A model driven approach for Android applications development

Abilio G. Parada, Lisane B. de Brisolara
Grupo de Arquitetura e Circuits Integrados (GACI)
Centro de Desenvolvimento Tecnológico (CDTec)
Universidade Federal de Pelotas (UFPel) - Pelotas, Brazil
{agparada,lisane}@inf.ufpel.edu.br

Abstract—The mobile application development industry is increasingly growing up due to the intensive use of applications in mobile devices, most of them running Android Operating System. However, developing applications for mobile platforms demands additional worries such as code efficiency, interaction with device resources, as well as short time-to-market. Model-driven Engineering (MDE) combined with UML, as already used in software engineering, could provide abstraction and automation for mobile software developers. To support that, adequate tools and approaches are required. This paper presents a MDE approach for Android applications development, which includes UML-based modeling and code generation in order to facilitate and accelerate the development of mobile applications.

Keywords-Android applications; mobile; Model-Driven Engineering; UML model; code generation; modeling approaches

I. INTRODUCTION

Smart phones become an alternative to personal computers, emerging as a mobile computation device. Most of these devices run Android Operating System. Android [1] is the Google’s mobile platform, which represents an open solution with development tools, a large support to many devices, as well an operating system.

Android applications are developed in Java language, however these do not run on the traditional Java Virtual Machine (JVM). Moreover, a desktop Java application developed to desktop should be adapted mainly due to new concepts, such as Activity and Service, and specific APIs used in the development of Android applications.

Mobile applications are significantly different from traditional applications. Typically, mobile applications are event-driven and have to constantly react with an input from user interfaces or sensors [2]. Additionally, mobile applications as any embedded software should efficiently use limited device resources. On the other hand, the development demands additional worries about the short time-to-market.

In traditional software development, there are many approaches to accelerate design and development. Among these approaches, Model-driven Engineering (MDE) has received attention because provides abstraction through high-level models (e.g. UML), thus facilitating the design of complex software [3]. MDE [4] considers models as key artifacts, and these are used in all phases of software engineering (analysis, design, implementation, and testing). The main objective of this approach is to reduce the gap between the problem domain and the software implementation through the use of technologies that support systematic model transformations. Typically, the effective use of this paradigm, combining models and tools, can provide abstraction and automation, properties surveyed mainly by embedded software designers. According to a study by the Motorola, MDE may reduce the time spent on software production by 70% [5].

More recently, Wang [6] studies the use of UML models for the smartphone applications development. However, most of available modeling tools do not completely address UML-based modeling and code generation for Android applications. Recent efforts propose solutions for this problem [7] [2], but these present some limitations, as discussed in the related work section.

This paper proposes a model driven approach for Android applications development. This approach includes UML-based modeling and automatic code generation to facilitate and accelerate the development of Android applications.

The remaining of the paper is organized as follows. Section II presents the background. The proposed approach is presented in Section III. A case study is described in Section IV, while related works is discussed in Section V. Section VI concludes and points out directions for future work.

II. DEVELOPING ANDROID APPS

The Android is an Operating System (OS) based on the Linux kernel held by Google to mobile devices. Its applications are developed in Java, but run on a specific virtual machine called Dalvik [8]. This platform provides also an API with several components that can be reused to define an Android application. Between them, activity, service, broadcast receiver, and content provider are core components.

A typical Android application consists of one or more activities. An activity usually shows a single visual user interface. Service is a special type of activity that does not have a visual user interface and usually run in the background for an indefinite period of time. Among the core...
components, activity and service typically encapsulate the most part of the behavioral code.

Android applications have to adhere to strict life cycle protocols, typically specified by methods [2]. An activity has its lifecycle defined by the following methods: onCreate, onStart, onResume, onPause, onStop, and onDestroy. The onCreate method is invoked by the OS when starts an application. The onStart puts the activity in foreground in the screen, while onResume is invoked if it lost the foreground, and onPause when the foreground is changed to other application. The onStop method is invoked when the activity is not visible, and onDestroy to run the last task before the activity be closed. To define the lifecycle for a service, the onBind and onDestroy methods are used. The service’s onBind method is used to make a persistent connection with a service, while onDestroy to terminate it.

An activity may transfer control and data to another activity through an interprocess communication protocol called intents. To promote this, a Broadcast Receiver (BR) component is used, which is a dedicated listener that waits for broadcasts messages. When an intent is broadcasted, the BR responds by executing a specific activity. Requesting for common datasets (e.g. contacts, pictures, messages, audio, and files) are handled by the Content Provider (CP), which is a data-centric service that makes persistent datasets available for Android applications. A data requesting is indicated through Uniform Resource Identifier (URI), which provides the standard access to CP.

III. MDE APPROACH FOR ANDROID

This section presents the proposed MDE approach for Android application development, which supports UML modeling and automatic code generation.

A. Modeling

This paper proposes how completely model the structure and behavior of an Android application using standard UML, focusing on specific aspects of these applications, not completely covered by traditional modeling approaches. Our modeling is based on UML, using class diagram to describe the application structural view, and sequence diagrams to represent the behavioral view. Subsection III-A1 presents modeling of main structural components, while the behavioral aspects are described in subsection III-A2.

1) Structural View: Activity and Service (in gray in Fig. 1) are special components used to define the application structure and the developer should specify which application classes are activities and which are services. Fig. 1 illustrates the proposed approach, where Main is defined as an Activity and Background as a Service using inheritance relationship.

The Activity and Service classes have several default methods (see Section II) that often are customized by the designer. Thus, with the indication in the model of which methods must be customized, standard code (e.g. calls for the superclass constructors) can be generated following characteristics of each method. In the example illustrated in Fig. 1, two methods (onCreate from Main and onBind from Background) are indicated to be customized for the application.

This section focuses on specific aspects of Android applications structure. To define more details about the structure (as definition of classes/interfaces, attributes, operations and relationships), the designer can use UML class diagram notations as used on traditional software modeling.

2) Behavioral View: In our approach, the application behavioral view is represented using sequence diagrams, which can represent iteration, conditional, and message exchanges, as usual for traditional object-oriented software.

However, the behavior of an Android application can evolve components as Content Provider and Intention, as discussed in Section II, when an application class can request some data or some service from another component. It should be described in the UML model to provide a complete application behavioral view. Fig. 2 illustrates the proposed representation for a data request to the Content Provider (CP), modeled by the contact message sent by the Main class to CP.

A service request is handled by the Broadcast Receiver (BR), as explained in Section II, and our proposal for modeling it is illustrated in Fig. 3. The Main lifeline indicates that this activity will be held in the Main class. Firstly, the intent request is performed, sending a message to the Intent lifeline and using ACTION_CALL to define the required intents action. The return of this operation will be assigned to the intentName object. The second action is the functionality request, performed by the startActivity message, which has as argument the Intent object.

The methods inherited from Activity or Service are often customized by the programmer and the behavior for each customized method should be detailed. The onCreate
method, inherited from Activity, describes the first task executed when the application is started. In the model illustrated in Fig. 4 is represented the basic behavior of this method, which is related to CP and BR components, as indicated by Ref fragments. Thereupon, a service is requested through the sent messages onStart and onBind. The first message is used to start the service, while the second to make a persistent connection to this.

**B. Code Generation**

The main benefit of the MDE adoption in the development of Android applications is the automation. This way, to demonstrate the automation support provided by our MDE approach, we are extending the GenCode [9] tool to generate Android code. As the original GenCode, our Android code generation is based on class and sequence diagrams.

From the class diagram, GenCode generates the application structure, producing a Java file for each class. It embodies class definition with attributes and methods, including also constructor, setting and getting methods. Relationship between classes or interfaces, as inheritance and association, are considered during the code generation. When the class represents an Android API component, the generation also includes necessaries imports, according with attributes declaration and parameters of methods. Furthermore, the code generation also provides invocation for Activity and Services standard methods, like onCreate.

The behavioral code generation is based on sequence diagrams, what limits the code generation until the method invocations level, thus, simple operations like variable attributions or math operations cannot be generated. From this diagram, the sequence of method invocations is captured, including argument and return for each invocation. Moreover, loops and conditionals are also captured from this diagram producing the corresponding Java statements. When this diagram presents an Android element, like an Intent, the tool provides the intent creation according to the action call made and the invoked service.

**IV. CASE STUDY**

The Snake, an open source application from Android developer web site [10], was chosen as case study. Although it is a simple example of Android apps, it allows us to demonstrate the main features of the proposed approach.

To obtain a UML model from Snake code, manual reverse engineering was used. As result, a UML model was built using Papyrus [11], representing static and dynamic aspects of the application. This model consists of one class diagram and twenty-three sequence diagrams.

The Snake static view, represented by the class diagram is depicted in Fig. 5. The Android classes, Activity, Intent, View, Handler, Bundle, and Canvas, are highlighted in gray in this model, while the application classes (Snake, SnakeView, and TileView), and two SnakeView inner classes (RefreshHandler and Coordinate) are illustrated as white-boxes.

Snake is the main application class and represents an Android activity, as indicated by the inheritance relationship. For this class, onCreate, onPause, onSaveInstanceState, and onBackPressed methods are explicitly defined, indicating that these methods will be customized. This class is responsible by the user interface and is associated to the SnakeView, which is also used in the graphic user interface definition.

The graphic view descriptions are given by the TileView class, SnakeView class, and its inner classes, RefreshHandler and Coordinate. TileView is responsible by the described view, as indicated by the inheritance relationship with the...
View class. TileView overrides the onSizeChanged, and onDraw methods, inherited from View. The SnakeView class is responsible for the game management, describing and managing the game elements. The RefreshHandler inner class, a specialization of Handler, performs the animation, while Coordinate is responsible for elements management, as mAppleList and mSnakeTrail.

The customization for the onCreate method is represented by the onCreate sd (sequence diagram) from Fig. 6. This diagram describes the behavior when the application is started. Basically, it checks the last game and chose to start the game for a new or a previous state. In this, two lifelines represent the mSnakeView and savedInstanceState objects, which interact on this scenario. The alt fragment represents an if/else, when the conditional is true, a new game is load, otherwise, a previous state is load. The Bundle component is used to restore a previous state.

Figure 7 depicts the onBackPressed sd, in which the modeling of an intent can be observed. This diagram represents the tasks executed when the Back button is pressed, pausing the game and calling the mainscreen. The game state is saved and when the application is called again, it is started from the previous game scene. To implement this behavior, an intent is used. Following the represented sequence, the intent object should be created and the argument passed in the message, in this case ACTION_MAIN, identifies the required action. After that, the two following messages, addCategory and setFlags, represent the addition of new features to the intent. In the sequence, the startActivity service is invoked using as argument the setIntent object.

Applying our approach, we produced Java code from the Snake UML model using the extension of GenCode. Listing 1 illustrates the code generated for the Snake class, declaring imports for Android API components, the Activity extension, and class attributes.


```java
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class Snake extends Activity {
    /** Attributes */
    private SnakeView mSnakeView;
    private String ICICLE_KEY = "snake_view";
    ...
```

Listing 2 illustrates the generated code for the oncreate method, where can be observed the invocations to the super.onCreate method and setContentView, standard invocations to the onCreate method and also the behavior specified by oncreate sd from Fig. 6.

```
/** Methods */
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.Snake);
    /** Specified by sd onCreate */
    mSnakeView.setMode(SnakeView.READY);
    if (savedInstanceState == null) {
        mSnakeView.restoreState(savedInstanceState);
    } else {
        Bundle map = savedInstanceState.getBundle(SnakeView.PAUSE);
        if (map != null) {
            mSnakeView.restoreState(map);
        }
    }
```
Listing 2. Generated Code for Snake - onCreate Method.

The generated code from the onBackPressed method, which demonstrates an intent usage, is depicted in Listing 3. In this code, the creation of the setIntent Intent, as ACTION_MAIN, is specified and after that, two methods, addCategory and setFlags, are invoked to setIntent. Finally, the service is invoked through the startActivity message with the setIntent argument. The tool also generates standard calls for Activity methods as super.onPause inside of onPause, and outState.commit in onSaveInstanceState, as can be observed in Listing 3.

Listing 3. Generated Code for Snake - onBackPressed, onPause and onSaveInstanceState Methods.

Listing 4 demonstrates the generated code for the onDraw method modeled in the onDraw sd from Fig. 8. This code includes the declaration of the loop statement (for) with its conditionals and counter along with the conditional if, as well as the invocation of the drawBitmap method from the canvas object, which embodies its arguments.


As the original GenCode, its extension also supports the generation of constructor methods, attributes with default values, setting and getting methods, and inner classes. Listing 5 illustrates code fragments from the SnakeView class, demonstrating the generated code lines, which include declarations of attributes, and setting and getting methods.

Listing 5. Generated Code for SnakeView.

V. RELATED WORKS

There are a huge number of tools that provide software modeling support using graphical notations as UML, the standard object-oriented modeling language. Some of these are able to generate code from UML models, as UModel [12], and Artisan Studio [13]. However, these tools focus on traditional software development, and do not completely support Android applications development.

To turn easier the development of Android Apps, App Inventor [14] was proposed, which is a web-based tool to produce simple Android applications. However, its approach is considered as more “visual programming” instead of “modeling”.

Recent efforts have focused on MDE support for Android applications proposing specific solutions and tools [2] [15] [7]. AndroMate [15] is an Eclipse modeling tool for Android development which supports graphical modeling of Android applications and generate some code. Although this approach is proposed as model-driven, it does not use UML and we consider more close to visual programming approaches than model-driven ones.
IBM Rational Rhapsody [7] is a commercial tool for software modeling, and recently, it was extended to support modeling and code generation for Android applications. Although, it is a complete solution, in order to turn possible the complete code generation, the application behavior should be specified in low details, making the code generation almost a mapping 1-to-1.

The use of UML activities, like building blocks to construct applications, is proposed by Arctis [2]. These activities are after translated to a state machine in order to generate code and other files needed to wrap the state machines into an executable Android application. Authors use a small UML profile that defines specific stereotypes for Android modeling. Differently from this approach, we propose the use of UML sequence diagram to represent application behavior. Furthermore, our approach was defined using UML standard notation without new stereotypes, and thus, allowing the use of any UML tool for Android application modeling and not requiring any new knowledge.

VI. CONCLUSIONS AND FUTURE WORKS

This paper proposes a MDE approach for Android applications development, which addresses how to model specific aspects of Android applications, as an intent and a data/service request, using standard UML notations. Moreover, it supports static and behavioral code generation from UML class and sequence diagrams, according to the rules imposed by the Android platform. To demonstrate our approach, a case study was conducted, in which an Android application was modeled in UML and code was generated from it. To generate code, the extension of GenCode was used.

However, the actual version of GenCode tool that supports the proposed approach, only made an automatic transformation from UML class and sequence diagrams to the target Android Java code, without consider any optimization in the generated code. As future work, we plan to extend this tool in order to consider the good practices for Android development [16], and thus generating efficient code.

ACKNOWLEDGMENT

The authors acknowledge financial support received from CNPq, and NESS project (PRONEX-10/0043-0).

REFERENCES


