2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing

A Simple Software Development Methodology Based on MVP for Android Applications in a Classroom Context

C. N. Ojeda-Guerra
Department of Telematics Engineering
University of Las Palmas de Gran Canaria
e-mail: carmennieves.ojeda@ulpgc.es

***Abstract*—With the world becoming a more mobile place for everyone through the use of smartphones, tablets and laptop computers, the need to help people to get mobile programming skills has never been greater. If anyone wants to become a mobile applications developer, the best approach might just be to learn how to develop and start doing it, and it's "to learn how to develop", the really important point.**

In this paper, we present a simple methodology which takes many features of different programming methodologies and uses its own. The main purpose of our methodology is that professionals in any field, with limited skills in mobile programming, can perform simple Android applications for use in their work environment or undergraduate engineering students can develop amazing apps with a minimum effort.

***Index Terms*—Android, MVP, Developing Process or methodology, Prototyping, Simple Sequence Diagrams, Learning.**

I. Introduction

Programming is an essential component in the curricula of most technical degrees in universities. Earning a college degree in computer programming can open up a wide range of career opportunities for individuals interested in the ever-growing field of technology [1]. These degree programs teach the student how to bring a computer program into existence, constructing, testing and debugging the program.

Today, one of the hottest new areas of computer programming is the mobile programming because the popularity of the mobile devices is driving increasing demand for IT professionals with mobile skills. According to the International Telecommunication Union [2], in 2014 mobile phone access reached more than three quarters of planet's population. The portability, relatively low-power usage, and wireless connectivity have allowed this technology to penetrate areas where people could not even dream of using a computer or connecting to the outside world. This makes mobile application development an exciting field to be in.

Mobile apps are developed for a certain platform, and the two most popular platforms today are iOS and Android. Mobile developers write programs inside of a mobile development environment using the Objective C, Swift or Java programming languages, among others. Due to significant differences in the environment and in platform specifications, mobile application development requires a suitable development methodology. The applications are usually small-sized, are not safety-critical, and do not have to satisfy interoperability or reliability constraints. They are delivered in rapid releases in order to meet market demands, and are targeted at a large number of end-users.

In these modern times, applications impact our normal lives every single day so the ability to develop mobile apps is an extremely hot skill to know right now. App developers are at the centre of this app revolution, and they're always on a mission to create something that people will find useful every day. However, building the first mobile app can be a challenging experience and new developers need a special guide to design and build them with a minimum effort. This is the main problem that we find in our working day: recent college grads or entry-level or mid-career professionals who want to develop mobile apps to solve specific problems related with their: jobs (e.g. Calculations for low-voltage installations, Vehicle diagnosis with OBDII, Employee time and attendance system, Spare parts inventory control,...), hobbies (e.g. Ornithology guide,...), quality of life (e.g. Balanced shopping list, Monitoring of medication intake,...), and so forth. In this context, we present a paper to explain a simple software development process to implement mobile applications in a classroom context. This process is based on *MVP* architecture because its specific characteristics make it the most simple way to introduce people, with limited or minimal programming skills, in the mobile apps world, as we explain after. The paper is organized as follows: In mobile applications methodologies section, we outline some important methodologies for designing mobile applications and we motivate our approach. Next, we introduce the proposed software development process that the students use in our bachelor engineering degree (non-CS) and post-graduate courses. We briefly describe the process that we follow to implement Android applications. Next, we present the planning process to complete the job on time. Finally, we conclude by outlining the main goals of our approach.

II. Mobile applications methodologies

The traditional methods used to define and develop desktop applications will not work with mobile application development, according to Gartner, Inc [3]. Hence, it's imperative to employ agile development to quickly iterate on improving the mobile applications. Thus, there are several well known *agile* methodologies such as: Extreme Programming (XP), Feature-driven Development (FDD), Lean Development, Adaptive Software Development, Crystal, Scrum, and so forth, all of them are focused on business value, adaptability, high customer involvement, and the frequent delivery of production-quality software [4], [5]. In another way, there are some methodologies based on *rapid-prototyping* which quickly mocking up the future state of a system. Prototypes range from rough paper sketches to interactive simulations that look and function like the final product. Some authors use these methodologies for designing mobile applications [6], [7]. Among them is the *Rapid application development* (RAD) that is a concept that products can be developed faster and of higher quality through: gathering requirements from customs, prototyping, reiterative user testing of designs, the re-use of software components, less formality in reviews and other team communication, and so forth. RAD employs a model-driven and object-oriented approach to developing complete solutions. In [8] there is a deep explanation about this methodology, its core elements, processes, development tools and so forth.

In this paper, we propose a simple process based on the features of the three previous methodologies and some of our own. But, if there are a lot of different methodologies, why are we proposing a new one? The main reason is that all of them are very complex to teach students or professionals who have a low level of knowledge about programming but they are interested in learning to program mobile devices. Perhaps, mobile programming won't be their future work, but they want to solve small and simple problems using mobile devices in their future work [9], [10], [11]. So, our proposed process teaches programming and design fundamentals that provide students with the tools to implement an application for an Android platform. This model is being used in the last year of a bachelor engineering degree (non-CS) with good results (all students perform a complete and useful mobile app in four months, some of which can be downloaded from Play Store). Also, we have tested the model with students in an access course (post-graduate). These students are entry-level or mid-career professionals in fields different from computer science. Due to our students' knowledge, we have met the challenge of designing a simple process which abstract away the details of the programming and provide a project overview, through prototype and sequence diagrams, make it easier to examine big design issues. This proposed process fits our thinking and visualize the behavior of the objects and the interactions with others.

III. Proposed methodology

We can define *methodology* as a set or system of methods, principles, and rules for regulating a given discipline, as in the arts or sciences. In the programming of our mobile applications, it will correspond to a set of methods which will be used to achieve the following goals: clear design, knowledge of application behavior, highly decoupled architecture (easy code modification), a team-based approach to development, quick implementation, code reuse and easy testing and verification process (it has not wait until the end of the development cycle).

In this context, the main idea of the proposed methodology is to design a set of methods, clear and simple, that any student can follow to develop a complete android app (useful and exiting) in a quadmester (the duration of our course) although he/she hasn't got previous knowledge and skills about mobile programming, so the learning objectives are to prepare students with limited (or minimal) programming background to advance on to mobile device computing.

A. Requirements specification and prototyping

Having a great idea is the starting point into every new project. Before developer goes straight into detailing though, he/she must clearly define the purpose and mission of his/her mobile app. What is it going to do? What is its core appeal? What concrete problem is it going to solve, or what part of life is it going to make better? In this step of the methodology, students must establish the initial requirements of the application using simple sentences. The requirements should be clear, concise and realistic. Also, it includes a set of *use cases* that describe interactions the users will have with the app. Once identified them, the students must create the *user interface prototype* (User-Centred Focus).

The prototype is reviewed and evaluated by the customers until they give their approval. According to Crosstalk, the Journal of Defense Software Engineering [12], the most failures in software products are due to errors in the requirements and design phases (as high as 64 percent of total defect costs), so it is important to have a proper process of analyzing the requirements in place to ensure that the customer needs are correctly translated into product specifications. Prototyping helps the users and designers to discuss options, early evaluation and shows a preview of the applications. Software prototypes run on a computer and include computer animations, interactive video presentations, applications developed with interface builders, users interaction and so forth. Designers must take the context of use into account when designing the details of the interaction and this interaction is pretty important in the design of the sequence diagrams.

One of the most popular mockup and prototype tool is *Balsamiq Mockups* [13]. Balsamiq team offers a program for high schools and universities which want to use the tool in a classroom setting, free of charge. Using this tool, our students develop their prototypes in a very short time (about 1-2 weeks) and test the mockup specifying the tasks they want the users to perform (usability test) [14] and involve evaluators examining the UI and judging its compliance with Nielsen's heuristics [15]. Students's usability studies provide them with a trove of insight and feedback, which they use to re-built the final prototype, and justify their design decisions. Figure 1 shows some prototype examples made by the students in our course. After the approval of the prototype, students can design the layouts of the different views of the app. In Android, these layouts are xml files that include the screen elements that will appear in the views.

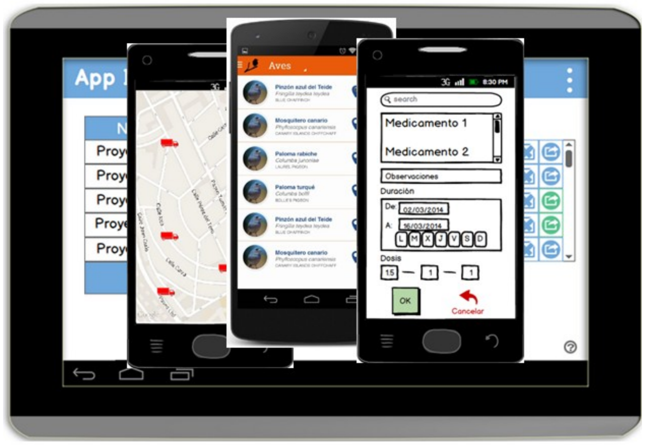


Fig. 1. Prototype examples made with *Balsamiq Mockups*.

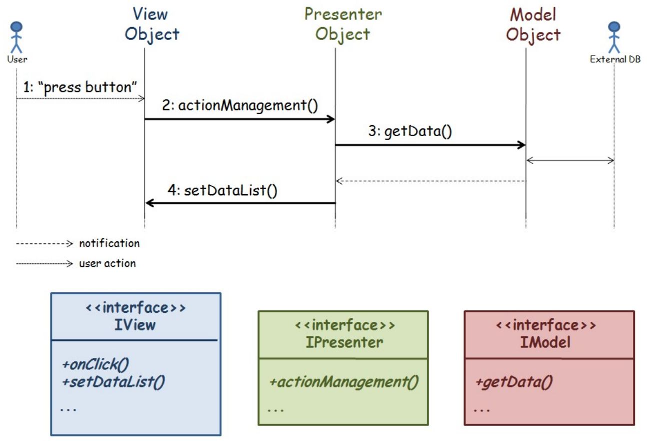


Fig. 2. Example of a sequence diagram and related interfaces.

B. Simple sequence diagrams

In our methodology, we use the *MVP* (*Model-View-Presenter*) architecture that first appeared in IBM and more visibly at Taligent the 1990's [16]. This architecture, much like other design patterns, decouples development in a way that allows multiple developers to work and test simultaneously. It's important to mention that the Android developers have a problem arising from the fact that Android activities are closely coupled to both interface and data access mechanisms. For an application to be easily extensible and maintainable developers need to define well separated layers. Thus, using MVP: a) Background tasks are separated from activities/views/fragments to make them independent of most lifecycle-related events, b) Complex tasks are split into simpler tasks and are easier to solve, c) Makes views independent from the data source and d) Facilitates automated unit testing. All these features make MVP architecture more convenient for students with limited or minimal programming skills who want to develop mobile apps.

The MVP architecture is composed by: *model* (that encompasses business objects), *view* (that contains all of the UI components that make up our application and forwards all interaction operations to the presenter) and *presenter* (that contains all of the logic for our application). In order to bind the three components of the architecture, we use interfaces: one for each view, model and presenter contained in the application. This allows us to define the behavior of each class independent of its internal implementation and enable to develop the different components of the architecture at the same time. The model, view and presenter are the mainsprings of the sequence diagrams that depend on the context of use of the application (*use cases*). In these diagrams, we show the behavior of each object (model, view and presenter) of the architecture and they are the entry point to the design of the classes and interfaces related with these components.

Sequence diagrams describe dynamic aspect of a system and use objects as basic elements. Each object in a diagram is represented by a vertical line, which corresponds with time axis, where time moves down. This diagram shows the messages sent between objects over time (so, it's a message sequence chart). The order of the objects is not so important, the truly important is the order of the messages and the dependencies between them i. e. what consequences have the sending of a message. Also, the sequence diagrams involve *actors* that represent how the application is used. In most applications, there are at least two actors: user (person who normally uses the application) and a database (entity used for storing the information). In order to make our simple sequence diagrams, we use the previous prototype to figure out how the application is used (*use cases*). Thus, if we assume that, in the view, there are a button and a list that is updated after pressing the button, the user can perform the "press button" action. When the view object receives the user action, delegates the management of the action (application logic) in the presenter object. If the presenter object needs data from the database, it sends a message to the model object in order to retrieve these data. The presenter object keeps on "observing" (it is an *Observer* object) to the model object (it is an *Observable* object), so when it finishes the data recovery process sends an event that will be "observed" by the presenter. After this, the presenter object sends a message to the view object for it updates the list with the new data.

In the design of our simple sequence diagrams, we only take into account the messages sent between objects, because these messages correspond with methods that have to be defined in the interfaces implemented by these objects' classes. Thus, a straight horizontal arrow that joins two vertical lines (named *A* and *B*) in the diagram depicts a message sent between an object (object *A*) and another (object *B*), and this arrow has an identifier that matches with a method's name defined in the object *B*'s class (figure 2), because object *B* is the destination of the message. Also, in the sequence diagram we don't include the parameters of the methods, only the methods' name.

For a particular application, our students follow the next

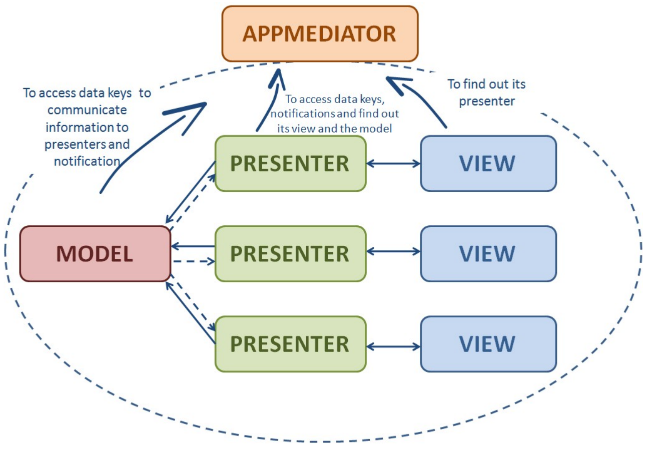

APPMEDIATOR
To access data keys to communicate information to presenters and notification
To access data keys, notifications and find out its view and the model
To find out its presenter
PRESENTER
VIEW
MODEL
PRESENTER
VIEW
PRESENTER
VIEW

Fig. 3. MVP architecture with *AppMediator* component.

rules to make the sequence diagrams:

- Based on the prototype, there is one class (so, one object) for each view of the application (*view class*), which implements an interface (*view interface*).
- Each *view class* has a related *presenter class*, which implements a *presenter interface*.
- In most cases, there is only one *model class* that implements a *model interface*.
- Each *view class* can only get the information or update the content of its particular view objects (e.g., *TextView*, *EditText* objects...) defining accessor methods.
- The *view classes* can not process any information or carry out any task. They are completely passive.
- Each user's action over a screen view (defined in a *view class*) invokes an action method of the corresponding *presenter class* that has no parameters and no return value.
- Any data interaction (insertion, updating, deletion, search) must be performed by the model at the request of any presenter, invoking a model method that has no return value.

Following the previous rules and the behavior of the prototype, our students make a set of sequence diagrams, in order to define the different methods of the view, presenter and model interfaces.

C. Implementation of the MVP architecture

After designing the sequence diagrams relating to the possible uses of the application (by the actors), we can implement the MVP architecture using Android tools. First, to handle logic that is not specific to any presenter and instead resides at the application layer, an *AppMediator* component is introduced (figure 3). In order to design the singleton *AppMediator* class, we took a look to the Android reference that describes the *Application* class [17] (Android creates an instance of this class when the application is started).

We implement the *AppMediator* class following the Android specification and completing it with our special features, as:

- Define a private instance variable (object) for each view, presenter and model interface.
- Define *setter* and *getter* methods for these variables.
- Define the constants for "data communication" (notifications, key of the data sent between Android components,...).
- Re-define particular Android methods to: launch activities, start services, broadcast messages, register receivers,...
- Define methods to navigate inside the application.

The aim of the two last items, is to remove the tasks over the Android components and the navigation in the application, from the particular classes of the application and merge them into one common class (*AppMediator* class). Thus, each screen view has a string identifier (for example, "initView", "loginView", "birdList",... related with its content) and when a specific presenter wants to launch the screen view *x*, it only asks the mediator that it launches the view class related with the identifier *x* (carrying out the user's navigation).

The *AppMediator* class is very similar in all our applications with the obvious changes: the name of the view, presenter and model interfaces, the name of the constants and the particular navigation. We can recover the *AppMediator* object from most Android components through the *getApplication* method or directly using the static *getInstance* method in *AppMediator* class.

On the other hand, each view class in the application extends *Activity* class (or any sub-class of *Activity* class) and has a related layout xml file (implemented in the prototyping phase). In the *onCreate* method of the view class (method invoked when the activity is launched), we must update the reference of this view in the *AppMediator* object. Also, the view class must implement the methods defined in its interface, which corresponds to the methods "discovered" in the respective sequence diagrams (these include the accessor methods). As mention before, in the case of any user interaction (press a button, select a menu item,...), the view requests its presenter that performs the corresponding task (using the action method).

Furthermore and as mentioned earlier, each view class is related with a presenter class. The main presenter, which is the presenter of the main view, must create the model and update the corresponding private instance variable in the *AppMediator* object. Any other presenter, which wants to access the model, can get it through the mediator. Also, each presenter object knows its related view object, using the mediator. Similar to the view class, the presenter class must implement the methods defined in its interface ("discovered" in the corresponding sequence diagrams). In all cases, these methods perform tasks related with the user's actions on its view.

Following guidelines developed by MVP architecture, if a presenter needs to access data, it has to request them to the model object. In our methodology, this access is performed as follows:

- The presenter defines a broadcast receiver (object of a *BroadcastReceiver* type). This object has a callback

onReceive method that is invoked when a specific notification arrives (we call it *callback notification* and it's defined as a public constant in the mediator).
- The presenter registers a notification receiver, which waits for the callback notification.
- After that, the presenter requests the data to the model.
- When the data are available, the model packs and sends them, to broadcast, in the callback notification. Any receiver, which is listening to this notification, responds with the execution of its callback *onReceive* method.
- Into the *onReceive* method, the presenter manages the data and performs the corresponding task (it can be to show the data in the view in a particular way).

In the listing 1, we can see the broadcast receiver object (*receiver* object), its *onReceive* method and the callback notification (*callbackNotification* variable). If the callback notification is equal to the expected notification (constant of the mediator called CALLBACK_NOTIFICATION), the presenter does something.

Listing 1. Code of the *MainPresenter* class to access data

```
private BroadcastReceiver receiver = new
     BroadcastReceiver() {
   @Override
   public void onReceive(Context cxt, Intent i) {
      String callbackNotification = i.getAction();
      if (callbackNotification.equals(
         AppMediator.CALLBACK_NOTIFICATION)) {
            // do something
      }
   }
};
@Override public void actionManagement() {
   appMediator.registerReceiver(receiver,
      AppMediator.CALLBACK_NOTIFICATION);
   appMediator.getModel().getData();
}
```

Also, in the *actionManagement* method, which was "discovered" in the corresponding sequence diagram (and it's invoked when the user interacts with the view i.e. pressing a button), we show the register of the notification receiver (invoking the *registerReceiver* method in the mediator) and the request for data to the model object (invoking the *getData* method in the model object).

Listing 2. Code of the *getData* method in the *Model* class

```
@Override
public void getData() {
   // Recover the data from somewhere
   Bundle extras = new Bundle();
   // Pack the data in the bundle object
   appMediator.sendBroadcast(
      AppMediator.CALLBACK_NOTIFICATION, extras);
}
```

In the same way, we show the code of the *getData* method in the model class (listing 2), "discovered" in the sequence diagram. In this method, the model has to recover the data from the external or internal database or from any other place where the data are stored. Once the data are available, they are packed into a *Bundle* object (*Bundle* is a class of Android) and finally, they are sent to broadcast using the callback notification. The data are fetched from the presenter object as mentioned above. As well as on the view and presenter classes, the model class must implement the methods defined in its interface ("discovered" in the sequence diagrams). All these methods will be invoked by a presenter in order to access the data.

The behavior previously described is the same in all our applications with the obvious changes: the name of the communication constants, the data fetched and the tasks performed in the presenter after the data are collecting, because all of them depend on the specific application developed.

D. *Testing*

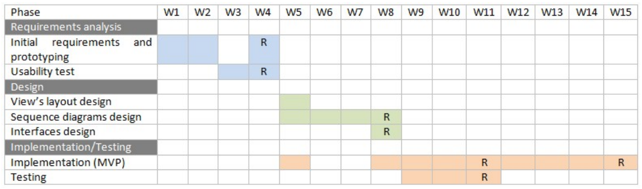
After writing the code, the developer must test it. These testing can be performed at the same time that other parts of the application are being implemented. In this phase, we use the Android's JUnit extensions to test Android components. The main areas to test are: a) Activity life cycle events, b) File system and database operations and c) Different device configurations.

In our applications, students are encouraged to test the database operations, i.e. the model's methods and their behavior. Write and read access from and to the database (internal or external) are tested including its handling. Also, we focus on fostering the small size of the methods in order to avoid programming errors. For testing the model, we register a receiver for the callback notification, set the pre-conditions (as if it was a normal test), invoke the tested method and wait for the notification when the data are available (the same steps as those carried out in the presenter class).

Listing 3. Test of *addRegister* method (model's method) in the test class

```
// test for adding a new register in the database
public void testAddNewRegister() throws Exception {
   // register a receiver waiting for the
   // callback notification. When the notification
   // arrives, it checks the database
   setTheReceiver();
   // set the pre-conditions if it's needed
   appMediator.getModel().addRegister("1234567",
      "John Doe", "123456", "johnny@qwe");
}
```

In listing 3, we show an example of a tested method. The *testAddNewRegister* method implements a code to add a new register in a database. In this example, we suppose that the tested method is the *addRegister* method in model class (its implementation is similar to that described in listing 2). Before calling this method, we call to the *setTheReceiver* method (in the testing class) that defines a *receiver* object in a similar way as the *receiver* object in the *MainPresenter* class (listing 1). If the *addRegister* method is well-implemented, after the data are available, it sends to broadcast a callback notification. When the receiver receives the callback notification, the database must be checked to test if the new register is in it (this is the code inside the *onReceive* method of the receiver in the



Phase	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
Requirements analysis															
Initial requirements and prototyping	🟦	🟦		🟦 R											
Usability test			🟦	🟦 R											
Design															
View's layout design					🟩										
Sequence diagrams design					🟩	🟩	🟩	🟩 R							
Interfaces design								🟩 R							
Implementation/Testing															
Implementation (MVP)					🟧			🟧	🟧	🟧	🟧 R	🟧	🟧	🟧	🟧 R
Testing									🟧	🟧	🟧 R				

Fig. 4. Methodology's working planning.

setTheReceiver method). Also, in the *onReceive* method, we have to execute the post-conditions code in order to keep the database in the same state as before the testing.

IV. Planning process

The proposed methodology is developed in 15 weeks (8 hours per week). In figure 4, we show the initial Gantt chart in which we illustrate the start and finish weeks of each sub-task in the project. At the end of each sub-task, the students should report about the job made in this sub-task. These reports are a formal and consistent project review. The focus of these project reviews is to ensure that the tasks are being completed on time and if it's not the case, to take all necessary and appropriate actions. These reviews are monitored for a third party which looks at whether the state of the project is as expected.

The planning shown in figure 4 is the result of several tests on different project in which not all students had the same knowledge level. The most critical part for us, it's the implementation phase. In this phase, we start implementing and testing the model class. Many times, this code includes access to external information that can involve unpredictable delays. We must take this into account to avoid a poor user experience.

After the implementation and testing of the model class is finished, we implement the view and presenter classes. In this part, we analyze the different use cases of the application (represented in the sequence diagrams) and implement every one of them.

V. Conclusion

This paper presents a simple programming methodology based on MVP architecture to help our students to implement mobile applications in Android. The main idea of this methodology is to design a set of methods, clear and simple, that any student can follow to develop a complete android app (useful and exiting) in a quadmester.

The reason why proposing this methodology is the complexity of the current methodologies when the developers are relatively inexperienced. This is particularly the case of many students in technical bachelor degrees (non-CS) or entry-level or mid-career professionals with some skills in programming languages, that have to enter the labor market where the mobile devices and their programming, cover almost all the strategic markets or the most profitable. Many of these students are interested in learning to program mobile devices, not because they want to be a mobile programming developer, but because they want to solve small and simple problems using mobile devices in their future and current work.

Our methodology is based on the features of some mobile methodologies: agile, rapid prototyping and RAD, and some of our own. It consists of the following steps: requirements specification and prototyping, design of the simple sequence diagrams, implementation of the *MVP* architecture and testing and it achieves the following goals: clear design, knowledge of application behavior, highly decoupled architecture, easy code modification, a team-based approach to development, quick implementation, code reuse and easy testing and verification process. We have used this methodology in the fourth year of a bachelor engineering degree and in a post-graduate course and have received positive feedback from our students (all students perform a complete and useful mobile app in four months, some of which can be downloaded from Play Store).

Acknowledgment

The author would like to thank to her students to have agreed to be guinea pigs!

References

[1] Web site of WorldWideLearn. Available: http://www.worldwidelearn.com/.

[2] Web site of the International Telecommunication Union. Available: http://www.itu.int/en/ITU-D/Statistics/Pages/publications/wtid.aspx.

[3] *Gartner Says Traditional Development Practices Will Fail for Mobile Apps*. Available: http://www.gartner.com/newsroom/id/2823619, 2014.

[4] L. Vijayasarathy, D. Turk, *Agile Software Development: A Survey of Early Adopters*. Journal of Information Technology Management. ISSN 1042-1319. Vol. XIX, No. 2, 2008.

[5] *8^{th} Annual State of Agile. Survey*. Journal of Information Technology Management. Available: http://stateofagile.versionone.com/, 2014.

[6] M. Aleksy, *An Approach to Rapid Prototyping of Mobile Applications*. IEEE 27th International Conference on Advanced Information Networking and Applications, 2013.

[7] B. Mulloy, A. Languirand, *From Napkin to App: Rapidly Prototype and Build for Mobile*. Available: http://apigee.com/about/resources/webcasts/napkin-app-rapidly-prototype-and-build-mobile.

[8] Rapid Application Development. Available: http://www.blueink.biz/RapidApplicationDevelopment.aspx, 2005.

[9] C. Lee Ventola, *Mobile Devices and Apps for Health Care Professionals: Uses and Benefits*. Journal for Managed Care and Hospital Formalary Management. 39(5): 356364. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4029126/, 2014.

[10] *Young lawyers seek to shake up legal profession with mobile apps*. Available: http://www.bostonglobe.com/business/2014/11/24/young-lawyers-seek-shake-legal-profession-with-mobile-apps/bnNLhfoceZumFg9CrVA3gI/story.html.

[11] *Spiceworks' New Android and iOS Mobile Apps Help IT Professionals Manage Technology Environments While on the Go*. Available: http://www.spiceworks.com/press/releases/2014-05-13/mobile-apps/.

[12] P. Mohan, A. Udaya Shankar, K. JayaSriDevi, *Quality Flaws: Issues and Challenges in Software Development*. Computer Engineering and Intelligent Systems (www.iiste.org). ISSN 2222-1719 (Paper). ISSN 2222-2863 (Online). Vol 3, No.12, 2012.

[13] Web site of Balsamiq Mockups. Available: http://balsamiq.com/products/mockups/.

[14] T. Lowdermilk, *User-Centered Design*, 1st ed. O'Reilly, 2013.

[15] J. Nielsen, *Ten Usability Heuristics* [Online]. Available: http://www.nngroup.com/articles/ten-usability-heuristics/.

[16] M. Potel, *MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java* [Online]. Available: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[17] Web site of Android developers. Available: http://developer.android.com/index.html