Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 19 (2013) 1004 – 1011

Procedia
Computer Science

The 3rd International Symposium on Frontiers in Ambient and Mobile Systems
(FAMS)

Component Based Framework to Create Mobile Cross-platform Applications

Joachim Perchat[a,b], Mikael Desertot[b], Sylvain Lecomte[b]

[a]*Keyneosoft, 31 rue de la Fonderie, 59200 Tourcoing, France*
[b]*Univ Lille Nord de France, F-59000 Lille, France, UVHC, LAMIH, F-59313 Valenciennes, France*
CNRS, UMR 8201, F-59313 Valenciennes, France
jperchat@keyneosoft.fr, Mikael.Desertot@univ-valenciennes.fr, Sylvain.Lecomte@univ-valenciennes.fr

Abstract

Smartphones provide a set of native functionalities and another set of functionalities available through third-party applications. The emergence of more and more actors, without standards to provide their devices or OS, stops the cross-platform development. Indeed, a developer would have to learn many programmatic languages and create many user interfaces for many devices. To resolve this problem, several solutions often consist in the creation of a common SDK to only write the application once. Then, the application code is translated in native code for each target platform. In this paper, we propose a solution based on a component model. A set of configurable components is implemented for the targeted platforms. A component will have a common interface independent from the host OS. Finally, a new language will offer developers a single instruction call to any component through its interface. This instruction is common on any platforms to simplify the implementation of a cross-platform application.

Keywords: Cross-platform, Cross-compiler, Generation of code, Component integration, Universal language

1. Introduction

Smartphones are more and more powerful and their OS support different hardware and software facilities like multi-touch screens, 3G, Wi-Fi, and particularly the ability of installing third-party applications. Through these new applications, new functionalities have emerged, especially for the user's mobility. For example, thanks to them, a user can track his way in a city, shows information in real time (augmented reality), pays his metro tickets ... However, the mobility causes many variations of the context (indoor/outdoor, city/highway, area without 3G coverage ...). Therefore a new type of applications is born: the context-aware applications.

Four problematics can be highlighted in the mobile domain [1]. The first one is user experience: the goal is to always define the best user interface, for a simple and intuitive use. The second problematic is linked to resource and power consumption: even if the devices are more powerful than before, they are still limited in memory, power, etc. The third one is application maintenance: a mobile OS is often updated, making some applications unusable on the new version. So, how can we test and maintain an application efficiently? Finally, a major problem is due to the heterogeneity of devices and mobile operating systems. There are various operating system providers and device constructors on the mobile market. For the time being, it is impossible to implement an application once, and deploy it on all smartphones. Indeed, they all use different technologies and programming rules. In this paper we tackle the last problematic. Our goal is to smooth the differences between the smartphones in order to simplify the development of cross-platform third-party applications. The paper is structured as follows : Section 2 shows the mobile development context, Section 3 presents some cross-platform tools, Section 4 and 5 explain our proposal and our first implementation. We conclude and present our perspectives in Section 6.

2. Mobile development context

We consider the market of smartphones as divided in two categories: the device constructors and the operating system providers. The telephone service providers have no influence on our problematic.

The device constructors build, assemble the hardware (screen, GPS chip ...) and configure one OS on their devices. The configuration allows the use of specific equipments like camera, GPS, accelerometer, Wi-Fi, 3G, NFC ... Of course, each of these companies has its own rules in defining the structure of a phone. The operating system providers produce operating systems that make the link between the users and the devices. This market is constituted of Apple with iOS, Open Handset Alliance (OHA) with Android, Microsoft with Windows Phone, RIM with BlackBerry, Samsung with Bada, Nokia with Symbian, etc. Today, the two most popular smartphone operating systems are iOS and Android. But, like the device constructors, these companies create operating systems which are not compatible with one another. The architectural differences due to the mobile constructors are not a problem from our point of view. Indeed, the developers of third-party applications don't have directly access to this layer. They only have access to the APIs of the operating system which accesses afterward the hardware layer. With the Table 1, we give rise to some of the points that differ when a developer implements an application on several mobile platforms.

In the following sections, we are going to present the problems resulting from these differences. Then, we give the main requirements to resolve our problematic.

Table 1: Some differences between several mobile operating systems.

Operating system	Virtual machine	Program. language	User interface	Memory mgmt	IDE	Development on:	devices
iOS	No	Objective-C	Cocoa Touch	reference counting	XCode	Mac OS X	homogenous
Android	Dalvik VM	Java	XML files	garbage collector	Eclipse	multi-platform	heterogenous
Windows Phone 7	CLR	C# and .Net	XAML files	garbage collector	Visual studio	Windows Vista / 7	homogenous
BlackBerry OS	Java ME	Java	In code	garbage collector	Eclipse	multi-platform	heterogenous
Symbian OS	Possible	C++	Qt	manual	Qt Creator	multi-platform	heterogenous

2.1. Problems

Each platform uses different tools, programming languages, user interface declarations and memory management. If a developer wants to create an application for every platform, he will have to buy one PC and one Mac. Then, he will have to follow different formations, one by platform. Finally, he will have to buy at least one phone for each kind of platform and sometimes even multiple phones for one platform, like for the Android case.

Most of those constraints must be respected. For example, we are not going to change the development environments. Indeed, these tools are very efficient (debug, device monitoring ...), so the implementation of a new environment is useless.

The application design and implementation steps are the two critical phases to create a cross-platform application. Indeed, even if an application must run the same functions on every platform, it is impossible to design it once and run it everywhere. Depending of the host, the application behaviors can be different too.

To demonstrate the importance of these two steps in the application creation process, we have developed a basic third-party application for iOS and Android. This application is made up of a tab bar with two tabs. On each tab, there are basic views (buttons, text fields) with the target platform look. Each button launches a functionality like the creation of a database, the capture of a picture, the navigation in photo album, the opening of a web page ... From one tab, users can open a web site. They can also open a new page to manage the camera or a database. On the camera management page, users can launch the camera and take a picture or navigate through the photo album. On the database management page, they can create or remove a database. Finally, from another tab, they can send a text to another view.

The implementation for the two platforms of this basic application is already complicated for a developer. He must navigate between different concept. For example the developer can use a navigationController on iOS whereas this object doesn't exist on Android. Then, on Android, to launch certain actions, we must used intent process (navigation between views, hardware access ...) whereas on iOS, this notion is missing. For a mobile developer, it is very difficult to change its thinking manner, according to the platform, during the implementation process. Therefore, we have to hide the uncommon notion between target platforms.

2.2. Requirements

Developers need a unified way to design, to implement and/or to run a third-party application on all available platforms, which allows to use every components (software or hardware) on each host. The unification doesn't mean that we want to lose the specificity of each platform.

The applications generated with such solution must be efficient. The application success is often due to its reactivity and its appealing design. At the same level, if the solution must be installed with the generated application on the smartphone (e.g. virtual machine), it must be lightweight because smartphones have limited resources.

Besides, because of the smartphones way of use, another constraint is added: the user's mobility. The mobility causes many context modifications (network loss, location variations) and according to this context, third-party applications must able to change their behavior. Therefore, we want a solution that considers and supports the creation of context-aware applications. This constraint accents the previous need: the efficiency, to manage properly the user's mobility.

Finally, this solution needs to be easily adaptable. Indeed, the mobile domain can evolve. For example, in a couple of years, Apple's iOS might not be present anymore and a new participant might take its place. So, the possibility of adding extensions must be considered in order to manage new platforms easily. In the best case, as soon as a new platform brings out, our proposal must be able to integrate it without modifying its internal architecture.

3. Related work

The solutions which enable the implementation of cross-platform applications, can be classified in three categories: the cross-compilers, the solutions based on the model-driven engineering and finally the interpreters of source code.

MoSync[1], **Corona**[2], **Neomades**[3] and **XMLVM** [2] are based on **cross-compilers**. These solutions aim the cross-platform application creation through a common language. Then, the instructions written with this common language are translated in native language. In this case, the reused code is complete but the mapping between the application language and the native target language is very difficult to achieve. That's why, in the most cases, the cross-compiler only manages few platforms and is limited to common elements from each platform.

One other part of existing solutions is based on **model-driven engineering**. To one side, **UsiXML**[3] and **Jelly**[4] allow users to produce user interface for multiple platforms. On the other hand, **MobiA modeler**[5] and **AppliDE**[6] (that is integrated in CAPucine [7]) allow users to produce a complete application and even a context-aware application. To create context-aware application, CAPucine's designers allow users to separate their applications in modules. Some of them will be integrated in the application at the transformation whereas the others will be loaded at the execution according to the context. To resume, with these solutions users can define theirs applications from models. Then, these models are transformed into source code for each target platform. But like cross-compilers, the translation between models and native source code is difficult to achieve if the developers want to manage any native component.

Interpreters translate, in real time with a dedicated engine, a source code to executable instructions. Users implement their cross-platform application and the interpreter manages their execution on many platforms. We can identify two categories in the mobile interpreter domain: virtual machines (VMs) and solutions based on web languages.

The most famous technology based on **VM** is **Java ME**. But, this technology is unpopular and isn't used by mobile developers. For all that, many variations based on it exist: **J2ME Polish**[4], **Bedrock**[5], **AlcheMo**[6]. They often consist to port (such as cross-compilers) a Java ME application with some extensions on several platforms. In [8], the language isn't Java but a new language dedicated to the mobile domain: **MobDSL**. Thereafter, the applications written with MobDSL would run on a VM.

[1] MoSync: http://www.mosync.com/ [2] Corona: http://www.anscamobile.com/ [3] Neomades: http://neomades.com/ [4] J2ME Polish: http://www.enough.de/ [5] Bedrock: http://www.metismo.com/ [6] AlcheMo: http://www.innaworks.com/

Finally, there are the solutions based on web technologies. **PhoneGap**[7], **QuickConnectFamily**[8], **Rhomobile**[9] allow to load, on the device, web applications (similar to websites) able to access the device hardware. Another solutions allow the execution on multiple platforms of widgets (”small” applications for mobile devices) from a cross-platform engine as **xFace**[9] or **Opera**[10]. In [10], xFace's designers explain how xFace is ported on many platforms. They defined a porting layer which is the combination of several components (e.g. file systems, graphics) common to each platform. This separation simplify the mapping between the engine and the target OS. Furthermore, in [11], the authors allow the automatic generation of widgets with a new language: Widget Markup Language. As for **Titanium mobile**[11] and **Flex** linked to Flash builder[12], they enable the use of web languages like any programmatic language in order to implement cross-platform application.

Even so, these solutions give us some interesting directions to follow, after running some tests, no tool could respond to all our needs. They are often limited to most common hardware features. Therefore, it is impossible to provide an acceptable user experience for our needs, and applications obviously have less interesting performances than native implementation. That's why, we propose our solution.

4. Proposition

We propose a hybrid solution to tackle the limitation described above. It is a combination between native code and a new universal language based on a declarative language similar to Java annotations. This new language consists of a layer added to the application, giving access to a set of components and their methods.

The third-party application developers will have to implement the minimal structure of the application (i.e. the view and the navigation between views) in native code in order to specify how and where to integrate the components declared with our language. Then, they will specify with our declarative language the use of components and let our associated tool manage the code integration inside their native code. From another point of view, the component developers must provide the implementations of components for any target platforms, as well as the interfaces (one by component and one common to its different implementations) in another language.

This solution need several steps to be fully achieved. The first step is the definition of a set of components that can be integrated in any application (on any platform). The second consists of defining a declarative language for describing components, or rather the use of a component. The last step is their integration in a native application.

4.1. Component layer

During the first step, we will consider the component developers' point of view. The goal is to implement a set of components which perform one or several functions on any mobile platform. These components must be able to access any hardware native functions like camera, GPS and any native software like buttons, text fields, map, etc.

First of all, we define a component. It consists of one public interface that represents its functions and its internal process that is hidden to user. In addition our components should be integrated on different platforms. Therefore, each targeted platform will have a particular native implementation (like Objective-C for iOS, etc.) by component. Using native languages gives the best possible efficiency and a full access to each platform. The public interface is common to all its implementations. This is the only access point of a component.

Because the problem of platform fragmentation isn't resolved for the component developers, we are working on a methodology that unifies component implementation for the different platforms. Main goals are both creating a unified component lifecycle and a common component access. Component lifecycle is already following similar patterns in the component programming domain (init, start, stop ...). We intend to standardize the component creation and access through naming conventions and common internal structure (same filenames, same method names). Using these rules, we want to create a tool able to generate a component template according to the target platforms and its functions. Even if this tool won't hide the differences between platforms, it will ease and unify the components development.

A last point to tackle is the specification of a common interface, which must be the same for each implementation of a component independently of the target platform. Each component defines its behavior through this interface, and provide a link for each target platform. In order to access this shared interface, we must implement a declarative language that users could use.

[7] Phonegap: http://phonegap.com [8] QuickConnectFamily: http://www.quickconnectfamily.org/ [9] Rhomobile: http://www.rhomobile.com/ [10] Opera: http://dev.opera.com/addons/widgets/ [11] Titanium mobile: http://www.appcelerator.com [12] Flash builder: http://www.adobe.com/products/flash-builder.html

4.2. Universal language layer

The second step of our solution is the design of a universal language representing a link between component methods and applications. In this part, we refer to a component as being an entity with a public interface and several native implementations. The different implementations are hidden to the users (the third-party application developers who benefit from our solution to write applications on multiple target platform).

To get rid off the difficulty of developing for more than one platform, we propose to abstract components by platform independent shared interfaces that the users access and call thanks to our language. This language will allow users to declare, instantiate a component and use its methods from its public interface. A single instruction will be enough to completely declare a component with its behavior for a specific application. In addition, this instruction could be reused on all platforms and to hide to users the differences between component's implementations.

This solution would be efficient only if the users can integrate our set of components from any place of their source code. Our language must be enough flexible to achieve that. A language similar to annotations could be suitable for this. Indeed, annotations can be written in any place of a source code and linked with any variables or methods.

4.3. Component integration

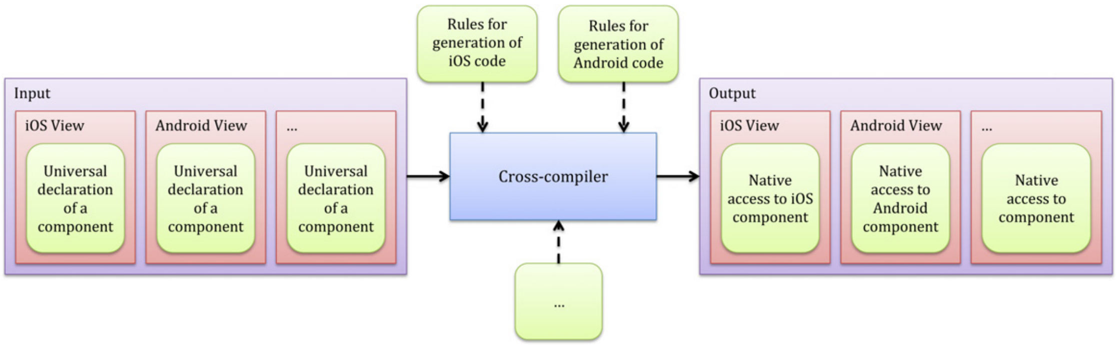

Rules for generation of iOS code
Rules for generation of Android code
Input
iOS View
Universal declaration of a component
Android View
Universal declaration of a component
...
Universal declaration of a component
Cross-compiler
...
Output
iOS View
Native access to iOS component
Android View
Native access to Android component
...
Native access to component

Figure 1: Translation of the declaration of a component to native instructions that call this component on any platform.

Two solutions could be considered to integrate a component declared with our language into a native application: either at compilation time or at execution time. We choose the implementation of a cross-compiler that manages the native source code as well as the instructions in our language, as shown in [Figure 1]. The integration of components in the source code at compilation allows to have an efficient application. Besides, the translation between our language and native instructions will be very simple because a component is developed for any target platform with native instructions. We won't have to create complicated rules to link a component, our compiler will only have to instantiate and generate the native function calls with parameters declared by the users in our language.

In this case, the applications generated by our proposition will be static whereas we would like to have dynamic applications, or rather context-aware applications. To handle this application type, it is interesting to manage two types of components: static and context-aware. The static components will be integrated at the compilation time whereas the others will be loaded at the execution time by a module dedicated to the context monitoring [12]. Therefore, this module would be implemented like any cross-platform component defined in our solution. However, the components loaded by this module could be different according to the context (the platform itself being a context element).

Regarding component integration, we must consider that new operating systems can appear on the market. Our solution would have to adapt to this new situation with two main additions : implementing components with the new SDK and just adding a new platform to the cross-compiler (a simple file of rules).

5. First implementation

In this section, we followed the three steps previously presented. We began by the implementation of a number components (for iOS and Android). Then, we defined two possible instructions with our universal language. These two instructions enable to use component methods in a mobile native project. Finally, we developed a prototype of our cross-compiler which is able to translate the two possible instructions in native code for an Android project. The implementation for a iOS project is in progress.

```java
public interface DeviceInfoManagerInterface {

    public void initWithTag(String tag);
    public String getUUID(Context context);
    public String getBatteryLevel();
}
```

Figure 2: Native interface on Android

```objc
@interface DeviceInfoManager : NSObject

-(void)initWithTAG:(NSString*)tag;
-(NSString*)getUUID;
-(NSString*)getBatteryLevel;

@end
```

Figure 3: Native interface on iOS

Here, we present only one component which enables to get information on the device like its UUID and its level of battery. Before using it, the users must initialize the component with a TAG. This TAG will be used in the log files. This component has a native interface on each target platform [Figures 2, 3] which are gathered together in a common interface [Figure 4]. Of course, only the common interface will be visible whereas all the native code (interfaces or implementations) will be hidden to users. We want to hide the native code to facilitate and unify the use of components. The common interface is deliberately vague, the main goal of this interface is to inform users on the component's possibilities independently of a platform. This interface is coming with another one which contains the details on the implementations for the different target platform [Figure 5]. To summarize, the first interface enables users to understand the component whereas the second will be used by our tools to handle the components.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="CommonInterfaceSchema.xsd" >

    <description>
        <nameComponent>DeviceManager</nameComponent>
        <version>1.0</version>
        <date>2012-11-12</date>
        <description>Get different information of the host device</description>
        <extraFile nameFile="ComplementInterface.xml"></extraFile>
    </description>

    <platforms>
        <target name="Android" nameExecutable="DeviceManager.jar"/>
        <target name="iOS" nameExecutable="DeviceManager.framework"/>
    </platforms>

    <configurations>
        <configuration name="initWithTag" description="init the tag used in log files">
            <parameters name ="tag" type="String"
                io="in" description="Tag used in log files"/>
        </configuration>
    </configurations>

    <methods>
        <method name="getUUID" description="Get the device's UUID">
            <parameters type="String" io="out" description="UUID of the device" />
            <!-- Others parameters can be dependant of the platform -->
        </method>
        <method name="getBatteryLevel" description="Get the device's battery level">
            <parameters type="String" io="out" description="Battery level of the device" />
        </method>
    </methods>
</Component>
```

Figure 4: Common interface

```xml
<methods>
    <extraInformationOnMethod methodName="getUUID"
        description="Get the device's UUID">
        <iOS className="DeviceInfoManager">
            <parameters type="NSString*" io="out"
                description="UUID of the device" />
        </iOS>
        <android className="DeviceInfoManager"
            packageName="fr.keyneosoft.devicemanager.implementations">
            <parameters type="String" io="out" description="UUID of the device" />
            <parameters name="context" type="android.content.Context" io="in"
                description="Context of the android application" />
        </android>
    </extraInformationOnMethod>

    <extraInformationOnMethod methodName="getBatteryLevel"
        description="Get the device's battery level">
        <iOS className="DeviceInfoManager">
            <parameters type="NSString*" io="out"
                description="Battery level of the device" />
        </iOS>
        <android className="DeviceInfoManager"
            packageName="fr.keyneosoft.devicemanager.implementations">
            <parameters type="String" io="out"
                description="Battery level of the device" />
        </android>

    </extraInformationOnMethod>
</methods>
```

Figure 5: Part of the complement interface

For the first implementation we defined two possible instructions in our universal language. The first one enables the use of a component method [Figure 6] whereas the second enables users to declare an input parameter of a method [Figure 7]. These two instructions are common to any platform. In order to have flexible instructions, they are based on annotations. Indeed, thanks to this type of language, users can insert our instructions anywhere in their code. Usually, annotations allow users to declare a particularity of their class or their variable. There, the main goal is to link a method or a method parameter of a component with a native variable (iOS, Android ...) declared in the code of users. For that, each annotation describes either a method or a method parameter of a component and must be integrated before the variable to link. For example, in the [Figure 6], a method instruction is placed before the native variable 'UUID'. Therefore, the result of the method described in the annotation (getUUID) will be assign to the variable 'UUID'.

Finally, we have developed a compiler which is able to translate our instructions in native codes. For now, only the method annotation needs translation [Figure 6]. The var annotation only allows to declare the variables used in

```
@Method(componentName = "DeviceInfoManager", configurationName = "initWithTag", methodName = "getUUID")
String UUID = null;
```

Figure 6: Example of method instruction on Android. It enables to link the variable 'UUID' with the result of the method 'getUUID' in 'DeviceInfoManager'. Before using the method 'getUUID', the component must be initialized with the configuration method 'initWithTag'.

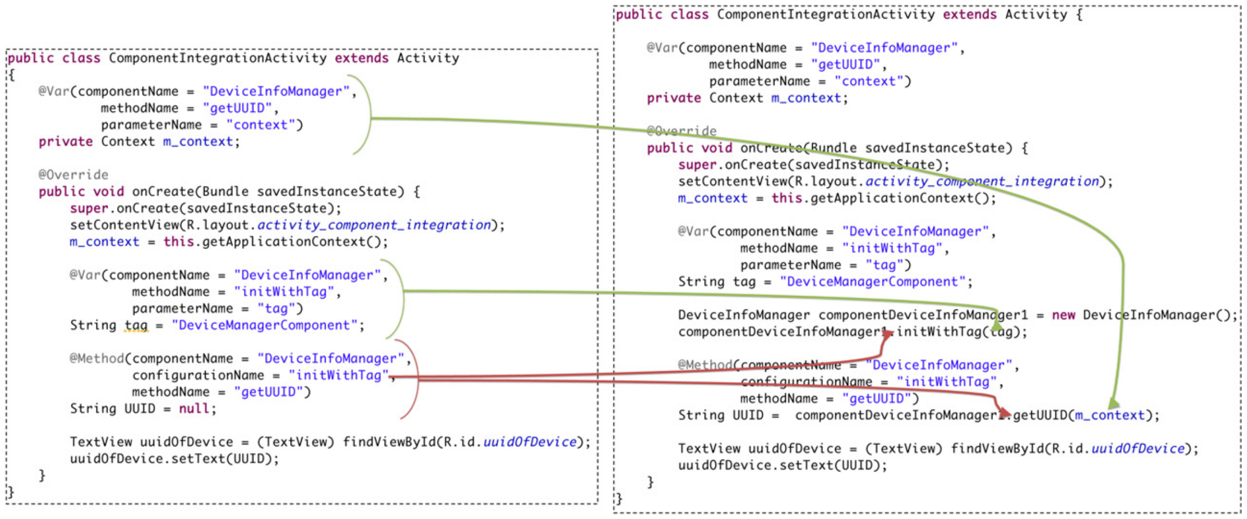
```
@Var(componentName = "DeviceInfoManager", methodName = "getUUID", parameterName = "context")
private Context m_context;
```

Figure 7: Example of var instruction on Android. It enables to link the parameter 'context' of the method 'getUUID' in 'DeviceInfoManager' with the var 'm_context' declared in the native code of the user.

the method calls [Figure 7]. Therefore, we must translate the method annotation to native method call with the input parameters declared in the annotation. To show the result of our compiler, we have developed a basic application on Android with one label that will contain the UUID of the device.

In the first part of the [Figure 8], we have developed an activity (android view) which use our component 'DeviceInfoManager' and more precisely the method 'getUUID'. To use it, we must declare a variable 'context' which will be passed in the input of the method [Figure 2]. So, we wrote a var instruction before the variable 'm_context'. Therefore, 'm_context' will be passed in the input of the method 'getUUID'. Then, we want to initialize the component with the method 'initWithTag' which needs a String in the input. Like 'm_context', we declared a variable 'tag' with a var annotation to link it with the method. Finally, we called the method 'getUUID' with a method instruction before the native variable 'UUID'. As a result, the variable 'UUID' will receive the result of the method.

In the second part of the [Figure 8], our compiler has generated the code associated to our universal language. The generation is immediate. The translation between the method instruction result in the addition of three native instructions. The first one is the instantiation of the component in the variable 'componentDeviceInfoManager1'. Then, this new object calls 'initWithTag'. Finally, the variable 'UUID' receives the result of the method 'getUUID'.



```
public class ComponentIntegrationActivity extends Activity
{
    @Var(componentName = "DeviceInfoManager",
          methodName = "getUUID",
          parameterName = "context")
    private Context m_context;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_component_integration);
        m_context = this.getApplicationContext();

        @Var(componentName = "DeviceInfoManager",
              methodName = "initWithTag",
              parameterName = "tag")
        String tag = "DeviceManagerComponent";

        @Method(componentName = "DeviceInfoManager",
              configurationName = "initWithTag",
              methodName = "getUUID")
        String UUID = null;

        TextView uuidOfDevice = (TextView) findViewById(R.id.uuidOfDevice);
        uuidOfDevice.setText(UUID);
    }
}
```

```
public class ComponentIntegrationActivity extends Activity {

    @Var(componentName = "DeviceInfoManager",
          methodName = "getUUID",
          parameterName = "context")
    private Context m_context;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_component_integration);
        m_context = this.getApplicationContext();

        @Var(componentName = "DeviceInfoManager",
              methodName = "initWithTag",
              parameterName = "tag")
        String tag = "DeviceManagerComponent";

        DeviceInfoManager componentDeviceInfoManager1 = new DeviceInfoManager();
        componentDeviceInfoManager1.initWithTag(tag);
        @Method(componentName = "DeviceInfoManager",
              configurationName = "initWithTag",
              methodName = "getUUID")
        String UUID =  componentDeviceInfoManager1.getUUID(m_context);

        TextView uuidOfDevice = (TextView) findViewById(R.id.uuidOfDevice);
        uuidOfDevice.setText(UUID);
    }
}
```

Figure 8: First : code written by users, Second : code after integration of our component.

The component integration process will be the same on any platform. If the user wants to implement the same application on several platforms, he will use the same instructions. Indeed, the method instructions will always be the same. Only, the var instruction could change. According to the platform, methods can have more or less parameters.

For example, in the android and iOS interfaces of our component [Figure 2, 3], the method 'getUUID' has a parameter on Android whereas on iOS the method doesn't have parameter.

To summarize, the users can write the same instructions to integrate a component in any application running on any platform. Then, it is our cross-compiler that will manage the code integration and, therefore, hide the component access and the component implementation details.

6. Conclusion & perspectives

The development of third-party applications for smartphones is difficult because of the multitude of platforms. Each platform has its own standards (application architecture, user interface, etc.). All these differences limit cross-platform development and innovation for the mobile domain. Indeed, the developers mainly focus on the management of many platforms instead of new functionalities. After running some tests, no tool could respond to all our needs. That's why we propose an alternative, allowing users (i.e. the developers of mobile application), to declare and integrate components from a new language in a native application. This language will soften all the differences between target platform and users will only focus on the business logic layer.

Components are implemented in native language to allow the use of all hardware and software native elements and provide the best possible efficiency for applications. Besides, even if the final result of a component is always the same on any target platform, its intermediate process could be different according to the specificities of target platforms. The goal is to keep the special features of each platform. However, the component developers will have to manage the differences between platforms. But, to simplify this development, we are defining a list of rules enabling the definition of a component with a uniform (naming conventions) and platform independent manners. Then, to unify the component integration, we are defining a universal language based on annotations. Finally, we are designing a cross-compiler able to generate a native source code from our universal language.

Our proposition is currently in its first phase. Development and maintenance will then be experimented in collaboration with mobile application developers. From this work, we will enhance our universal language (addition of new instructions and semantic between instructions) and the compiler. Then, we will formalize a cross-platform module capable of loading components according to the context.

References

[1] A. I. Wasserman, Software engineering issues for mobile application development, in: Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER '10, ACM, 2010.

[2] A. Puder, I. Yoon, Smartphone cross-compilation framework for multiplayer online games, in: Proceedings of the 2010 Second International Conference on Mobile, Hybrid, and On-Line Learning, ELML '10, IEEE Computer Society, 2010.

[3] F. J. Martinez-Ruiz, J. Vanderdonckt, J. M. n. Arteaga, Context-aware generation of user interface containers for mobile devices, in: Proceedings of the 2008 Mexican International Conference on Computer Science, IEEE Computer Society, 2008.

[4] J. Meskens, K. Luyten, K. Coninx, Jelly: a multi-device design environment for managing consistency across devices, in: Proceedings of the International Conference on Advanced Visual Interfaces, AVI '10, ACM, 2010.

[5] F. Balagtas-Fernandez, M. Tafelmayer, H. Hussmann, Mobia modeler: easing the creation process of mobile applications for non-technical users, in: Proceedings of the 15th international conference on Intelligent user interfaces, IUI '10, ACM, 2010.

[6] C. Quinton, S. Mosser, C. Parra, L. Duchien, Using multiple feature models to design applications for mobile phones, in: Proceedings of the 15th International Software Product Line Conference, Vol. 2 of SPLC '11, ACM, 2011.

[7] C. A. Parra, C. Quinton, L. Duchien, CAPucine: Context-Aware Service-Oriented Product Line for Mobile Apps, ERCIM News 88.

[8] D. Kramer, T. Clark, S. Oussena, Platform independent, higher-order, statically checked mobile applications, International Journal of Design, Analysis and Tools for Circuits and Systems 2 (1).

[9] F. Jiang, Z. Feng, L. Luo, xface: A lightweight web application engine on multiple mobile platforms, in: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, CIT '10, IEEE Computer Society, 2010.

[10] B. Pan, K. Xiao, L. Luo, Component-based mobile web application of cross-platform, in: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, CIT '10, IEEE Computer Society, 2010.

[11] D. Cammareri, C. Raibulet, An enhanced approach to automatic generation of mobile widgets, in: Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services, iiWAS '10, ACM, 2010.

[12] D. Popovici, M. Desertot, S. Lecomte, T. Delot, When the context changes, so does my transportation application: Vespa, Procedia CS 5.