Android Passive MVC: a Novel Architecture Model for Android Application Development

Karina Sokolova*†, Marc Lemercier*
*University of Technology of Troyes, France
{karina.sokolova, marc.lemercier}@utt.fr

Ludovic Garcia†
†EUTECH SSII, France
{k.sokolova, l.garcia}@eutech-ssii.com

***Abstract*—Nowadays the demand for mobile application development is very high. To be competitive, a mobile application should be cost-effective and 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 some widely used architectural design patterns and propose a unified architecture model adapted to Android development. We provide the implementation example and test the efficiency of the proposed architecture by implementing it on a real application.**

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

I. INTRODUCTION

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 very high especially for the two market leaders: Apple (iOS) and Google (Android). Multi-platform solutions, such as Phone-Gap, Rhodes Rhomobile and Titanium Appcelerator reduce development time, as one application is developed for several platforms [1], but have limited possibilities – often requiring native plug-ins. Multi-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. Native solutions enable use of all the platform's options with better performance and lighter code, 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 [2]. Android requires no particular architecture [3] – 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 [4]. 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 [5].

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 identify 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.

The remainder of the paper is organized as follows: the second section presents several architectural patterns used in software development. Section 3 presents briefly the Android SDK and existing difficulties in adapting one known architecture to Android. In Section 4, we propose an adaptation of the MVC design pattern to the Android environment and provide an implementation example. Section 5 evaluates the Android Passive MVC model and Section 6 concludes this work and presents some perspectives.

II. FUNDAMENTAL DESIGN PATTERNS

We present four popular MVx-based 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.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

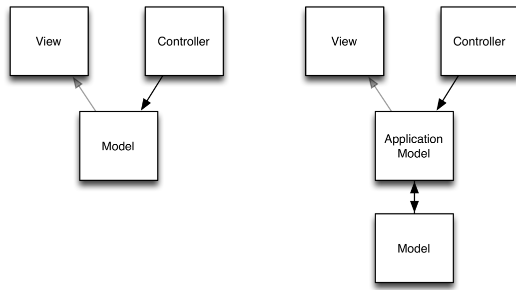

Fig. 1. Classic MVC and Application Model MVC

A. Model-View-Controller (MVC)

Presented in 1978 [6], Model-View-Controller is the oldest design pattern and has been successfully applied for many systems since it's creation [7], [8]. 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 [6]. MVC consists of three main components: *Model*, *View* and *Controller*. The *Model* represents a 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 *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*.

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

In all variants, *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 [9]. 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. Model-View-Presenter (MVP)

The Model-View-Presenter was introduced in 1996 as an MVC adaptation for the modern needs of event-driven systems [10]. The model consists of three components: *Model*, *View* and *Presenter*. In this model, the *View* represents a full screen and it handles events from the user actions. The *Presenter* is responsible of the presentation logic. The *Model* is a Domain model.

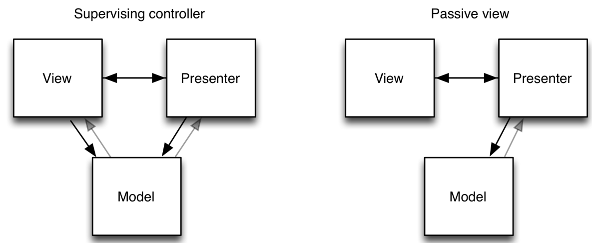

Fig. 2. Supervising controller and Passive view

There are two types of MVP: Supervising controller and Passive view. Both models are shown in Figure 2. 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* [10].

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

The Hierarchical-Model-View-Controller was first introduced in 2000 as an Classic MVC adaptation for Java programming [11]. 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 3.

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.

D. 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 [12]. The realization of this model is done with binding between components [13]. The binding is not supported in Android by default but could be implemented using the very recent Android-binding framework.

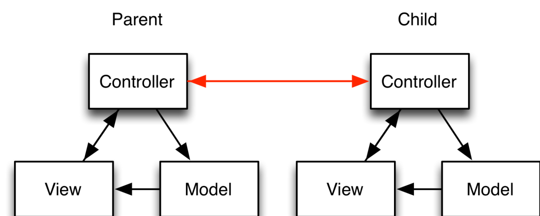

Fig. 3. Hierarchical-Model-View-Controller

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

As stated in [14], 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. Android Application Development Experience

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 and NASA also choose Android for their projects.

Android applications are mainly written in Java using the Android SDK [15]. 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 [16] – 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. Activity is a main component of Android applications created while the application that is also the entry point to the application is open. Many Activities can exist in the application but only one is active at a time. The service works on the background of an application permitting an execution of long tasks (e.g. file download). When the application is closed, unlike Activity, the work of the Service is not interrupted. The Content provider component gives access to the local data stored in SQLite databases. The Broadcast receiver is a messaging system that enables communication inside the application and between multiple Android applications installed on the phone.

Activity causes major difficulties in implementing the known architecture: is it a *View*, a *Controlle*r, a *Presenter* or none of them? Some developers say Android actually imposes the MVC model where the layout.xml (file, defining the layout of the screen) is a *View*, Activity component is a *Controller* and the rest is a *Model*. This proposition is not really the MVC: layout.xml only defines what the screen looks like, but button actions, text information and other presentation logic are usually placed in Activity. Therefore, Activity handles events as *Controller* and manages the visualization as *View*, replacing the *View-Controller*. It leads to the creation of a heavy and complex Activity class [17]. Huge classes that have many responsibilities (event handler, presentation logic, etc.) violate the Single Responsibility Principle of Object Oriented Programming and could be hard to understand, test and extend [18].

Other developers place Activity as a *View* of MVC creating *Controller* apart. This solution works for simple applications where one Activity represents one visual block, while Activity usually manages several *Views*: main screen, menu, dialogue box, etc. In complex visual applications Activity becomes heavy; *View* components are strongly linked to each other and are not reusable. *Controller* will be either complex or divided into parts by a number of embedded Activity *Views* that go against the MVC statement of one *Controller*, one *View*. Replacing MVP *View* with Activity can cause similar problems.

Some developers observed that Android have predefined *View*s as ViewFlipper. It brings another solution where the Activity became a *Controller* and *Views* are created apart. Solution seems more adaptable to Android as event interception in Activity can be defined in layout.xml but actually creates problems similar to previous implementation: many *Views* make the only *Controller* (Activity) complex. *Views* are reusable but the corresponding *Controller* should always be added to the new Activity using the *View*. To delete or modify the *View* developer should modify the full Activity. Final application is complex and hard to maintain.

Even if MVC and MVP architectures seem suitable for Android developments they are not intuitive to implement. A new architecture should be easily implemented with Android-specific components, such as Activity. The implementation of the model should improve the application and code quality. More precisely, the model should 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 [14], [18]. A mobile phone has a limited memory and a garbage collector could have unexpected behaviour therefore the creation of unnecessary objects should be avoided. Finally, objects remaining in the memory should be lightweight [16].

IV. Novel Design Pattern for Android Platform

The first part of the section explains the novel architecture for Android application development we named Android Passive MVC. The second part of the section presents a simple example implementing the Android Passive MVC. The third part of the section recommends an architecture 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 could also simplify the development process.

A. Android Passive MVC Presentation

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. We decided to create MVC model around Activity making the Activity the fourth component. We can also think of Activity as a main screen (parent) controller in HMVC model.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

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

In our model, Activity becomes an intermediate component between the Views and the Controllers, thereby Controllers take the event handling responsibility and the Views take the presentation logic making the Activity lightweight. The scheme of the Android Passive MVC model is shown in Figure 4. The grey dashed arrows show the interaction via Android native methods. Black arrows indicate direct calls and grey arrows represent listener events.

The Activity is like a screen controller. The starting Activity creates a link between a View and a corresponding Controller to make them communicate directly. The communication between Controllers is made via Activity.

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 independent and do not communicate.

The Controller handles events from the user action (e.g. button click), calls necessary methods from the Model and then notifies the View to be updated on Model response. The Controllers are independent from one another and do not communicate directly.

This solution makes Activity lightweight by moving all event handlers to Controllers and interface management to Views. Views and Controllers created on demand avoid unnecessary objects, saving memory. Developers can easily modify or remove application components by only modifying 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 thereby the maintainability of the application is high.

We perform the communication between Activity, Controller and Model via message listeners implemented via interfaces as proposed by [19]. Figure 5 shows the Android Passive MVC implementation diagram. Listeners increase the performance of the application and create a weak coupling between components that improve maintainability.

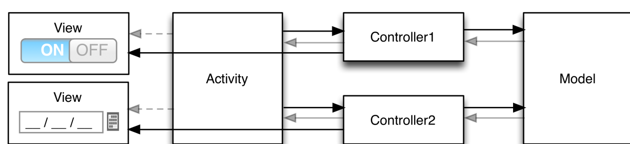

Fig. 4. Android Passive MVC

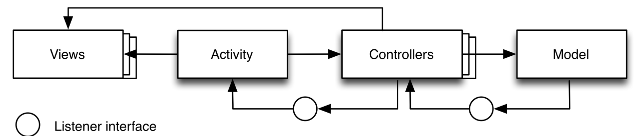

Fig. 5. Android Passive MVC implementation

B. Android Passive MVC Implementation

This section presents an implementation example of communication between Android Passive MVC components. This implementation is suitable for the new manually created Activities. Some predefined Activities, especially from third-party libraries, will possibly not fit the implementation. 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 the 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 6.

Login View contains methods for obtaining login and password (getters), methods to set button listener and methods to set errors. Login Controller handles event from the login form implementing the onClickListener; while the button is pressed Controller launches simple verifications and calls the model. If login is successful, the answer goes back to the Login Activity, which opens a welcome screen. To simplify the example we do not include the model, but the communication between the Controller and the Model can be implemented similarly. A full code example can be found on [20].

C. Android Domain Model

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 usually used in 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 7.

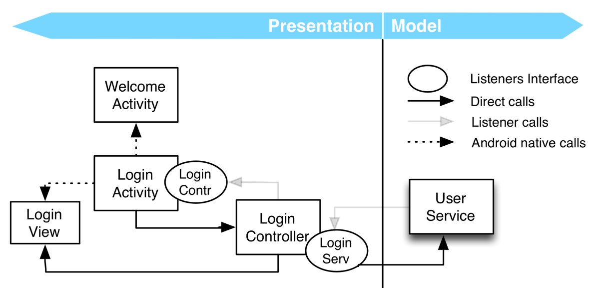

Fig. 6. Login implementation example

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

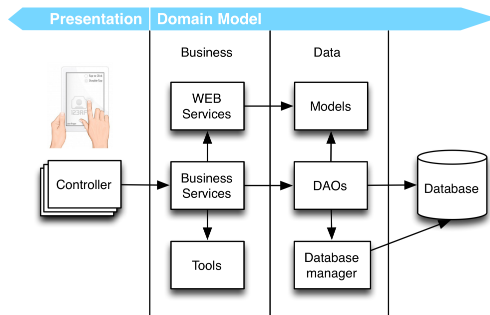

Fig. 7. Domain Model Architecture

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 [21].

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. Data is manipulated and transferred through the 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 use a lightweight 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 the 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, increases cohesion. High cohesion facilitates the maintainability of the software. The final code of the application could be organized in packages by architectural components: 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.

V. Architecture Evaluation

We evaluated the architecture in two steps. First, we ensured that the architecture fit the lists of code quality criteria proposed by [14], [16]. Second, we ask an Android developer to rewrite one of his latest applications using Android Passive MVC, compare results and give feedback regarding the model.

A. Code quality

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

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. Architecture implementation

We asked an Android developer with three years' experience to test the Android Passive MVC. He chose to redevelop one of his latest applications which had become complex, 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 calendar of tasks on the screen by day, week and month. The application also generates reports by given parameters.

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

The Android Passive MVC helps with organizing classes in packages. The original version of the application had many

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

packages created partly using the MVP model, partly the application logic, and partly the Android components named. The limited number of packages of the Android Passive MVC version gives the application a clear structure.

The full code became smaller: both the number of classes and the number of functions were reduced. The Android Passive MVC enables high reusability of components.

The code complexity is evaluated using Cyclomatic Complexity Number (CCN). 'Cyclomatic complexity measures the number of linearly independent paths through a program module.' [5]. 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 and with CCNs greater than 50 the program is untestable. Table I shows that the average complexity of the application 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, while the new version has the only JSON parser with a CCN of 30 and several methods with a CCN of 10 to 15.

The developer's feedback was 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. Application components are not only reusable in the application, but could also be reused in future Android development.

VI. CONCLUSION AND FUTURE WORK

We have analysed some well-known architectural design patterns and proposed an Android architecture solution based on an MVC design pattern and the Domain Model organization. The architecture defined can simplify the work of novice and experienced developer alike and enable creation of less complex and well-structured applications. The existing Android application was reimplemented using the Android Passive MVC, resulting in better maintainability, extensibility and performance. The complexity of the new implementation was lower.

We consider a wider evaluation by the Android community. We are currently working on a user-friendly model description and several well-commented implementation examples. We are also drawing up on a questionnaire for the developers who have tested the model. We plan to spread the documentation, examples and a survey over the important websites and blogs to reach a larger audience.

TABLE I
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

This work can be continued by testing the observer-observable design pattern integrated in the Android Passive MVC. The adaptation of the MVP model can be envisaged. The same testing software could be redeveloped to compare the results. Finally, the same test using the Android-binding MVVC framework could be implemented to choose the most effective solution for different types of applications.

REFERENCES

[1] 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.

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

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

[4] B. Foote and J. Yoder, *Big Ball of Mud*. Addison-Wesley, 1997.

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

[6] 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, pp. 26–49, 1988.

[7] 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, pp. 100–107, Mar. 2005.

[8] 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 Press, Mar. 2003, pp. 140–149.

[9] S. Burbeck. (1997, Mar.) 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 [retrieved: March 2013]

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

[11] J. Cai, R. Kapila, and G. Pal, "HMVC: The layered pattern for developing strong client tiers," *Java World*, pp. 07–2000, 2000.

[12] J. Smith. (2009, Feb.) Wpf apps with the model-view-viewmodel design pattern. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx [retrieved: March 2013]

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

[14] S. McConnell, *Tout sur le code : Pour concevoir du logiciel de qualité*, 2nd ed. Dunod, Feb. 2005.

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

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

[17] F. Garin, *Android - Concevoir et développer des applications mobiles et tactiles*, 2nd ed. Dunod, Mar. 2011.

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

[19] W.-Y. Kim and S.-G. Park, "The 4-tier design pattern for the development of an android application," in *Proceedings of the Third international conference on Future Generation Information Technology*, ser. FGIT'11. Springer-Verlag, Dec. 2011, pp. 196–203.

[20] K. Sokolova. Android passive mvc implementation example. [Online]. Available: https://github.com/KarinaSokolova/android-mvc-example [retrieved: March 2013]

[21] P. D. Sheriff, *Fundamentals of N-Tier Architecture*. PDSA Inc., May 2006.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9