The clear separation of presentation and business logic cannot ensure the application of good quality alone. The core of the application should also be implemented through patterns and gold architectures. Android application business logic structure is similar to any Java application core logic, therefore all patterns that can be applied to Java could also be applicable to the Android Domain Model, although we observe difficulties in Android Domain Model organisation.

In this section we go further and give some guidelines of the business logic of the application – the Model. Android applications have similar needs: internal database management and access, web service access and reusable components use. Clear main architecture of business logic is necessary to obtain the application of quality.

The Model of Android Passive MVC is a Domain Model containing business methods, web service call methods, database access objects, reusable methods and data model objects.

A Domain Model architecture should include components that are usual for Android applications, such as Database manager, Web services manager and Business logic. Those components should be independent, as the architecture should be adaptable. Reusable components should be also separated. The basic model architecture is shown in Figure 23.

The architecture of Domain Model proposed in this document is inspired by 3-tier architecture that separates the presentation, the business and the data access layers [24].

The business layer of our model regroups objects and methods that use web services, business services and reusable tools. Business services contain business logic. If an application works via Internet as well as locally, all necessary verifications are done in Business services, which calls corresponding methods. The communication between a presentation and a domain model layer are made via Business services.

The data layer contains Models, Data Access Objects (DAO) and Database Manager. DAO and Model are the implementation of the Data Access Object pattern. Model contains data being persisted in the database or retrieved by web services calls. Model is a simple Plain Old Java Object (POJO) that contains only variables and their getter and setter methods. To avoid transcription of the Android Native Cursor object to Model objects, Model can encapsulate the Cursor object proposing getters and setters for a concrete value type available in Cursor. Data is manipulated and transferred through the

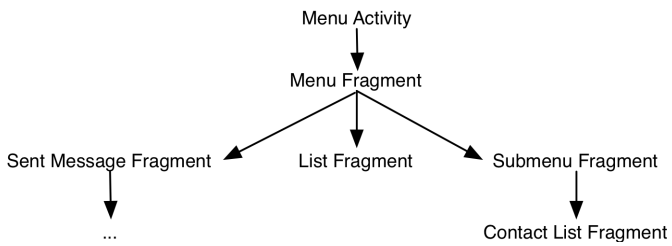

Figure 22. Activity as Main Menu

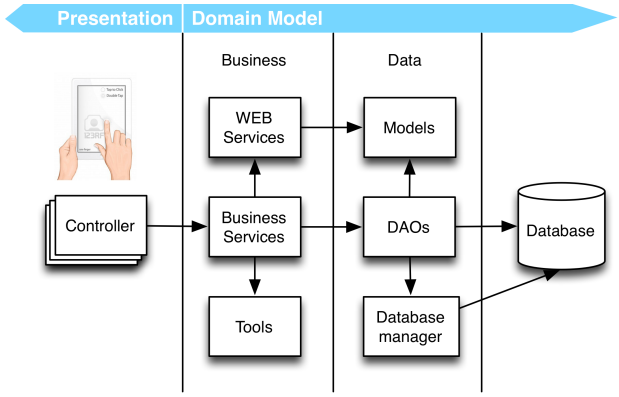

Figure 23. Domain Model Architecture

application using those lightweight objects that are often called Data Transfer Object (DTO).

Persistence methods are organized in DAOs. DAO contains methods that enable the data in a database to be saved, deleted, updated and retrieved. Even if Android proposes an abstraction on the data access level with Content Provider, DAO simplifies the code of the application. The DAO design pattern creates a weak coupling between components and uses a Model object instead of an Android Cursor object in the application. DAO can also be used for the data stored in XML or text files. Good practice is to make DAO accessible via interfaces. It allows DAO modification (for example the change of SQLite to XML storage) without any change in Business services, which increases maintainability.

Database manager is in charge of database creation. Database manager exists only if SQlite database is used by the application. It stores the name of the database, and of its tables and methods to be able to create, drop, open and close the database.

This architecture regroups logically similar methods together, increasing cohesion. High cohesion facilitates the maintainability of the software. The final code of the application could be organized in packages by architectural component: Activities, Views, Controllers, Business Services, Tools, Web Services, Model, DAOs and Database. It gives the clear structure of an application and limits the package number. Additional packages could be created for interfaces, parsers (e.g., XML, JSON) and constants.

IX. Architecture evaluation

We evaluate the architecture in two steps. First, we ensure that the architecture fits the lists of code quality criteria proposed by [17][19]. Second, we propose modification scenarios that can be applied to the 'Tweetle' and discuss the impact of each scenario on the implementation. Third, we ask an experienced Android developer to rewrite one of his latest applications using Android Passive MVC, compare results and give feedback regarding the model. Finally, we proposed to two developers that they use the architecture for 10 months in their real life projects and obtained their feedback.

A. Code quality

The evaluation of our architecture is based on the following three code quality evaluation criteria: maintainability, extensibility and reusability.

1) Maintainability: option of modifying the system.
2) Extensibility: option of adding new functionalities to the system.
3) Reusability: option of reusing the same components in different functionalities of the system or in different systems.

The use of standard platform techniques is important for the model: the support of third-party functionalities could be interrupted making implementation of the model impossible. The Android Passive MVC could be implemented using Android SDK without any additional libraries.

A high-quality application has high maintainability and extensibility: codes have weak coupling between components, easy code suppression possibility and high testability. The Passive MVC architecture ensures high maintainability. Clear separation between presentation and business logic simplifies testability of components. Weak coupling between all layers is carried out via listeners. One component (ex. interface, DAO, web service) could be replaced or modified without changes in others. The extension or modification of the user interface itself is done by simply adding, deleting or modifying the view-controller couples.

The reusability of components make the code clearer and boost development time. The view-controller components of the Android MVC model could be reused through the application and could be easily embedded in other Android applications made with Android Passive MVC.

Good performance is especially important in mobile environments: resource utilization should be limited as mobile devices have little memory. Short response time is essential for modern users. The Android MVC architecture makes a very lightweight Activity component. Controllers, View and Model objects are also small and kept in memory only if used, which minimizes resource utilization. The use of listeners also slightly increases response speed.

B. Scenario-based evaluation

We chose the scenario-based software architecture evaluation method to validate Android Passive MVC; the overview of such methods can be found in [25]. Scenarios enable evaluation of the architecture of a specific system and comparison of several architectures of the same system regarding modifiability. We apply scenario-base evaluation to previously described implementations of Android Passive MVC to show the benefits of this architecture. Most scenario-based methods involve shareholders, software designers and an evaluation team for the real project to define possible modification scenarios and the ability of the architecture to support those modifications. Our example is an illustration of the architecture, we define the most likely modification scenarios for the implementation. The two architectures of 'Tweetle' are described in Section 7.

1) Adapt the phone version to the tablet
2) Add new tab to the main menu
3) Move the main menu to the separate independent screen
4) Modify the appearance of the list
5) Add a bar containing the name of the active tab

We analyse and explain the impact of each scenario on the both implementations if different.

Table I presents the quality criteria evaluated for each scenario.

1) Scenario 1: Adapt the phone version to the tablet

'Tweetle' is an application dedicated to the smartphone usage, but can be adapted to smart tablet. Tablets in landscape mode have enough space to keep all three screens visible at one time, therefore the main menu becomes just an indicative

name menu to define each list without any action. Tablet in portrait mode will have a mobile application behaviour.

The adaptation can be easily achieved with Android Passive MVC. Application should only be adapted for the tablet landscape mode. The developer should define a new layout.xml for the new tablet appearance: the new layout mainly consists of combining the existing layouts into one. Developers do not need to define a Controller (Fragment) for the main menu as there is neither action nor presentation logic needed. Other Controllers remain the same. Developers should add to the Activity a verification of whether the tablet landscape mode is active or not and set the corresponding layout. All Controllers defined in the layout.xml will be attached automatically. One can see that a very few modifications are needed to make the adaptable interface. This scenario shows the high maintainability and reusability allowed by the Android Passive MVC.

2) Scenario 2: Add new tab to the main menu

It is very probable to add new tab to the existing menu. For example, 'Tweetle' need an extension with a map showing the newest geolocated tweets nearby. For both implementations, the developer should create a new map Fragment generating the map and Controlling actions on the map. Then, the developer can modify the main menu Fragment (controller) to add an action to the new button. For the first implementation the developer should also add a new activity-initialising fragment and an active button. Domain Model would be enriched with several new components as a new web service recovering geolocated tweets or a new DAO method recovering geolocated tweets from existing database have to be used.

One can see that only one Controller should be modified for this extension and several independent components are created. The modification of existing components is very light in Domain Model too.

3) Scenario 3: Move the main menu to the separate screen

The client wants to change the style of the mobile application creating a Windows 8-style menu screen with big square buttons and icons taking up the full screen. For the first implementation, the developer should create a new layout for the main menu and attach it to the new activity with the exact same Controller. The developer needs to check other Activities corresponding to the menu tabs to delete the initialisation of the active button, as it is not used any more if the initialisation was made in Activity.

The second implementation requires greater modifications: Activity can take a Mediator Controller role and replace the main menu with another Fragment, as the Fragment cannot replace itself. The developer could also pass to the first implementation modifying entirely the main menu Controller and creating additional Activities reusing all other fragments.

TABLE I
EVALUATION CRITERIA BY SCENARIO

Scenario #	Maintanability	Reusability	Extensibility
1	x	x	
2	x		x
3	x		
4	x		
5			x

This example shows that for maintenance reasons the developer should preferably choose different Activities for the independent screens, as in the first implementation. In spite of the common menu, all tabs are completely independent and could be arranged differently in the interface while the application evolves. Fragment Mediate Controllers are less reusable but as they are very small they can be reimplemented easily. This example shows that the architecture resists extensive visual modifications and most Controllers remain reusable.

4) Scenario 4: Modify the appearance of the list

It is possible to improve the visualisation of messages: to add an avatar, nickname, and make different colours for different lists. This can be done easily for both implementations. The developer should create an adapter to adapt the Tweet object from the Domain Model to the new visualisation in the list. The developer should only modify the adapter in the list Controller to modify the visualisation of all Controllers. If different visualisations are needed for different lists, the developer can create different Controllers or different Adapters and set up the visualisation in parent controllers.

This example also shows the maintainability of an application made with Android Passive MVC and the reusability of components.

5) Scenario 5: Add a bar containing the name of the active tab

We assume we should add a new name bar to the initial 'Tweetle' application. The visual appearance of this bar is the same for all tabs; the name corresponds to the tab name. For the first implementation, the easiest way is to add this bar directly into the layout.xml without any modification in the code. This is not possible for the second implementation. If the developer adds the modification of the view of the name bar to the main menu, two interfaces becomes dependent and cannot be used separately. Main menu could also notify the Activity to set up the name bar, but in this case the bar is not reusable in other Activities. The most reusable way to carry out the second implementation is to create a Fragment for the name bar. The main menu could pass the data to initialise the name bar as it initialises the lists instead of manipulating the view directly.

This example shows how the first implementation has maintainability advantages over the second implementation as the parent Fragments implementing Fragment transactions could be trickier to manage in case of the interface modification, but child Fragments can always be reused. This example also shows that Fragments have advantages even for the interface having no action but presentation logic.

C. Architecture application

We asked an Android developer with three years' experience to test the Android Passive MVC in real life projects. He chose to redevelop one of his latest applications which had become complex and hard to maintain, extend and test. The application is called 'TaskProjectManager' and it enables tasks to be assigned to different employees and to view the full

TABLE II
TASKPROJECTMANAGER STATISTICS

	Original	Android MVC	% Gain
# Packages	25	17	32
# Classes	393	275	30
# Functions	2186	1683	23
Avg CCN	2,30	1,87	19
Max CCN	110	30	73

calendar of tasks on the screen by day, week and month. The application also generates reports according to parameters.

We choose to compare the old version and the novel version developed using the Android Passive MVC without Fragments. In spite of the fact that developers produce slightly better code while redeveloping the same application due to greater experience, our measurements show the impact of Android Passive MVC on the redevelopment.

Measurements of both versions of the application are made with JavaNCSS, a source measurement suite for Java, and the results are shown in Table II. Android Passive MVC reduces all code parameters.

For each comparison feature denoted i, the gain is calculated as the difference between the original and the Android Passive MVC applications scores (resp. $Original_i$ and $AndroidMVC_i$) divided by the original application score (i.e., $Original_i$).

$$Gain_i = \frac{Original_i - AndroidMVC_i}{Original_i} \quad (1)$$

where:
$Original_i$ - measurement of feature i taken on the originally implemented application
$AndroidMVC_i$ - measurement of feature i taken on the Android Passive MVC implementation
i - the comparison feature

The Android Passive MVC helps with organizing classes in packages. The original version of the application had many packages created partly using the MVP model, partly the application logic, and partly the Android components names. The limited number of packages of the Android Passive MVC version gives the application a clear structure deciding the Domain Model from the interface management.

The full code became smaller: both the number of classes and the number of functions were reduced. We observed the application had a main menu appearing while the calendar was visible. Calendar had different modes of functionality managed by different activities with a huge presentation method managing the appearance of different activities. Menu actions were found multiplied in those activities. The Android Passive MVC enables high reusability of components and structuration of presentation logic.

The code complexity is evaluated using Cyclomatic Complexity Number (CCN). 'Cyclomatic complexity measures the number of linearly independent paths through a program module' [6]. Normal method complexity without any risks is 1-10 CCN, with 11-20 CCN the complexity is moderate, with 21-50 CCN the complexity is very high and with CCNs greater than 50 the program is untestable. Table II shows that the average complexity of the application has decreased slightly. The maximum CCN dropped significantly: an original version has methods with CCNs of 40, 50 and even 100 and 110 mostly for Activities handling a huge number of events, while the new version has the only JSON parser with a CCN of 30 and several methods with a CCN of 10 to 15 in the application core.

The developer's feedback explained that the Android Passive MVC model is easy to understand and to follow. The final application was visibly more reactive: the response time became almost nil, while the users of the original version complained about a very long response time for each screen. The Android Passive MVC version is open to extensions and easily modifiable. The developer said that he had already added more functionalities to the new application before transmitting the code for the CCN analysis. Application components are not only reusable in the application, but could also be reused in future Android development.

The same developer and his colleague continued to use the Android Passive MVC in their everyday job of Android application development for clients. They have tested the version with Fragments and consider it even easier due to the many predefined Android actions.

We obtained a new feedback after 10 months of testing. The developers recognise the improvement of development process since the architecture was introduced: they were able to test both types of Fragment implementations but mostly the first one. The software design state became shorter. They were able to reuse components from one application in another with slight controller modification: photo gallery, search views, 3D and PDF visualizers, horizontal scroll view, etc. They reported the shorter development time due to the clear structure defined by the architecture and easier group work. The division of projects on tasks became simpler and conflicts in code merges became less frequent.

They also noted that they were able to integrate updates rapidly while some old projects without the architecture were redeveloped entirely because of updates demanded by the client.

The developers also discovered that the architecture helps students in practical training to do better and gives them more autonomy. In older project, students without experience needed continual supervision and without it produced little maintainable code. Student, having a short practical training during the experimental Android Passive MVC usage, showed better results while having the same background as previous students involved into Android development.

The developers also noted that the architecture simplified the work with a colleague's code if he is absent, thanks to the common logic, naming and structure.

X. ANDROID AND MVP

In the MVP, the View and the Presenter correspond to a full screen interface. It is not suitable for Android, as visual block embedded into one View could be reused on several screens. Although architecture similar to MVP can be implemented

on Android around the Activity making triads for each visual piece (such as Android Passive MVC).

The Presenter manages the presentation logic of the View and communicates with the model. The difference between the implementation of both architectures is the event handling.

Events in MVP are handled by Views and the action is transmitted to the Presenter. Unlike the Android Passive MVC, in MVP View listens to the events and only calls corresponding methods in Presenter instead of letting the Controller listen to events directly. This implies the creation of additional classes for each View (visual component) implementing event listeners.

The code of Controller/Presenter remains the same. The only difference is that instead of one method handling an event (e.g., click event) Presenter uses many methods corresponding to the action to be carried out on one particular event (e.g., click on search button). This slightly reduces the Controller complexity and allows easier testing. The initial event handling method is placed in the View and remains very simple and does not need any tests.

We did not take the MVP implementation as a base architecture: the implementation of two models is very similar. The existence of Controller having a very complex event handlers can justify the use of MVP instead of Android Passive MVC but we assume it is a rare case. We assume the MVP implementation is the extension of Android Passive MVC for the exceptional particular use.

XI. Android and AM-MVC

The AM-MVC adds the Application Model (AM) component to triads moving the presentation logic to this component. We assume this component can be used in Android Passive MVC in some cases where the action of visual component remains the same but the visual presentation changes. This kind of situation is visible on 'Tweetle' implementation with reusable lists. Instead of unambiguity as to whether the population of the list should be done in the Controller or in the parent Controller, different AM components could be created for each list then, depending on the screen, Controller can initialise the necessary AM object.

We did not find this situation very common and the improvement sufficient to include this case directly into the Android Passive MVC. Similar to the MVP, we consider the AM component as a possible extension for the exceptional use.

XII. Related works

The question of mobile architecture and mobile development process was investigated since the first mobile devices, such as mobile phones and Personal Digital Assistants (PDA), appeared.

Several works were conducted on high-level (methodology) aspects studying the appropriateness of Agile methodology and proposing new methodologies for mobile application development: A Hybrid Method Engineering Approach [26], MobileD [27], etc. Among other aspects, the reusability of components was noted as a very important one.

Many projects are concentrated on web service-based mobile applications. [28] proposes the Balances MVC architecture to partition optimally the core of the application between client and server for different types of application. [29] studies the gap in mobile service-oriented application and propose a mediator layer between the mobile device and the server to fill the gap. Those authors do not include any concrete client-side architecture.

Other works were conducted on the low-level (architecture) issues. [6] conducted an experiment on the possibility of applying the Agile development on mobile systems using design patterns and proved that rapid development only benefits from defined architectures and patterns. [30] analysed the possible cases in application of MVC and PAC architectures in mobile J2ME and Symbian development and concluded that PAC is slightly more suitable due to the modularity of the interface. Authors such as [31] propose some guidelines for designing and developing mobile applications based on a single concrete implementation example.

Concerning Android systems, authors mostly concentrated on security and privacy problems rather than on application architecture. We only found the work of [23] proposing to perform a communication between all Android components via interfaces. We aimed to fill in the gap of the client-side architecture for modern Android system.

XIII. Conclusion and Future Work

The architecture plays an important role in the development of good quality applications. We identified the gap in the Android development, which was missing unified defined architecture.

We have analysed some well-known architectural design patterns and proposed an Android architecture solution based on an MVC and PAC/HMVC design pattern. We have also proposed the Domain Model organization for the Android application that helps to structure the core functionalities. We have provided implementation examples for several common cases in Android development and a concrete implementation of a Twitter client mobile application - 'Tweetle'.

The architecture defined can simplify the work of novice and experienced developers alike and enable the creation of less complex and well-structured applications.

The architecture was evaluated in several ways: scenario-based evaluation showed the high maintainability of the example implemented with Android Passive MVC. One existing Android application was reimplemented using Android Passive MVC, resulting in better maintainability, extensibility and performance. The complexity of the new implementation was considerably lower. We also involved two developers in long-term testing of the architecture on real projects and collected positive feedback on Android Passive MVC. We provided architecture explanation online to reach a larger population and to collect more feedback.

We aim to create a plug-in for Android development environments such as Eclipse or Android Studio to generate common structure, components and classes for an application. For example, the model and database classes can be generated using the database structure.

It is important to note, that Android Passive MVC could also be applied to other systems similar to HMVC and PAC architectures. It requires the main component (main Controller) implemented as Activity in Android but can be represented otherwise in another system.

REFERENCES

[1] K. Sokolova, M. Lemercier, and L. Garcia, "Android Passive MVC: a Novel Architecture Model for the Android Application Development," in Patterns 2013, IARIA, Ed., 2013, pp. 7–12.

[2] S. Allen, V. Graupera, and L. Lundrigan, Pro Smartphone Cross-Platform Development: IPhone, Blackberry, Windows Mobile and Android Development and Distribution, 1st ed. Berkely: Apress, Sep. 2010.

[3] D. Mark and J. LaMarche, More IPhone 3 Development, ser. Tackling Iphone Sdk 3. Berkely: Apress, Jan. 2010.

[4] J. Steele, N. To, S. Conder, and L. Darcey, The Android Developer's Collection. Addison-Wesley Professional, Dec. 2011.

[5] B. Foote and J. Yoder, "Big Ball of Mud," in Pattern Languages of Program Design. Addison-Wesley, 1997, pp. 653–692.

[6] T. Ihme and P. Abrahamsson, "The Use of Architectural Patterns in the Agile Software Development of Mobile Applications," in ICAM 2005 Internation Conference on Agility, Aug. 2005, pp. 155–162.

[7] G. Krasner and S. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," Journal of object oriented programming, vol. 1, 1988, pp. 26–49.

[8] P. Sauter, G. Vögler, G. Specht, and T. Flor, "A Model–View–Controller extension for pervasive multi-client user interfaces," Personal and Ubiquitous Computing, vol. 9, no. 2, Mar. 2005, pp. 100–107.

[9] M. Veit and S. Herrmann, "Model-view-controller and object teams: a perfect match of paradigms," in AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development. ACM Request Permissions, Mar. 2003, pp. 140–149.

[10] S. Burbeck. Applications Programming in Smalltalk-80TM: How to use Model-View-Controller MVC. [Online]. Available: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html (1997)

[11] J. Coutaz, "PAC," ACM SIGCHI Bulletin, vol. 19, no. 2, Oct. 1987, pp. 37–41.

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

[13] M. Potel, "MVP: Model-View-Presenter the taligent programming model for C++ and Java," Taligent Inc, 1996.

[14] J. Cai, R. Kapila, and G. Pal. HMVC: The layered pattern for developing strong client tiers. [Online]. Available: http://www.javaworld.com/article/2076128/design-patterns/hmvc–the-layered-pattern-for-developing-strong-client-tiers.html (2000)

[15] J. Smith, "Wpf apps with the model-view-viewmodel design pattern," MSDN magazine, Feb. 2009.

[16] R. Garofalo, Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern. Microsoft Press, Mar. 2011.

[17] S. McConnell, Tout sur le code : Pour concevoir du logiciel de qualité (Everything about code: make software of quality), 2nd ed. Dunod, Feb. 2005.

[18] R. Meier, Professional Android 4 Application Development (Wrox Professional Guides), 3rd ed. Birmingham: Wrox Press Ltd., May 2012.

[19] I. Salmre, Writing Mobile Code: Essential Software Engineering for Building Mobile Applications. Addison-Wesley Professional, Feb. 2005.

[20] S. Brahler, "Analysis of the android architecture," Karlsruher Institute of Technology, Tech. Rep., 2010.

[21] F. Garin, Android - Concevoir et développer des applications mobiles et tactiles (Android - Comprehend and develop mobile and tactile applications), 2nd ed. Dunod, Mar. 2011.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed. Addison-Wesley Professional, Nov. 1994.

[23] W.-Y. Kim and S.-G. Park, "The 4-tier design pattern for the development of an android application," Lecture Notes in Computer Science, vol. 7105, Dec. 2011, pp. 196–203.

[24] P. D. Sheriff, Fundamentals of N-Tier Architecture, pdsa inc. ed. PDSA Inc., May 2006.

[25] M. T. Ionita, D. K. Hammer, and H. Obbink, "Scenario-based software architecture evaluation methods: An overview," in Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering, 2002.

[26] V. Rahimian and R. Ramsin, "Designing an agile methodology for mobile software development: A hybrid method engineering approach," in Research Challenges in Information Science, 2008. RCIS 2008., 2008, pp. 337–342.

[27] P. Abrahamsson et al., "Mobile-D: an agile approach for mobile application development," in OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Oct. 2004, pp. 174–175.

[28] H. J. La and S. D. Kim, "Balanced MVC Architecture for Developing Service-Based Mobile Applications," in e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on, 2010, pp. 292–299.

[29] A. Papageorgiou, B. Leferink, J. Eckert, N. Repp, and R. Steinmetz, "Bridging the gaps towards structured mobile SOA," in MoMM '09: Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia, Dec. 2009, pp. 288–294.

[30] D. Plakalovic and D. Simic, "Applying MVC and PAC patterns in mobile applications," Journal of Computing, vol. 2, no. 1, Jan. 2010, pp. 65–72.

[31] D. Zissis, D. Lekkas, and P. Koutsabasis, "Design and Development Guidelines for Real-Time, Geospatial Mobile Applications: Lessons from 'MarineTraffic'," in Mobile Web and Information Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 107–120.