Journal of Computer Science 10 (11): 2165-2181, 2014
ISSN: 1549-3636

doi:10.3844/jcssp.2014.2165.2181 Published Online 10 (11) 2014 (http://www.thescipub.com/jcs.toc

COMMON FRAMEWORK: A HYBRID APPROACH TO INTEGRATE CROSS-PLATFORM COMPONENTS IN MOBILE APPLICATION

[1,2]Joachim Perchat, [2]Mikael Desertot and [2]Sylvain Lecomte

[1]Keyneosoft, 31 Rue De La Fonderie, 59200 Tourcoing, France
[2]Univ Lille Nord de France, F-59000 Lille, France, UVHC, HAMIH,
F-59313 Valenciennes, France CNRS, UMR 8201, F-59313 Valenciennes, France

Received 2014-04-06; Revised 2014-04-07; Accepted 2014-07-22

ABSTRACT

There is a multitude of mobile OS: iOS android, Windows Phone 8 and each OS provides its own standards and tools. This heterogeneity in the mobile domain forces developers to implement an application for each mobile platform. To achieve that, developers need to master several languages (Java, Objective-C…). They also need to have several devices at their disposal (PC, Mac, many smartphones …). Then, after applications distributions, developers have to maintain several source codes. In this study, we tackle this problematic. Our goal is to soften the differences between each OS in order to simplify the development of cross-platform third-party applications. To achieve that, we have defined a framework called COMMON (Component Oriented programming for Mobile Multi OsiNtegration). This framework allows the integration of cross-platform components in any application (iOS android). To run our components on any OS, we provide an implementation for each platform. However, to make their integrations easier, we also provide a common public interface of each component, which is platform-independent. Besides, we provide a common language, also platform-independent, allowing the integration and use of any component in any native application (iOS android). This language is based on annotations. Finally, we have implemented a cross-compiler, which translates the source code written with our language to native source code: Objective-C for iOS, Java for Android,… In this study, we have shown that our solution offers performance and memory consumption closed to native applications. Finally, with COMMON, mobile developers implement less lines of source code than with a native application. In your test application, we have saved 30%.

Keywords: Cross-Platform, Components, Common Language, Component Integration, Cross-Compiler, Hybrid Application

1. INTRODUCTION

With the success of smartphones and their application stores, many companies and individual developers have chosen to implement mobile applications. Indeed, smartphones are more and more powerful and their OS support more and more new features and particularly the ability of installing third-party applications. In 2013, more than 50 billion third-party applications were downloaded from the App Store (Apple press info, May 2013: http://www.apple.com/pr/library/2013/05/16Apples-App-Store-Marks-Historic-50-Billionth-Download.html. (iOS store) and more than 25 billion from Google play (Android official blog, September 2012: http://officialandroid.blogspot.fr/2012/09/google-play-hits-25-billion-downloads.html) (Android store). With all these new applications available through the different markets, the usages of this kind of phones have changed. Smartphones are not only used to call or send SMS but also to connect to the Internet, to get points of interest according to the user's location, to play video game in the subway, to read a book on the beach, to share

Corresponding Author: Joachim Perchat, Keyneosoft, 31 Rue De La Fonderie, 59200 Tourcoing, France

pictures via Facebook and many other things. With these new usages, the user's mobility has opened new perspectives in the mobile research. Now, the developers must implement context-aware applications in order to provide the best possible user experience.

In our previous works (Popovici *et al*., 2011), we have designed the CATS for Context-Aware Transportation Services framework, which helps the implementation of context-aware applications. This kind of applications adapts their behaviors according to the user's context. For example, with the same application, if a user is inside his car in a town, the application could launch a parking space search service. Whereas, if the user is a pedestrian, the application could launch a point of interest search service. Of course, the application detects itself the context changes and adapts its behavior without restart it. To achieve that, the application is divided between several context-aware components. Then, the components will be loaded at runtime in the application according to the user's context. We have implemented this framework, using OSGi (Hall *et al*., 2011), for Android but it is impossible to easily port this version on other mobile operating systems.

Ideally, CATS and more generally mobile applications must be available at least on iOS and Android. These are the two most popular platforms for smartphones: 93% of the market (IDC Worldwide Quarterly Mobile Phone Tracker, May 2013: http://www.idc.com/getdoc.jsp?containerId=prUS24108913). But, the development of cross-platform applications is very hard to carry through to a successful conclusion. Cross-platform applications means that applications are able to run on several platforms and are implemented entirely or in part from the same source code. The major problem is due to the heterogeneity of mobile operating systems. A developer who wants to implement a cross-platform application must provide a significant effort. He must use several programming languages (Objective-C, Java,), IDEs (Xcode, Eclipse android studio,), memory management systems (garbage collector, reference counting,). Furthermore, the maintenance of several source codes is also difficult.

In this study, we tackle the problematic of mobile OS heterogeneity. Our goal is to soften the differences between smartphones and more generally between devices running a mobile OS (e.g., tablets) in order to simplify the development of cross-platform third-party applications. To do that, we have introduced the component oriented programming in the mobile development domain (Perchat *et al*., 2013). Our tools and components are integrated in the Component Oriented programming for Mobile Multi OsiNtegration (COMMON) framework. Our cross-platform components have an implementation per platform. Of course, all these implementations are hidden to developers. Only a common interface written in XML is visible. This interface is platform-independent and represents the component features. To integrate our components, we introduce an intermediate language based on annotations that allows components integration in any application implemented in native language from common interfaces. The instructions written with our language are common to any target platform. Finally, a cross-compiler translates the code written with our language to native language. At the end, our tools provide a complete native application.

Up to now, we have presented and published the concepts of this new approach and we remind them in section 4. For this study, we have implemented and tested our proposal in order to entirely create an application, like presented in section 5. We also evaluated our prototype by comparing three approaches: Developing an application only with the Android SDK, then using our approach and finally with Titanium mobile. We focused on the application feasibility with our approach and Titanium mobile and we compared the performances of each application version.

This study is structured as follows: Section 2 shows the smartphone market with its problems, section 3 presents the existing solutions to implement cross-platform applications, section 4 and 5 explain our proposal and its use for implementing an application on Android. Section 6 and 7 show the evaluation of our proposal and the discussion about it. In section 8, we conclude and present our perspectives.

2. SMARTPHONES MARKET

The mobile market is divided between many smartphones manufacturers, the main ones being Apple, Samsung, LG or HTC. Each one provides theirs smartphones with a different mobile OS. Among these OS, there are iOS from Apple android from Open Handset Alliance (from a Google initiative), Windows Phone from Microsoft … With the **Table 1**, we give rise to some of the points which differ when a developer implements an application on several mobile platform.

2.1. Current Situation

Each platform uses different tools, programming languages, user interface declarations and memory management. If a developer wants to create an application that works on all platforms, he should buy one PC with Windows 8 and one Mac. Then, he will have to follow different trainings, one per platform. Finally, he will have to buy at least one phone for each kind of platform and sometimes even multiple phones for one platform, the same as in the case of Android.

Table 1. Some differences between several mobile operating systems

Operating system	Virtual machine	Programmi-nglanguage	User interface	Memory management	IDE	Development on:	Devices
iOS	No	Objective-C	CocoaTouch	Reference counting	Xcode	Mac OS X	Homogenous
Android	Dalvik VM	Java	XML files	Garbagecollector	Eclipse	Multi-platform	Heterogenous
Windows Phone 8	CLR	C# and VB.Net	XAML files	Garbagecollector	Visual studio	Windows 8	Homogenous

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 on the host, the application behaviors can be different too.

To demonstrate the importance of these two steps, we have developed a basic application for iOS and Android. This application consists in only one view. On this view, we have added a button “close the application”. When the user clicks on this button, the application displays a popup that contains the message “The application will be closed” and a button “OK”. When the user clicks on the button “OK” the application is closed.

We have analysed the source code and found multiple differences. First, the source code is implemented in two different development environments: Xcode for iOS and Eclipse for Android. Second, the user interface is implemented in two different manners. In iOS, we use interface Builder. This editor allows us to choose the graphical elements and to drag and drop them on a view. Whereas on Android, we define the user interface from XML files. It is also possible to use the same process as on iOS (drag and drop) with the graphical layout proposed by eclipse but developers rarely use this tool. Indeed, it is easier to implement the views in XML than with the graphical layout because this tool does not provide a simple mechanism to take into account the screen heterogeneity of the Android devices. Third, the languages to implement the application behaviours are different: Objective-C for iOS and Java for Android as shown in **Fig. 1 and 2**. Fourth, the links between the source code and the user interface are different for each platform. For example, to link an action to the button “close the application”, on iOS, we must use Interface Builder and link an IBAction on the “press button” event whereas on Android, we must get an element reference from the XML file and then add a listener to it. Fifth, when we want to create a popup on iOS, we must provide a delegate to the popup in order to get the click events on its buttons. Whereas, on Android, we must set a listener on each popup button.

```
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    exit(0);
}

-(IBAction)closeApp:(id)sender {
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:@"Information"
        message:@"The application will be closed"
        delegate:self
        cancelButtonTitle:@"ok"
        otherButtonTitles:nil];
    [alertView show];
}
```

Fig. 1. Popup creation on iOS

```
closeButton=(Button)this.findViewById(R.id.closeButton);
closeButton.setOnClickListener(new OnClickListener() {
    // display a popup
    @Override
    public void onClick(View v) {
     Builder builder = new Builder(MainActivity.this);
     builder.setMessage("The applicaton will be closed");
     // when a user clicks on the ok button
     builder.setPositiveButton("ok",
       new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog,
                            int which) {
            // quit the application
            MainActivity.this.finish();
            }
        });
    }
});
```

Fig. 2. Popup creation on android

With this basic example, we have found five differences when implementing the same application on iOS and Android. Of course, the complexity increases when you add new features and target platforms. The differences are not only located at the programming language level but also in the manner of thinking. Indeed, the developers will have to change their manner of thinking according to the target platform. For example: There are delegates for iOS and listeners for Android. These changes are really difficult to grasp for the developers during the implementation process. Therefore, we want to hide the notions that are different 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 them to use every component, software or hardware, on each host. The unification does not mean that we want to lose the specificity of each platform.

The applications generated with such solution must be efficient. The success of an application 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.

Finally, this solution needs to be easily adaptable. Indeed, the mobile domain can evolve. For example, in a couple of years, Apple 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 comes out, our proposal must be able to integrate it without modifying its internal architecture.

In the next section, we have studied the existing solutions that allow the development of mobile cross-platform applications.

3. RELATED WORK

The solutions which enable the implementation of cross-platform applications, can be classified in four categories: Cross-compilers, solutions based on model-driven engineering, source code interpreters and finally the solution which allows to run certain parts of an application on the cloud.

The solutions based on cross-compilers enable the developers to write their applications from a common language for each target platform. Then, they generate the associated native code for each of them (iOS android...). In this case, the reused code is complete but the mapping between all the common language APIs and all the native target language APIs is very difficult to achieve. That's why, in most cases, cross-compilers only manage few platforms and are limited to common elements from each platform. This is the case of MoSync, (http://www.mosync.com/) Corona (http://www.anscamobile.com/) and Neomades (http://neomades.com/). This limitation is even more present when the common development language is based on a usual mobile SDK (e.g. android SDK or iOS SDK). In this principle, XMLVM (Puder and Yoon, 2010) enables the implementation of an application for Android, iOS and Palm Pre from Android source code.

One other part of existing solutions is based on model-driven engineering. With these kinds of solutions, developers can define theirs applications from models once for several target platforms. Then, these models will be translated into source code for each target platform. But like cross-compilers, the translation between models and native source code is difficult to achieve, especially, if the solution providers want to manage any native component. On one hand, UsiXML (Vellis *et al.*, 2012) and Jelly (Meskens *et al*., 2010) allow developers to produce user interface for multiple mobile platforms. On the other hand, MobiAmodeller (introduced by Balagtas-Fernandez *et al*., 2010) and AppliDE (Quinton *et al*., 2011), which are integrated in CAPucine (Parra *et al*., 2012), allow developers to produce a complete application and even a context-aware application. To create context-aware application, CAPucine designers allow developers to separate their applications in modules. Some of them will be integrated in the application during the transformation whereas the others will be loaded at execution time according to the context.

Interpreters translate, in real time with a dedicated engine, a source code to executable instructions. Developers implement their cross-platform application and the interpreter manages their execution on many platforms. In this case, the interpreter developers must implement a module able to interpret the code for each target platform. 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 is not used by mobile developers because the fragmentation of devices and operating systems is always present and even emphasized with the multitude of existing JSRs in which the application development is based. For all that, many variations based on it exist: J2ME Polish (http://www.enough.de/) Bedrock (http://www.metismo.com/) AlcheMo (http://www.innaworks.com/). They often consist in porting, such as cross-compilers, a Java ME application with some extensions on several platforms. Kramer *et al*. (2011), the common language is not Java but a new language dedicated to the mobile domain: MobDSL. Thereafter, the applications written with MobDSL would run on a VM.

Today, web languages are accessible to everyone, that's why several solutions based on it have emerged. Multiple strategies were defined for mobile web applications. One of them allows uploading, on the device, of web applications, which can be compared with mobile websites being able to access the device hardware. PhoneGap (http://phonegap.com) QuickConnectFamily (http://www.quickconnectfamily.org/) Rhodes (http://www.rhomobile.com/) follow this strategy. Several of these solutions are presented in Allen *et al*. (2010). Another mobile web development branch is based on widgets (Duarte and Afonso, 2011): "Small"applications for mobile devices. These widgets are implemented with web languages and run through a cross-platform widget engine such as xFace (Jiang *et al*., 2010) or Opera (http://dev.opera.com/addons/widgets/). Pan *et al*. (2010), the xFace designers introduced a lightweight engine of widgets running on several platforms. To port this engine on many platforms, they define a porting layer, which is the combination of several components (e.g., file systems, graphics) common to each platform. This separation facilitate the mapping between the engine and the target OS. Finally, there are solutions, which are using the web languages like any programmatic language in order to allow the implementation of an application with mobile specificities. Titanium mobile (http://www.appcelerator.com) and Flex linked to Flash builder (http://www.adobe.com/products/flash-builder.html) are based on this strategy. In the section 6, we compare our approach with Titanium mobile. This solution probably provides the most mature framework. Indeed, when we show the available features, we can easily think this is the best solution. All these solutions often allow the use of all hardware features (such as camera, gps). But, regarding more precisely, the available features are often limited. For example, it is often possible to use the camera in order to take pictures or videos but it is impossible to exploit its stream. This is a real problem when the developers want to implement a barcode scanner. Therefore, these solutions do not allow the implementation of advanced applications.

Finally, several solutions propose to use the cloud as an application platform (Mikkonen and Taivalsaari, 2013). The main goal is to delegate certain parts of an application to the cloud. For example, an application that allows face recognitions will be divided in two parts. The first one, executed in the device, allows capturing pictures from the camera stream and the second one will handle the face recognition in the cloud. Currently, the real goal of this research domain is to save device energy in distributing the heavy processes on servers and also to provide new features to mobile applications (Zhang *et al*., 2011). But, these solutions can also facilitate the development of mobile cross-platform applications. March *et al*. (2011) with µCloud, the developers must divide their mobile applications into many components. Each component is classified by its location: Cloud, mobile or hybrid. Then, at runtime, a conductor orchestrates the application execution. Here, the components running on the cloud are developed once and are reused in any mobile application (iOS android, ...). However, this kind of solutions does not work if the device is disconnected. A possible perspective is to implement hybrid components with another existing solution (e.g., javaScript linked with PhoneGap or Titanium mobile). In this case, the component could work on the cloud or on any device (iOS android...). If the connection to the network is good, the component will run on the cloud, else it will run on the device.

The above solutions cited propose some interesting directions to consider, however up to now no tool could respond to all our needs. The most contributions are limited to most common hardware features for a basic use. Besides, the user experience provided is not acceptable for our needs and applications obviously have less interesting performances than native implementation.

4. COMMON FRAMEWORK

Perchat *et al*. (2013), we introduced the component oriented programming for the mobile domain. Our framework called COMMON for Component Oriented programming for Mobile Multi OsiNtegration allows the developers of mobile cross-platform applications to integrate cross-platform components in any native mobile application.

With this framework, mobile developers must implement the minimal structure of their application, views and navigation between them, with the native SDK of each target platform. So, mobile developers must provide the implementation of their applications structure with the Android SDK, then with the iOS SDK and all the other target platforms. By implementing the application structure with native SDKs, we allow mobile developers to provide the best possible user experience for their applications for each host platform. Indeed, on each platform, they will be able to use any graphic element specific to each native SDK. For example, in their application, on the iOS version, users will be able to use a navigation bar to navigate between their views, whereas, on the Android version, this element will not be

necessary because devices have a back button. Instead of providing a unique user interface for each target platform, we allow to provide applications that will be entirely integrated in the host platform.

Mobile developers will integrate cross-platform components in their applications, thanks to a language based on annotations. Indeed, we provide a set of cross-platform components that are application-independent. A valid component must be reusable in several applications. Besides, our components are platform-independent. To do that, we have defined the structure of our components as shown in **Fig. 3**. A component has one implementation per target platform: Implemented with the iOS SDK, then with the Android SDK and so on. By providing native implementations, our components are able to use any hardware or software element of any platform. Therefore, our components, graphics or running in background, will be perfectly integrated in a native application without altering the host application.

Of course, each component implementation is hidden to mobile developers. We want to have platform-independent components. All the components features are represented in the common public interface, which is platform-independent. This interface is written in XML because it is a flexible enough language that does not depend on any platform. Finally, a component has a complementary interface, which is also written in XML. This interface lists all the component native methods from all implementations. Therefore, each native method (iOS android…) will have its XML representation in the complementary interface. This XML interface is platform dependent and is hidden to developers. Our tools mainly use it: Cross-compiler and component visualization software. Our tools from native implementations generate the two XML interfaces.

After having defined our components structure, we provide a common language to integrate them, or rather to unify their integration. In our solution, the minimal structure of an application is implemented in native languages. Therefore we allow the integration of components directly in the native source codes. To do that, we provide a language based on annotations. It is based on annotations because they are flexible enough to be integrated anywhere in a native source code written with any language such as Java, Objective-C,. This common language allows the use of any method of any component from their common public interface. The instructions written with this language are also be platform-independent and thus, are the same on each target platform as displayed in **Fig. 4**. Mobile developers define only once the use of a method and reuse it in any native application.

Finally, we have defined a cross-compiler that translates all instructions written with our language to native language (for Android, iOS,). To achieve that, the translation process is based on the complement interface of each component. Indeed, mobile developers specify the use of a method with our common language and the complement interface contains all the methods of a component in XML. This is a light process with some simple rules. First, our compiler parses all the complement interfaces of our components. Then, it parses the developer's project; it gets all instructions written from our annotations. Of course, it checks the validity of each instruction (existing methods, good input parameters...). Thus, if the source code is correct, the cross-compiler only translates the XML representation method to a native call with the parameter defined by mobile developers. Finally, it replaces our annotation with the native source code. Therefore, if we want to add a new platform to the compiler, it is possible to do that very fast (some days). We present a concrete example in the section 6.

Besides, our cross-compiler are installed on the mobile developer PC or Mac and our components are shared as executable files. Under this condition, our compiler cannot be accessed to component source code to translate annotations. Thus, we must provide an alternative to source code, which are the complement interfaces of each component.

To sum up, a developer of mobile applications, using our solution, implements the minimal structure of its applications with the native SDKs of each target platform. Then, he writes the required instructions to integrate our cross-platform components or rather to call their methods. All the instructions written with our language will be the same for each target platform. We unify the integration of our components. Finally, our cross-compiler transforms all annotations in native languages. At the end of the process, our framework provides complete native applications for iOS android and so on.

In this study, we differentiate two kinds of developers: Mobile developers and component developers. The first one uses our solution to implement mobile applications. They will integrate our components. Whereas, the second one implements components which will be used by mobile developers. The mobile developers do not implement components.

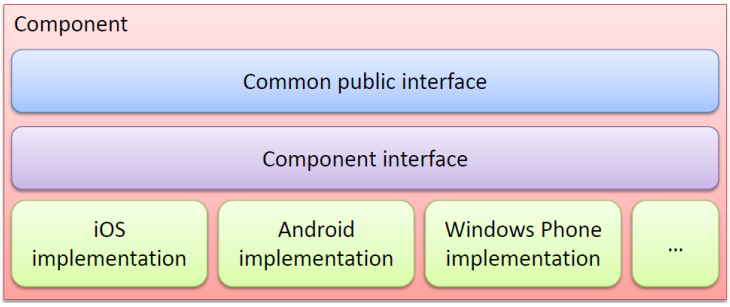

Fig. 3. Component internal structure

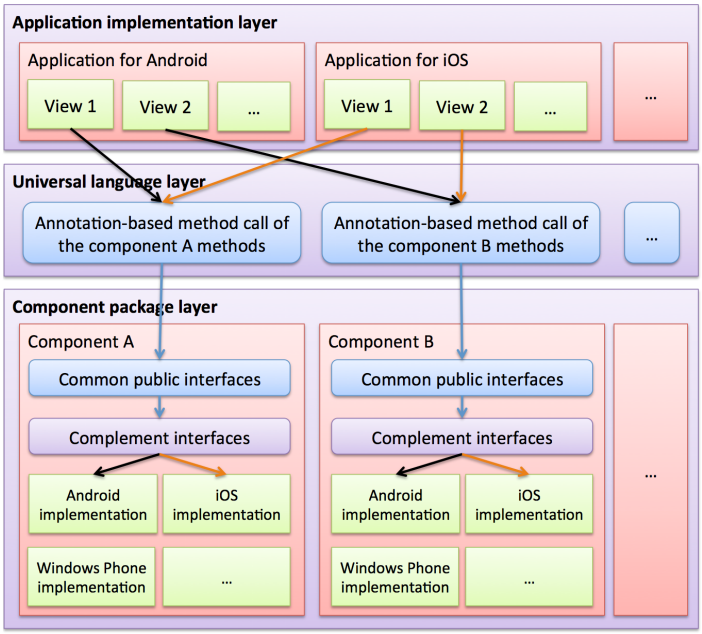

Fig. 4. Component-oriented framework to create cross-platform mobile applications

In the next section, we are going to present our solution to implement a concrete application. We are going to focus to component implementation by component developers and integration by mobile developers.

5. COMMON FRAMEWORK IN SITUATION

To validate our solution, we have chosen to develop a utility application called LocaPlace. This application is a concrete application with a professional style and an advanced user experience. The goal is to provide a deployable application for mobile users. Thus, we will be able to generalize our approach for all possible applications.

The user interface and the navigation between views have been developed in native languages whereas most of the other features were integrated with our common language based on annotations. The instructions written with our language allow the use of six different cross-platform components. In this section, we present the Locaplace application, then the required functionalities, which are provided by our framework. Finally, we present the integration process of a cross-platform component, which is representative of our components.

The LocaPlace application consists in two parts illustrated in **Fig. 5**. The first one allows users to

search for a list of Points Of Interest (POIs) in their proximity or in the city of their choice. After discovering the POIs, the application displays them (name, address) in two alternate views, a list or a map. Then, users can get more information about each POI such as phone number, website, reviews, photos. Users can also find an itinerary from their location to any POI. The itinerary is displayed either as a list or on a map. In the second part, the application allows the decoding of QR-Code. The goal is to get information contained inside and to display it in the application. The information contained in the QR-Code represents a POI (name, address, phone number). After displaying the information, users can find an itinerary from their location to the place.

In this application, the developer needs six application-independent services:

- Auto-completion from an array
- Auto-completion from a database with several columns selected
- Sending of Http Requests to Google Web Services
- XML Parser for the responses of Google Web Services
- Getting information about the device such as location service and network information
- QR-Code Scanner

To make these services available to mobile developers, we provide a set of components in our framework. Among them, we have six cross-platform components that implement the required services.

In the next parts of this section, we focus on another component: HttpRequestManager. We are going to present its native, common and complement interfaces in 5.1. We will only focus on component developer tasks; this part is hidden to mobile developers. Then, in 5.2, we will present the component integration in the application. We will focus on users of our solution: Mobile developers. Finally, in section 5.3, we are going to present the generated code associated to component integration with our cross-compiler.

5.1. Components

This part concerns only component developers. Here, we show how a component is implemented for our solution. This part is entirely hidden to mobile developers, which use our solution. In the future, we will provide them with the tools enabling the implementation of components.

The HttpRequestManager component is representative of our components and it can be integrated in almost all applications. It is used to send http requests to web services. First, we have defined its functions independently from any platform:

- Send an http request with an asynchronous process. The results will be returned through a delegate
- Cancel all the requests in progress
- Cancel a specific request

Then, we provided its implementations for Android and iOS as shown in **Fig. 6 and 7**. Each native interface takes the functions and adapts them according to the target platform. For example, on Android, we do not have a generic type for an http request. So, we must duplicate the method sendHttpRequest. The first version takes as input anHttpGet object and the second one takes aHttpPost object. Of course, these interfaces are hidden to mobile developers. We intend to hide the native source code to simplify and unify the use of components.

After providing the component implementations, we have generated its common and complement interfaces as illustrated in **Fig. 8 and 9**. The generation process is based on the component native interfaces and is performed from a third-party software that we provide.

The common public interface described in **Fig. 8** is entirely platform-independent. We do not consider the input parameters types, the parameters that are non-functional. The main goal is to present the general component features and to focus only on these features and not on the potential differences between the target platforms. The common public interface only contains public component methods. Indeed, our components can have several internal methods, which are hidden to users.

Unlike this interface, the complement interface shown in **Fig. 9** is entirely dependent from all supported target platforms. It gets component native function signatures presented in **Fig. 6 and 7** to translate them into XML. For example, in **Fig. 9**, we show the representation of the “sendHttpRequest” method signature on Android and iOS in XML. Our compiler, as presented in section 5.3, uses this interface to translate annotations in native language.

We have presented here the necessary tasks to implement and transform a native component into cross-platform component in our solution. Finally, we provide to mobile developers an executable file and a common public interface, which is independent from the platform. The complement interface is hidden.

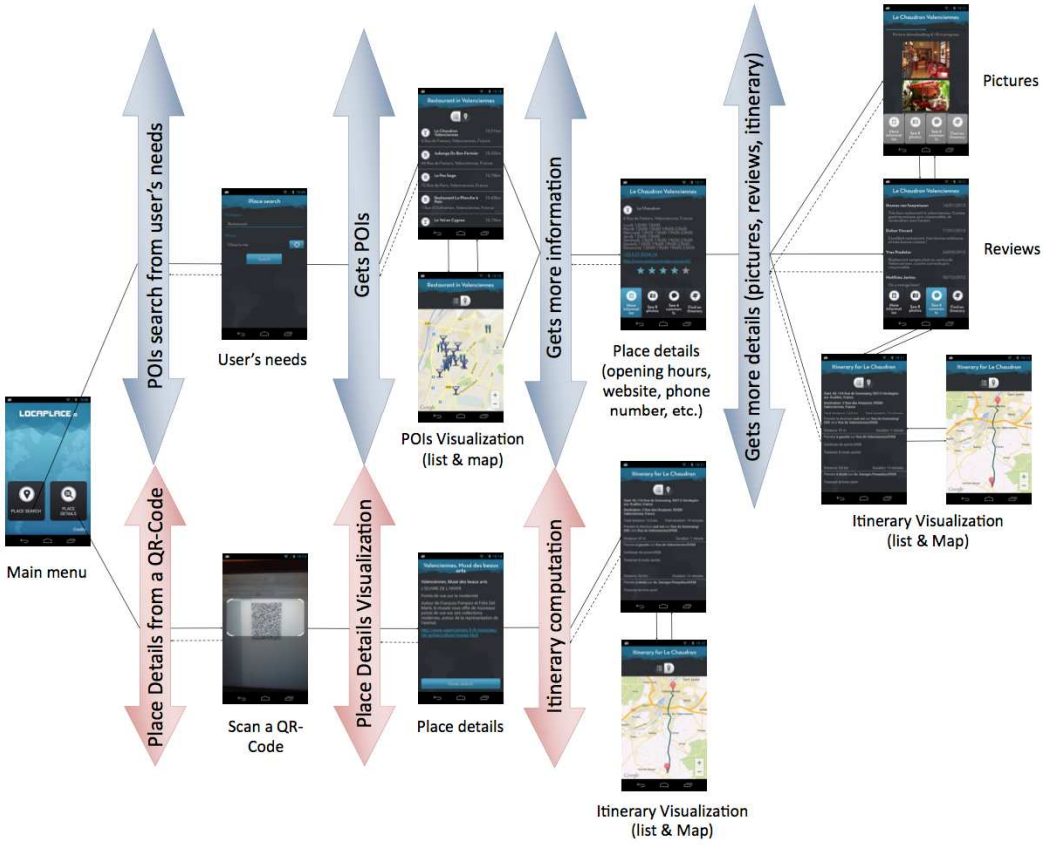

Fig. 5. LocaPlace application views and navigation

```
public interface HttpRequestServiceInterface {

    public void sendHttpGetRequest(Context context,
            HttpGet getRequest,
            HttpGetRequestServiceDelegate delegate);

    public void sendHttpPostRequest(Context context,
            HttpPost postRequest,
            HttpPostRequestServiceDelegate delegate);

    public void cancelAllHttpRequestsInProgress();

    public void cancelTheRequest(HttpGet request);

    public void cancelTheRequest(HttpPost request);
}
```

Fig. 6. Native interface on Android

5.2. Integration

This part only concerns mobile developers, which are using our solution. Indeed, we show how mobile developers can integrate our components in Android or iOS project.

```
@interface HttpRequest : NSObject {
    BOOL activityIndicator;
}

+(HttpRequest*)sharedInstance;

-(void)sendRequest:(NSURLRequest*)request
          delegate:(id<HttpRequestDelegate>)delegate
andActivityIndicatorVisible:(BOOL)visible;

-(void)cancelAllRequests;
-(void)cancelARequest:(NSURLRequest*)request;

@end
```

Fig. 7. Native interface on iOS

To integrate our components, we allow the call of each component method using a new language based on annotations from its common public interface. Among these annotations, there are the "var" annotation and the "method" annotation. These are placed before instructions in native language. The first annotation enables developers to declare the input parameters of a component method.

```
<Component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="CommonInterfaceSchema.xsd" >
    <description>
        <componentName>RequestManager</componentName>
        <version>1.0</version>
        <date>01/01/2014</date>
        <description>send http requests to any web service</description>
        <dependancies>
            <dependance componentName="DeviceInfoManager"/>
        </dependancies>
    </description>
    <targetPlatforms>
        <target platformName="android" executableName="RequestManager.jar"/>
        <target platformName="iOS" executableName="RequestManager.framework"/>
    </targetPlatforms>
    <methods>
        <method methodName="sendHttpRequest"
            description="send a http request (Get, Post, Soap) with an asynchronous process">
            <parameter io="in" description="the http request to send"/>
            <parameter io="in" description="the delegate which receives the responses"/>
        </method>
        <method methodName="cancelAllHttpRequestsInProgress"
            description="cancel all http requests in progress">
        </method>
        <method methodName="cancelTheRequest" description="cancel a particular http request">
            <parameter io="in" description="The http request to cancel"/>
        </method>
    </methods>
</Component>
```

Fig. 8. HttpRequestManager common public interface

```
<extraInformationOnMethod methodName="sendHttpRequest"
    description="send a http request (Get, Post, Soap) with an asynchronous process">
    <android methodName="sendHttpGetRequest" className="HttpRequestService"
        packageName="com.keyneosoft.requestManager.implementations" isSingleton="true"
        description="send a http request (Get) with an asynchronous process">

        <parameter parameterName="context" parameterType="Context" io="in" description="the application context"/>
        <parameter parameterName="getRequest" parameterType="HttpGet" io="in" description="the http request to send"/>
        <parameter parameterName="delegate" parameterType="HttpGetRequestServiceDelegate" io="in"
            description="the delegate which receives the responses"/>
    </android>

    <android methodName="sendHttpPostRequest" className="HttpRequestService"
        packageName="com.keyneosoft.requestManager.implementations"
        isSingleton="true" description="send a http request (Post) with an asynchronous process">

        <parameter parameterName="context" parameterType="Context" io="in" description="the application context"/>
        <parameter parameterName="postRequest" parameterType="HttpPost" io="in" description="the http request to send"/>
        <parameter parameterName="delegate" parameterType="HttpPostRequestServiceDelegate" io="in"
            description="the delegate which receives the responses"/>
    </android>

    <iOS className="HttpRequest" isSingleton="true">

        <parameter parameterName="request" parameterType="NSURLRequest*" io="in" description="the http request to send"/>
        <parameter parameterName="delegate" parameterType="id\x3CHttpRequestDelegate\x3E" io="in"
            description="the delegate which receives the responses"/>
        <parameter parameterName="visible" parameterType="BOOL" io="in"
            description="true if the network indicator in the status bar must be activate, else false "/>
    </iOS>
</extraInformationOnMethod>
```

Fig. 9. An HttpRequestManager complement interface part

This annotation will link the variable placed after it with an input parameter of a component method, whereas the method annotation enables the call of component methods. This annotation will link the method result with the following variable.

To make the integration of components easier, we provide developers with third-party software, which describes the different steps to integrate them in a mobile project. This software takes as input a component and presents its different features. It facilitates the understanding of our components because mobile developers don't need to read the XML interfaces. Then, the software provides users with the different annotations that are necessary in order to call a component method as displayed in **Fig. 10**. Thus, mobile developers only need to copy the annotations and paste them into their native source code (for example, in an Android application). As shown in **Fig. 10**, if a developer wants to call the "sendHttpRequest" method of our HttpRequestManager component, our software provides the instructions needed to declare the input

parameters (request to send, delegate…) as well as the instruction used to call the method.

In **Fig. 11a**, the mobile developer has declared three variables from "var" annotations: "Context", "getRequest" and "myDelegate". He has linked each variable to the "sendHttpRequest" method with the var annotation parameter "methodName". Then, mobile developer called "sendHttpRequest" method from a "method" annotation.

The component integration process will be the same on any platform. Indeed, the developers will use the same annotations in an Android or iOS application. There might be slight changes in the var annotation. According to the platform, the methods can have more or less parameters. For example, in the Android interface of our component, the sendHttpRequest method has a parameter "context" which does not exist on iOS, see **Fig. 6 and 7**. In **Fig. 12a**, we have used the same annotations in an iOS application. However, in some cases, we cannot provide a component for a particular platform. For example, on Android and Windows Phone 8, it is possible to provide applications that enable NFC tags capture. On iOS, it is impossible, Apple doesn't integrate this technology. In this case, developers will not be allowed to integrate our annotations for this component in iOS source codes, our compiler will generate errors. Of course, as shown in **Fig. 10**, our tools show platform list for which each component is compatible.

5.3. Generated Code

After calling the methods of our cross-platform components with our language, mobile developers can launch our cross-compiler. It translates all method annotations found to native languages (Java for Android, Objective-C for iOS…). The source code shown in **Fig. 11** ais transformed into the code shown in the **Fig. 11b**. In this example, our compiler translated the method annotation, which allows the call to the method "sendHttpRequest" to native language (Java).

Fig. 10. Steps to follow in order to call the sendHttpRequest method

```
@Var(componentName = "RequestManager",
        methodName = "sendHttpRequest",
        parameterName = "context")
Context context = this.getApplicationContext();

@Var(componentName = "RequestManager",
        methodName = "sendHttpRequest",
        parameterName = "getRequest")
HttpGet getRequest = new HttpGet(url);

@Var(componentName = "RequestManager",
        methodName = "sendHttpRequest",
        parameterName = "delegate")
HttpGetRequestServiceDelegate delegate = this;

@Method(componentName = "RequestManager",
        methodName = "sendHttpRequest")
```

(a)

```
@Var(componentName = "RequestManager",
        methodName = "sendHttpRequest",
        parameterName = "context")
Context context = this.getApplicationContext();

@Var(componentName = "RequestManager",
        methodName = "sendHttpRequest",
        parameterName = "getRequest")
HttpGet getRequest = new HttpGet(url);

@Var(componentName = "RequestManager",
        methodName = "sendHttpRequest",
        parameterName = "delegate")
HttpGetRequestServiceDelegate delegate = this;

@Method(componentName = "RequestManager",
        methodName = "sendHttpRequest")
HttpRequestService componentHttpRequestService = HttpRequestService
        .getSharedInstance();
componentHttpRequestService.sendHttpGetRequest(context, getRequest,
        delegate);
```

Generated code

(b)

Fig. 11. (a) Necessary annotations to call (b) Generated code to call the sendHttpRequest thesendHttpRequest method on Android. method with native language on Android (Java)

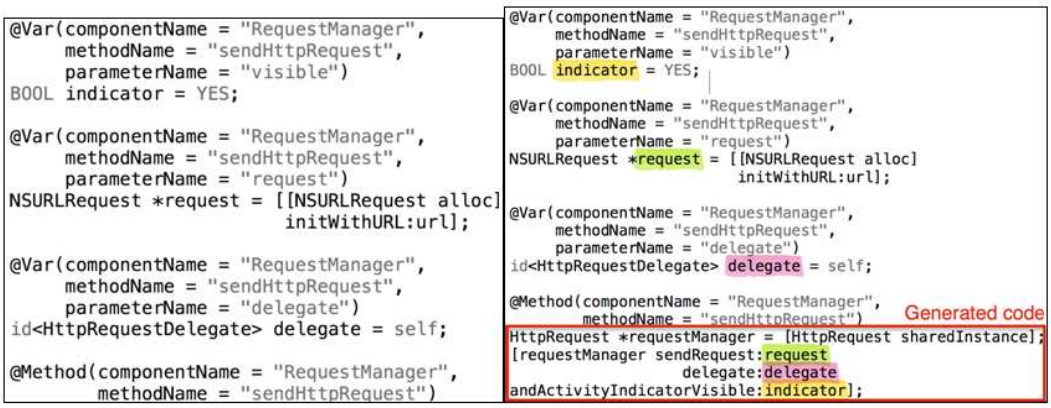

Fig. 12. (a) Necessary annotations to call (b) Generated code to call the sendHttpRequest thesendHttpRequest method on iOS. method with native language on iOS (Objective-C)

First, it inserted the declaration of the component and it initialized it. In this example, the component is a singleton, so, the initialization method is "getSharedInstance". Then, the compiler inserted the method call "sendHttpGetRequest" from the variable previously declared. Finally, it filled in the input parameters from the var annotations declarations.

Today our compiler is implemented in order to translate annotations in Java language for Android. The transformation for iOS application is not implemented yet but we are able to extrapolate the same process on iOS. In **Fig. 12a**, we show the necessary source code in objective-C and annotations to call the "sendHttpRequest" method. In **Fig. 12b**, we have added the source code, which will be generated by our compiler.

6. RESULTS

Before beginning the COMMON implementation for several platforms, we wanted to calculate the potential additional cost on applications using our solution (limitation, performance, memory consumption…). To do that, we have chosen Android because Android uses a virtual machine to execute applications: Dalvik. We wanted to ensure us that Dalvik will be able to support the loading and execution of several components.

We have developed the LocaPlace application for Android under three different ways. The first implementation has been developed entirely in Java with the Android SDK. The second one has been implemented with our framework and the Android SDK. This version was presented in the previous section. The third implementation has been realized with Titanium mobile.

We have chosen Titanium mobile to implement the third version because this solution seems representative of mobile web-applications (Hashimi *et al*., 2010; Allen *et al*., 2010) and it is the most mature solution in the market. With Titanium mobile, the application developers write their entire application in JavaScript. Then, the applications are embedded on the phone with an engine, which is able to interpret JavaScript and the Titanium APIs. By choosing Titanium mobile, we can compare our solution with web-applications. However, the Titanium mobile version is not reliable. Depending on the device on which the application is deployed, the application can stop its execution on certain views in a random way. Analysing the logs, bugs are not coming from the provided application but from Titanium SDK itself. Today, such application cannot be subject to a widespread deployement for the general public.

In the next subsections, we have first calculated the lines of code saved using our approach, compared to the native application. Secondly, we have compared the performances regarding the three versions of LocaPlace. Results will be discussed in section 7.

6.1. Lines of Code Saved

We have compared the number of lines of code written in the native version and in the version with our framework, see **Table 2**. For this evaluation, we do not consider the Titanium mobile version because it is unusable. We are not able to determine the number of lines of code that need to be written to make it usable. To calculate the line numbers of each application version, we have used the "metrics" plugin installed in eclipse (Metrics plugin website: http://metrics.sourceforge.ne).

Table 2. Lines of code written for each application

Application version	Lines of code	Saved
100% native	6932	
Application with components	5004	1928 (28%)

Using our approach, the developer writes two thousand lines of code less than by using the native SDK. Besides, the parts, which are not implemented, are often the most complex ones (parsing, scanning, sending of http request…). With our approach, the developer only needs to implement the application views and the navigation between them.

6.2. Performances

Even if the application Titanium mobile version is not marketable, we are able to compare the performance of several services between the three versions. The compared tasks can be implemented in many applications and not just in our application. Therefore, this comparison can be considered as application-independent. However, it is mandatory to evaluate our approach from a real application and not just from a "test" or "basic" application because a real application uses more resources than a "test" application (images loaded in memory, cache.).

As shown in **Fig. 13**, we have measured the execution time of certain tasks implemented in the Locaplace application. For each task, we have measured the execution time (ms) one hundred times. Then, for each task, we have calculated the average between results. We have gotten these results on the Samsung Galaxy Nexus i9250. This phone was released in 2011, it is equipped with a Dual-core 1.2 GHz Cortex-A9, it has 1 GB of RAM and a screen of 4.65 inches. The installed Android version on the device is Jelly Bean (4.2.2, API 17).

During our evaluation, we have compared the weight of each application version on the device. As shown in **Table 3**, the version with our approach has the same weight as the native version. The Titanium mobile version is 2.3 times heavier than the native version. The Titanium mobile engine, which interprets the web-application, weights around 10 MB.

Finally, in **Fig. 14**, we have compared the memory consumption (RAM). For each task, we have measured the used RAM before and after their execution. The used RAM is the addition of the used RAM for the OS, then for the other applications in execution and finally for our application. To get these measures from the native version and the one implemented with COMMON, we have used the standard APIs provided by Android SDK. In the same way, for the Titanium mobile version, we have used the APIs provided by Titanium mobile SDK.

7. DISCUSSION

Before any commentsabout the performances evaluation, we will discuss the feasibility of LocaPlaceusing the three tools, especially in terms of user interface and user experience. Then we will comment the lines of code saved and analyse the performances.

7.1. Feasibility

Of course, we succeeded in implementing the complete application with the native SDK. It also was a success with our framework. Indeed, we allow the native SDK use for the user interface implementation. Thus, the developer can use any native graphic element.

With Titanium mobile, the implementation is laborious. Indeed, the Titanium mobile SDK does not provide all the existing graphical elements available on Android (or other platforms). Thus, we cannot provide the same user interface and user experience defined in the LocaPlace application specifications. Moreover, it is impossible to put the graphical elements anywhere in a view. We must place each element with pixel precision. This is really problematic for Android devices because there are different screen sizes. In order to hide this heterogeneity, the Android SDK provides an automatic mechanism to create dynamic views without fixed size. The developers do not need to calculate the position of each graphical element, for example they can use a RelativeLayout, which allows the graphical element positioning according to others. With the Titanium mobile SDK, this mechanism does not exist. Another issue is related to the dynamical views of an application (some elements are hidden whereas other ones appear). In the Titanium mobile SDK, this situation is not really considered. Therefore, when the views change their states, the application freezes.

With our approach, developers do not face any limitation when implementing an Android application and the generated applications will be reliable and professional quality. They will be able to use any graphical or hardware elementaccessible through the Android SDK. On the contrary, with Titanium mobile, developers will have to adapt their applications according to the existing elements inside the Titanium mobile SDK.

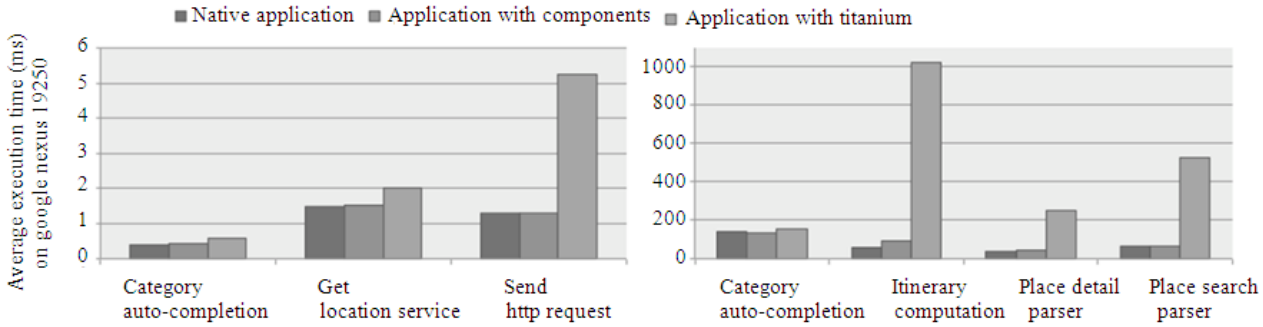

Fig. 13. Average execution times of some LocaPlace applications tasks

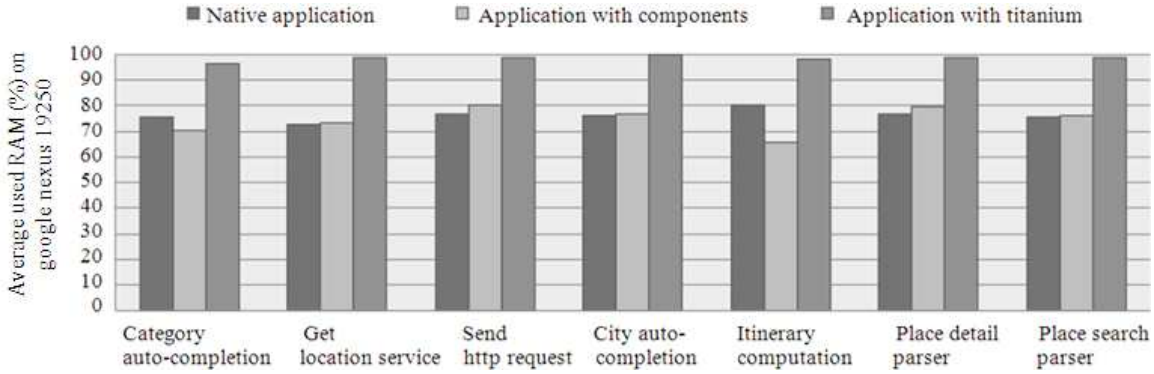

Fig. 14. Used RAM for each LocaPlace application version after several tasks execution

Table 3. Weight of each application version

Application version	Weight
100% native	8.49 MB
Application with component	8.62 MB
Application with Titanium mobile	19.43 MB

7.2. Lines of Code Saved

As shown in **Table 2**, with our solution, LocaPlace developers saved 28% of lines of code compared to native version. However, we have written more than four thousand lines of code to provide the required components for this application, **Table 4**. This is a classic analysis in the component-oriented programming. Our components must be generic to be integrated in any application. This adaptability has a cost especially in number of lines of code. Besides, components often provide more functions than the ones required in the LocaPlace application.

The second time that our components will be integrated into a similar application, we can assume that the developers will, once again, save two thousand lines of code. At that moment, the number of lines written for the components will be the same as the number of lines saved by developers. Starting from the third integration, our solution will become profitable.

Table 4. Lines of code written for each Android version of our components

Component (Android version)	Lines of code
Auto-completion	1044
City auto-completion	325
DeviceInfoManager	320
HttpRequestManager	782
GoogleWSParser	1773
Total	4244

7.3. Performances Analysis

As shown in **Fig. 13**, for almost every evaluated task, the application implemented with our approach provides the same (or almost the same) execution time as the native version. For example, for the sending of http requests, with our approach, the application takes 1.28 ms whereas with the native SDK, the application takes 1.27 ms. The time difference is negligible. Therefore, with our approach, we can consider that the generated application will have the same performance as a native one. The calls of methods through our cross-platform components do not take more time than native calls.

Moreover, for all the evaluated tasks, our approach provides better results than the application implemented with Titanium mobile.

Table 5. Some differences between web-apps solutions and COMMON framework

Solutions	Performance	Resources consumption	User interface	Reliability	General limitation
COMMON framework	As native	As native	No limitation	As native	No limitations
Web-apps solutions (Titanium mobile)	Less efficient	More consumption	limited to common elements	Not reliable	Limited to common elements

In the best case, Titanium mobile is 1.15 times less efficient than our approach (city auto-completion). This good coefficient is due to the low-level of the task. Indeed, in this task, we executed sql requests to sqlite database. We can think that Titanium mobile delegates these actions to the native SDK. In these conditions, we can consider that it is a similar approach with ours and therefore, the solution is able to provide good results. Unlike the low-level tasks, the others are much slower. In the worst case, Titanium mobile is 11.22 times less efficient than our approach (itinerary computation). Here, the process is entirely executed from JavaScript source code. This means that the execution of a web language (like JavaScript) has an additional cost that is not insignificant. PhoneGap, another popular web-based solution, provides the same ratio in terms of execution time as Titanium mobile (Corral *et al*., 2012).

Finally, as shown in **Fig. 14**, with our approach, the application uses the same ratio of RAM as the native version (to around 70% until 80%). The loading of components (jar files) has no influence on the RAM consumption. Whereas, with the Titanium mobile version, the application uses all the time to 95% until 99% of the available RAM on the device. This consumption may have a cost in energy consumption and probably the application often stops its execution because it has no more available RAM.

Nowadays, we can read many discussions about Web-apps versus native applications (Mikkonen and Taivalsaari, 2013; Corral *et al*., 2011; Charland and Leroux, 2011). But as shown with the results, **Table 5**, we can conclude that the use of web languages to implement mobile application has a non-negligible cost in terms of performances and consumes more RAM than our application. This supports our decision to keep our solution tied to native code.

8. CONCLUSION

The development of mobile cross-platform applications is very hard to accomplish due to the OS heterogeneity. Indeed, mobile developers must provide one version per target platform. Each version will be implemented with different standards, programming languages. These differences have a significant cost for the developer. That's why, in this study, we propose a new approach which soften the differences between each OS. Our framework, called COMMON for Component Oriented programming for Mobile Multi OsiNtegration, allows developers of mobile cross-platform applications to integrate cross-platform components, with platform-independent a language, in any native application.

COMMON is based on a set of cross-platform components. Components have one implementation per target platform. Each one is implemented in native language to allow the use of any native software or hardware element. In order to hide the differences between each implementation, we have defined a common public interface written in XML. This interface is entirely platform-independent. Then, to unify the integration of our component, we have defined a new language based on annotations. This language is also platform-independent because it is based on the common public interface of our components. Finally, we have designed a cross-compiler, which translates the instructions written with our language to native code. The generated applications only contain native code.

In this study, we have evaluated our framework on Android. To do that, we have developed the same application with the native SDK and with our framework. In term of development tasks, the developer has saved 28% of code lines for our sample application. The evaluation has also shown results closed to native application. Indeed, our solution provides execution times similar to native applications. We observe the same result in term of used RAM, or application weight. We can conclude that our solution does not bring an additional cost to mobile applications. Furthermore, we have also compared our approach with Titanium mobile. We obtained better results than Titanium mobile on all evaluation criterias and especially, we do not limit the use of native elements (software or hardware) contrary to Titanium mobile.

Using our solution, developers can implement the user interface and its behaviour in native languages. This flexibility allows them to implement any application withoutany limitations. However, this part of source code is very important in an application (around 50%).

This process could be replaced by a solution based on model-driven engineering. Today, most of these solutions only allow the generation of user interface. By coupling this kind of solutions with COMMON, developers will be able to generate an application, writing a minimum amount of code.

In our future works, we will provide an implementation of COMMON allowing the integration of components from iOS and Windows 8 native source code. The main goal is to evaluate our approach on several platforms. We also want to assist the component developers. Today, the component providers in our solution must implement each component for each platform. In the future, we want to generate the structure of components and even generate one implementation from another one. For example, we will be able to generate an iOS implementation from an Android implementation. However, the translation must not impact the possibilities offered by each platform such as the existing solutions. Finally, we want to help implementing cross-platform context-aware applications using our previous works on the context (Popovici *et al*., 2011).

9. ACKNOWLEDGEMENT

The present research work has been supported Keyneosoft. The authors gratefully acknowledge the support of this company.

10. REFERENCES

Allen, S., V. Graupera and L. Lundrigan, 2010. Pro Smartphone Cross-Platform Development: iPhone, Blackberry, Windows Mobile and Android Development and Distribution. 1st Edn., Apress, New York, ISBN-10: 1430228687. pp: 288.

Balagtas-Fernandez, F., M. Tafelmayer and H. Hussmann, 2010. Mobia Modeler: Easing the creation process of mobile applications for non-technical users. Proceedings of the 15th international conference on Intelligent user interfaces, Feb. 07-10, ACM New York, pp: 269-272. DOI: 10.1145/1719970.1720008

Charland, A. and B. Leroux, 2011. Mobile application development: Web Vs. native. Commun. ACM, 54: 49-53. DOI: 10.1145/1941487.1941504

Corral, L., A. Sillitti and G. Succi, 2012. Mobile multiplatform development: An experiment for performance analysis. Procedia Comput. Sci., 10: 736-743. DOI: 10.1016/j.procs.2012.06.094

Corral, L., A. Sillitti, G. Succi, A. Garibbo and P. Ramella, 2011. Evolution of mobile software development from platform-specific to web-based multiplatform paradigm. Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms and Reflections on Programming and Software, Oct, 22-27, New York, pp: 181-183. DOI: 10.1145/2048237.2157457

Duarte, C. and A.P. Afonso, 2011. Developing once, deploying everywhere: A case study using jil. Procedia Comput. Sci., 5: 641-644. DOI: 10.1016/j.procs.2011.07.083

Hall, R., K. Pauls and S. McCulloch, 2011. OSGi in Action: Creating Modular Applications in Java. 1st Edn., Manning Publications Company, Greenwich, ISBN-10: 1933988916, pp: 548.

Hashimi, S., S. Komatineni and D. MacLean, 2010. Titanium Mobile: A Webkit-Based Approach to Android Development. In: Pro Android 2, Hashimi, S.Y., S. Komatineni and D. MacLean (Eds.)., Apress, ISBN: 1430226595, pp: 627-660.

Jiang, F., Z. Feng and L. Luo, 2010. Xface: A lightweight web application engine on multiple mobile platforms. Proceedings of the IEEE 10th IEEE International Conference on Computer and Information Technology, Jun. 29-Jul. 1, IEEE Xplore Press, Bradford, pp: 2055-2060. DOI: 10.1109/CIT.2010.349

Kramer, D., T. Clark and S. Oussena, 2011. Platform independent, higher-order, statically checked mobile applications. Int. J. Design, Analysis Tools Circuits Syst.

March, V., Y. Gu, E. Leonardi, G. Goh and M. Kirchberg *et al*., 2011. Mcloud: Towards a new paradigm of rich mobile applications. Procedia Comput. Sci., 5: 618-624. DOI: 10.1016/j.procs.2011.07.080

Meskens, J., K. Luyten and K. Coninx, 2010. Jelly: A multi-device design environment for managing consistency across devices. Proceedings of the International Conference on Advanced Visual Interfaces, May 26-28, New York, pp: 289-296. DOI: 10.1145/1842993.1843044

Mikkonen, T. and A. Taivalsaari, 2013. Cloud computing and its impact on mobile software development: Two roads diverged. J. Syst. Software, 86: 2318-2320. DOI: 10.1016/j.jss.2013.01.063

Pan, B., K. Xiao and L. Luo, 2010. Component-based mobile web application of cross-platform. Proceeding of the IEEE 10th International Conference on Computer and Information Technology, Jun. 29-Jul. 1, IEEE Xplore Press, Bradford, pp: 2072-2077. DOI: 10.1109/CIT.2010.352

Parra, C.A., C. Quinton and L. Duchien, 2012. CAPucine: Context-aware service-oriented product line for mobile apps. ERCIM News, 88: 38-39.

Perchat, J., M. Desertot and S. Lecomte, 2013. Component based framework to create mobile cross-platform applications. Procedia Comput. Sci., 19: 1004-1011. DOI: 10.1016/j.procs.2013.06.140

Popovici, D., M. Desertot, S. Lecomte and N. Peon, 2011. Context-aware transportation services (cats) framework for mobile environments. Int. J. Next-Generat. Comput.

Puder, A. and I. Yoon, 2010. Smartphone cross-compilation framework for multiplayer online games. Proceeding of the Second International Conference on Mobile, Hybrid and On-Line Learning, Feb. 10-16, IEEE Xplore Press, Saint Maarten, pp: 87-92. DOI: 10.1109/eLmL.2010.13

Quinton, C., S. Mosser, C. Parra and L. Duchien, 2011. Using multiple feature models to design applications for mobile phone. Proceeding of the 15th International Software Product Line Conference, Aug. 22-26, New York. DOI: 10.1145/2019136.2019162

Vellis, G., D. Kotsalis, D. Akoumianakis and J. Vanderdonckt, 2012. Model-based engineering of multi-platform, synchronous and collaborative UIs-extending UsiXML for polymorphic user interface specification. Proceeding of the 16th Panhellenic Conference on Informatics, Oct. 5-7, IEEE Xplore Press, Piraeus, pp: 339-344. DOI: 10.1109/PCi.2012.27

Zhang, X., A. Kunjithapatham, S. Jeong and S. Gibbs, 2011. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. Mobile Netw. Applic., 16: 270-284. DOI: 10.1007/s11036-011-0305-7