Component Technologies on Google Android

Michael Eckel

University of Applied Sciences Gießen-Friedberg,
Department of Mathematics, Natural Sciences and Computing,
michael.eckel@mni.fh-giessen.de
http://www.fh-giessen-friedberg.de/

Abstract. Within the last years mobile devices have evolved into powerful computers capable of running fairly sophisticated applications [8]. Since almost every bigger software application relies on an underlying component technology—such as OSGi—it is obvious to develop mobile applications based on components, too. In 2008 Google entered the mobile device market with an open-source operating system called *Android* introducing a new component system. In contrast OSGi is a technically mature component technology with a wide-spread area of application, but Android by default doesn't support it.
Within this paper I will compare and discuss both component systems and will provide a way to get OSGi working on Android. Furthermore I want to introduce ROCS (Remote OSGi Caching Service), a remotely provisioned OSGi framework for ambient systems. ROCS enables mobile devices to become cloud-aware by running applications directly from network without storing them locally [4].

Keywords: Android, Mobile Devices, Mobile Programming, Component Technologies, OSGi, Cloud Computing, Java, Java Class Loading, RMI

Table of Contents

1 Introduction

This article is about component systems on Google Android. Even if Android is shipped with its own component model it is a desirable objective to run OSGi—a mature component system for the Java platform—on Android powered mobile devices. In this article advantages of both component systems will be discussed and implementation details will be given. Since OSGi features hot-plugging a comparable approach on Android's component system is being presented. With respect to cloud computing ROCS—a remotely provisioned OSGi framework making mobile devices cloud-aware—is being introduced.

The paper is organized as follows. Section 2 introduces Google Android in general explaining aspects of system architecture and characteristics before in section 3 the component system of Android is being examined in more detail. In addition an approach to realize a plugin system with Android's component system is presented. Section 4 describes the differences between OSGi and Android's component system and explains how OSGi components can be run on Android. As a last topic ROCS is being presented in section 5 describing an approach of making mobile devices cloud-aware by running all applications directly from network without storing them locally. Finally, section 6 concludes the article.

2 Android

Android is a mobile operating system developed by Google. The first version was brought to market in 2008. Since then Android is experiencing a great increase in market share. It is largely open-source and is based on Linux 2.6. Application programming is done in Java but is being compiled to so called DEX bytecode and is being run by the Dalvik Virtual Machine (DVM). This section describes fundamentals of Android including system architecture and runtime environment.

2.1 System Architecture

Android uses a 4-layered system architecture depicted in figure 1 [6,2]. The layers are described in the following.

Linux Kernel Provides low level operating system functionality such as memory management, security, process management etc. It also facilitates all the hardware drivers.

Android Runtime Includes many libraries mostly written in the Java programming language, which provide a lot of the functionality of the Java core libraries. The Android Runtime layer also offers the Dalvik Virtual Machine (DVM) which is beeing described in 2.2.

Libraries Provides C/C++ libraries which are used by many other components within Android. Those libraries are e. g. the System C library, SQLite, 3D libraries and so on.

Application Framework Android offers an extensive framework enabling application to easily communicate and share data with each other. Furthermore the framework simplifies accessing systems resources such as hardware, location data and background services.

Applications On this layer Android applications accessible by the user are located. Android comes with some standard applications such as Contacts, Messaging etc.

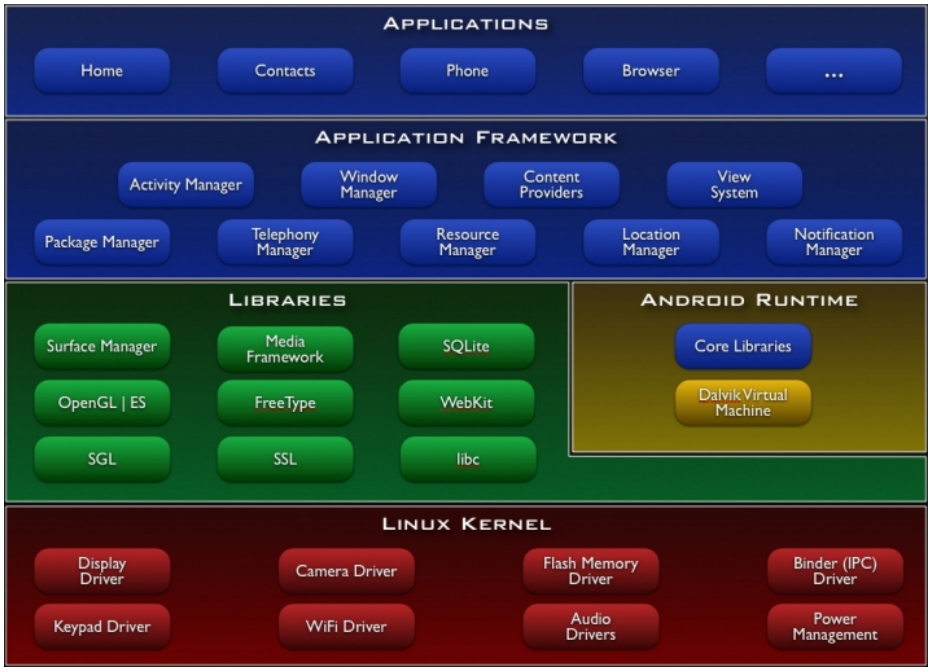

Fig. 1. System architecture of Android [6]

2.2 Dalvik Virtual Machine

Android applications are developed using the Java programming language. Nevertheless you cannot make use of the full Java framework since Android ships with its own framework (cf. section 2.1). Java language features such as *Reflection* are not supported by this framework due to performance issues.

The Dalvik Virtual Machine (DVM) is a virtual machine which is optimized for (low-resource) mobile devices. Unlike the virtual processor model of the Java Virtual Machine (JVM) the DVM's processor model makes use of machine registers thus taking advatage of modern mobile CPUs. In addition bytecode for

the DVM (called DEX bytecode[1]) is much smaller than Java bytecode and can be executed much faster [2].

How a Java source file becomes DEX bytecode is described in the following and illustrated in figure 2. Given a Java source file (`*.java`) the Java compiler *javac* creates a Java bytecode file (`*.class`). Now the *dx* tool transforms it to a `*.dex` file which is capable of being executed by the DVM.

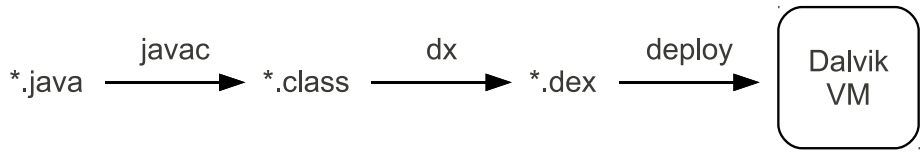

Fig. 2. From Java sourcecode to DEX bytecode

2.3 Android Applications

Android applications consist of DEX bytecode, resources and data and are deployed as `*.apk` files. They are subject to a lifecycle and are executed in a so called "Sandbox". That is every application runs in its own DVM in an independent Linux process. When an application is being installed to an Android device it is being assigned a new user and group ID. Concerning security there is no possibility to illegally access another process or its data since the user and group IDs are different.

3 Android's Component System

Before going into the details of Android's component system I first want to introduce what a software component really is. In 1996 the European Conference on Object-Oriented Programming (ECOOP) defined it as follows:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [3,16]

In short the following defines a component:

1. contractually specified interface
2. explicit context dependencies
3. independent deployment

Components must conform to a component model which specifies form and properties of a component as well as how components interact with each other and how they can be combined.

[1] DEX stands for Dalvik EXecutable

3.1 Components in Android

Android allocates four different types of components [2]:

Activities are suitable for interacting with the user and presenting the user interface.

Services can be used to run background tasks. They do not have an user interface.

Content Providers are suited for providing data to other applications. Thus it is possible to "break out" of the Sandbox and provide data through a defined interface.

Broadcast Receivers are used to receive broadcasts either from the Android system or from other applications.

3.2 Communication between Components

As a major aspect of a component model is the interaction between components this subsection is to clarify how it is implemented in Android. The communication between components in Android is done by using *Intents* and *Intent Filters*. The calling component uses an Intent to announce the wish for communication with another component. The called component has an Intent Filter to receive the Intent whereat the Intent must match the conditions of the Intent Filter.

Intent Filters Intent Filters are defined in a manifest file called `AndroidManifest.xml`. An example is shown in listing 1.1. The XML tag `<receiver>` stands for a Broadcast Receiver. This Broadcast Receiver is annotated with an Intent Filter whose action name is `org.example.TEST` meaning the Broadcast Receiver is able to process actions with that name. So we have an explicitly defined interface through which the component can be used. That applies to the first point of the definition of a component.

Listing 1.1. Intent Filters in the Android manifest file

```
1 ...
2 <receiver android:name="org.example.MyReceiver">
3   <intent-filter>
4     <action android:name="org.example.TEST" />
5   </intent-filter>
6 </receiver>
7 ...
```

Intents Amongst other information Intents contain information about the desired target component. An Intent can either be explicit or implicit. Explicit means that the target component is known by its Java name (see listing 1.2). Implicit means that the target component is not known but the desired action is. In case of an implicit Intent the Android system is responsible for finding

a component suitable for executing the action. If more than one component is found the Android system asks the user interactively which one to choose. If none is found Android will display an error message.

Listing 1.2. Explicit and implicit intents

```
1 /* explicit intent */
2 Intent ei = new Intent(org.example.MyReceiver.class);
3
4 /* implicit intent */
5 Intent ii = new Intent("org.example.TEST");
```

When preparing an Intent it should be attached with additional data that are of interest to the receiving component (as far as needed). Listing 1.3 shows an example of invoking some component that is capable of composing an email. The appropriate additional data—in this case the email text and subject—is attached to the Intent. While in line 1 to 4 the Intent is being prepared line 5 really invokes the desired component—in this case an Activity. Line 2 defines an URI which the target component must be able to process. While the Intent `ACTION_SENDTO` is quite unspecific the URI is more specific (`mailto:you@mail.com`). URI parsing goes beyond the scope of this article and is not needed for further understanding. Nevertheless it is documented very well in [7, page 104ff].

Listing 1.3. Sending an Intent for Email Composition

```
1 Intent i = new Intent(Intent.ACTION_SENDTO);
2 i.setData(Uri.parse("mailto:you@mail.com"));
3 i.putExtra(Intent.EXTRA_SUBJECT, "Lottery");
4 i.putExtra(Intent.EXTRA_TEXT, "You won the Jackpot!");
5 startActivity(i);
```

In contrast to Intent Filters Intents do not have an explicit definition in the Android manifest. So point 2 of the definition of a component is eventually violated. It is up to the component developer to document the component e. g. via source code documentation. The example in listing 1.3 uses parameters complying to the standard email application shipped with Android. There is no documentation available, not even source code documentation. But the application is open source and the correct parameters can be obtained via source code inspection. Since many Android software is open source the possibility to look up the correct parameters in the source code is given even though it is not quite pretty and/or satisfying. It would be nice to see both all dependencies and all published interfaces of an Android component in the manifest file but the developers decided only to list published interfaces in it represented by Intent Filters. Thus errors finding a missing component occur at runtime instead of being solved or recognized before starting.

The aspect of independent deployment (point 3 of the definition of a component) is explained in the following subsection with the help of an example.

3.3 Realization of a Plugin System in Android

Since hot deployment and plugin systems are a desirable objective to component systems the following section will show an example of how to realize such a plugin system in Android. Therefore one basic application is being defined that is extensible by different plugins.

Communication The communication principle is quite simple. When the basic application starts it sends a broadcast message addressing all plugins to the Android system. Thereupon all plugins reply to that broadcast message with another broadcast message addressing the basic application. The following will provide some implementation details of the basic application and the plugins.

Basic Application Listing 1.4 shows how all plugins are being requested on startup by sending a broadcast message. The `onCreate` method is being invoked by the Android framework when an Activity is started.

Listing 1.4. Request all Plugins

```
1  ...
2  public class BasicApplication extends Activity {
3    @Override
4    public void onCreate(Bundle savedInstanceState) {
5      ...
6      /* request all plugins */
7      Intent i = new Intent("org.example.REQUEST_PLUGIN");
8      sendBroadcast(i);
9      ...
10   }
11   ...
12 }
13 ...
```

Furthermore the basic application needs a Broadcast Receiver to receive the plugins' responses and an Intent Filter for routing the response to that Broadcast Receiver. Listing 1.5 shows an appropriate Broadcast Receiver. The `onReceive` method is being invoked by the Android framework upon reception of a broadcast message. The possibility of attaching some extra data to the Intent in form of key/value pairs is being used here. The package name of the plugin is put into that extra data and is reachable under the key `package_name`. The package name is used here just for demonstration purposes. You can put every kind of data in here[2].

[2] Not all data types are supported out of the box—only primitive types and those that implement the interface `Parcelable`—because components use inter process communication (IPC) to communicate due to every application runs in its own Linux process. So custom data types need to implement the interface `Parcelable`. For more information see [7, p. 111ff].

Listing 1.5. Broadcast Receiver for Responses from Plugins

```
...
public class BasicApplicationResponseReceiver extends
    BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    /* process plugin answer */
    String packageName = intent.getStringExtra("package_name"
        );
    ...
  }
}
...
```

Listing 1.6 shows the Intent Filter entry in the Android manifest file. For receiving responses from plugins the name `org.example.PLUGIN_RESPONSE` is used. So broadcast messages that are addressed to this name are being routed to the `BasicApplicationResponseReceiver`.

Listing 1.6. Intent Filter for Responses from Plugins

```
...
<receiver android:name=".BasicApplicationResponseReceiver">
  <intent-filter>
    <action android:name="org.example.PLUGIN_RESPONSE" />
  </intent-filter>
</receiver>
...
```

Plugins Just like the basic application plugins also need a Broadcast Receiver and an Intent filter. The Broadcast Receiver needs to put its package name into the Intent's extra data and send a broadcast to the basic application (see listing 1.7). The Intent Filter is really straightforward and is shown in listing 1.8.

Listing 1.7. Broadcast Receiver of a Plugin

```
...
public class Plugin1RequestReceiver extends BroadcastReceiver
    {
  @Override
  public void onReceive(Context context, Intent intent) {
    /* send broadcast response to basic application */
    Intent i = new Intent("org.example.PLUGIN_RESPONSE");
    i.putExtra("package_name", context.getPackageName());
    context.sendBroadcast(i);
  }
}
...
```

Listing 1.8. Intent Filter of a Plugin

```
...
<receiver android:name=".Plugin1RequestReceiver">
  <intent-filter>
    <action android:name="org.example.REQUEST_PLUGIN" />
  </intent-filter>
</receiver>
...
```

This example is just rudimentary. Considering useful applications some possibility of invoking a plugins' methods must be present. With the above example it can easily be realized. Since the basic application now knows the package names of all installed plugins[3] you can easily introduce another broadcast type and put the package name into the extra data, so every component can check if itself is being addressed. Obviously quite much overhead is produced in this approach as the broadcast is addressed to all plugins. The decision is made within every plugin itself. Luckily Android supports the mechanism of URI parsing with what the problem can be circumvented. As mentioned above URI matching goes beyond the scope of this article thus it will not be discussed in here.

WebSMS At this point I want to refer to WebSMS [1] as an existing example of a plugin system in Android. WebSMS is used to send text messages via different online services instead of the cellular phone network. For that purpose it also uses a basic application extensible by different so called "connectors". Connectors send text messages via specific online services and can be installed independently from the basic application. The basic application notices the existence of all available connectors at runtime and offers configuration and usage of those to the user. In terms of plugin systems connectors are plugins for the basic application and ca be installed/deployed independently. So components in Android comply to the definition of a software component.

4 OSGi on Adroid

OSGi is a dynamic module system for Java [10] and is specified by the OSGi Alliance [11].

> "OSGi technology is Universal Middleware. OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. These capabilities greatly increase the value of a wide range of computers and devices that use the Java™ platform." [10]

Since this article is not about introducing OSGi no further explanation of OSGi fundamentals will be given.

Running OSGi on Android has several advantages [13] e. g. when

[3] Package names must be unique in Android [5].

– OSGi specific features are needed
– porting an existing application based on OSGi to Android
– reusing an already existing OSGi component

4.1 Differences between OSGi and Android's Component System

Figure 3 depicts the differences between OSGi and Android's component model. As mentioned above Android applications run as an independent process within their own DVM. OSGi in contrast executes all its components in the same JVM in one single process.

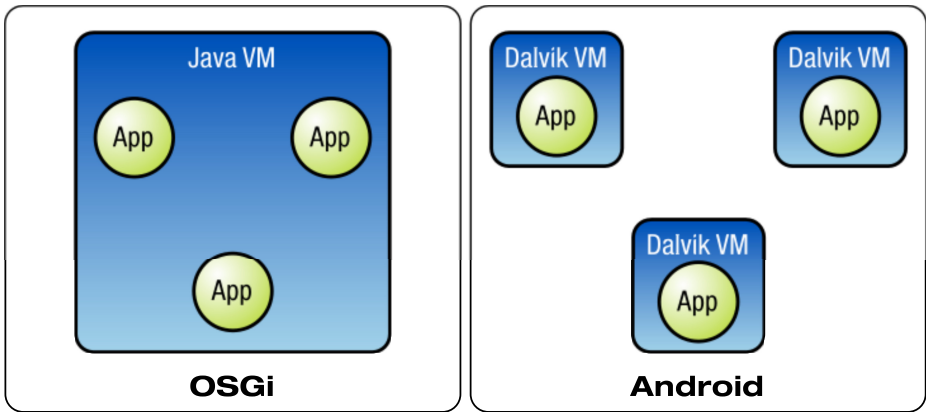

Fig. 3. OSGi vs. Android's Component System [10]

The advantage of Android's component system is that when an application crashes it does not influence other processes. The advantage of using OSGi is the decreased communication overhead. While in Android inter process communication must be used there is no need to in OSGi due to the fact that all components run in the same process.

4.2 OSGi Components on Android

Apache Felix [15] is an OSGi implementation that has successfully been ported to Android [9,13,8]. This subsection will describe the steps necessary to get OSGi components running with Apache Felix on Android. There are two simple steps to follow [14]:

1. We create a normal OSGi bundle named `Dummy.jar` with the following contents:
   – `META-INF/MANIFEST.MF`
   – `org/example/Dummy.class`

   - `org/example/impl/DummyImpl.class`

   The interface `org.example.Dummy` must be exposed in the Manifest file.
2. We convert the bundle to Dalvik format using `dx` tool:
   ```
   dx --dex --output=DummyDex.jar Dummy.jar
   ```

   The contents of `DummyDex.jar` file should be as follows:
   - `META-INF/MANIFEST.MF`
   - `classes.dex`

An alternative to Apache Felix is R-OSGi, a distributed OSGi platform that features lightweight OSGi service invocation over different transport protocols [14]. For that purpose another step is necessary: We need to add the exposed interface `Dummy.class` to the JAR file since it is needed by the R-OSGi framework to send it via a remote connection. After that the `Dummy.jar` should have the following content:

- `META-INF/MANIFEST.MF`
- `classes.dex`
- `org/example/HelloWorld.class`

Now the bundle is ready for deployment on Android. The next section will present ROCS, a remotely provisioned OSGi framework for ambient systems where components are not stored locally on a device rather than loaded into system memory directly from network.

5 ROCS: a Remotely Provisioned OSGi Framework for Ambient Systems

This section is completely based on [4]. The authors introduce the need for and objective of ROCS as follows:

> "One of the challenges of ambient systems lies in providing all the available services of the environment to the ambient devices, even if they do not physically host those services. [...] The ROCS (Remote OSGi Caching Service) framework is a novel proposal which relies on a heavy-weighted standard Java/OSGi stack. [...] The ROCS framework provides improvements in two areas. First, it defines a minimal bootstrap environment that runs a standard Java/OSGi stack. Secondly, it provides an architecture for loading any necessary missing class from remote servers into memory at runtime."

Since the authors are talking of *ambient devices* and mobile devices are ambient devices I prefer to use *mobile devices* as this article is about those. Another point worth of mentioning is that the authors have based and tested their work on a DSL router with a memory capacity of 8 MB. Today even average mobile device have memory capacities of 100 MB and above. So the first point of providing a minimal bootstrap environment plays a minor role and will not be discussed in this paper.

5.1 ROCS on OSGi

ROCS is based on OSGi since it takes advantage of Java's type safety and relies on a service-oriented approach. Nevertheless OSGi has one major drawback: components or rather bundles are deployed into a local cache before they are loaded into the device's memory. ROCS is an elegant solution that makes use of Java's remote class loading mechanisms for relocating the bundle cache from the mobile device to a remote server. So application resources are never stored locally on the mobile device. Resources are loaded from the remote server directly into the device's memory on demand. The legitimation of remote resource loading is given by considering the fact that downloading from the network is mostly equivalent to loading from a slow flash drive [4]. Given the fact that nowadays almost every mobile device is equipped with a WiFi unit the throughput of data is comparable to notebooks or desktop computers which are mostly equipped with similar hardware.

5.2 OSGi Issues

One major benefit of using OSGi is that applications can be installed, updated and uninstalled during runtime without requiring a restart of the JVM. According to the OSGi specifications bundles must be saved along with their state [4]. Normally OSGi implementations achieve this by using a local file system cache. The cache stores all bundles that are currently installed. When a bundles gets started from a remote URL it is first loaded into the local cache and then being started.

5.3 The ROCS Solution

The ROCS architecture consists of two instances (cf. figure 4):

1. Mobile devices
2. Remote cache servers (ROCS server)

Mobile devices run standard OSGi frameworks such as Apache Felix (see section 4). Remote cache servers deliver class bytecode on demand. The only difference between ROCS and standard OSGi implementations is that bundles are initially not downloaded to the device. When a remote bundle is requested the following steps are being performed:

1. The ROCS server downloads the bundle from its repository and stores it locally.
2. The framework on the mobile device then loads the meta-data contained in the bundle manifest.
3. A local shadow cache is created having the same structure as standard implementations but no bundle resources are stored in it. After that the bundle state is *Resolved*.

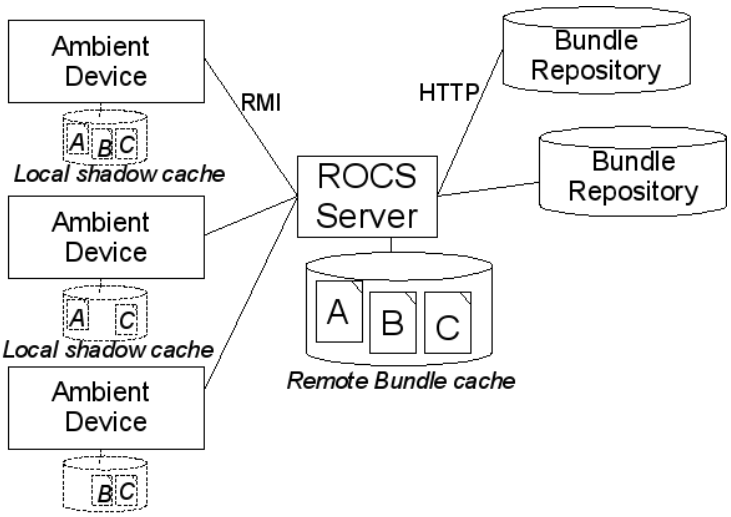

Fig. 4. Architecture of ROCS

4. The starting process invokes the bundle activator's `start` method what triggers the bytecode of the class to be loaded via RMI from the ROCS server directly into the JVM. The bundle is now in the state *Started*.
5. Any further class loading event triggers the same machinery.

In order to get the above working some modifications and additions have to be made to the standard OSGi framework running on the mobile device. For instance a new handler to manage the new URL scheme called `reference:http:` has to be added to the framework. When the OSGi framework encounters such a scheme it will use the ROCS server. Implementation details are omitted at this point since it goes beyond the scope of this paper. For further information see [4].

By using ROCS mobile devices become cloud-aware as applications can be run directly from network without the need to install them locally. This aspect raises many new application scenarios e. g. administrators have to manage just one application repository instead of managing many mobile devices; security constraints can be checked and enforced at a single point etc.

6 Conclusion

This article presented the usage of component technologies on Android. After examining Android and its component system an approach of realizing a plugin system—i e. extending the functionality of an application at runtime—in Android was presented. Thereafter Android's component system was compared to OSGi and pros and cons have been discussed. An approach of running OSGi on Android was also presented. Finally, ROCS was introduced, a framework for

running OSGi applications on a mobile device without storing bundles locally but rather loading them from network directly into the JVM.

As seen in this paper Android's component system is quite specific and of course optimized for mobile devices. Nevertheless OSGi is a mature component system adopted by many software applications. The usage of OSGi bundles on Android is a great benefit even though Android does not support it by default. Maybe in future releases Google will ship Android with an OSGi framework but in the meantime it has to be installed and bundles to be ported manually.

References

1. Bechstein, F.: Sourcecode of WebSMS, `https://github.com/felixb/websms/`
2. Becker, A., Pant, M.: Android – Grundlagen und Programmierung (Mar 2009), `http://www.androidbuch.de`
3. European Conference on Object-Oriented Programming (ECOOP): Definition of a Software Component (1996), `http://www.eecs.berkeley.edu/~newton/Classes/EE290sp99/lectures/ee290aSp994_1/tsld009.htm`, [Online; Accessed December 24th, 2010]
4. Frenot, S., Ibrahim, N., Mouel, F.L., Hamida, A.B., Ponge, J., Chantrel, M., Beras, D.: ROCS: a Remotely Provisioned OSGi Framework for Ambient Systems (2009)
5. Google Inc.: The AndroidManifest.xml File (Dec 2010), `http://developer.android.com/guide/topics/manifest/manifest-element.html`, [Online; Accessed December 24th, 2010]
6. Google Inc.: What is Android? (Dec 2010), `http://developer.android.com/guide/basics/what-is-android.html`
7. Hashimi, S., Komatineni, S., MacLean, D.: Pro Android 2. Apress (Mar 2010)
8. Heuser, S.: Entwicklung eines Frameworks zur sicheren mehrseitigen Aushandlung von Policies in Ambient-Intelligence Umgebungen. Master's thesis, Technical University Darmstadt (Apr 2010)
9. Luminis: Apache Felix on Google Android (Nov 2007), `http://lsd.luminis.nl/osgi-on-google-android-using-apache-felix/`
10. Offermans, M., Pauls, K.: OSGi on Google Android using Apache Felix (Apr 2008), `http://opensource.luminis.net`
11. OSGi Alliance: OSGi Alliance Specifications, `http://www.osgi.org/Specifications/HomePage`, [Online; Accessed December 19th, 2010]
12. OSGi Alliance: OSGi Technology, `http://www.osgi.org/About/Technology`, [Online; Accessed December 20th, 2010]
13. Schütte, J.: Felix OSGi on Android (2010), `http://linkality.org/felix-osgi-on-android/`, [Online; Accessed November 24th, 2010]
14. Schütte, J.: R-OSGi on Android (2010), `http://linkality.org/r-osgi-on-android/`, [Online; Accessed November 25th, 2010]
15. The Apache Software Foundation: Apache Felix, `http://felix.apache.org/`
16. Wikipedia, The Free Encyclopedia: Komponente (Software) (2010), `http://de.wikipedia.org/w/index.php?title=Komponente_(Software)&oldid=75243743`, [Online; Accessed December 24th, 2010]