UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea specialistica

Developing Google Android Mobile Clients for Web Services: a Case Study

Anno Accademico 2007/2008

relatore
Ch.mo prof. Stefano Russo

correlatore
Ing. Marcello Cinque

candidato
Vito Daniele Cuccaro
matr. 885/83

"Anyone who has never made a mistake
has never tried anything new."

Albert Einstein

A chi ha condiviso con me tratti di vita, lunghi o brevi,
camminando al mio fianco e sorreggendomi quando inciampavo.

To whoever shared with me stretches of life, long or short,
walking by me and holding me up while I was stumbling.

Introduction

This work was born a few months ago from the curiosity for the new mobile platform created by Google: Android. Many rumors had been around for a while but most of the people didn't even have a vague idea of what Android really was. In fact the available documentation was still poor and nebulous, the SDK hadn't reached the 1 version and no real devices had been launched yet. Nevertheless this looked a compelling subject for a thesis. I started to study the Android platform during my Erasmus exchange period at *University of Strathclyde in Glasgow* but I began to work on my case study after coming back to Italy, at *Mobilab*, a research lab of the Computer Science Department. While looking for something useful and interesting to test with the Google mobile platform, a good topic emerged. "Is Android ready for Web Services?" - we wondered. The answer was anything but obvious so that, after searching more about this, we decided that we would delve into the issue.

Web Services, by far the most popular implementation of Service Oriented Architectures, have been standing out for some years as the leading technology in the world of Distributed Computing, come to its third generation. It's therefore clear that the ability of accessing and consuming Web Services is definitely a feature a mobile device should have nowadays.

At this stage we only needed a good case study and we found a perfect one: a framework for monitoring sensor networks exposing some of its functions as Web Services, developed within a research project of our Department.

The goals of our work are therefore various: on one hand studying the Android platform

and testing some of its capabilities, on the other hand developing an Android client for Web Services, using the mentioned framework as a case study.
Moreover, as we found out that a Java mobile client for the same framework was already available, this turned out to be as well a good test-bed for experiencing a porting from a very common mobile platform (J2ME) to the new one.
Now we will summarize in a few lines the contents of this work.

In Chapter 1 we will introduce Web Services (and the Service Oriented Architecture they implement) as the evolution of distributed systems, describing the basic model and architecture and the standards involved, with a special focus on SOAP.

In Chapter 2 we will discuss the Google Android mobile platform, illustrating its architecture, main features and philosophy, with some final considerations about its relationship with Web Services.

In Chapter 3 we will describe the key tool of our case study: the ICAAS framework for accessing sensor networks. We also correlate it with Web Services and SOAs. A final presentation of the J2ME client available for the framework is also provided.

In Chapter 4 we will illustrate in details our case study, the development of an Android client for the ICAAS framework, following the three phases of Analysis, Design and Implementation and comparing this development with a standard porting process.

In Chapter 5 we will show some examples of use and scenarios of the Android application developed.

Finally in Chapter 6 we will present a comparative critical analysis of our development experience on Android.

Chapter 1

Web Services for a Service Oriented Architecture

In this chapter we will briefly introduce Web Services, which are probably the most popular application of the SOA paradigm, as a result of the astonishing evolution occurred during the latest years in the world of distributed computing, explaining the motivations for their success and analyzing their key features.

Then we will illustrate the basic operational model, the architecture and the communication models of Web Services.

Afterwards we will describe the main standards involved in the Web Services technology, with a particular focus on SOAP, which is the main protocol employed in our case study, as we will see hereafter.

Finally we will say something about one of the most common SOAP Java-based implementations, used for the present work: Apache Axis.

1.1 Web Services: the latest evolution of distributed computing

Distributing Computing can be defined a type of computing in which different components and objects comprising an application can be located on different computers connected to a network.

According to Coulouris-Dollimore-Kindberg's reference book about this subject: "A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages".

Less formally but very effectively Leslie Lamport said: "You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done."

The first model that went over the mainframe architecture, dominating the early years of computer science, and opened the way to distributed computing was the well-known two-tier client-server architecture. However it turned out soon not so fit for complex robust applications. The concept of middleware was born to target the main issues of distributed computing: heterogeneous computing environment and the need for enterprise application integration. The *middleware* can be described as a software layer interposed between operating system and applications in order to provide useful abstractions and services for developing distributed applications.

Many different models of middleware have been proposed during the years, contributing to the evolution of distributed computing from the initial basic client-server model. They can be classified in two main categories: *Data Oriented* (Remote Data Access and Transaction Processing) and *Communication Oriented* (Remote Procedure Call or RDA, Message Oriented Middleware or MOM, Tuple Space or TS, Distributed Objects Middleware or DOM and Component Model or CM).

In fact the real revolution in the world of distributed computing was introduced by DOM, which is the result of the integration of the object programming paradigm with the client-server model. That's why DOMs are regarded as the second generation of distributed software systems. The most popular examples of DOM are OMG CORBA, Java RMI and Microsoft DCOM. These technologies have been quite successful in integrating applications within a homogeneous environment inside a local area network but not equally effective in dealing with the problems of a heterogeneous network environment like the Internet. Here are the main challenges issued by DOM systems:

- extremely complex maintenance of various versions of stubs/skeletons in the client and server environments;

- hard achievement of Quality of Service goals;
- almost impossible interoperability of applications implementing different protocols;
- most of the protocols designed to work within a local network, not very firewall friendly and often unable to access over the Internet.

Most of these problems are due to tight coupling and implicit dependence among objects typical of DOM.

The natural evolution of DOM, which tried to overcome this limit, is the *Component Model* (CM), wherein not objects but more complex items called components are distributed. Common examples of CM are J2EE, Microsoft COM/DCOM (now replaced to some extent by .NET) and CORBA3.

CM, along with the concept of *Service Oriented Architecture* (SOA), gave life to the third generation of distributed systems.

The evolution of Distributed Systems is schematically represented here:

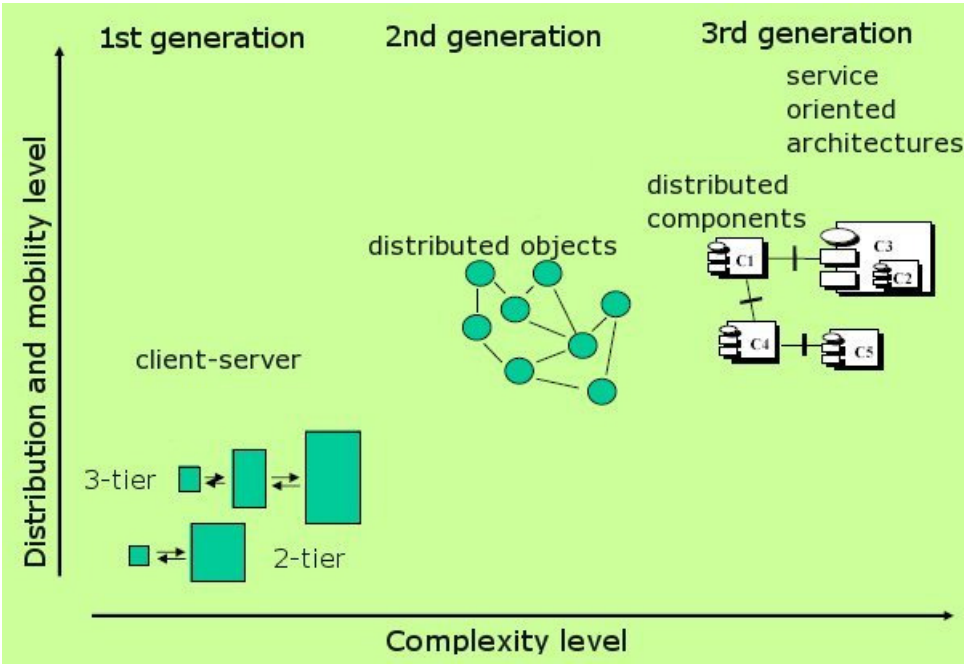

Figure 1.1

The term SOA is one of the most complex, ambiguous and overloaded as it's been used in several meanings by many people. However we will try to explain what a SOA is by giving some basic hints.

Two not completely new but successful ideas are at the basis of SOA: the interface of an application, independent from its implementation, as a contract between provider and consumer, typical of the object-oriented model, and the loose coupling between client and server, performed by the Web. In fact the main goal of SOAs is combining the vantages of DOM with the power of the Web.

The key concept of Service Orientation is basically about the exposition of software components as services so that they can be accessed by heterogeneous consumers over the Internet.

By a recursive definition we could say: "Service oriented architectures define the overall structure that is used to support service oriented programming for service oriented applications in the service oriented enterprise".

There are many alternative conceptual models for a SOA and most of them are strongly related to an enterprise organization as SOAs are mainly thought for integrating not only the applications but the departments themselves of a company. Nevertheless that is beyond our goals.

Instead we will now try to understand how SOA and Web Services relate each other: are they synonyms? One way to answer this compelling questions is through specialization: SOA is the general architectural framework that supports service oriented programming and Web Services are just one of its possible realizations. Most of the technologies mentioned before, like CORBA, MOM, .NET, J2EE, can also be seen as specializations of SOA as they let you create a service with an interface, which specifies how to call it, and deploy it on a server to make it consumable by a client.

So the general SOA model can be implemented by a number of different technologies but Web Services emerged as the best SOA realization presently available and the reason is

mainly related to the achievable independence level. The following diagram compares some SOA-oriented frameworks under this aspect.

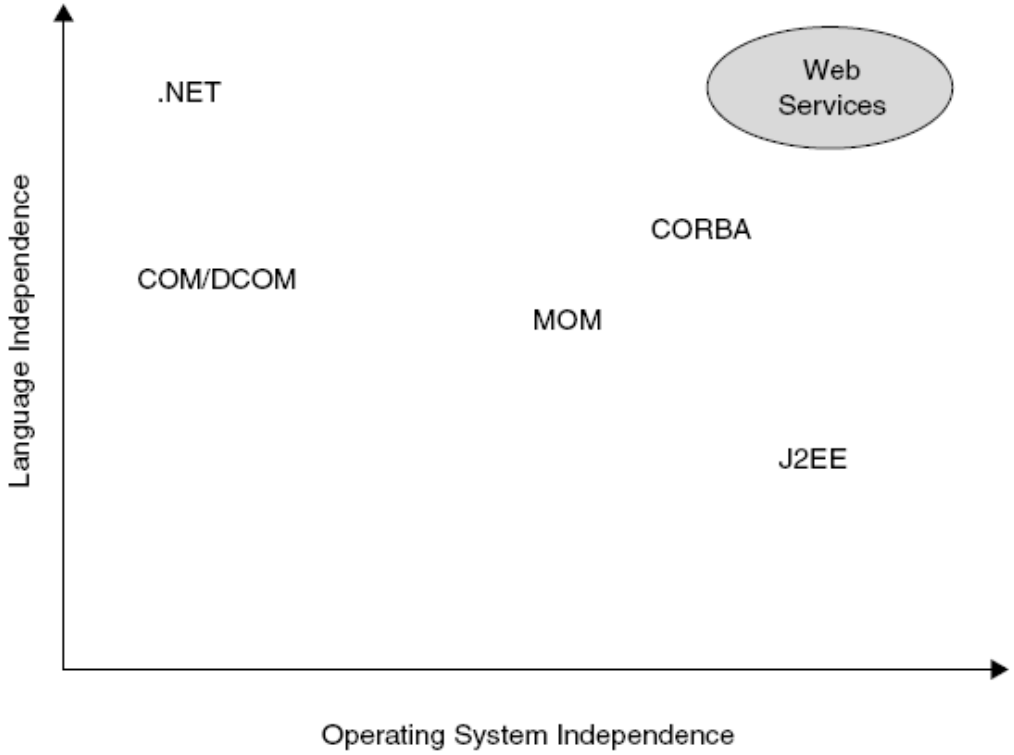

Figure 1.2

As we can see .NET grants the highest independence from the programming language but it's tightly connected to Microsoft operating systems while SUN J2EE is completely OS independent but obviously Java-based. MOM and CORBA are placed in the middle and Web Services are by far the best technology in relation to the independence features here taken into account.

According to Gartner "Web services are loosely coupled software components delivered over Internet standard technologies". More exactly they could be defined as "self-describing and modular business applications that expose the business logic as services over the Internet through programmable interfaces and using Internet protocols for the purpose of providing ways to find, subscribe, and invoke those services".

From a functional point of view the main motivation for introducing Web Services is

meeting the challenges of B2B (Business To Business) communication: while traditional web applications simply allow interaction between an end user and a web site, Web Services enable application-to-application communication and make easier accessing heterogeneous applications and devices over the Internet. The key choice for this purpose was the adoption of standard protocols and data formatting (basically XML).

This is a list of the main features of Web Services:

- data exchange based on XML (eXtensible Markup Language)
- cross-platform integration of business applications over the Internet
- programming language independence
- use of industry-standard protocols like HTTP which make them easily accessible through corporate firewalls
- a wide range of possible clients and functionalities (from a trivial request to a complex business transaction)
- platforms like J2EE, CORBA and .NET support creation and deployment of Web Services
- dynamic location and invocation from public and private registries based on industry standards such as UDDI and ebXML

1.2 Key benefits and known challenges of Web Services

The key benefits of implementing Web services are as follows:

- applications can easily become services accessible by anyone, anywhere, and from any device.
- service-based application connectivity facilitates EAI, and intra-enterprise and inter-enterprise communication.
- businesses requiring flexibility and agility in application-to-application

communication over the Internet are supported

- dynamic location and invocation of services through service brokers (registries).
- collaboration with existing applications that are modelled as services to provide aggregated Web services.

More technically here are the major reasons for choosing Web Services:

- they can be invoked through XML-based RPC mechanisms across firewalls.
- they provide a cross-platform, cross-language solution based on XML messaging.
- they facilitate application integration using a lightweight infrastructure without affecting scalability.
- they enable interoperability among heterogeneous applications.

Therefore Web services seem to be the next major technology for introducing a new way of communication and collaboration. Nevertheless we cannot ignore some known challenges associated with the mission-critical business requirements addressed by Web Services:

- *Distributed transactions*. If the environment requires distributed transactions with heterogeneous resources, it should be studied and tested with standard solutions.
- *Quality of Service* (QoS). In case of a mission-critical solution, the service providers must examine the reliability and performance of the service in peak load and uncertain conditions for high availability. The exposed infrastructure must provide load balancing, and failover and fault tolerance, to deal with these scenarios.
- *Security*. As Web services are exposed to the public using http-based protocols they must be implemented using authentication and authorization mechanisms and SSL enabling encryption of the messages. Security standards like SAML, XML Encryption, XML Signature, or XACML may be a solution.

- Other challenges include the manageability and testing of the Web services deployment, which is subjected to different operating system environments and platforms and managing service descriptions in public/private registries.

1.3 The basic operational model of Web Services

Web services operations can be conceptualized by the following simple operational model.

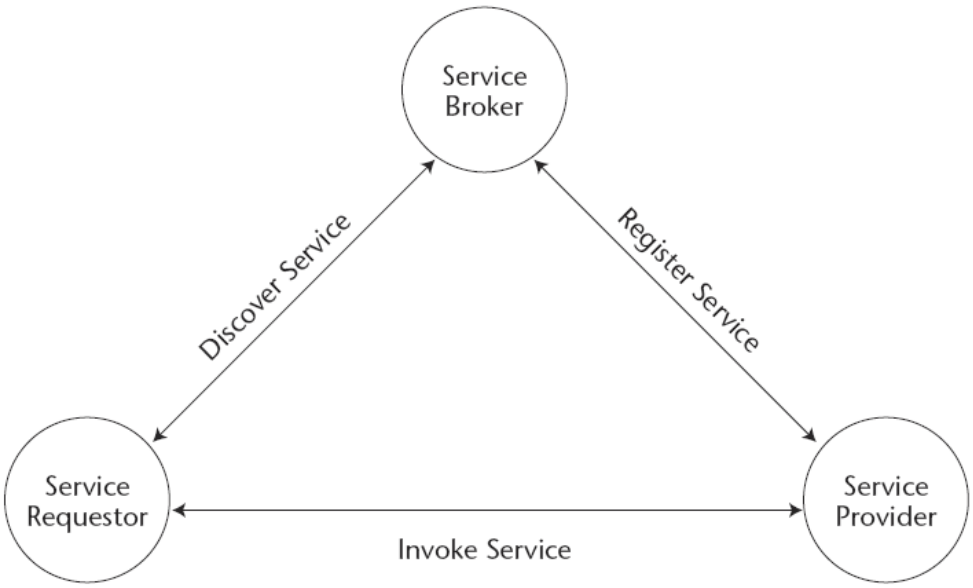

Figure 1.3

Let's briefly examine the roles involved in the model and their relationships:

- the *Service Provider* is responsible for developing and deploying the Web services; it also defines the services and publishes them with the service broker.
- the *Service Broker* (also commonly referred to as a *Service Registry*) is responsible for service registration and discovery of the Web services; it lists the various service types, descriptions, and locations of the services that help the service requestors to find and subscribe to the required services.
- the *Service Requestor* is responsible for the service invocation; it locates the Web

service using the service broker, invokes the required services, and executes it from the service provider.

1.4 The SOA-based Web Services Architecture

Let's see now how the basic operational model can be translated into a slightly more detailed architectural model

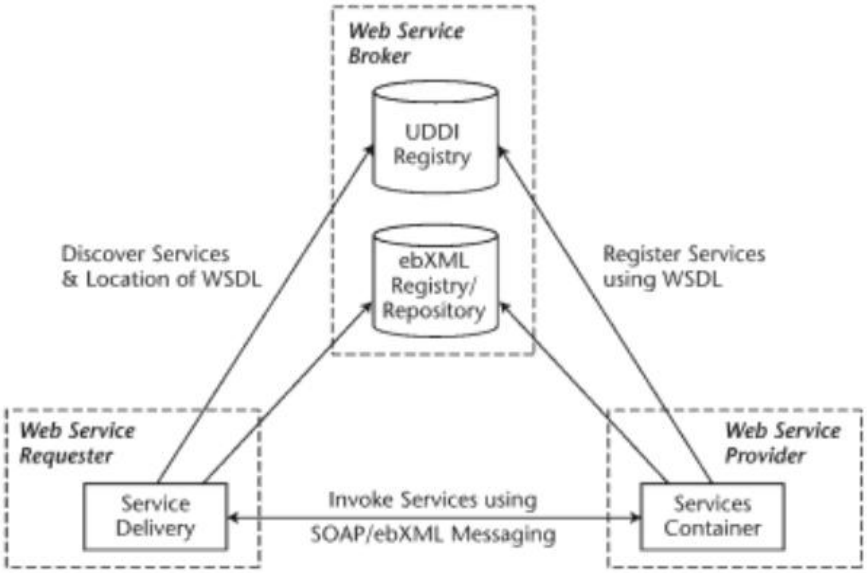

Figure 1.4

The three logical components of this schema are the core building blocks mapping the operational roles and relationships of the conceptual model seen before:

- the *Services container* acts as the Web services runtime environment and hosts the service; it exposes the potential components of the underlying applications as interfaces and facilitates the services deployment and administration. In addition, it also handles the registration of the service description with the service registries. It is usually implemented by the Web services platform provider.

- the *Services registry* hosts the published services and acts as a broker facilitating the publishing and the storing of the description of Web services registered by the service providers. Furthermore, it defines a common access mechanism for the service requestors for locating the registered services.
- the *Services delivery* acts as the Web services client runtime environment by looking up the services registries to find the required services and invoking them from the service provider. It works as a presentation module for service requestors, which exposes the appropriate interfaces or markups for generating content and delivery to a variety of client applications, devices, platforms, and so on.

To concretely build this logical architecture we need a communication model and some universally accepted standards (already mentioned in the previous figure) for describing, publishing and invoking the services.

We will analyze these two key requirements in the following paragraphs.

1.5 Web Services Communication Models

The Web Services architecture can be built following one of two possible communication models, depending upon the functional requirements.

The *RPC-based* communication model is synchronous: when the client sends a request it waits until the server sends a response back before doing any other operation. This model, typical of CORBA and RMI communication and implemented by using remote objects, makes the web services tightly coupled (in opposition to their basic principle) but allows both service providers and requestors to register and discover services. When invoking a service the client sends the appropriate parameters to the service provider, which executes the required methods and sends back the return values.

Here is the schema of a typical RPC-based communication:

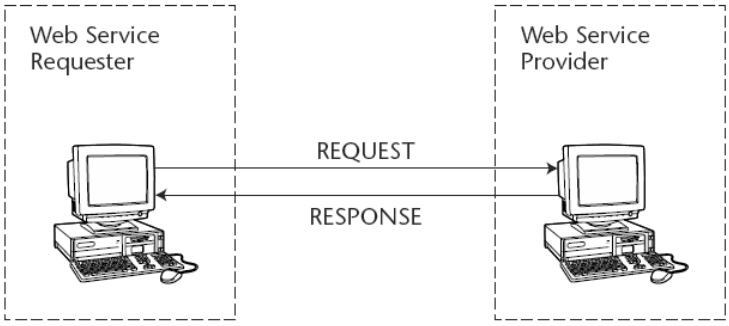

Figure 1.5

The *Messaging-based* model defines a loosely coupled and document-driven communication. The service requestor invoking a service provider does not wait for a response. It typically sends an entire document rather than a set of parameters. The service provider receives the document, processes it, and then may or may not return a message. Depending upon the implementation, the client can either send or receive a document asynchronously to and from a Web service, but it cannot do both functionalities at an instant. Furthermore, it also is possible to implement messaging with a synchronous communication model where the client sends the service request to the provider, and then waits until it receives the document back.

The following figure represents a messaging-based communication:

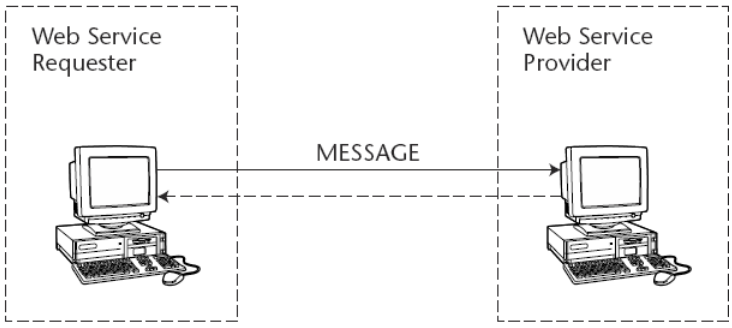

Figure 1.6

1.6 Core Standards for Web Services

The key standards and technologies for building and enabling Web Services are: XML, SOAP, WSDL, UDDI and ebXML. Let's have a quick look at them.

Extensible Markup Language (XML)

In February 1998, the Worldwide Web Consortium (W3C) officially endorsed the Extensible Markup Language (XML) as a standard data format. An XML file, coded by Unicode, contains self-describing neutral complex data that can be stored as a simple readable text document. Today, XML is the *de facto* standard for structuring data, content, and data format for electronic documents. It has already been widely accepted as the *lingua franca* for exchanging information between applications, systems, and devices across the Internet.

In the core of the Web services model, XML plays a vital role as the common wire format in all forms of communication and is also the basis for other standards.

Simple Object Access Protocol (SOAP)

Simple Object Access Protocol, or SOAP, is a standard for a lightweight XML-based messaging protocol. It enables an exchange of information between two or more peers and enables them to communicate with each other in a decentralized, distributed application environment. Like XML, SOAP also is independent of the application object model, language, and running platforms or devices. SOAP is endorsed by W3C and key industry vendors like Sun Microsystems, IBM, HP, SAP, Oracle, and Microsoft.

In the core of the Web services model, SOAP works as the messaging protocol for transport on top of Internet protocols such as HTTP, SMTP and FTP. SOAP uses XML as the message format, and a set of encoding rules for representing data as messages. It can also operate on a request/response model by exposing the functionality in a RPC-based communication model. In addition SOAP can be used with J2EE-based application frameworks. We will say more about SOAP further on.

Web Services Definition Language (WSDL)

WSDL is an XML format for describing the network services and its access information. It enables a binding mechanism for attaching a protocol, data format, abstract message, or

set of endpoints defining the location of services.

In the core of the Web services model, WSDL is used as the metadata language for defining Web services and describes how service providers and requesters communicate with one another. It also describes the Web services functionalities offered by the provider, where the service is located, and how to access the service. The service provider usually creates Web services by generating WSDL from its exposed business applications. A public/private registry is utilized for storing and publishing the WSDL-based information.

Universal Description, Discovery, and Integration (UDDI)

UDDI defines the standard interfaces and mechanisms used by registries for publishing and storing descriptions of network services in terms of XML messages. It is similar to the yellow pages or a telephone directory where enterprises list their products and services. Web services brokers use UDDI as a standard for registering the Web service providers. By communicating with the UDDI registries, the service requestors locate services and then invoke them.

In the core Web services model, UDDI makes the registry for Web services function as a service broker allowing the service providers to populate it with service descriptions and service types and the service requestors to query it to find and locate the services. Web applications interact with a UDDI-based registry through SOAP messages. These registries can be either private or public services. The UDDI working group includes leading technology vendors like Sun Microsystems, IBM, HP, SAP, Oracle, and Microsoft.

ebXML (electronic business using XML)

ebXML defines a global electronic marketplace where enterprises find one another and conduct business process collaborations and transactions. It also defines a set of specifications for enterprises to conduct electronic business over the Internet by

establishing a common standard for business process specifications, information modeling, process collaborations, collaborative partnership profiles, agreements and messaging. ebXML is an initiative sponsored by the United Nations Center for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organization for the Advancement of Structured Information Standards (OASIS).

In the Web services model, ebXML provides a comprehensive framework for the electronic marketplace and B2B process communication by defining standards for business processes, partner profile and agreements, registry and repository services, messaging services, and core components.

It complements and extends with other Web services standards like SOAP, WSDL, and UDDI.

Other Standards

Many industry initiatives and standards supporting Web services are currently available and many more will be available in the future.

1.7 SOAP: the *de facto* standard for Web Services development and invocation

With the emergence of Web services, SOAP has become the de facto communication protocol standard for creating and invoking applications exposed over a network as services. SOAP is similar to traditional binary protocols like IIOP (CORBA) or JRMP (RMI), but it adopts text-based, instead of binary, data representation using XML.

SOAP defines a lightweight wire protocol and encoding format to represent data types, programming languages, and databases. It can use a variety of Internet standard protocols (such as HTTP and SMTP) for the transport, and provides conventions for representing communication models like remote procedural calls (RPCs) and document-driven messaging. This enables inter-application communication in a distributed environment and

interoperability between heterogeneous applications over the networks.

The protocol consists of the following components:

- *SOAP Envelope* describes the message, identifying the contents and the processing information; it contains a SOAP body that can carry RPC method and its parameters, a target application specific data or a SOAP fault for reporting errors and status information
- *SOAP Transport* defines the bindings for the underlying transport protocols such as HTTP and SMTP.
- *SOAP Encoding* defines a set of encoding rules for mapping the instances of the application-specific data types to XML elements.
- *SOAP RPC conventions* defines the representation of the RPC requests and responses, which are marshalled in a data type and passed into a SOAP body.

Basically, SOAP is a stateless protocol by nature and provides a composed one-way messaging framework for transferring XML between SOAP applications which are referred to as SOAP nodes. These SOAP nodes represent the logical entities of a SOAP message path to perform message routing or processing. They can play one of the following roles:

- *SOAP sender*, which generates and sends the message.
- *SOAP receiver*, which ultimately receives and processes the message with a SOAP response, message, or fault
- *SOAP intermediary*, which can act as a SOAP sender or receiver. It can be *active* and perform some additional functions or just *forwarding*. There can be zero or more SOAP intermediaries between the SOAP sender and receiver to provide a distributed processing mechanism for SOAP messages.

This figure represents a basic SOAP message exchange model:

Figure 1.7

From a Web services point of view SOAP is the messaging protocol for exchanging information between the service provider and the service requestor. It defines and provides:

- A standardized way to transmit data using Internet-based protocols and a common-wire format (XML) between the Web service provider and its requestors.
- An extensible solution model using an XML-based framework enabling the Web service providers and requestors to interoperate with each other in a loosely coupled model and without knowing the underlying application architecture (such as programming languages and operating systems). This enables the creation of Web services over existing applications without modifying the underlying applications.

In a Web services implementation model, SOAP can be implemented both as a client and as a server application.

A *SOAP client* application acts as a Web services requestor, which typically handles an XML-based request/response, a message containing a XML document, parameters required to invoke a remote method, or the calling of a SOAP server application.

A *SOAP server* application works as a Web services provider, which processes the SOAP requests and messages from calling SOAP clients. It interacts with its encapsulated applications to process the requests or messages and then sends a response to the calling client.

A *SOAP intermediary* in the middle of a SOAP communication route plays both roles.

SOAP supports the two types of web services communication models already seen:

- *SOAP RPC* defines a remote procedural call-based synchronous communication where the SOAP nodes send and receive messages using request and response methods and exchange parameters and then return the values. In this case the SOAP request message body contains a RPC method call with its parameters while the response body returns the results of the remote processing with zero or more out parameters.
- *SOAP Messaging* defines a document-driven communication where SOAP nodes send and receive XML-based documents using synchronous and asynchronous messaging. In this model the SOAP message body (request and response) is basically represented by a XML document.

Interestingly, the SOAP specifications do not specify and mandate any underlying protocol for its communication as it chooses to bind with a variety of transport protocols between the SOAP nodes.

The *SOAP bindings* define the requirements for sending and receiving messages using a transport protocol. They also define the syntactic and semantic rules for processing the incoming/outgoing SOAP messages and a supporting set of message exchanging patterns. This enables SOAP to be used in a large number of applications and on OS platforms using a variety of protocols.

1.8 Apache Axis: a Java-Based SOAP implementation

The SOAP specifications do not mandate a single programming model nor do they define programming language-specific bindings for its implementation. It is up to the provider to choose a language and to define the implementation.

That being stated, the use of Java for developing SOAP applications grants scalability, portability and a good interoperability with heterogeneous applications residing on different platforms. Additionally, SOAP-based applications that adopt a J2EE-based infrastructure and component framework allow the inheritance of the characteristics of J2EE container-based services such as transactions, application security, and back-end application/databases connectivity. The release of the Java Web Services Developer Pack (JWSDP) also provides a full-fledged API solution for developing SOAP-based Web services. A long list of open source communities, Web services platform providers, and J2EE vendors also have released their SOAP implementations adopting Java platform and Java-based APIs.

Apache Axis is an open-source Java-based SOAP implementation for developing Web services.

As a packaged solution, the Apache Axis environment provides the following:

- A SOAP-compliant runtime environment that can be used as a standalone SOAP server or as a plug-in component in a compliant Java servlet engine (such as Tomcat, iPlanet, and Weblogic)
- An API library and runtime environment for developing SOAP RPC and SOAP messaging-based applications and services
- A transport-independent means for adopting a variety of transport protocols (such as HTTP, SMTP, and FTP)
- Automatic serialization and deserialization for Java objects to and from XML in SOAP messages
- Support for exposing EJBs as Web servicesTools for creating WSDL from Java classes and vice-versa
- Tools for deploying, monitoring, and testing the Web services

The process of installing Axis for building a Web services environment is quite simple.

Axis can be installed as part of a Java servlet engine (e.g. Apache Tomcat) or as a J2EE-compliant application server, or it also can run as an independent server. We refer to Appendix A for more details.

Chapter 2

The Google Android mobile platform

"What is Android?". This is the first question we will try to answer in this chapter.

To avoid any misunderstandings we will begin by saying briefly what Android is not.

Android is not a mobile device (the so called "G-Phone", that is the Google answer to the Apple iPhone) and neither a programming language, even if it includes a SDK with its APIs and a runtime environment (its own Virtual Machine) for applications (all written in Java). At the same time Android is not only an operating system for mobile devices, as one may think.

In a few words Android is a software stack for mobile devices that includes an operating system, middleware and key applications.

2.1 The Open Handset Alliance and the Android Challenge

The Android software platform was developed by Google within the Open Handset Alliance (OHA), a business alliance established in November 2007 by 34 mobile and technology companies, and now involving almost 50 members, among which international mobile operators (such as T-Mobile, Telecom Italia, Telefonica, Vodafone, NTT DoCoMo), software companies (the most important is Google itself), commercialization companies, chip makers (Intel, Broadcom, Nvidia, Texas Instruments, ARM, etc.) and handset producers (e.g. Motorola, Samsung, LG, ASUSTek, Sony Ericsson, Toshiba and HTC, the first to launch a "G-Phone" into the American and British market during last

Autumn). This consortium was created with the main aim of developing open standards for mobile devices in order to accelerate innovation in the mobile devices world and compete against leading companies such as Apple (that recently launched its I-Phone), Microsoft (with Windows Mobile), Nokia and Symbian. The OHA members describe it this way: "A commitment to openness, a shared vision for the future, and concrete plans to make the vision a reality. To accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience".

The OHA hopes to deliver a better mobile software experience for consumers by providing the platform needed for innovative mobile development at a faster rate and a higher quality without licensing fees for software developers or handset manufacturers.

What really makes Android attractive is its open philosophy, which should ensure that any deficiencies in user interface or native application design can be easily fixed by the developers community.

Google presents Android as "the first truly open and comprehensive platform for mobile devices, all of the software to run a mobile phone but without the proprietary obstacles that have hindered mobile innovation".

Open access to the details of the underlying system has always driven software development and platform adoption. So far, that's not happened for mobile phones, and that's one of the reasons why there are so few good mobile phone applications and fewer still available for free with the consequence that most mobile phones remain nearly identical to themselves for years.

On the contrary, Android allows and even encourages a revolution in the mobile world. It gives developers a great opportunity to change the way consumers use their phones as every Android application is conceived as a native part of the phone, not just software to be run on top of it.

In this open perspective, the Android platform is completely open source since October 2008. It has been released under an Apache license (except the parts under GPL such as the Linux kernel) which allows handsets vendors to add proprietary extensions and

customizations to it and developers to make closed source software. However Google updates and extensions will still be open source.

Soon after releasing the first Android version Google also promoted an Android Developers Challenge with an overall jackpot of 10 Million Dollars in order to encourage the diffusion of the platform and above all the development of a large number of compelling applications so that they could be immediately available for the first Android phones.

2.2 Android Software Stack

We will now consider the overall Software Architecture of the Mobile Platform:

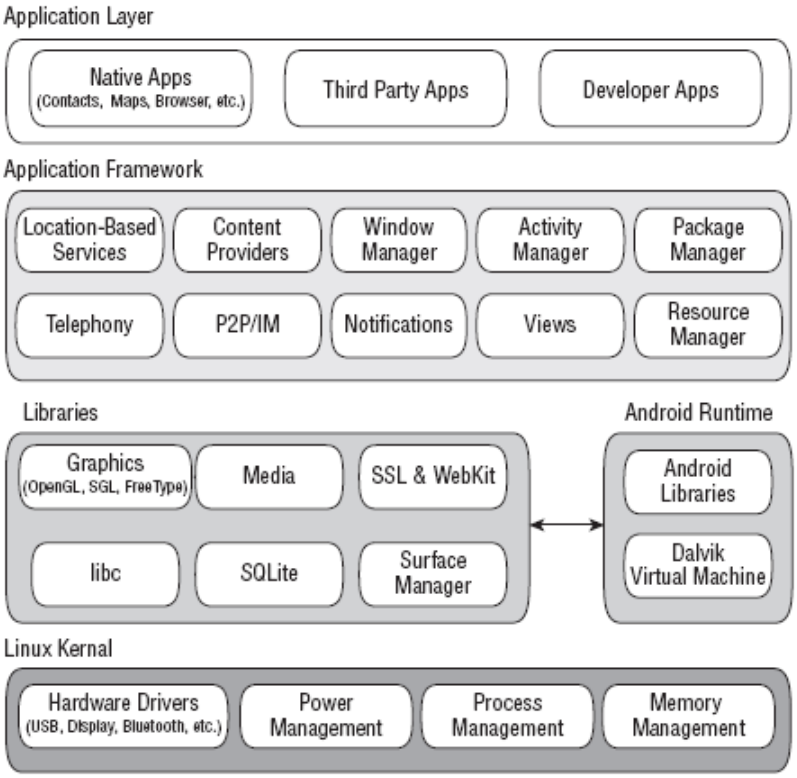

Figure 2.1

Let's briefly describe the layers that form the Android stack from bottom-up:

Linux Kernel. Core services (including hardware drivers, process and memory management, security, network, and power management) are handled by a Linux 2.6 kernel. The kernel also provides an abstraction layer between the hardware and the remainder of the stack.

Android Runtime. What makes Android something completely different from a simple mobile Linux implementation is the Runtime layer, formed by the *core libraries* and the *Dalvik Virtual Machine*. This layer is the basis for the Application Framework.
The core libraries provide most of the functionalities available in the Java API as well as some Android-specific libraries. Indeed, though all Android applications, included the native ones, are written in Java, they are not run on a Java Virtual Machine but on a special Dalvik Virtual Machine that's not simply a JVM implementation. Every application runs in its own process, with its own instance of the Dalvik Virtual Machine. We will describe it in details in the next paragraph.

Libraries. Android uses a set of C/C++ libraries supporting several components of the Android system. Developers can access these functionalities through the application framework. Some of the core libraries are:

- *System C library* - a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices
- A *media library* for playback of audio and video media
- A *Surface manager* to provide display management
- *Graphics libraries* that include SGL and OpenGL for 2D and 3D graphics
- *SQLite* for native database support
- *SSL* and *WebKit* for integrated web browser and Internet security
- *FreeType* - bitmap and vector font rendering

Application Framework. The application framework provides the classes used to create Android applications as well as a generic abstraction for hardware access and manages the user interface and application resources.

Developers have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This same mechanism allows components to be replaced by the user.

The set of services and systems underlying the applications includes:

- A rich and extensible set of *Views* that can be used to build an application
- *Content Providers* enabling applications to access data from other applications or to share their own data
- A *Resource Manager* for accessing non-code resources such as graphics and layout files
- A *Notification Manager* used by applications for displaying custom alerts
- An *Activity Manager* that manages the lifecycle of applications and provides a common navigation backstack

Applications. All applications, both native and third party, are built using the same API libraries on the application layer which runs within the Android runtime, thanks to the classes and services made available by the application framework.

The native preinstalled applications normally include:

- An e-mail client compatible with Gmail
- An SMS management application
- A full PIM (personal information management) suite with calendar and contacts list
- A fully featured mobile Google Maps application
- A WebKit-based web browser
- An Instant Messaging Client

- A music player and picture viewer
- The Android Marketplace client for downloading third-party Android applications
- The Amazon MP3 store client for purchasing DRM free music

All native applications are written in Java using the Android SDK and are run on Dalvik.

2.3 The Dalvik Virtual Machine

As mentioned before, rather than use a traditional Java virtual machine (VM) such as Java ME (Micro Edition), Android uses its own custom VM, the Dalvik Virtual Machine, that plays a key role in the mobile platform. It was designed by Dan Bornstein, who named it after the fishing village of Dalvík in Iceland, where some of his ancestors lived.

Unlike most virtual machines and true Java VMs which are stack-based, the Dalvik VM is a register-based architecture. The relative advantages of the two approaches depend on the interpretation/compilation strategy chosen. However, stack based machines generally require more instructions than register machines to implement the same high level code as they must use instructions to load data on the stack and manipulate that data. Nevertheless the instructions in a register machine must encode the source and destination registers and therefore tend to be larger. According to Bornstein himself this choice is mainly motivated by the need for avoiding instruction dispatch and unnecessary memory access as well as consuming instruction stream efficiently thanks to a higher semantic density per instruction. He reports some stats to compare a register based VM like the Dalvik with a stack based virtual machine: 30% fewer instructions, 35% fewer code units and 35% more bytes in the instruction stream (but the instructions are consumed two at a time).

The Dalvik VM uses the device's underlying Linux kernel to handle low-level functionality including security, threading, and process and memory management. It's also possible, at least in principle, to write C/C++ applications that run directly on the underlying Linux OS. However all hardware and system service accesses performed by applications are in fact managed using Dalvik as a middle tier. By using a VM to host

application execution, developers have an abstraction layer that ensures they never have to worry about a particular hardware implementation.

DVM is optimized for low memory requirements, and is designed to allow multiple VM instances to run efficiently at once. It is often referred to as a Java Virtual Machine, but this is not accurate, as the bytecode on which it operates is not Java bytecode. Instead, it runs Dalvik executable files, a format optimized to ensure minimal memory footprint so allowing good performance on systems that are constrained in terms of memory and processor speed. An uncompressed .dex file is typically a few percent smaller in size than a compressed .jar (Java Archive) derived from the same .class files. The .dex executables are created by transforming Java language compiled classes through the dx tool included in the SDK.

The Dalvik executables may be modified again when they get installed onto a mobile device. In order to gain further optimizations, byte order may be swapped in certain data, simple data structures and functions libraries may be linked inline, and empty class objects may be short-circuited, for example.

Here are some further features that differentiate the Dalvik VM from other standard VMs:

- The VM was slimmed down to use less space
- Dalvik has no Just-in-time compiler
- The constant pool uses only 32-bit indexes to simplify the interpreter

2.4 Android Development Framework

The Android software development kit (SDK) includes everything you need for developing, testing, and debugging Android applications. Let's see what's available for download:

- The *Android APIs* that let developers access the Android software stack. These are the same libraries used by Google to create native Android applications.

- *Development Tools* to compile and debug your applications such as Dalvik Debug Monitoring Service (DDMS) for monitoring and controlling the Dalvik virtual machines on which you're debugging your applications, Android Asset Packaging Tool (AAPT) for creating distributable Android package files (.apk) and Android Debug Bridge (ADB) that provides a link to a running emulator letting you copy files, install .apk files and run shell commands.
- The *Android Emulator*, a fully interactive device emulator letting you see how your applications will look and behave on a real Android device. It is an implementation of the Dalvik Virtual Machine providing a number of user interfaces and full network connectivity as well as the ability to simulate sending and receiving SMS and voice calls. All Android applications run within the Dalvik VM so that the emulator is an excellent environment as it is hardware-neutral therefore providing a better independent test environment than any single hardware implementation.
- Full *Documentation* of the available APIs.
- *Sample Code* that demonstrates some of the possibilities available using Android and how to use individual API features.
- *Online Support* provided by the developers community included the proper Google Groups, with regular participation of the Android development team.
- A *plug-in* for the development environment Eclipse that simplifies project creation and tightly integrates Eclipse with the Android Emulator and debugging tools. Lately a similar plug-in for Netbeans has become available too.

2.5 Android Application Anatomy

An Android application consists of loosely coupled components, bound using a project manifest that describes each component and how they interact.

A peculiarity of Android is that an application can use elements of another application (provided that application permits it). For this purpose the system must be able to start an

application process when any part of it is required. That's why Android applications don't have a single entry point. Rather, they have essential components that the system can instantiate and run as needed.

There are four types of independently launchable components:

Activities form the presentation layer of an application: an Activity usually coincides with a single screen. Each activity is implemented as a single class that extends the Activity class, displays a user interface composed of Views and responds to events. Most applications consist of multiple screens. Each of these screens would be implemented as an activity. Moving to another screen is accomplished by starting a new activity. When a new screen opens, the previous screen is paused and put onto a history stack. The user can navigate backward through previously opened screens. Android uses a special class called Intent to move from screen to screen. An intent describes what an application wants to do while an IntentFilter is another class describing what kind of intents an Activity is able to handle. Navigating from screen to screen is accomplished by resolving intents. The process of resolving intents happens at run time. This gives two key benefits: Activities can reuse functionality from other components simply by making a request in the form of an Intent and can be replaced at any time by a new Activity with an equivalent IntentFilter.

Services are the invisible workers of an application used to perform regular processing that needs to continue even when the Activities aren't active or visible. A Service is long-lived code without a UI but rather running in the background for an indefinite period of time. However it's possible to communicate with a service through an appropriate interface.

A typical function generally accomplished by a service is a background music player.

Broadcast Receivers enable the development of event-driven applications. They let applications do some actions in reaction to external events by listening to broadcast Intents

that match specific filter criteria (as we will see hereafter). When triggered by an event a Broadcast Receiver will automatically start the application component best matching the related Intent Filter. Broadcast receivers are usually registered in the AndroidManifest.xml file (described afterwards) of the application, which does not have to be running for its broadcast receivers to be called; the system will start the application, if necessary, when a Broadcast Receiver is triggered. Broadcast Receivers do not display a UI, although they may use the Notification Manager to alert the user if something interesting has happened.

Content Providers form shareable data stores. They are the preferred way for sharing data across application boundaries. This means that it's possible to configure Content Providers to permit access from other applications and use Content Providers exposed by others to access their stored data. The data can be stored in the file system, in an SQLite database or otherwise. Android devices include some native Content Providers allowing to share useful databases like contact information. A Content Provider can be made synchronizable by setting the proper attribute of the tag provider in the manifest. However a built-in synchronization service seems to be still unavailable at the moment so that one should implement its own synchronization logic to get a Provider actually synchronized.

Not every application needs to have all four, but all Android applications will be written with some combination of these components.
Now we will describe a few more elements which are not, strictly speaking, core components of an Android application but are essential for its working:

Intents are simple message-passing frameworks providing a facility for late run-time binding between components in the same or different applications. The Intent object itself is a passive data structure holding an abstract description of an operation to be performed or, in the case of broadcasts, a description of something that has happened and is being announced. Intents can be divided into two groups.

Explicit intents designate the target component by its name. Since component names would generally not be known to developers of other applications, they are typically used for application-internal messages — such as an activity starting a subordinate service or launching another activity.

Implicit intents do not name a target but broadcast messages system-wide. They are often used to activate components in other applications. In this case the system must find the best component (or components) to handle the intent (a single activity or service to perform the requested action or the set of broadcast receivers to respond to the broadcast announcement) by comparing the contents of the Intent object to Intent Filters, structures associated with components that can potentially receive intents (see below).

Intent Filters advertise the capabilities of a component (activity, service or broadcast receiver) to handle implicit intents. If a component does not have any intent filters, it can receive only explicit intents. A component with filters can receive both explicit and implicit intents. An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

Notifications allow to signal users without interrupting their current Activities. They're the preferred technique for getting a user's attention from within a Service or Broadcast Receiver. For example, when a device receives a text message or an incoming call, it alerts you somehow (by flashing lights, making sounds, displaying icons, or showing dialog messages).

2.6 The Android Manifest

To start an application component the Android system must know that it exists. For this reason applications declare their components in a manifest file. Every project must have in

its root directory an AndroidManifest.xml file (with exactly that name) defining the structure and metadata of the application and its components.

This structured XML file, in addition to declaring the application's components, does some other things, among which:

- naming the Java package so that the application has a unique identifier
- determining which processes will host application components.
- declaring the permissions the application must have in order to access protected parts of the API and interact with other applications and the permissions that others are required to have in order to interact with the application's components
- listing the Instrumentation classes that provide profiling and other information as the application is running (these declarations are present in the manifest only during development and testing but are removed before the application is published)
- declaring the minimum level of the Android API that the application requires
- listing the libraries the application needs to be linked to

The general structure of the manifest is showed below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
   <uses-permission />
   <permission />
   <permission-tree />
   <permission-group />
   <instrumentation />
   <uses-sdk />
   <application>
      <activity>
         <intent-filter>
```

```
            <action />
            <category />
            <data />
        </intent-filter>
        <meta-data />
    </activity>
    <activity-alias>
        <intent-filter> . . . </intent-filter>
        <meta-data />
    </activity-alias>

    <service>
        <intent-filter> . . . </intent-filter>
        <meta-data/>
    </service>
    <receiver>
        <intent-filter> . . . </intent-filter>
        <meta-data />
    </receiver>
    <provider>
        <grant-uri-permission />
        <meta-data />
    </provider>
    <uses-library />
  </application>
</manifest>
```

As we can see, the file includes nodes (represented by tags) for each of the components

(Activities, Services, Content Providers, and Broadcast Receivers) that make up the application and, using Intent Filters and Permissions, determines how they interact with each other and with other applications.

It also offers attributes to specify application metadata (like its icon or theme), and additional top-level nodes can be used for security settings and unit tests.

The root *manifest* tag includes all the nodes of the file, among which:

- the *application* node, one for each manifest, specifying the metadata for the application (including its title, icon, and theme). It also acts as a container that includes the Activity, Service, Content Provider, and Broadcast Receiver tags used to specify the application components.
- the *activity* tag, required for every Activity or any other screen or dialog to be displayed, with the android:name attribute to specify the class name. Trying to start an Activity that's not defined in the manifest will throw a runtime exception. Each Activity node supports intent-filter child tags that specify which Intents launch the Activity.
- the *service* node, one for each Service class used in the application, also supporting intent-filter child tags to allow late runtime binding.

Broadcast receivers can either be declared in the manifest, or they can be created dynamically in code.

2.7 Android Application Lifecycle

An important and unusual feature of Android is that applications have no control over their own life cycles. Instead, application components must listen for changes in the application state and react accordingly, being prepared for untimely termination.

As mentioned before, by default, each Android application is run in its own process executing a separate instance of Dalvik VM. Memory and process management of each

application is handled exclusively by the runtime layer.

When the first of an application's components needs to be run, Android starts a Linux process for it with a single thread of execution. By default, all components of the application run in that process and thread. However, components can be arranged to run in other processes, additional threads can be spawned for any process.

The process where a component runs is controlled by the manifest file. The component elements — <activity>, <service>, <receiver>, and <provider> — each have a process attribute that can specify a process where that component should run. These attributes can be set so that each component runs in its own process, or so that some components share a process while others do not. They can also be set so that components of different applications run in the same process — provided that the applications share the same Linux user ID and are signed by the same authorities. The <application> element also has a process attribute, for setting a default value that applies to all components.

All components are instantiated in the main thread of the specified process, and system calls to the component are dispatched from that thread. Separate threads are not created for each instance. Even though an application may be confined to a single process, there will likely be sometimes need for spawning a thread to do some background work. Threads are created in code using standard Java Thread objects.

Android aggressively manages its resources to ensure that the device remains responsive. This means that processes (and their hosted applications) can be killed, without warning if necessary, to free resources (above all memory) for higher-priority applications (generally those currently interacting directly with the user).

The order in which processes are killed to free resources depends on the priority of the hosted applications. An application's priority is equal to its highest-priority component currently active.

If two applications are the same priority level, the process that has been at a lower priority longest will be killed first. Process priority is also affected by interprocess dependencies: an application will have at least the same priority level as the application it supports.

All Android applications remain running and in memory until the system needs its resources for other applications.

Clearly it is important for developers to understand how different application components impact the lifetime of the process hosting them. Not using these components correctly can result in the system killing the application's process while it is doing important work.

Processes are therefore placed into a hierarchy based on the state of the application components running within them. These are, in order of importance:

1. *Foreground processes*, hosting applications currently interacting with the user. These are generally the processes Android is trying to keep responsive by reclaiming resources and are therefore killed only as a last resort being moreover very few. They include: Activities in an "active" state; that is, in the foreground and responding to user events; Activities, Services, or Broadcast Receivers currently executing an onReceive event handler; Services executing an onStart, onCreate, or onDestroy event handler.
2. *Visible processes*, hosting Activities visible to the user on-screen but not in the foreground or responding to user events. This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity). There are generally very few visible processes, and they are only be killed in extreme circumstances to allow active processes to continue.
3. *Service processes*, hosting Services that have been started. Services support ongoing processing that should continue without a visible interface. As Services don't interact directly with the user, they receive a slightly lower priority than visible Activities. They are still considered important and won't be killed unless resources are needed for active or visible processes.
4. *Background processes*, hosting Activities that aren't visible and that don't have any Services started and therefore have no direct impact on the user experience. The system can kill such processes at any time to reclaim memory for one of the most

important processes types. As there are usually many of these processes running, they are kept in an LRU list, so that the system gives higher priority to most recently used processes.

5. *Empty processes*, not holding any active application components. They are at the end of their lifetime. The only reason to keep such processes in memory is as a cache to speed up startup the next time the application it hosts is launched. Therefore these processes are routinely killed as required.

The lifecycle just discussed is here represented:

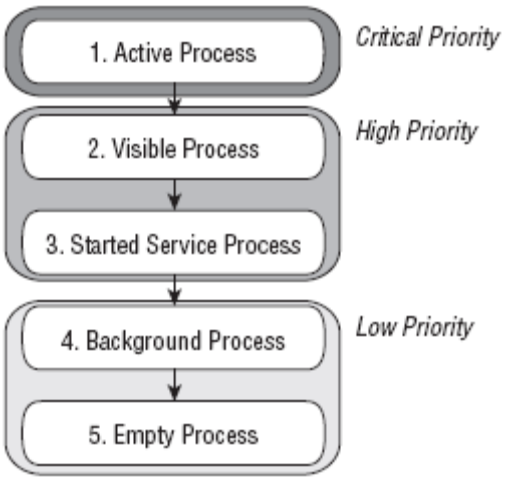

Figure 2.2

Since the lifecycle of an Android application is tightly connected to the lifecycle of its components, it can be useful to present schematically the lifecycle of an Activity and of a Service, two of the most important parts of an application.

Activity Lifecycle

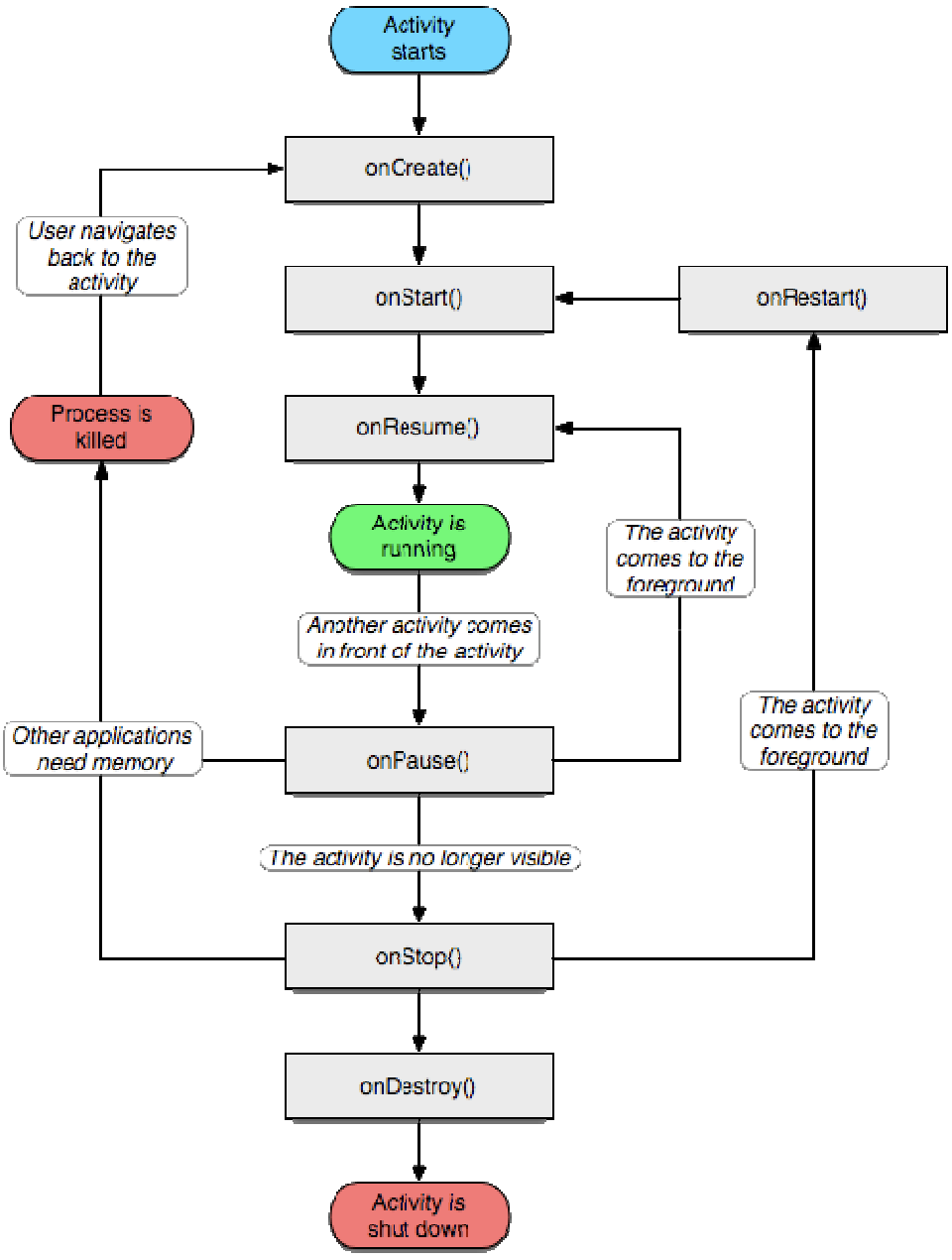

Figure 2.3

Service Lifecycle

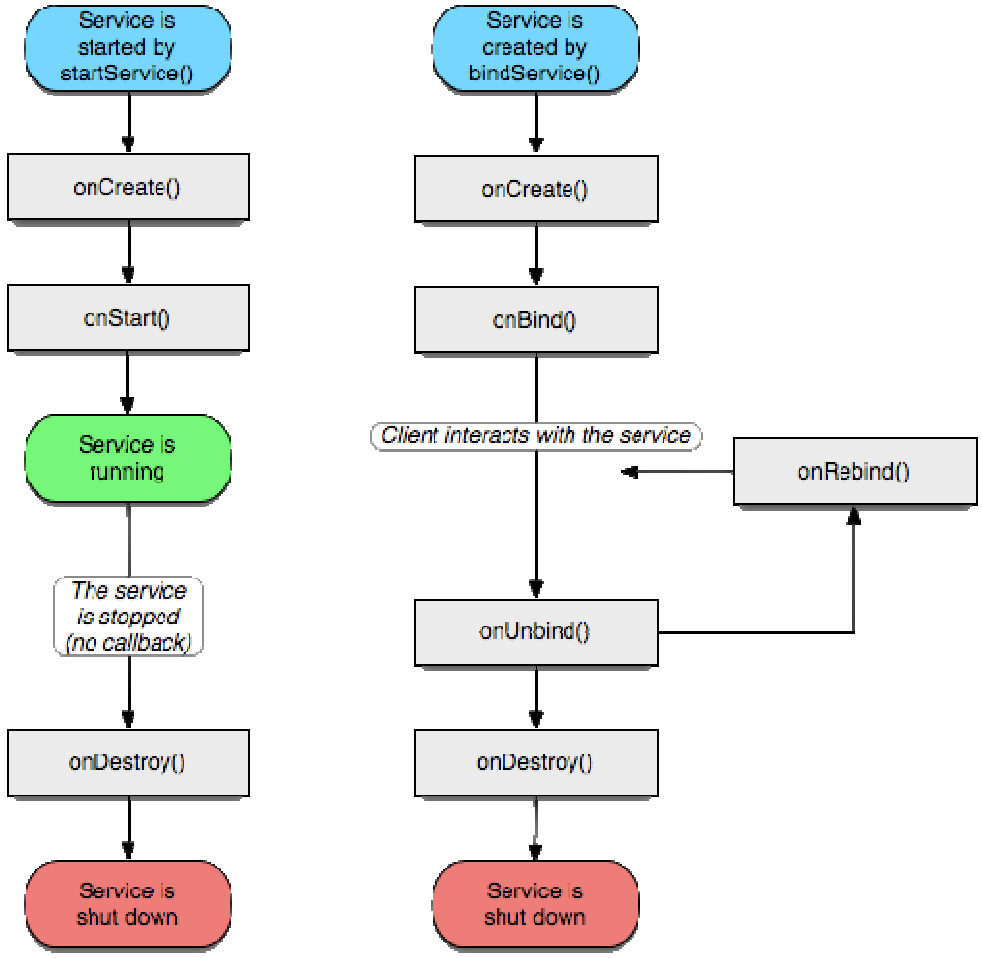

Figure 2.4

2.8 Android APIs

Android provides a large number of specific libraries for applications development, in addition to many Java APIs. The following list of core Android APIs gives an insight into what will be surely supported by all Android devices:

android.util contains low-level classes like specialized containers, string formatters, and XML parsing utilities;

android.os provides access to basic operating system services like message passing, interprocess communication, clock functions, and debugging;

android.graphics supplies the low-level graphics classes;

android.text for displaying and parsing text;

android.database provides the low-level classes required for handling cursors when working with databases;

android.content is used to manage data access and publishing through services for dealing with resources, content providers, and packages;

android.view supplies classes for building the Graphic User Interface;

android.widget for drawing elements of the GUIs such as lists, buttons and layouts;

com.google.android.maps, a high-level API providing access to native map controls;

android.app, a high-level package providing access to the application model (including Activity and Service APIs);

android.provider to ease developer access to certain standard Content Providers (such as the contacts database);

android.telephony allows to directly interact with the device's phone stack in order to make, receive, and monitor phone calls, phone status, and SMS messages.

android.webkit features APIs for working with Web-based content, including a WebView control for embedding browsers in your activities and a cookie manager.

In addition to these core APIs some advanced libraries for more complex applications are available. Their detailed features and functionalities depend on the specific device. Let's list some of them:

android.location gives applications access to the device's current physical location, using whatever position-fixing hardware or technology is available on the device;

android.media provides support for playback and recording of audio and video media files, including streamed media;

android.opengl offers a powerful 3D rendering engine using the OpenGL ES API for

creating dynamic 3D user interfaces for your applications;

android.hardware, where available, exposes sensor hardware including the camera, accelerometer, and compass sensors;

android.bluetooth, *android.net.wifi* , and *android.telephony* provide low-level access to the hardware platform, including Bluetooth, Wi-Fi, and telephony.

2.9 Android GUIs

We said that an Activity coincides with a screen of the application but we have to clarify that, by itself, it is graphically represented by an empty screen. The setContentView method is called within the onCreate handler of the Activity to set the Graphic User Interface to display: it accepts either a single View instance or a layout resource ID. This allows to define the User Interface either in Java code (through the specific Android APIs) or adopting the preferred technique of external layout resources (accomplished by xml files). The latter choice is the real Android way for GUIs and enables the decoupling of presentation layer from the application logic thanks to the opportunity of changing the presentation without changing code. Furthermore this makes possible to specify different layouts optimized for different hardware configurations, even changing them at run time based on hardware changes (such as screen orientation).

For example, a simple vertical layout with a text view and a button looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >
   <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Whichever technique is chosen for creating a GUI, this is accomplished by using a hierarchy of View and ViewGroup nodes, as shown in the diagram below. This hierarchy tree can be as simple or complex as needed and can be built up using either Android's set of predefined widgets and layouts or custom Views.

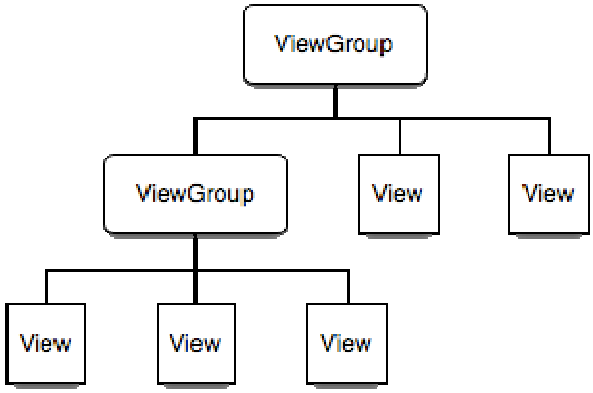

Figure 2.5

Views are the basic class for visual interfaces: they serve as the base for subclasses called "widgets," which are UI objects like text fields and buttons.

A View object is a data structure whose properties store the layout parameters and content for a specific rectangular area of the screen. A View object handles its own measurement, layout, drawing, focus change, scrolling, and key/gesture interactions for the rectangular

area of the screen in which it resides. As an object in the user interface, a View is also a point of interaction for the user and the receiver of the interaction events.
All User Interface controls, and the layout classes, are derived from Views.

View Groups are extensions of the View class that can contain multiple children Views. By extending the ViewGroup class, it is possible to create compound controls that are made up of interconnected Views. The ViewGroup class is also extended to provide the layout managers, classes useful for creating GUIs through the layout xml files.

Note that there is a tight connection between the Android classes for GUIs and the tags of an external layout file: even the names used for the visual elements are exactly the same.

2.10 A kind of magic: R.java and project resources management

Resources are external non-code files used by applications and compiled at build time.
It's always good practice to keep them external to the code in order to make them easier to maintain, update and manage and to facilitate the definition of alternative resource values to support different hardware and internationalization.
Android supports a number of different kinds of resources, such as XML, PNG, and JPEG files, and their externalization. The XML files have very different formats depending on what they describe and are compiled into a binary, fast loading format for efficiency reasons. Maybe the most powerful resources available for externalization are layouts: as mentioned before, xml-defined layouts can be easily changed depending, for example, on the screen size and orientation, so making GUIs device-independent.
Application resources need to be stored under the res/folder of the Android project hierarchy, including some subfolders for different kinds of resources.
There are seven primary resource types stored in different folders: simple values (like strings and arrays) and styles (in /res/values/), images (in /res/drawable/), layout-files (in

/res/layout/), animations (in /res/anim/), raw files such as mp3 and similar (in /res/raw/), and xml files with different functions.

As well as customized resources Android supplies several system resources that can be used in the applications directly from the code or through other resources referencing them. It's important to note that is not possible to choose a particular version of resources to use: Android will automatically select the most appropriate value for a given resource based on the current hardware and device settings.

When an application is built its resources are compiled as efficiently as possible and included in the package. This process also creates an R class file that contains references to all resources included in the project. The R.java file so generated (and, when working in Eclipse, automatically updated every time a resource is added or modified) is a sort of short-hand way to refer the resources available in the project. This allows to quickly and interactively locate every reference with the further advantage of compile-time syntax checking.

The R class contains static subclasses for each of the resource types for which at least one resource has been defined: they expose their associated resources as variables, with names matching the resource identifiers.

Therefore using resources in code is just a matter of knowing the full resource ID and what type of object your resource has been compiled into. The syntax for referring to customized resources is *R.resource_type.resource_name*, while Android system resources can be used by typing an expression like this: *android.R.resource_type.resource_name*. In both cases *resource_type* is the R subclass that holds a specific type of resource and *resource_name* is the name attribute for resources defined in XML files, or the file name (without the extension) for resources defined by other file types.

2.11 Developing for Android: design philosophy and best practices

Google explicitly defines a design philosophy for Android by recommending some best

practices useful to meet three key requirements:

- Performance
- Responsiveness
- Seamlessness

We will briefly analyze them.

Performance

An Android application should be fast or, more accurately, efficient, as in fact in a resource-constrained environment like a mobile, being fast means being efficient.

One of the keys to write efficient Android code is to not carry over assumptions from desktop and server environments to embedded devices because even the most powerful mobile device can't match the capabilities of a typical desktop system.

For this reason the main goal Android developers should pursue is saving resources as much as possible, with a particular focus on memory. This means keeping memory allocations to a minimum, writing tight code, and avoiding certain language and programming idioms that can make performance worse. In object-oriented terms, most of this work takes place at the method level, on the order of actual lines of code, loops, and so on.

The two basic rules for achieving this can be so summarized:

- Don't do work that you don't need to do.
- Don't allocate memory if you can avoid it.

Here are some programming suggestions inspired by these principles:

- Avoid creating objects when not essential: it’s never free with reference to memory.

- Use native methods rather than customized methods: they are usually much faster as they are tipically implemented in C/C++.
- Prefer Virtual over Interface but Static over Virtual.
- Avoid Internal Getters/Setters.
- Cache field lookups: copying an object field into a local variable prevents accessing the object at every iteration of a loop.
- Declare constants Final.
- Use Enhanced For Loop Syntax With Caution
- Avoid Enums and Float
- Use package scope with inner classes

Responsiveness

An Android application should be responsive, i.e. it should respond to user input as quickly as possible, in order to keep a good usability level. Lack of responsiveness could be caused for example by slow I/O and network operations or complex computations.

The Android system guards against applications that are insufficiently responsive for a period of time by displaying a dialog to the user, called the Application Not Responding (ANR) dialog. The user can choose to let the application continue, but of course he will not appreciate seeing this dialog too often. So it's important to design responsiveness into applications, so that the system displays an ANR to the user as seldom as possible.

The best solution for this problem is generally threading: the main thread (which drives the user interface event loop) can keep running while its children threads do most of the work. Since threading is accomplished at the class level, you can think of responsiveness as a class problem (while performance was a method-level issue).

Seamlessness

Even if an application is fast and responsive, it can still annoy users if it’s not seamless enough. An Android application must allow users to fluidly switch between applications.

A lack of seamlessness can cause several problems such as unplanned interactions with other applications or dialogs, inadvertent loss of data, unintended blocking, and so on. In short, an application should interact seamlessly with the system and with other applications. The goal is a consistent and intuitive user experience where applications can started, stopped and paused instantly, without noticeable delays, and switching from an application to another one is quick, fluent and easy.

Seamlessness is particularly important for an Android device that will typically run several third-party applications written by different developers.

The Android system is designed to treat applications as a sort of federation of loosely-coupled components, rather than chunks of black-box code, so that they can cooperate nicely with the system and integrate cleanly and seamlessly with other applications. Therefore seamlessness is clearly a component-level requirement.

Probably the most important practice to meet it concerns the way users are notified about events: using the proper system facility for notifying the user (the Notification classes) allows to signal events discreetly without interrupting him but leaving the user in control.

Some other good rules for developing a seamless application are listed below:

- Don't drop data: this could cause data loss if the application is killed by the system for any reason.
- Don't expose raw data: this doesn’t help interoperability with other applications; using Content Providers for data sharing is definitely better.
- Don't interrupt the user: as seen, Notifications are the correct way to draw his attention.
- Use Threads for expensive or long-running computations.
- Don't overload a single Activity screen
- Extend System themes when designing a GUI, in order to give the application a uniform look.
- Design a GUI to work with multiple screen resolutions.

- Assume the network is slow: minimizing network accesses and bandwidth will result in a good user experience even with a low connection.
- Don't assume touchscreen or keyboard as real devices can be quite different.
- Try to conserve the device battery as much as possible by minimizing the power drain.

Another key requirement for an Android application is Security, discussed in the next paragraph.

2.12 Paranoid Android: the Security Model

Android is a multi-process system, in which each application (and part of the system) runs in its own process.

Much of its security is native to the underlying Linux kernel, providing facilities, such as user and group IDs that are assigned to applications during their installation and remain constant for the duration of their life on the device. This has the effect of sandboxing every process and the resources it creates, so that it can't affect (or be affected by) other applications.

Because of this kernel-level security, additional steps are needed to make applications communicate with each other.

To provide application-level finer-grained security features Android uses the permission mechanism, enforcing restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data.

No Android application has, by default, any permissions associated with it, so that it cannot do anything that would adversely impact the user experience or any data.

To make use of protected features of the device, one or more <uses-permission> tags declaring the permissions the application needs, must be included in the AndroidManifest.xml.

For example, an application that needs to access to the network should specify: <uses-permission android:name="android.permission.INTERNET"></uses-permission>

Also many of Android native components have permission requirements. The permission strings used by native Android Activities and Services can be found as static constants in the android.Manifest.permission class.

When an application package is installed, the permissions requested in its manifest are analyzed and granted (or denied) by checks with trusted authorities and user feedback.

Unlike many existing mobile platforms, all Android permission checks are done at installation. After that, the user will not be prompted again to evaluate those permissions.

All Android applications (.apk files) must be signed with a certificate whose private key is held by their developer. This certificate identifies the author of the application.

As the certificate is used only to establish trust relationships between applications, not for wholesale control over whether an application can be installed, it does not need to be signed by a certification authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates. The most significant ways that signatures impact security is by determining who can access signature-based permissions and who can share user IDs.

Because of the process level security provided by the Linux kernel, any two applications cannot normally run in the same process, since they need to run as different Linux users. The sharedUserId attribute in the AndroidManifest.xml's manifest tag of each package can be used to have them assigned the same user ID. This way, for purposes of security the two packages are then treated as being the same application, with the same user ID and file permissions. In order to retain security, only two applications signed with the same signature can be given the same user ID.

Any data stored by an application is assigned that application's user ID, and is not normally accessible to other packages. When creating a new file, the MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE flags can be used to allow any other package to read/write the file.

2.13 Android and Web Services: state of the art

What gave life to this work was the aim of trying to understand if and how it would be possible to make Android work with a key technology like Web Services.

We have seen in the first chapter how this implementation of the SOA model has already changed the landscape of Distributed Computing introducing a large number of benefits for developers and users as well as for enterprises. Now we wonder if an Android application is able to access and consume Web Services as this would be a really useful and interesting feature for the new mobile platform.

Of course the first step was verifying if some proper library was available in the SDK, something similar to javax.xml.soap (the java API for implementing SOAP) to be clear. Unfortunately we didn't find anything like this although the Android SDK includes many Java libraries. Clearly the Web Services issue hadn't been officially faced yet and therefore no native solution was provided by Android, at least at the moment.

So we started to browse a number of developers communities in the Web, hoping to find out some useful information. Our problem turned out to be quite common and not trivial at all but some possible solutions had already been suggested and tested.

One of the hints was creating a HTTP connection using the java.net or org.apache.http APIs, available in the SDK, and then manually parsing the xml files by the org.xml.sax, available too. This seemed a sort of "homemade" solution, very expensive with regard to programming effort and time needed, and with no guarantee of success.

Other more reasonable solutions focused on the opportunity of trying to fit some existing library (most of them conceived for Java ME) on Android: the most interesting proposals concerned kXML-RPC and KSOAP.

kXML-RPC is a J2ME implementation of XML-RPC, a protocol that allows remote procedure calling using HTTP as the transport and XML as the encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be

transmitted, processed and returned. We will see KSOAP in Chapter 4.

Nevertheless our task looked still not easy and its outcome uncertain, as several troubles had emerged during many developers' attempts and also some SDK version compatibility problems had come out.

Anyway the current situation was outlined and this was a good starting point.

In chapter 4 we will say which solution we finally chose and why.

Chapter 3

ICAAS: an Interoperable Configurable Architecture for Accessing Sensor networks

In this chapter we will describe a key tool for our case study: the ICAAS framework for accessing and monitoring Wireless Sensor Networks. After saying what a WSN is, we will focus on the main goals and requirements of ICAAS, then describing its features and architecture. Finally we will look at the framework from the client-side, presenting an available J2ME client application.

3.1 Wireless Sensor Networks: some basics

Sensor networks are distributed frameworks for collecting, merging and aggregating data concerning an observed process or phenomenon, independently from its geographical extent. More in detail a Wireless Sensor Network (WSN) is a wireless network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations.

Wireless Sensor Networks are the wireless answer to the issues raised by traditional wired sensor networks, now outdated: WSN, compared to their “ancestors”, give a number of

key benefits, among which easy displacement, natural redundancy, reduced costs and great accuracy. In addition to one or more sensors, each node in a sensor network is typically equipped with a radio transceiver or other wireless communications device, a small microcontroller, and an energy source, usually a battery.

A WSN is normally considered a special wireless ad-hoc network also called "hybrid network" as each sensor supports a multi-hop routing algorithm (several nodes may forward data packets to the base station). This is not very accurate because of several differences existing between a WSN and a common ad-hoc network. However such analysis is beyond our goals.

The general architecture of a WSN is here represented:

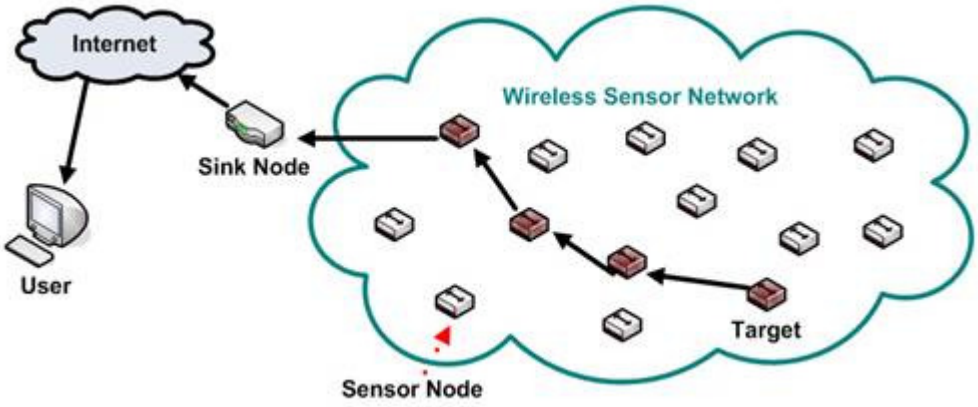

Figure 3.1

The target node is the node whose data are currently requested by the user while the sink node works as an intermediate between the WSN and the user by collecting data from the network and delivering them to the client application.

The development of wireless sensor networks was originally motivated by military needs such as battlefield surveillance but today they are used in many industrial and civilian application areas, including industrial process monitoring and control, machine monitoring, environment and habitat monitoring, medical applications, home and building automation, traffic control.

3.2 The ICAAS project within REMOAM: main goals and requirements

REMOAM (Italian acronym for "Sensor networks for environmental risk monitoring") is a research project pursued by Strago s.r.l. and CINI consortium with the goal of defining next generation systems for environmental monitoring and for on-line evaluation of risks.

More in detail, the project aims at the development of new wireless sensor devices, specifically targeted for monitoring relevant environmental parameters, and of innovative software architectures for gathering and managing the data collected from the sensors.

ICAAS (Interoperable and Configurable Architecture for Accessing Sensor networks) is one of the outcomes of the REMOAM project.

The main features that its designers aimed to achieve for their framework are as follows:

- *Interoperability* among: sensor networks, remote applications for analysis and monitoring, and distributed measurement systems.
- *Configurability* of: sensor networks, data processing, data access by other users.

The functional requirements identified were instead the following ones:

- ✓ **Processing of information coming from sensor networks**
  - o data capture and persistence
  - o data access
- ✓ **Configuration and setup of sensor networks to make them interact with iCAAS**
  - o configuration of network parameters
  - o definition of sensor groups
- ✓ **Customization and configuration of user-specific parameters**
  - o workflow configuration
  - o alarm levels definition

✓ **Users management**

The main non functional requirements are:

- **Interoperability** among sensor networks and applications, and with third-party systems.
- **Extendability** of the architecture.
- **Performance** for data access (requests for recent data should have priority over requests for historical data) and requests traffic (the outgoing traffic should be kept as light as possible in order to improve performance in case of a large number of requests).

3.3 The ICAAS Architecture: interoperability features and software stack

The ICAAS Architecture could be defined "interoperability-driven" as the key feature firstly taken into account when designing the overall framework was interoperability. Indeed, with a top-down approach, it was needed to consider which entities would have to interact and cooperate with the system, in order to identify the externally interfaced software modules.

The main external interactions expected concern:

- *Third Party Systems*: other measurement and data collecting systems that interact with ICAAS for both using data and providing the results of their own processing.
- *Desktop/Mobile Clients*: they run applications for analysis, monitoring, configuration, networks setup.
- *Sensor Networks*: middleware and/or applications running on network gateways collect and send data.
- *Data Base Management System*: the system that hosts and manages the database.

The following figure represents these interoperability features:

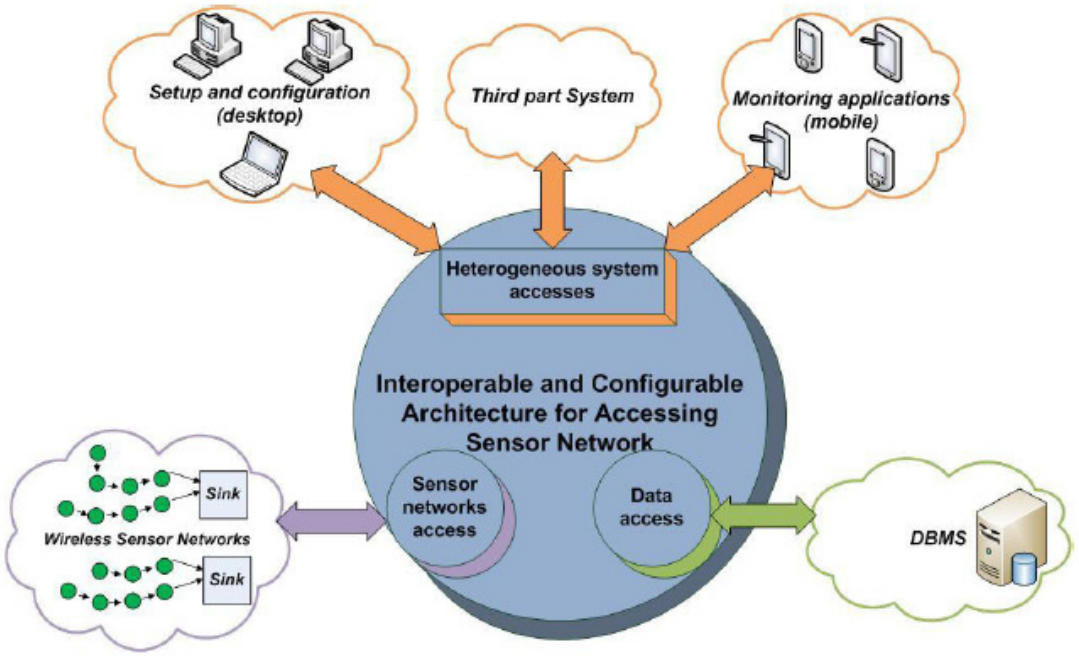

Figure 3.2 – From [15]

This analysis leads to identifying three key software modules externally interfaced:

- **Data access** is the layer enabling communication with the DBMS and therefore with the database, which is, in a sense, the core element of the entire framework, as all the processing depends on it.
- **Sensor network access** is the module allowing communication with sensor networks and is therefore as important as the previous one, because it handles the access to the data source of the processing.
- **Heterogeneous system accesses** is the layer that lets any external software system access ICAAS and use its functionalities, meeting the key interoperability requirement. However this has to be a complex module in order to manage the interaction with very heterogeneous systems. More in detail ICAAS has been

conceived as a ROA-oriented and SOA-oriented architecture at the same time. The ROA (Resource Oriented Architecture) model allows to export data as resources while the SOA model provides functionalities to external applications as services.

In the core of ICAAS have then been identified three more layers underlying the Data Access module:

- **Caching** manages the cache, that is the data structure containing the most recent data: the write-through policy chosen reduces the processing time by avoiding many queries.
- **Filtering** builds customized results focused on users' profiles using as well an optimization mechanism for producing responses.
- **Sessioning** manages users' sessions letting registered users access the ICAAS functionalities.

Here is the overall ICAAS architecture:

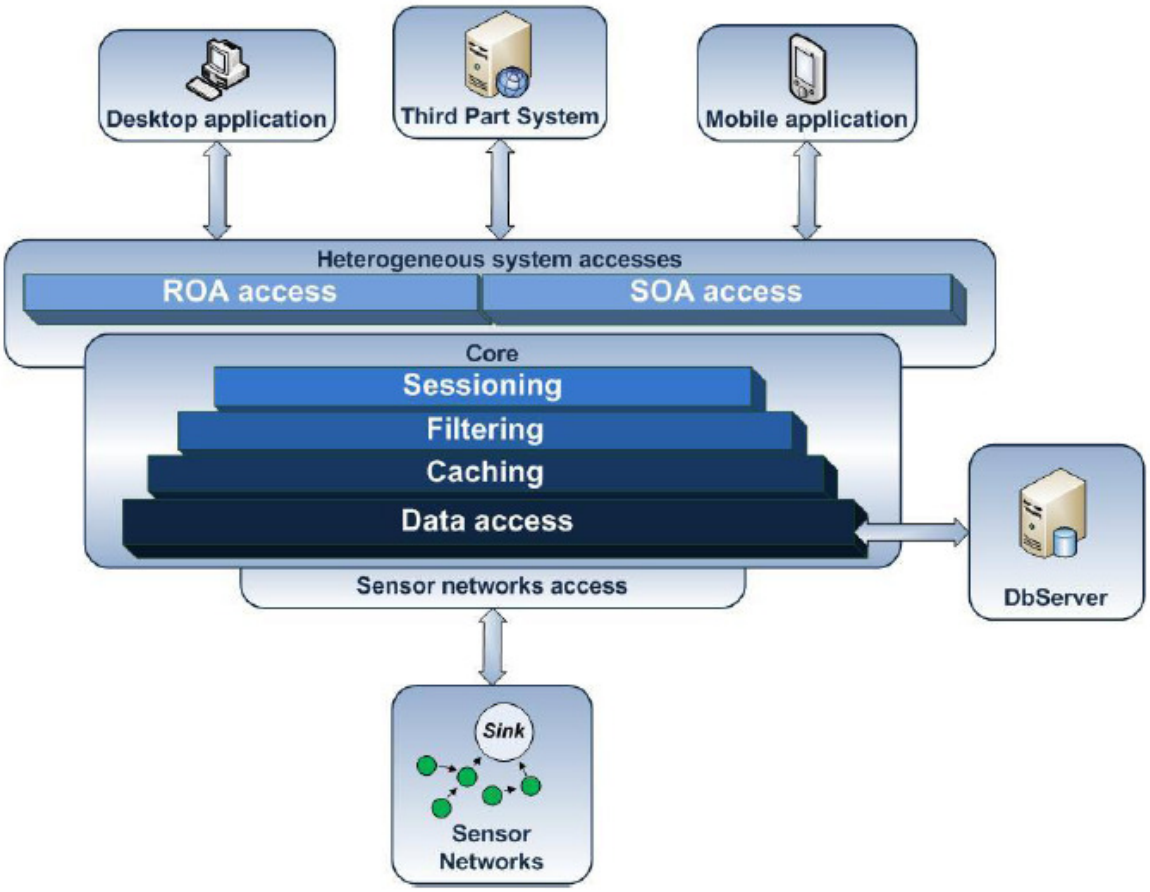

Figure 3.3 - From [15]

3.4 ICAAS as a SOA: Web Services for client applications

As mentioned previously, at the *Heterogeneous System Accesses* level the ICAAS architecture includes a ROA and a SOA module.

Providing *ROA access* implies exporting functionalities as resources: this basically means making them accessible through the Web, using HTTP and URI references. Probably the most popular ROA model is REST (REpresentational State Transfer), entirely based on HTTP and on some key requirements to meet when implementing it.

Now we will focus in more details on the *SOA access* also provided by ICAAS.

We have already seen in Chapter 1 what a Service Oriented Architecture is and how it can be implemented. The Sensor Web Enablement (SWE) standard, proposed by the Open Geospatial Consortium (OGC), provides some specifications and guidelines for

implementing a SOA specifically thought for Sensor Web (sensor networks connected to the Web).

These specifications include:

- **Observations & Measurements** (O&M) - Standard models and XML Schema for encoding observations and measurements from a sensor, both archived and real-time.
- **Sensor Model Language** (SensorML) - Standard models and XML Schema for describing sensors systems and processes associated with sensor observations; provides information needed for discovery of sensors, location of sensor observations, processing of low-level sensor observations, and listing of taskable properties, as well as supports on-demand processing of sensor observations.
- **Transducer Model Language** (TransducerML or TML) - The conceptual model and XML Schema for describing transducers and supporting real-time streaming of data to and from sensor systems.
- **Sensor Observations Service** (SOS) - Standard web service interface for requesting, filtering, and retrieving observations and sensor system information. This is the intermediary between a client and an observation repository or near real-time sensor channel.
- **Sensor Planning Service** (SPS) - Standard web service interface for requesting user-driven acquisitions and observations. This is the intermediary between a client and a sensor collection management environment.
- **Sensor Alert Service** (SAS) - Standard web service interface for publishing and subscribing to alerts from sensors.
- **Web Notification Services** (WNS) - Standard web service interface for asynchronous delivery of messages or alerts from SAS and SPS web services and other elements of service workflows.
- **Sensor Collection Service** (SCS) - Standard web service interface to fetch

observations from a sensor or constellation of sensors. Provides real time or archived observed values. Clients can also obtain information that describes the associated sensors and platforms.

Clearly SWE itself defines some standard services for a Sensor Web–specific SOA as Web Services, so suggesting to adopt them when implementing the architecture. This is what ICAAS did, as a sort of non-standard SWE implementation. The non standardized aspects concern the Sensor Network Access layer, the Configurability requirement, the Optimization of requests and the ROA Access module, available besides SOA Access.
However the standard was followed to a degree so that were implemented as many web service classes as the standard SWE web services.
The interfaces of these classes are shown in the following figure.

Figure 3.4 – From [15]

These web services, whose functions are self-explanatory, implement the SOA Access layer of the ICAAS framework. They can be easily deployed using the Apache Axis *AdminClient* tool with a proper .wsdd file. For more details about this procedure we refer to Appendix A.

3.5 ICAASMobile: a J2ME client for the framework

In order to test the capabilities of ICAAS, its designer also developed a J2ME client accessing the architecture through SOA Access, that is consuming the available web services (seen in the previous paragraph).

This client, called ICAASMobile, implemented using the J2ME APIs, NetBeans Mobility version for the GUI, KSOAP (a SOAP open source mobile-oriented implementation) and the kXml DOM parser, provides a good example of how the ICAAS framework can be exploited, viewing the data received and accessing other functionalities.

We tested it to figure out better how the framework works and how a client for accessing it could be conceived and structured.

This experience turned out to be an useful support for the project we had in mind about Web Services and Android, as we will see in the next Chapter.

Meanwhile let's give an idea of how ICAASMobile looks when running within the SUN emulator, showing a couple of screens from the application. The first one displays the list of the sensors received from the ICAAS Server and updated in real time, with the drop-down menu, while the other one lists the properties of a single sensor.

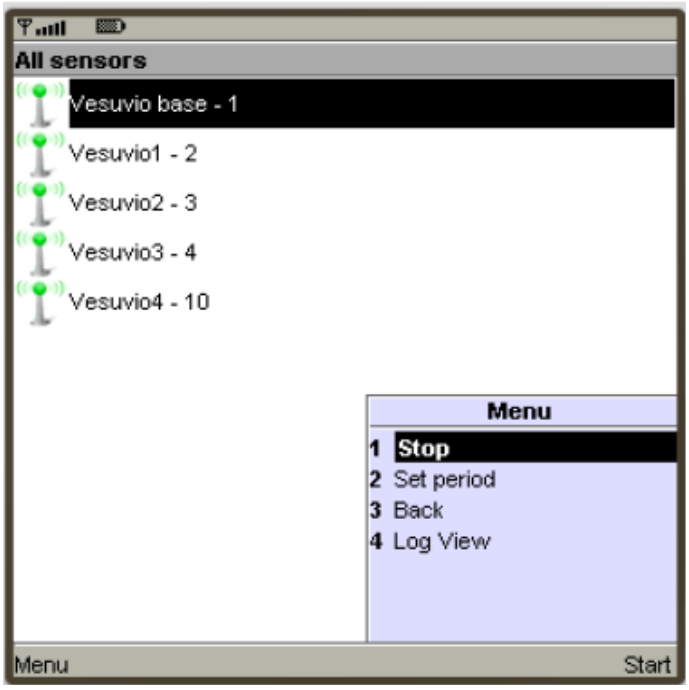

Figure 3.5 – List of the sensors

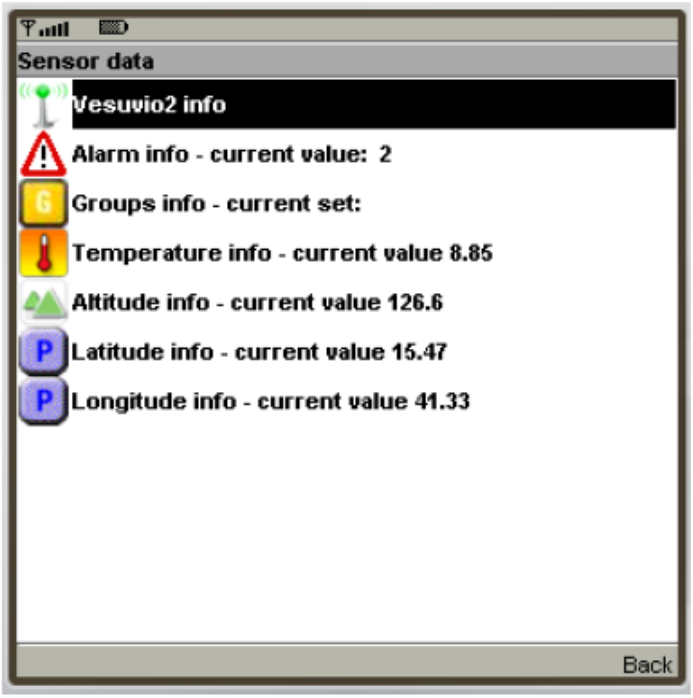

Figure 3.6 – Properties of a single sensor

Chapter 4

Case Study: Analysis, Design and Implementation of an Android client for ICAAS

In this chapter we will describe in details the case study taken into account in our work: the development of a client for using the ICAAS framework, presented in the previous chapter, to be run in an Android mobile environment.

The treatment is divided into three sections matching the steps of a typical software development process: Analysis, Design and Implementation.

In the Analysis part we will start from the J2ME client seen before, that has been reversely engineered in order to identify the requirements of the application to be carried out but also to understand to what degree the available client could be reused and fit on the Android platform, and which parts of the project had to be completely rethought and redesigned for our specific needs. In a few words it will be highlighted to what extent we have followed a porting process and where we have wandered off it, pursuing an *ex novo* traditional development. Then we will list the requirements of the application and some use cases, finally coming to the class and package diagram needed by an object oriented analysis.

In the Design section we will present the forward engineering phase, outlining the validation of the original design and the porting strategies adopted, and then focusing on

the main design issues we have had to deal with when trying to fit some parts of the existing project and conceiving the new components. The final outcome of the object oriented design is, as usual, the class and package diagram.

Finally, in the Implementation section some particularly interesting classes are shown with their source code for giving an idea of how the analysis and design outcomes have been translated into code.

4.1 Analysis

In this section we will firstly list the steps of a typical porting process with the aim of trying to fit them for our case study. Then we will analyze the ICAASMobile J2ME client mentioned in the previous chapter, and describe how it has been reversely engineered in order to understand it and to identify requirements and use cases useful for our purposes. Class and Package Diagram are the outcomes of this Object Oriented Analysis.

4.1.1 ICAASMobile as a starting point: the steps of a typical porting process

As already mentioned, when beginning to work on the idea of developing an Android client for the ICAAS framework in order to test the Google's mobile platform with regard to its capability of accessing and consuming Web Services, we firstly took a look at the available J2ME client.

We got both source code and an executable version of the ICAASMobile client as well as a Java sensor network simulator (for emulating data received from the sensors) and two .jars containing the core classes implementing the framework and the classes implementing the web services. So, after learning how to install the ICAAS software architecture and how to deploy the web services on our computer (we refer to Appendix A for any technical details concerning the setup procedure), we had everything we needed to test the client (and therefore the ICAAS framework, of course) and to figure out how it worked and how it had been conceived, structured and design.

Executing ICAASMobile and trying all its functionalities we found its working quite fulfilling as it did more or less what we needed for our main purpose, that was testing Android Web Services-related capabilities. So we began to believe that probably trying to directly port this J2ME application to Android was not a wrong idea; on the contrary this seemed quite compelling. Indeed this way we could have the chance to experience a Porting process from one of the currently most popular mobile platforms (J2ME) to the latest promising one (Google Android), as well as pursuing our original aim of making Android work with Web Services.

In fact we will see hereafter that a pure porting turned out to be impossible in this case and we had to choose rather a mixed approach, combining a porting process with a traditional development from scratch.

Anyway we will introduce some principles of Porting and then show the steps of a typical porting process to make clearer to what extent we followed them and where and why we diverged from them.

Software porting is the engineering process of transforming an existing application so that the resulting software will execute properly on a new platform. Depending on the nature of the software and the source and destination platforms, this can be as simple as recompiling the code on a new system and verifying that everything works properly, or as complex as redesigning the whole application and rewriting large sections of the code to accommodate the new platform.

Porting has some advantages over *ex novo* development but also some drawbacks, depending on the point of view from which we look at its features.

On one hand, when talking about Porting, we implicitly assume that:

- Specifications are clear
- A detailed prototype in form of the running application is available
- Architecture and design are already there
- Identifying the subsystems is easier, as one can actually see the application running

- Source code is there

On the other hand, one might object that:

- Specifications are generally biased towards the original platform/framework, hence they might actually be tougher for the target platform
- The application may not be well designed and architectured at all
- Original architecture and design may not suite the target platform
- Linking the functional subsystems with their corresponding source files is a difficult task
- Understanding somebody else's code is always difficult
- It is easier to feel lost while looking at all the source code at once

So, although at first sight porting can seem clearly much easier than developing from scratch, as reusing at least part of the code allows to save efforts and time, after a deeper analysis this is not so obvious and in some cases could turn out to be completely false.
Anyway it is important to understand that porting of an application is different from developing it from scratch even just because the original running application is available with all its source code. As a matter of fact, porting can facilitate the development under certain conditions but also introduces many decisional factors in the process, making it more complex. Even if there are several different types of porting (between two different Operating Systems or OS versions, DBMS, languages, frameworks, libraries, development environments, middlewares, or some combination of these circumstances) there are still some common issues to be faced in all of them. That's why adopting appropriate strategies might make the task easier. A good starting point is defining the steps of a standard porting process to use as a guideline, as shown below:

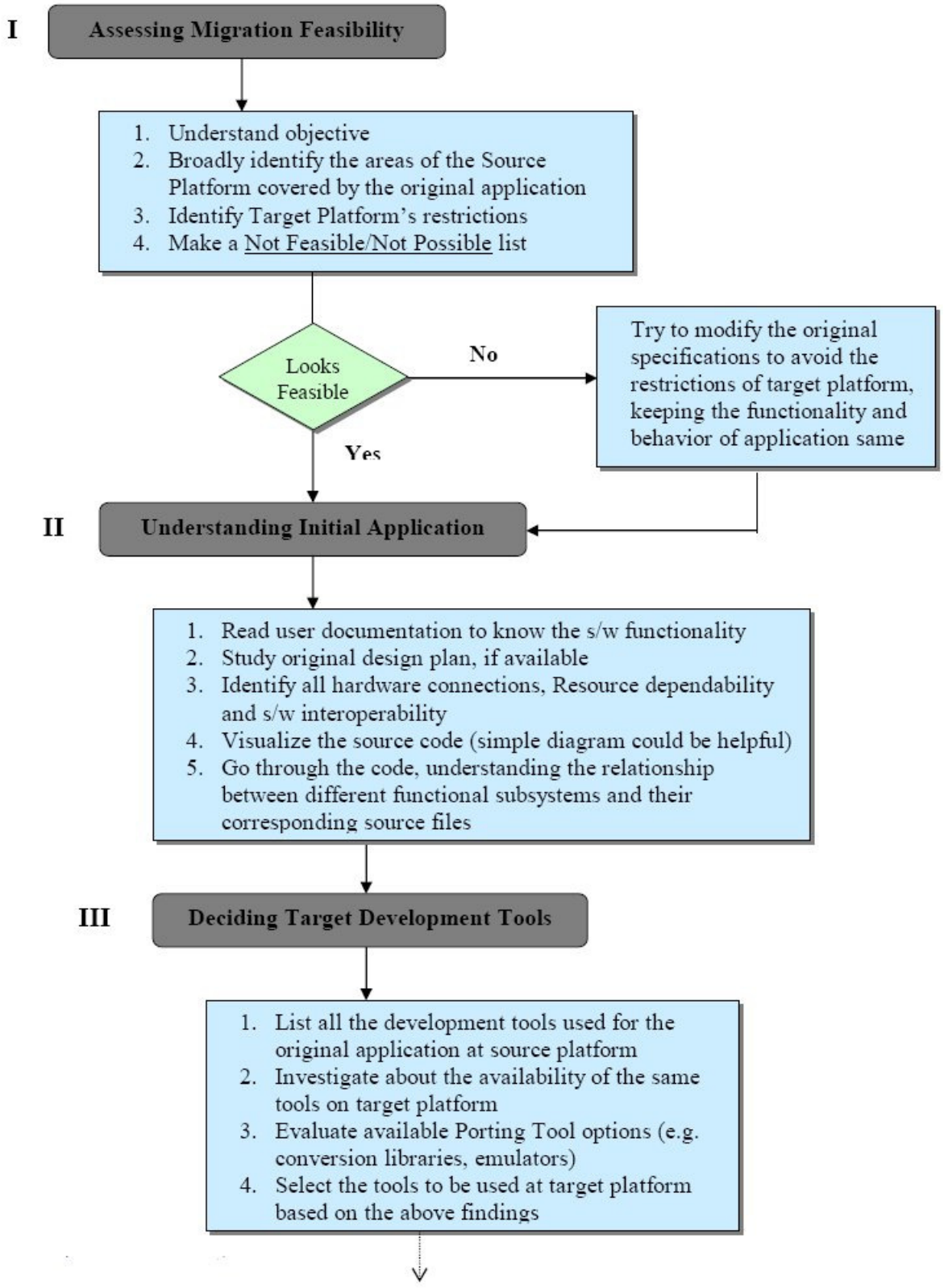
I
Assessing Migration Feasibility
1. Understand objective
2. Broadly identify the areas of the Source Platform covered by the original application
3. Identify Target Platform's restrictions
4. Make a Not Feasible/Not Possible list
Looks Feasible
No
Try to modify the original specifications to avoid the restrictions of target platform, keeping the functionality and behavior of application same
Yes
II
Understanding Initial Application
1. Read user documentation to know the s/w functionality
2. Study original design plan, if available
3. Identify all hardware connections, Resource dependability and s/w interoperability
4. Visualize the source code (simple diagram could be helpful)
5. Go through the code, understanding the relationship between different functional subsystems and their corresponding source files
III
Deciding Target Development Tools
1. List all the development tools used for the original application at source platform
2. Investigate about the availability of the same tools on target platform
3. Evaluate available Porting Tool options (e.g. conversion libraries, emulators)
4. Select the tools to be used at target platform based on the above findings

Figure 4.1

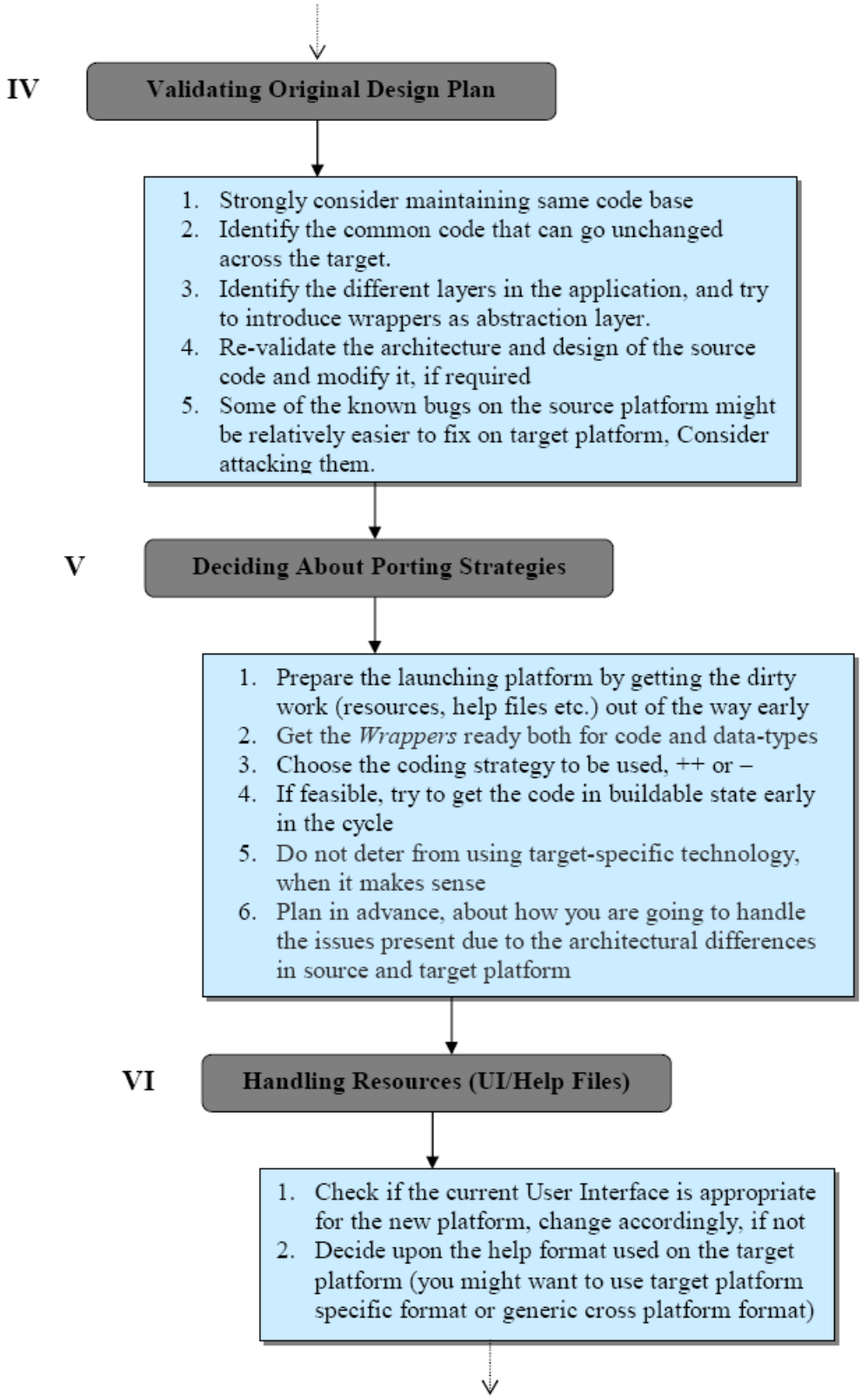
IV
Validating Original Design Plan
1. Strongly consider maintaining same code base
2. Identify the common code that can go unchanged across the target.
3. Identify the different layers in the application, and try to introduce wrappers as abstraction layer.
4. Re-validate the architecture and design of the source code and modify it, if required
5. Some of the known bugs on the source platform might be relatively easier to fix on target platform, Consider attacking them.
V
Deciding About Porting Strategies
1. Prepare the launching platform by getting the dirty work (resources, help files etc.) out of the way early
2. Get the Wrappers ready both for code and data-types
3. Choose the coding strategy to be used, ++ or –
4. If feasible, try to get the code in buildable state early in the cycle
5. Do not deter from using target-specific technology, when it makes sense
6. Plan in advance, about how you are going to handle the issues present due to the architectural differences in source and target platform
VI
Handling Resources (UI/Help Files)
1. Check if the current User Interface is appropriate for the new platform, change accordingly, if not
2. Decide upon the help format used on the target platform (you might want to use target platform specific format or generic cross platform format)

Figure 4.2

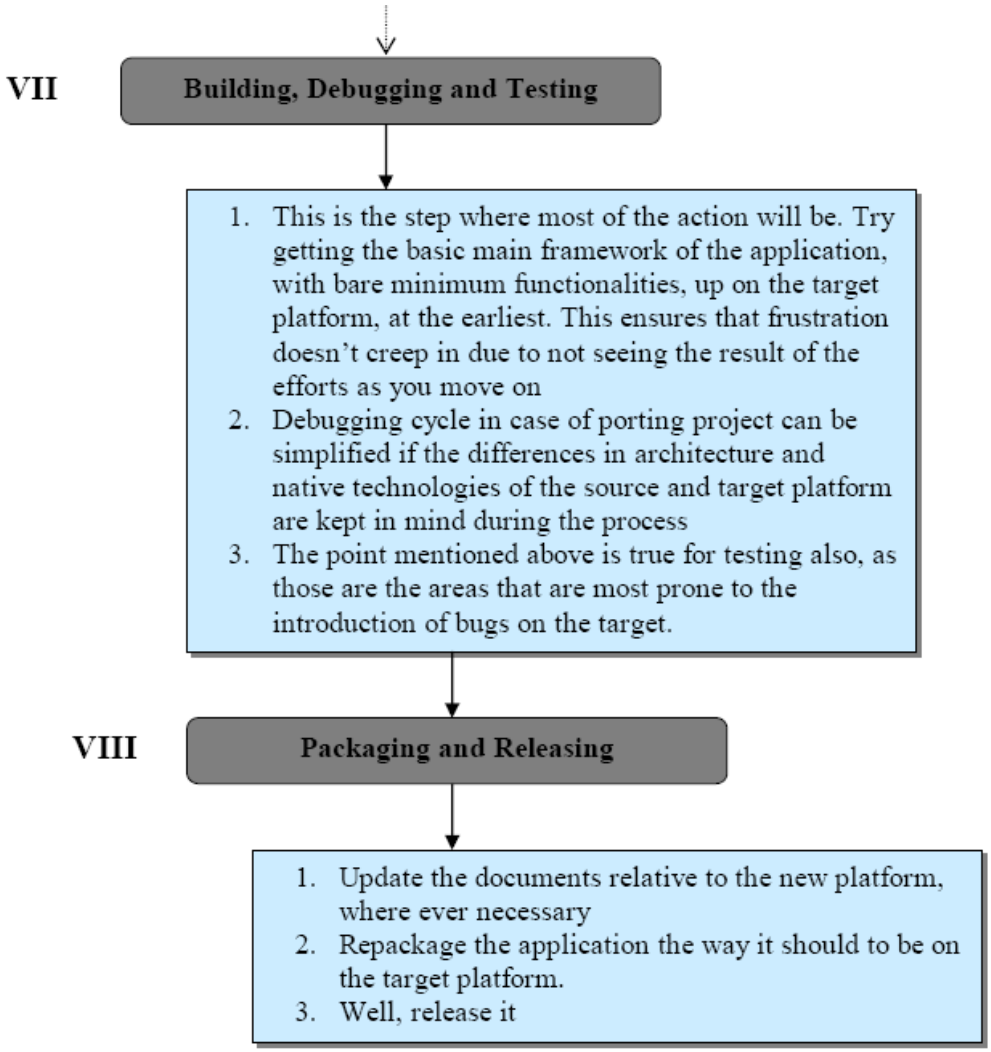

Figure 4.3

As said before, we will not follow step by step a standard porting process but some of the steps just seen will be very useful in the analysis and design phases of our work.

With regard to the first step, preliminary to the all process, as it explains how to assess migration feasibility, which of course can be never taken for granted, we performed it in a quite informal way: by testing the J2ME client we observed it working so learning its goals and functional requirements. After studying the Android platform, we didn't identify in it any restrictions making some function carried out in the original platform unfeasible

in the target one. The only doubt was about the invocation of Web Services but it was removed after finding out the KSOAP solution, as shown afterwards.

Therefore we concluded that we didn't need to modify any original specifications as we were able to keep the same functionalities and behaviour of the existing client.

We will analyze the second main step of the process in the next paragraph.

4.1.2 Reverse engineering of the ICAASMobile client: understanding the initial application

Reverse engineering can be referred to as the "process of analyzing a subject system to create representations of the system at a higher level of abstraction" (Chikofsky) or simply as "going backwards through the development cycle" (Warden), for example following an inverse waterfall model. In practice, two main types of reverse engineering emerge, depending on whether source code of the original software is available or not: in the first case the goal is getting a high-level overview and above all the requirements of the system; in the second case, all efforts are oriented towards discovering one possible source code (as the original one can never be known for sure).

We are clearly in the first case as we have the source code of the original application and we want to figure out its overall design structure and its requirements to be used for the forward engineering process. Such an approach helps us to accomplish at the same time the second step of the porting process: understanding the initial application.

The ICAASMobile J2ME client wasn't very documented as it had been conceived as a simple test application for the ICAAS framework. Moreover we did not even have the original design plan or class diagram. So what we firstly did was installing the framework and the sensor network simulator on our computer (as described in Appendix A) and testing the application with all its functionalities in order to understand its requirements. After that we analyzed the code, trying to get the function of every Java class, the relationships among the classes and the overall structure of the application. This was not a trivial task, as there were many poorly documented classes, divided in four packages and

some of them were really hard to understand as the User Interface had been written using an automatic GUI generator integrated in NetBeans and moreover adopting Java microedition-specific libraries rather than the Java Standard Edition APIs (like Swing). Anyway we made it also with the help of a UML class-diagram tool (OMONDO) integrated in Eclipse.

We will see the results of this reverse-engineering process, that is the software requirements specification, the use cases diagram and the package and class diagram, in the following section.

It's important to note that the Software Requirements Specification we got by testing and analysing the original application will be taken as the SRS for the new application to develop for Android. That's why we will sometimes mention the Google platform although the requirements described below exactly match the ones of the J2ME application we started from.

4.2 Software Requirements Specification and Use cases

For the sake of brevity we will follow the IEEE 830-1998 Standard only as regards the Specific Requirements part.

Anyway let's give an overall description of the application to be developed for Android. We want to make a mobile client for accessing sensor networks through the web services provided by the ICAAS framework in order to monitor the data coming from a sensor network in real time. Each available sensor network should be accessible by any user registered to monitor it, logging in from an Android device. The users are assumed to have a basic computer experience but a good knowledge of the application domain and therefore of the meaning of the data viewed, as the software is mainly thought for organizations and authorities interested to the monitoring of environmental processes and phenomena.

Specific Requirements

1. External interface requirements

1.1 User interfaces

A simple GUI with buttons and pull-down menus will help the user to interact with the application.

1.2 Hardware interfaces

No direct interaction with any hardware device is required.

1.3 Software interfaces

The application has to interact with the ICAAS framework through the provided web services.

1.4 Communications interfaces

The application will rely on the network connection available on the device to communicate with the ICAAS server (where web services will be deployed).

2. Functional requirements

2.1 Change Settings

2.1.1 Description

The user views and changes the settings concerning server URL, username and password to access the ICAAS framework.

2.1.2 Input

New URL and/or username and/or password.

2.1.3 Processing

The old settings are turned into the new ones.

2.1.4 Output

The new settings.

2.2 Login

2.2.1 Description

The user logs in to the ICAAS server.

2.2.2 Input

The settings previously saved.

2.2.3 Processing

The user's settings are sent to the server and a connection is started.

2.2.4 Output

The screen to view the list of the sensors in the current network.

2.3 View Sensors

2.3.1 Description

The user gets a static view of the sensors of the network he is interested in.

2.3.2 Input

A proper control.

2.3.3 Processing

The static list of the sensors is requested to the server to be displayed.

2.3.4 Output

A static view of the sensors of the current network.

2.4 Reset Sensors View

2.4.1 Description

The user resets the screen displaying the static list of sensors

2.4.2 Input

A proper control.

2.4.3 Processing

The screen is reset.

2.4.4 Output

Blank screen.

2.5 Set Period

2.5.1 Description

The user sets the observation period he is interested in.

2.5.2 Input

The new value for the observation period.

2.5.3 Processing

The observation period is set to the new value.

2.5.4 Output

The new value of the observation period.

2.6 View Log

2.6.1 Description

The user gets a dynamic view of the sensors in the current network, that is the log of their updates in real time.

2.6.2 Input

A proper control.

2.6.3 Processing

The dynamic view of the sensors is made to be displayed.

2.6.4 Output

A dynamic log view of the sensors of the current network.

2.7 View Single Sensor

2.7.1 Description

The user gets a view of the properties of a single sensor in the current network.

2.7.2 Input

The sensor to be viewed in details.

2.7.3 Processing

The properties of the selected sensor are extracted to be displayed.

2.7.4 Output

A screen showing the properties of the selected sensor and their values.

2.8 View Single Property

2.8.1 Description

The user gets a view of a single property of the current sensor in details.

2.8.2 Input

The property to be viewed in details.

2.8.3 Processing

The details of the selected property are extracted to be displayed.

2.8.4 Output

A screen showing the details of the selected property.

3. Performance requirements

Every processing should take a reasonable time, let's say a few seconds and in any case less than one minute, also taking into account the waiting for network connections.

4. Design constraints

No one.

5. Software system attributes

5.1 Reliability

The software system should be reliable enough to let users permanently or occasionally monitor the sensor networks they are registered for in any moment without too long delays.

5.2 Availability

As a consequence of the reliability requirement the software system should be always available.

5.3 Security

The software system handles confidential information concerning user accounts even if it does not manage user registration but only login.

5.4 Portability

The software system should work properly on any mobile device running the required platform or operating system (in this case Android).

The following Use Cases Diagram schematically summarizes the functional requirements of the application.

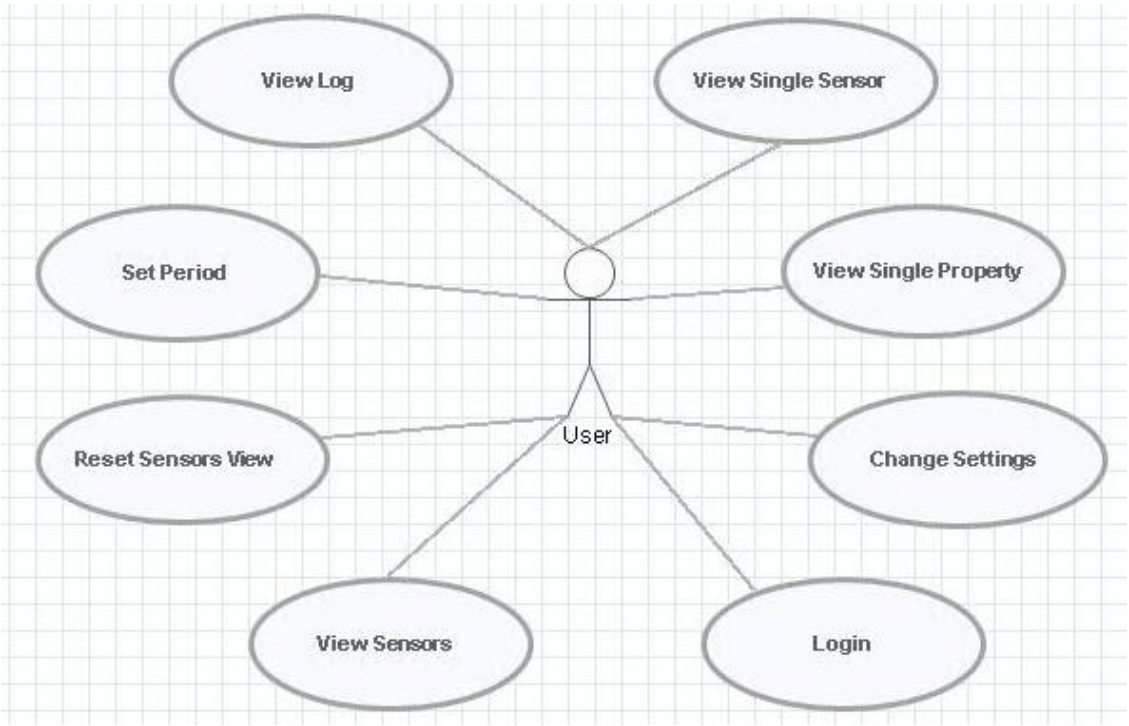

Figure 4.4

4.3 Object Oriented Analysis: Package Diagram and Class Diagram

Here is the package diagram of the original application, with the usage dependencies indicated by arrows going from user package to used package. Clearly the *client* package uses the other ones as utility collections: *mvc* provides classes for building the GUI, *service* handles connections to the server while *type* provides custom types such as Sensor, Property and Net as well as some metadata and xml utilities.

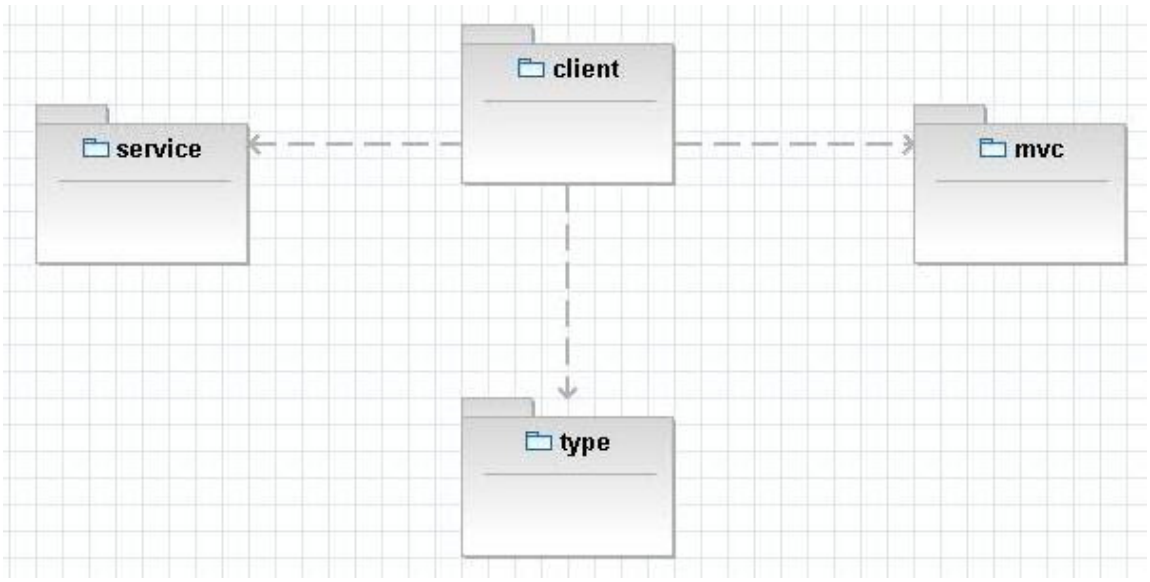

Figure 4.5 – Package Diagram of DroidICAASMobile

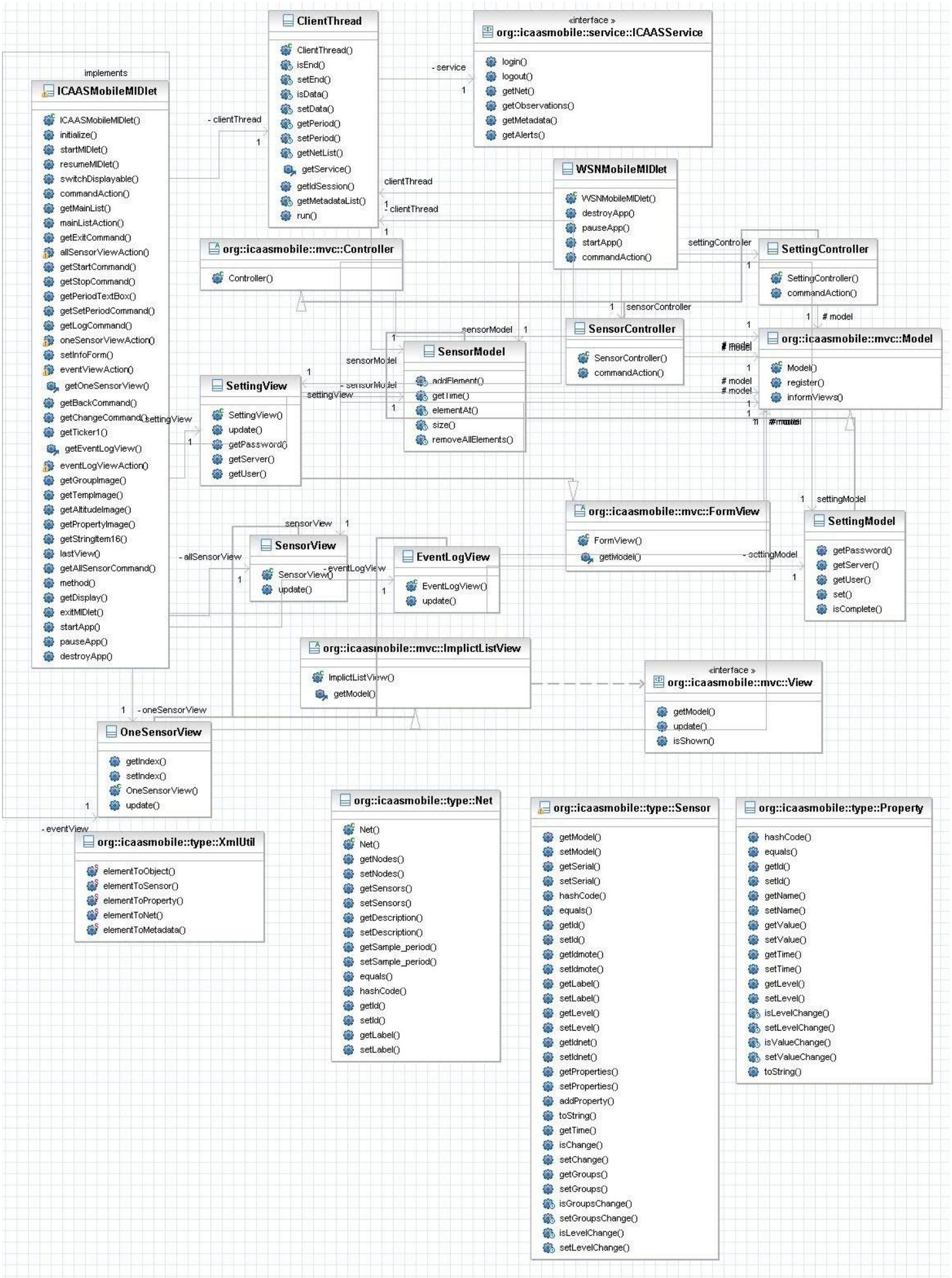
ClientThread
ClientThread()
isEnd()
setEnd()
isData()
setData()
getPeriod()
setPeriod()
getNetList()
getService()
getIdSession()
getMetadataList()
run()
«interface»
org::icaasmobile::service::ICAASService
login()
logout()
getNet()
getObservations()
getMetadata()
getAlerts()
- service
1
implements
ICAASMobileMIDlet
ICAASMobileMIDlet()
initialize()
startMIDlet()
resumeMIDlet()
switchDisplayable()
commandAction()
getMainList()
mainListAction()
getExitCommand()
allSensorViewAction()
getStartCommand()
getStopCommand()
getPeriodTextBox()
getSetPeriodCommand()
getLogCommand()
oneSensorViewAction()
setInfoForm()
eventViewAction()
getOneSensorView()
getBackCommand()
getChangeCommand()
getTicker1()
getEventLogView()
eventLogViewAction()
getGroupImage()
getTempImage()
getAltitudeImage()
getPropertyImage()
getStringItem16()
lastView()
getAllSensorCommand()
method()
getDisplay()
exitMIDlet()
startApp()
pauseApp()
destroyApp()
- clientThread
1
clientThread
1
- clientThread
1
WSNMobileMIDlet
WSNMobileMIDlet()
destroyApp()
pauseApp()
startApp()
commandAction()
settingController
org::icaasmobile::mvc::Controller
Controller()
SettingController
SettingController()
commandAction()
1
sensorController
1
SensorController
SensorController()
commandAction()
1
model
org::icaasmobile::mvc::Model
Model()
register()
informViews()
sensorModel
1
SensorModel
addElement()
getTime()
elementAt()
size()
removeAllElements()
sensorModel
1
SettingView
SettingView()
update()
getPassword()
getServer()
getUser()
settingView
- sensorModel
settingView
1
org::icaasmobile::mvc::FormView
FormView()
getModel()
1
settingModel
SettingModel
getPassword()
getServer()
getUser()
set()
isComplete()
- settingModel
1
sensorView
1
SensorView
SensorView()
update()
- allSensorView
1
- eventLogView
EventLogView
EventLogView()
update()
1
org::icaasmobile::mvc::ImplictListView
ImplictListView()
getModel()
«interface»
org::icaasmobile::mvc::View
getModel()
update()
isShown()
1
- oneSensorView
OneSensorView
getIndex()
setIndex()
OneSensorView()
update()
1
- eventView
org::icaasmobile::type::XmlUtil
elementToObject()
elementToSensor()
elementToProperty()
elementToNet()
elementToMetadata()
org::icaasmobile::type::Net
Net()
Net()
getNodes()
setNodes()
getSensors()
setSensors()
getDescription()
setDescription()
getSample_period()
setSample_period()
equals()
hashCode()
getId()
setId()
getLabel()
setLabel()
org::icaasmobile::type::Sensor
getModel()
setModel()
getSerial()
setSerial()
hashCode()
equals()
getId()
setId()
getIdmote()
setIdmote()
getLabel()
setLabel()
getLevel()
setLevel()
getIdnet()
setIdnet()
getProperties()
setProperties()
addProperty()
toString()
getTime()
isChange()
setChange()
getGroups()
setGroups()
isGroupsChange()
setGroupsChange()
isLevelChange()
setLevelChange()
org::icaasmobile::type::Property
hashCode()
equals()
getId()
setId()
getName()
setName()
getValue()
setValue()
getTime()
setTime()
getLevel()
setLevel()
isLevelChange()
setLevelChange()
isValueChange()
setValueChange()
toString()

Figure 4.6

The overall Class Diagram represented above can give an idea of the complex structure of the original application, conceived following the Model-View-Controller pattern, as we will see better in the Design section. Now we will show some more detailed partial diagrams that can be helpful to understand the application architecture and the relationships among classes. Here is a hierarchical layout showing how the MVC pattern is used.

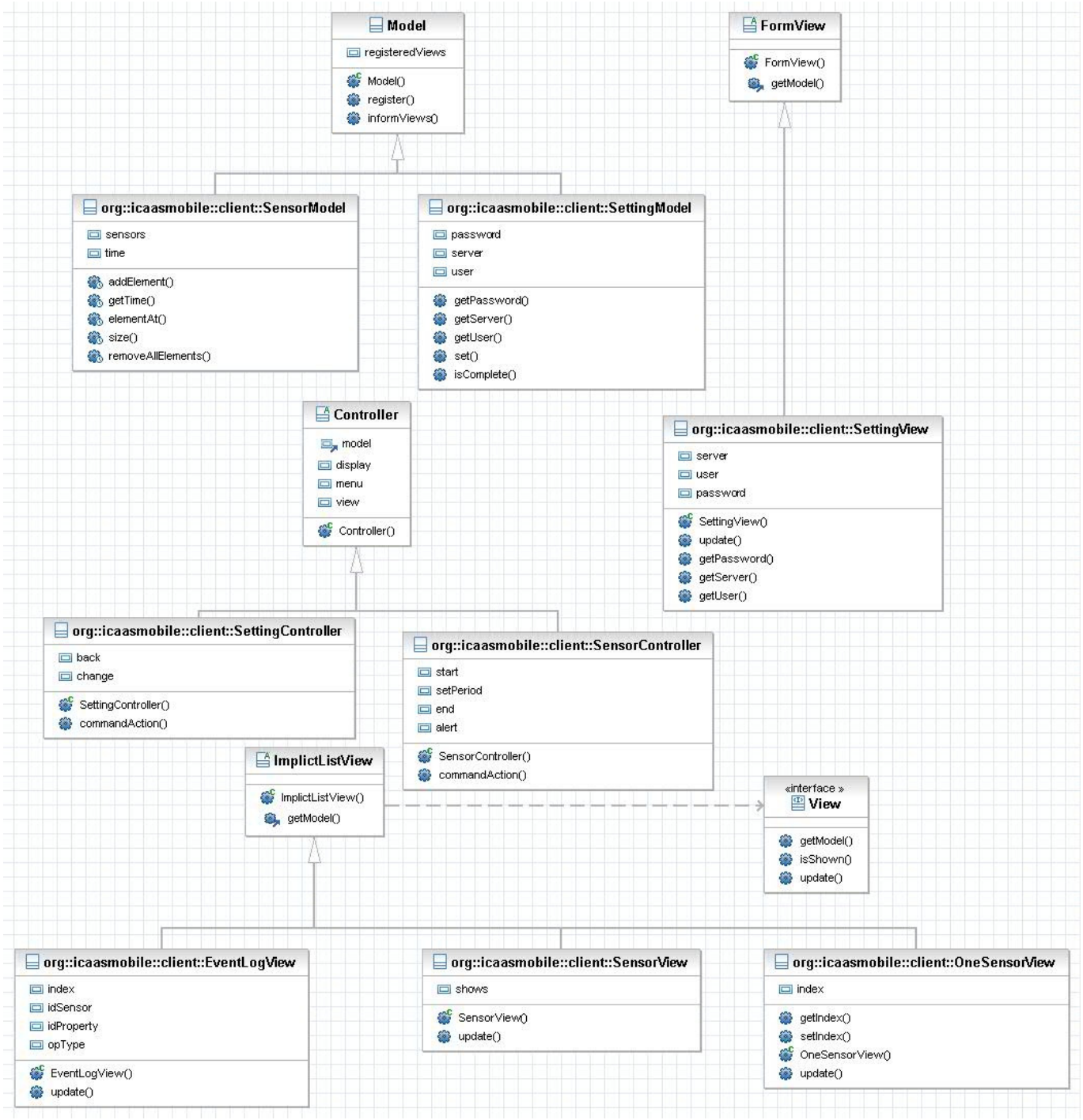

Figure 4.7

Here is instead represented the part of the application that manages the connections with the server, accomplished by the ClientThread Java thread, responsible for sending requests to, and receiving responses from, the Web Services running on the ICAAS Server, through an RPC-based client-side stub. The data so achieved are then passed by the Thread to the other classes that need them.

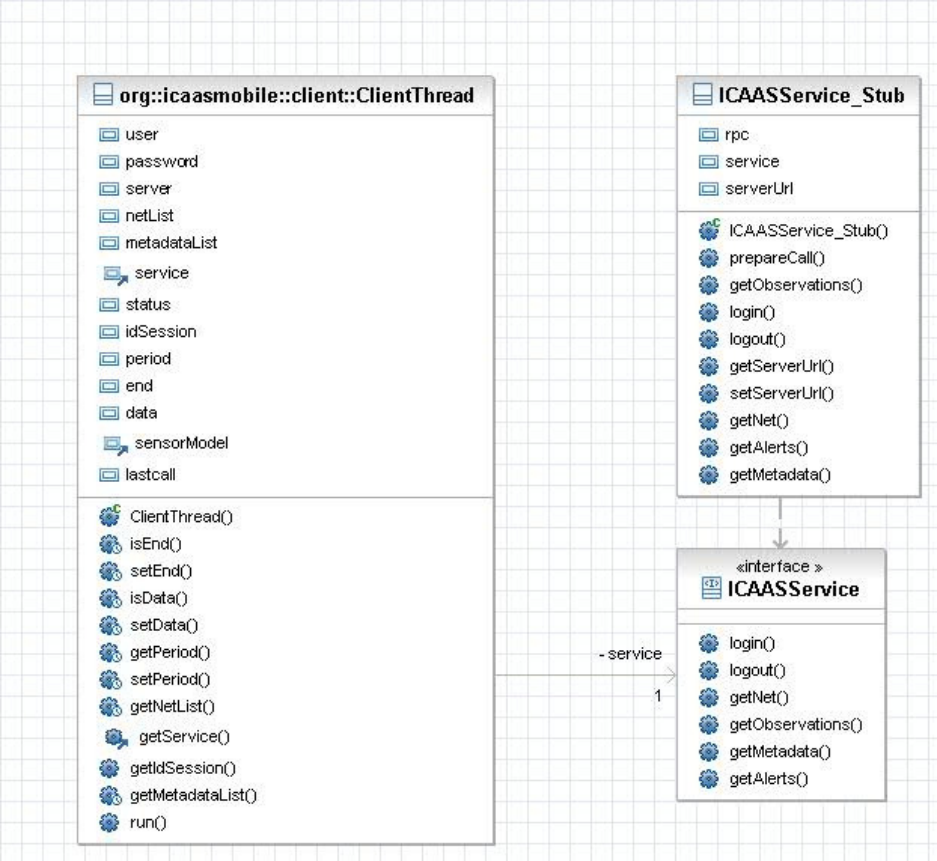

Figure 4.8

4.4 Design

In this section we will use the outcomes of the analysis phase to design the application we want to develop for the Android platform that we will call *DroidICAASMobile*.

Knowing the specific software requirements and the package and class diagram of the original application, we are now able to develop our target application. We will try to reuse the available code as far as it is possible, following the steps of a typical porting process (as shown before), while we will entirely redesign the parts of the application that

need it to work properly on the target platform. Therefore, after describing how we kept on performing the porting process started in the analysis phase, we will particularly focus on the main design challenges we had to deal with. As in every object-oriented design process, we will finally show the package and class diagram of our application.

4.4.1 Forward engineering of DroidICAASMobile: Validating the Original Design Plan and Deciding about Porting Strategies

When testing and analyzing the original application we didn't know its design plan but we got it through a reverse engineering process. Anyway we found the application properly working and not affected by any bug, and its overall architecture well designed, although quite complex to understand, as it strongly separates user interface and business logic (through the MVC pattern) as well as network communication management (thanks to the stub pattern). So we decided to keep the same code base or rather to reuse as much code as possible at least as regards the business logic of the application. The only part of the business logic that immediately raised several doubts about its adaptability to the target platform was the stub, which used the KSOAP 1.1 library for accessing the ICAAS web services: its working with Android was anything but granted. Hereafter we will see how we dealt with this issue. With regard to the Graphic User Interface, it was clear that it had to be completely redesigned as it used J2ME-specific APIs and moreover Android has its own special structures and libraries for developing GUIs, as explained in Chapter 2. That's why we didn't worry about the chaotic code of some classes implementing the GUI, clearly generated by an automatic drawing tool.

About the layers of the application they basically matched the packages it was divided into: the *client* package implemented the user interface, *service* managed the connections to the server (through the web services) and *type* provided some utility functions. The *mvc* was the one we probably didn't need any more as it provided some GUI utilities.

After validating the original design plan we had to choose a porting strategy: we picked the ++ (*plus-plus*) one, gradually adding the existing functions to our project in order to

figure out which parts of the original code could work on the target platform without any change and could be therefore taken as a starting point for the new application.

Of course we initially tried to test the business logic (leaving the GUI out) on Android, using also some test classes written on purpose, with a particular focus on the key function of invoking and consuming the ICAAS web services, performed by the stub class.

The first problem we had to solve was making it work on the new platform, as we will see in the next paragraph. After that we will describe the GUI design process of *DroidICAASMobile*.

4.5 Invoking Web Services in Android: the KSOAP2 solution

Before delving into the KSOAP2 solution adopted for accessing Web Services on Android it can be helpful taking a look at the conceptual model implemented by the original application as regards Web Services invocation, that we decided to keep, only modifying its implementation.

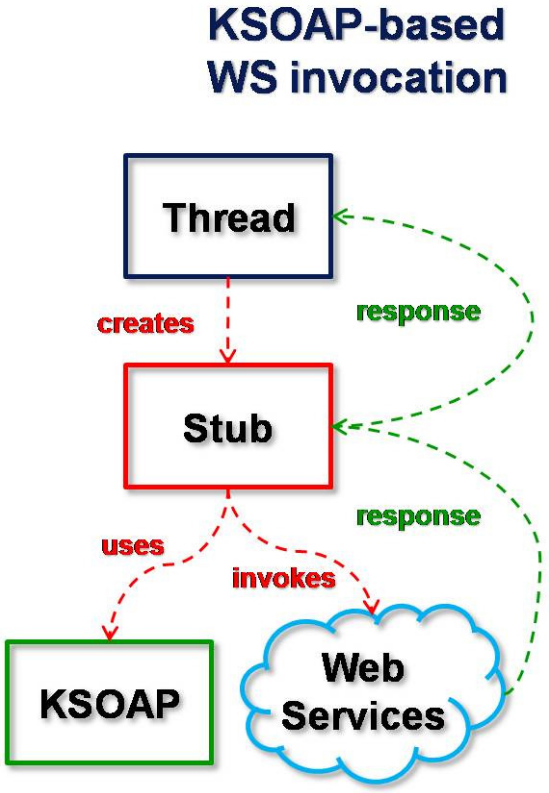

Figure 4.9

The thread (ClientThread in the class diagram) is responsible, as mentioned before, for sending requests to and receiving responses to the Web Services running on ICAAS Server. For this purpose it calls the Stub, that actually invokes the Web Services, using the methods provided by the KSOAP API.

The thread has been completely reused without any significant changes, while the stub has needed a code restructuring, as we will see hereafter. However the model adopted is unchanged.

In Chapter 2 we have already examined the state of the art about Android and Web Services. Failing a native API-level method for invoking web services on the Google mobile platform, two solutions emerged as the most popular among Android developers (leaving out the *home-made* ones): kXML-RPC and KSOAP.

Although the first one, according to someone, promised higher performances being lighter, the latter seemed to be much more common and tested on several mobile platforms (partially also on Android) and therefore more reliable. Moreover KSOAP turned out to be the same solution chosen for developing the original J2ME client so that it could make possible trying to reuse most of the code implementing web services invocation and consuming as well as the remainder of the business logic. However, after studying some developers' experience, we were still not sure that it would work properly with Android or rather we knew that in all likelihood we would have to change something in the code also because the KSOAP library used in the original application was the old 1.1 version rather than the latest KSOAP2.

Anyway KSOAP2 was our solution for dealing with the web services issue on Android.

4.5.1 KSOAP: a SOAP open source client library for mobile Java-based devices

Before describing in more details how we used this solution, let's briefly say what KSOAP

is. KSOAP is an open source SOAP web service client library for constrained Java environments such as Applets or J2ME applications. Therefore KSOAP was explicitly thought for mobile Java-based devices, although its official documentation warns that the overhead it introduces may be problematic for mobile devices.

KSOAP 2 is a complete redesign of KSOAP 1, that is now deprecated. Some important changes are:

- Structure cleaned up
- kSOAP2 has improved support for literal encoding
- SOAP Serialization support is now optional and contained in a separate package
- Several separate classes have been integrated into the class SoapSerializationEnvelope, providing SOAP serialization support. SoapSerializationEnvelope extends the base class SoapEnvelope.
- A dotNet flag can be used to switch the SoapSerializationEnvelope from standard behaviour to the namespace handling that seems to be default in .NET

The KSOAP2 API consists of four packages:

- **org.ksoap2** contains basic classes required for handling SOAP Envelopes and literal XML content.
- **org.ksoap2.serialization** supports the Soap Serialization specification.
- **org.ksoap2.servlet**
- **org.ksoap2.transport** mainly manages HTTP connections to transport SOAP calls.

4.5.2 Rewriting the ICAAS Stub: an example of code restructuring

In order to use the KSOAP2 API instead of KSOAP 1, now deprecated, we had to completely restructure the ICAAS Stub, changing many method calls so that they could work with the new library. As KSOAP documentation warns, the porting process from

KSOAP1 to KSOAP2, granting the benefits of a definitely improved API, requires some additional effort. However this allowed us to reuse most of the stub class and therefore all the business logic depending on it, so saving time and avoiding to redo what was already available and working, as the principle of code reuse, typical of object-oriented design, recommends. Moreover the KSOAP2 API didn't require any change in the HTTPTransport class whereas developers adopting the KSOAP1 library had to properly modify the HTTPTransport and ServiceConnection classes of the API.

For a more detailed analysis of the code restructuring performed we refer to the code of the original Stub and of the modified one, shown below.

In yellow we highlighted the parts of code that have been modified in the new stub compared to the old one. First, we use a new class for creating the Transport service: HttpTransportSE instead of HttpTransport (only working with J2ME). Then, in the prepareCall method, we set the Soap Action to be performed and the new Soap Object in a different way. Finally, when reading the Input Stream we cannot get the Bytes Array directly by call (rpc object) on the service object as we need to get the response through a proper envelope. This different call is present in every method that needs to read an Input Stream.

In green we highlighted the parts of code we had to add in the restructured stub to make it work with KSOAP2. First we create SoapSerializationEnvelope object that holds call and response of the Web Services invocation. Then we pass a SoapObject, to which have been previously added the right properties, to the envelope, to set the request we want it to hold. Finally we invoke the call method on the transport service created, passing to it the envelope and the String indicating the Soap Action we want to perform. At this point the response can be got through the envelope. This scheme is repeated more or less the same way by all methods that invoke Web Services.

No changes were needed as regards the xml parsing.

Original Stub

```
package org.icaasmobile.service;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
import org.kobjects.serialization.PropertyInfo;
import org.ksoap.SoapObject;
import org.ksoap.transport.HttpTransport;
import org.kxml2.io.KXmlParser;
import org.kxml2.kdom.Document;
import org.xmlpull.v1.XmlPullParserException;

public class ICAASService_Stub implements ICAASService {

	private SoapObject rpc;
	private HttpTransport service = new HttpTransport();
	private String serverUrl;


	public ICAASService_Stub(String serverUrl) {
		this.serverUrl = serverUrl;
	}

	void prepareCall(String serviceName,String soapAction) {
		service.setUrl(serverUrl+serviceName);
		service.setSoapAction(soapAction);
		rpc=new SoapObject(serviceName,soapAction);
	}

	public Document getObservations(String idSession, long time) throws
IOException {
		prepareCall("iCAASObservationService","getObservations");

		rpc.addProperty("idSession",idSession);
		rpc.addProperty(new PropertyInfo("time",new Long(time)), new
Long(time));

		Reader reader=new InputStreamReader(new
ByteArrayInputStream(((String)service.call(rpc)).getBytes()));
		KXmlParser parser=new KXmlParser();
		Document doc = null;
		try {
			parser.setInput(reader);
			doc=new Document();
			doc.parse(parser);
		} catch (XmlPullParserException e) {
			e.printStackTrace();
		}

		return doc;
	}

	public String login(String name, String password) throws IOException
{
		prepareCall("iCAASNotificationService","login");
		rpc.addProperty("name",name);
```

```
            rpc.addProperty("password",password);
            String str=(String) service.call(rpc);
            return str;
      }

      public String logout(String idSession) throws IOException {
            prepareCall("iCAASNotificationService","logout");
            rpc.addProperty("idSession",idSession);
            return (String) service.call(rpc);
      }


      public String getServerUrl() {
            return serverUrl;
      }


      public void setServerUrl(String serverUrl) {
            this.serverUrl = serverUrl;
            service.setUrl(serverUrl);
      }

      public Document getNet(String idSession,int idNet) throws
IOException {
            prepareCall("iCAASCollectionService","getNet");
            rpc.addProperty("idSession",idSession);
            rpc.addProperty(new PropertyInfo("idNet",new Integer(idNet)),
new Integer(idNet));


            Reader reader=new InputStreamReader(new
ByteArrayInputStream(((String)service.call(rpc)).getBytes()));
            KXmlParser parser=new KXmlParser();
            Document doc = null;
            try {
                  parser.setInput(reader);
                  doc=new Document();
                  doc.parse(parser);
            } catch (XmlPullParserException e) {
                  e.printStackTrace();
            }

            return doc;
      }

    public String getAlerts(String idSession, int idsensor, int
idproperty) throws IOException {
            prepareCall("iCAASAlertService","getAlerts");
            rpc.addProperty("idSession",idSession);
            rpc.addProperty(new PropertyInfo("idsensor",new
Integer(idsensor)), new Integer(idsensor));
            rpc.addProperty(new PropertyInfo("idproperty",new
Integer(idproperty)), new Integer(idproperty));

            return (String)service.call(rpc);
    }

    public Document getMetadata(String idSession) throws IOException {
        prepareCall("iCAASCollectionService","getMetadata");
        rpc.addProperty("idSession",idSession);
```

```
        Reader reader=new InputStreamReader(new
ByteArrayInputStream(((String)service.call(rpc)).getBytes()));
        KXmlParser parser=new KXmlParser();
        Document doc = null;
        try {
                parser.setInput(reader);
                doc=new Document();
                doc.parse(parser);
        } catch (XmlPullParserException e) {
                e.printStackTrace();
        }

        return doc;
    }
}
```

Restructured Stub

```
package vitodaniele.DroidICAASMobile;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
import org.ksoap2.*;
import org.ksoap2.serialization.SoapObject;
import org.ksoap2.serialization.SoapSerializationEnvelope;
import org.ksoap2.transport.HttpTransportSE;
import org.kxml2.io.KXmlParser;
import org.kxml2.kdom.Document;
import org.xmlpull.v1.XmlPullParserException;

public class ICAASService_Stub implements ICAASService {

     private SoapObject rpc;
     private SoapSerializationEnvelope envelope;
     private String action;
     private HttpTransportSE service= new HttpTransportSE(null);
     private String serverUrl;
     private static final String NAMESPACE = null;


     public ICAASService_Stub(String serverUrl) {
           this.serverUrl = serverUrl;
     }

     void prepareCall(String serviceName,String soapAction) {

           service.setUrl(serverUrl+serviceName);
           this.action=soapAction;
           rpc=new SoapObject(NAMESPACE,soapAction);
     }

     public Document getObservations(String idSession, long time) throws
IOException {
           prepareCall("iCAASObservationService","getObservations");

           rpc.addProperty("idSession",idSession);
```

```
            rpc.addProperty("time",new Long(time));
            envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
            envelope.setOutputSoapObject(rpc);
            try {
                  service.call(action,envelope);
            } catch (XmlPullParserException e1) {
                  e1.printStackTrace();
            }

            Reader reader=new InputStreamReader(new
ByteArrayInputStream(((String)envelope.getResponse()).getBytes()));
            KXmlParser parser=new KXmlParser();
            Document doc = null;
            try {
                  parser.setInput(reader);
                  doc=new Document();
                  doc.parse(parser);
            } catch (XmlPullParserException e) {
                  e.printStackTrace();
            }

            return doc;
      }

      public String login(String name, String password) throws IOException
{
                  String str;
             prepareCall("iCAASNotificationService","login");
             rpc=new SoapObject(NAMESPACE,"login");
             rpc.addProperty("name",name);
             rpc.addProperty("password",password);
             envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
            envelope.setOutputSoapObject(rpc);
             try {
                  service.call(action,envelope);
                  Object result = envelope.getResponse();
                  str=result.toString();
            } catch (XmlPullParserException e1) {
                  e1.printStackTrace();
                  str="error";
            }

             return str;
      }

      public String logout(String idSession) throws IOException {
            prepareCall("iCAASNotificationService","logout");
            rpc.addProperty("idSession",idSession);
            envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
            envelope.setOutputSoapObject(rpc);
         try {
                  service.call(action,envelope);
            } catch (XmlPullParserException e1) {
                  e1.printStackTrace();
            }
          String str=(String) envelope.getResponse();
          return str;
      }

      public String getServerUrl() {
            return serverUrl;
```

```
	}

	public void setServerUrl(String serverUrl) {
		this.serverUrl = serverUrl;
		service.setUrl(serverUrl);
	}


	public Document getNet(String idSession,int idNet) throws
IOException {
		prepareCall("iCAASCollectionService","getNet");
		rpc.addProperty("idSession",idSession);
		rpc.addProperty("idNet",new Integer(idNet));
		envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
		envelope.setOutputSoapObject(rpc);
		try {
			service.call(action,envelope);
		} catch (XmlPullParserException e1) {
			e1.printStackTrace();
		}

		Reader reader=new InputStreamReader(new
ByteArrayInputStream(((String)envelope.getResponse()).getBytes()));

		KXmlParser parser=new KXmlParser();
		Document doc = null;
		try {
			parser.setInput(reader);
			doc=new Document();
			doc.parse(parser);
		} catch (XmlPullParserException e) {
			e.printStackTrace();
		}
		return doc;
	}

    public String getAlerts(String idSession, int idsensor, int
idproperty) throws IOException {
		prepareCall("iCAASAlertService","getAlerts");
		rpc.addProperty("idSession",idSession);
		rpc.addProperty("idsensor",new Integer(idsensor));
		rpc.addProperty("idproperty",new Integer(idproperty));

		envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
		envelope.setOutputSoapObject(rpc);
		try {
			service.call(action,envelope);
		} catch (XmlPullParserException e1) {
				e1.printStackTrace();
		}
		String str=(String) envelope.bodyIn;
		return str;
    }

    public Document getMetadata(String idSession) throws IOException {
        prepareCall("iCAASCollectionService","getMetadata");
        rpc.addProperty("idSession",idSession);
        envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
        envelope.setOutputSoapObject(rpc);
        try {
            service.call(action,envelope);
```

```
            } catch (XmlPullParserException e1) {
                  e1.printStackTrace();
            }
            Reader reader=new InputStreamReader(new
ByteArrayInputStream(((String)envelope.getResponse()).getBytes()));
        KXmlParser parser=new KXmlParser();
        Document doc = null;
        try {
                parser.setInput(reader);
                doc=new Document();
                doc.parse(parser);
        } catch (XmlPullParserException e) {
                e.printStackTrace();
        }

        return doc;
    }
}
```

4.6 Porting the MVC Pattern to Android: a case of software reengineering

After solving the problem of web services invocation by fitting the KSOAP library to Android, the main issue that emerged in the design phase was about the Graphic User Interface. Leaving out the midlet J2ME-specific classes, not very interesting under the design aspect, being clearly generated using an automatic tool, the architecture underlying the GUI turned out to follow a MVC (Model-View-Controller) programming pattern, very common in Java and in object-oriented design in general, as shown in the partial class diagram of figure 4.7. However Android, although basically Java-based, has its own special APIs, structures and techniques for developing GUIs (as seen in Chapter 2) and is therefore very different, under this aspect, from a typical Java-based platform like J2ME. The key question we had to answer was then: "Is it possible to implement the MVC pattern in Android and how?". Clearly, MVC is a good design and programming pattern for a GUI, as we will see better below but, above all, keeping it would mean being able to exploit much more easily the all business logic of the existing application, included the ClientThread class accomplishing the main functions of logging in and retrieving the data from the Server through the stub class already analyzed.

Anyway our task wasn't trivial at all but it was really compelling, also because it was

about trying to port a key object-oriented design and programming pattern for GUIs to a new mobile platform that introduced a completely new approach to the development of Graphic User Interfaces, as already mentioned in Chapter 2. We could consider it with good reason an interesting case of software reengineering.

In the next paragraphs we will say what MVC is and how it can be fit to Android.

4.6.1 Model-View-Controller: separating GUI from Business Logic and Data Access

Model–view–controller (MVC) is an architectural pattern originally introduced in the Smalltalk object-oriented programming language and now commonly used in software engineering.

Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance or the underlying business rules without affecting the other. Moreover the data access is better controlled because it's also separated from GUI and processing. So the same data can have several visual representations (depending for example on the specific device and/or platform and/or user and so on) or can be processed in a different way.

In MVC, the *Model* encapsulates the data of the application, providing the methods for accessing them, the *View* displays these data and interacts with the user through elements of the GUI and the *Controller* listens for users' inputs and external events and reacts to them changing the state of Model and View when needed. Therefore the Controller acts as an intermediate between Model and View.

The diagram below shows how the MVC pattern works.

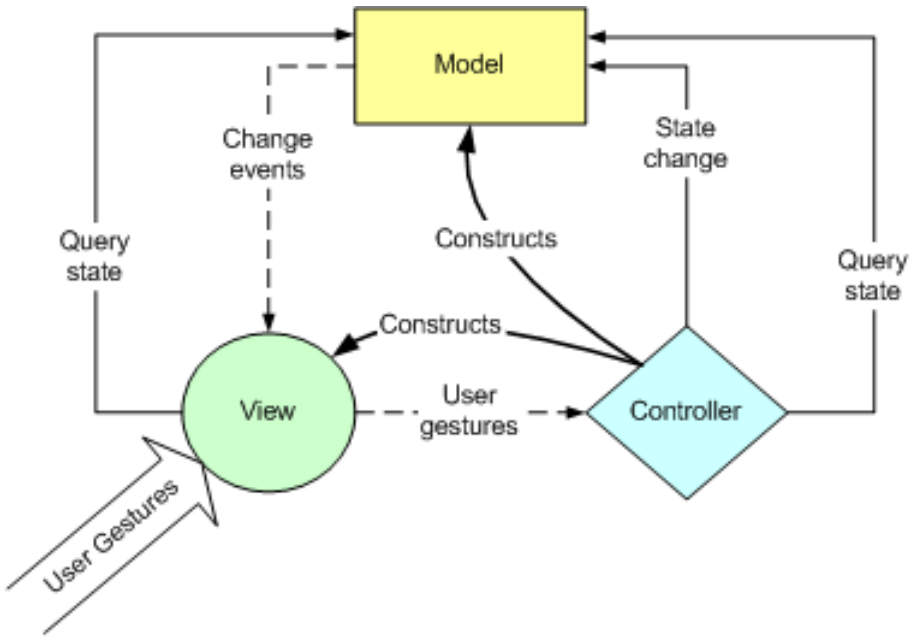

Figure 4.10

Actually the MVC pattern can be regarded as both a real architectural pattern and as a design pattern depending on whether physical separation of the three elements (Model View and Controller), that can be hosted by different networked machines is required or not.

In this case we are talking about MVC with regard to a single client application running on a mobile device: we will therefore consider it as a design and programming pattern more than an architectural pattern in the strict sense of the word.

In the specific J2ME implementation of MVC, shown as a hierarchical diagram in figure 4.7, Views register on Models so that they can change their state following data updates, while Controllers, implementing CommandListeners, are associated to both Views and Models so that they can listen for users' inputs received through the GUI and accordingly change the data managed.

It is clear that this schema cannot work without any change when developing for Android, as its GUI APIs are quite different from the Java Mobile ones, and we have therefore to understand who could play each role in an MVC-like pattern.

This is what we will try to do in the next paragraph.

4.6.2 AVA (Adapter-View-Activity): MVC, the Android way

We learnt in Chapter 2 that Android doesn't include among the other Java APIs, Swing and least of all any J2ME-specific library, as it has its own special APIs for creating GUIs, first of all the *android.view* library. Furthermore, although it is possible to define user interfaces directly in Java code (of course using the Android-specific APIs), this is not the preferred technique when developing for the Google mobile platform.

Rather Views implementing a GUI should be defined in external resource files (xml format) so allowing to decouple the presentation layer from the business logic as well as to change the presentation without changing the code. Moreover this makes possible to specify different layouts optimized for different hardware configurations, even changing them at run time based on hardware changes (such as screen orientation). This kind of approach to the development of user interfaces is a key point of the "Android Revolution" as it gives some important benefits over other mobile platforms. That's why we chose to adopt it without distorting the Android development philosophy.

However this completely changed our perspective when trying to fit the MVC pattern to our needs, as one can easily see. The main issue is that, with this approach, a View is no more a class at all as it's defined in an xml resource file. So we have to understand how Models and Controllers can interact with Views and, in addition to this, who Models and Controllers are in the Android world.

With regard to Controllers it was quite clear from the beginning that Activities had to play this role, as they work at same time as a sort of View Containers and Event Listeners. Indeed an Activity coincides with a screen containing the visual elements defined in the xml resource files and bound to it through the setContentView method as well as using the R.java file (as explained before) but also allows to associate to visual elements (such as buttons) listener classes (such as *OnClickListener*) to receive users' inputs and accordingly react to them.

Much less obvious was identifying which Android class could act as a Model. At first it

seemed that this task could be assigned to a Content Provider, as this class, similarly to a Model, provides data encapsulation, allowing applications store, share and access information. Actually using a Content Provider is only required if more applications need to share data and these data need to be stored in an SQL- based DBMS. In this case we don't need to share any data with other local applications and moreover we have no local database, as the data to be viewed are received in real time from a remote server that hosts a proper database to store them. That's why we rejected such hypothesis and focused our attention on another interesting Android-specific class: the Adapter.

An Adapter object acts as a bridge between an AdapterView and the underlying data for that view. The Adapter provides access to the data items and is also responsible for building a View for each item in the data set. An AdapterView is simply a View whose children are determined by an Adapter (let's remember from Chapter 2 the hierarchy of View and ViewGroup nodes building an Android GUI). So we had exactly what we needed: it was a matter of fitting it to our needs and making it work properly.

Although several native Adapters were available in the APIs we decided to build our own custom Adapter, inheriting from the BaseAdapter class, which provided the basic methods for implementing it. Adopting this technique we were able to partially reuse even the SensorModel class, that played the role of the Model with regard to the part of the original application managing the list of the sensors displayed. Basically we turned SensorModel into a BaseAdapter-kind class, keeping the original code as much as possible and implementing the needed methods (particularly getView, automatically called in the associated Activity after the Adapter is associated to an AdapterView, in order to get the View displaying the current item).

Therefore the custom Adapter class is used in the related Activity (SensorsList) where a SensorModel object is created and then associated to the data to manage (a Vector of sensors that will be received from the server) and to the View that will display the data (in this case a ListView, defined as usual in a Sensors.xml resource file). The layout needed for displaying every single item in the ListView (a row containing an icon and a text field)

is also defined in another row.xml proper file and passed to the SensorModel using the R.java file as an index. Finally a Handler object defined in the same Activity allows to force the updating of the Views associated to the Adapter every time the underlying data are modified by an external class (in this case ClientThread, that communicates with the server through the stub and receives data from it): when this occurs a message sent to the handler triggers the updating, accomplished by calling the notifyDataSetChanged method on the custom Adapter object created.

In conclusion we can say that the original MVC pattern becomes on the Android platform an AVA pattern, where the Adapter acts as a Model, the View is defined in an external .xml resource file and inflated into the code by proper methods using R.java (such as findViewById(int id)), and the Activity plays the role of a Controller, that registers a View with an Adapter and reacts to the users' inputs received through the GUI modifying the data managed by the Adapter and accordingly updating the associated Views.

We show below a conceptual model of the AVA pattern defined.

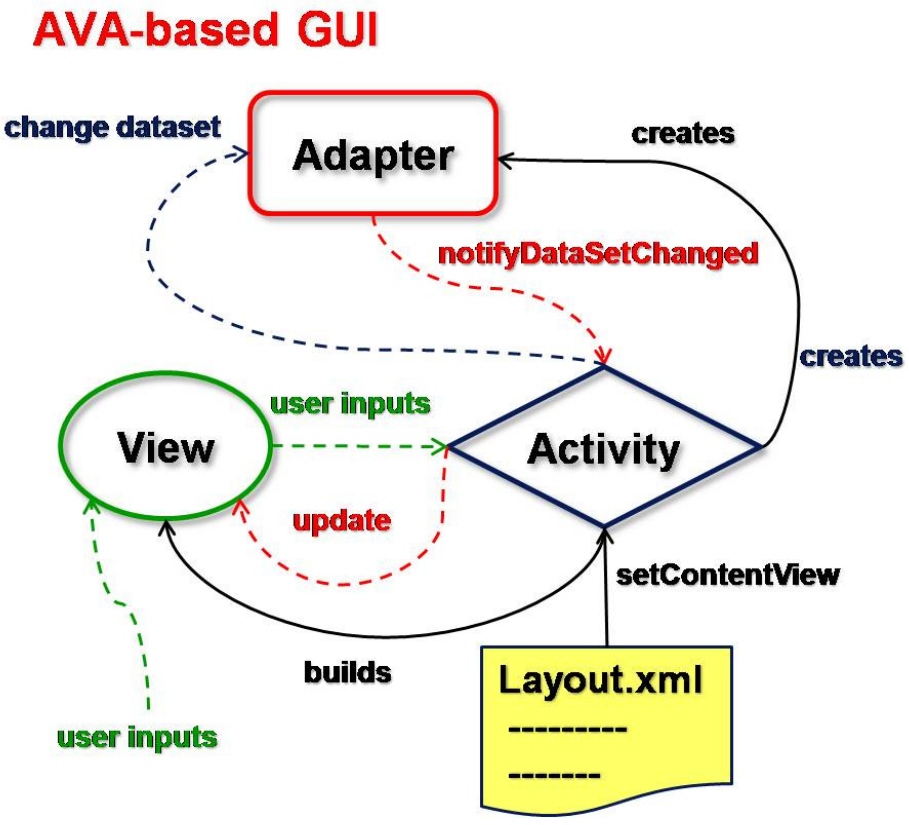

Figure 4.11

In the following figure is instead represented a partial class diagram of the application including the classes involved in the AVA pattern (SensorsList and SensorModel).

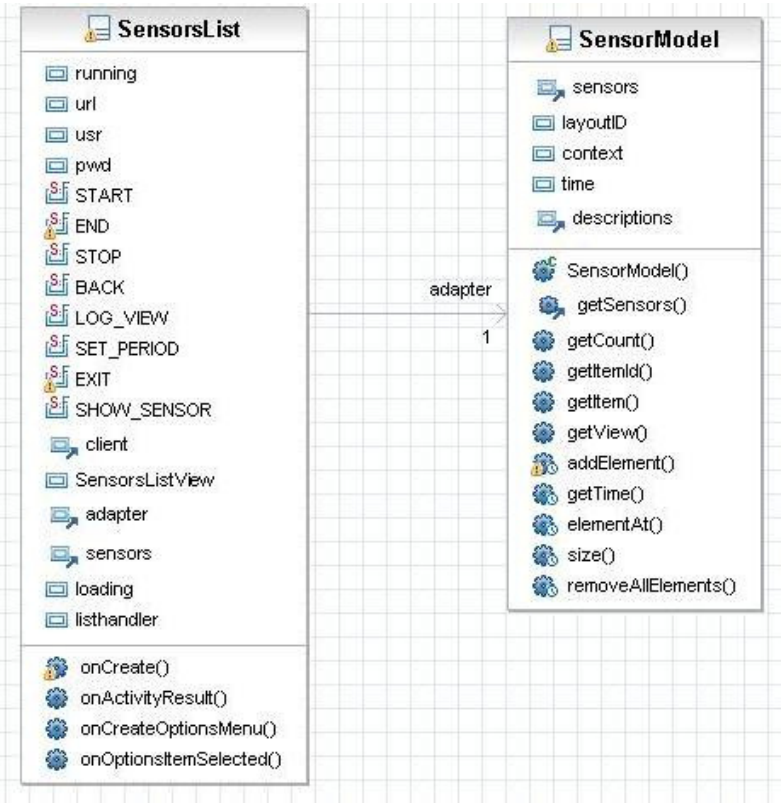

Figure 4.12

We used again the AVA pattern for the parts of the application managing the view of a single sensor and of the detailed properties of a sensor. We don't illustrate them as there are no conceptual differences but the schema is repeated more or less the same way.

For the source code implementing the reengineered design pattern here presented (including the related .xml resource files) we refer to the Implementation section (paragraph 4.8.3).

4.7 Object Oriented Design: Class Diagram and Package Diagram

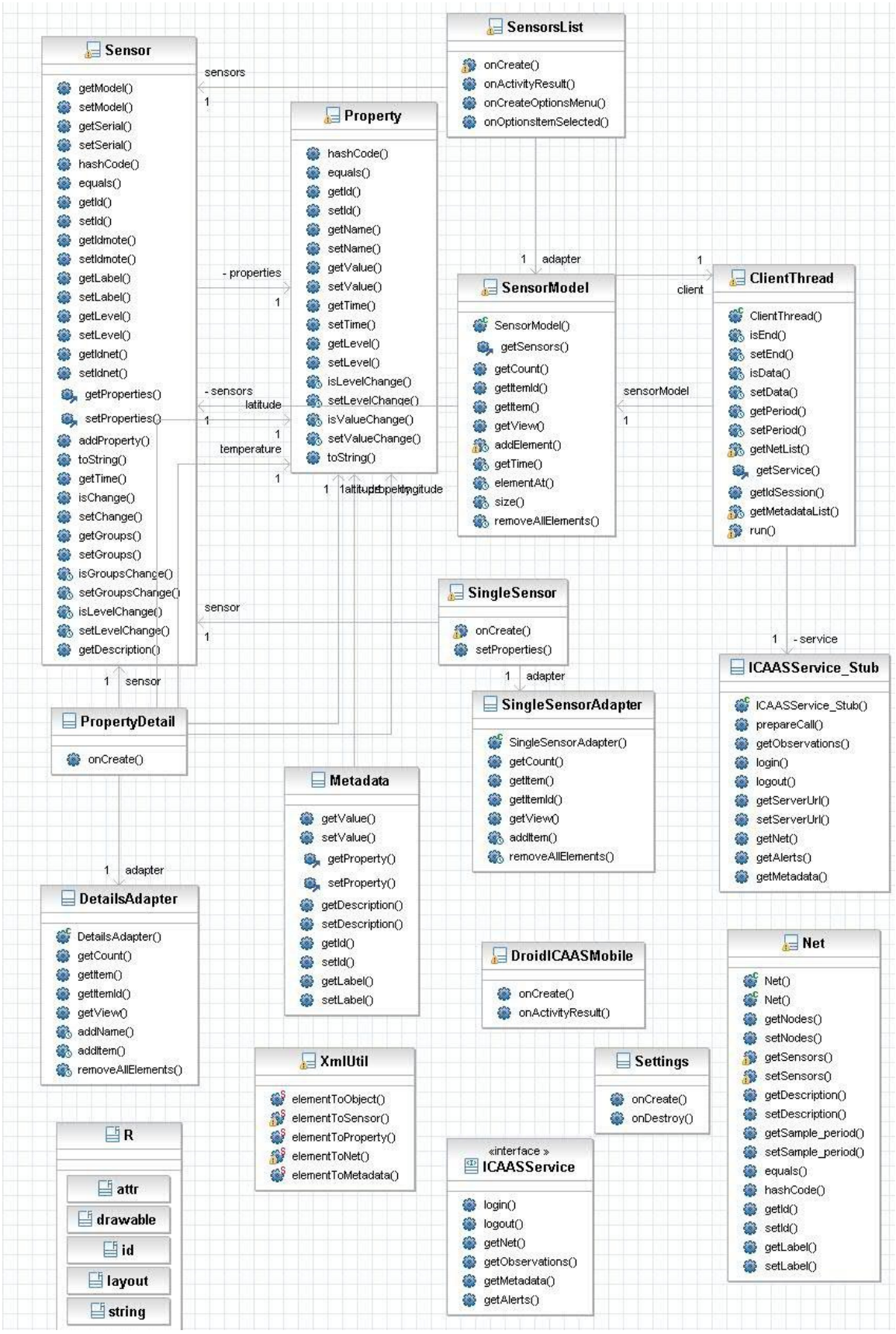

Figure 4.13

We will briefly describe some of the classes included in the Class Diagram above. DroidICAASMobile is the main Activity (building the welcome screen of the application) while Settings is the Activity for managing the login data. SensorsList, SingleSensor and PropertyDetail are the Activities responsible for showing: the list of the sensors, the properties of a single sensor and the details of a single property. SensorModel, SingleSensorAdapter and DetailsAdapter are the Adapters built by those Activities to manage the data they need to display in their screens and to get notified about dataset changes. As explained, the Activities build their screens exploiting the View definitions contained in the xml layout files. Net, Metadata, XmlUtil, Property and Sensor are utility classes inherited from the original application. ICAASService is the Stub interface and ICAASServiceStub is its (properly restructured) implementation. ClientThread is a java Thread responsible for consuming Web Services through the Stub, so receiving from the ICAAS Server the data handled by the Adapters and displayed by the Activities.

The overall class diagram of the DroidICAASMobile application looks definitely less complicated than the one of the original application, seen before. So our porting-oriented design process leaded us to get a good design plan, quite easy to understand and thereby to maintain and to modify.

This is confirmed by the package diagram, that includes one package less than the original one (the mvc package, for obvious reasons), as represented below. We decided indeed to store the new Android classes in the same package containing the rest of the user interface classes without replacing the deleted package with a new one. Instead the service and type packages are still helpful for the client application.

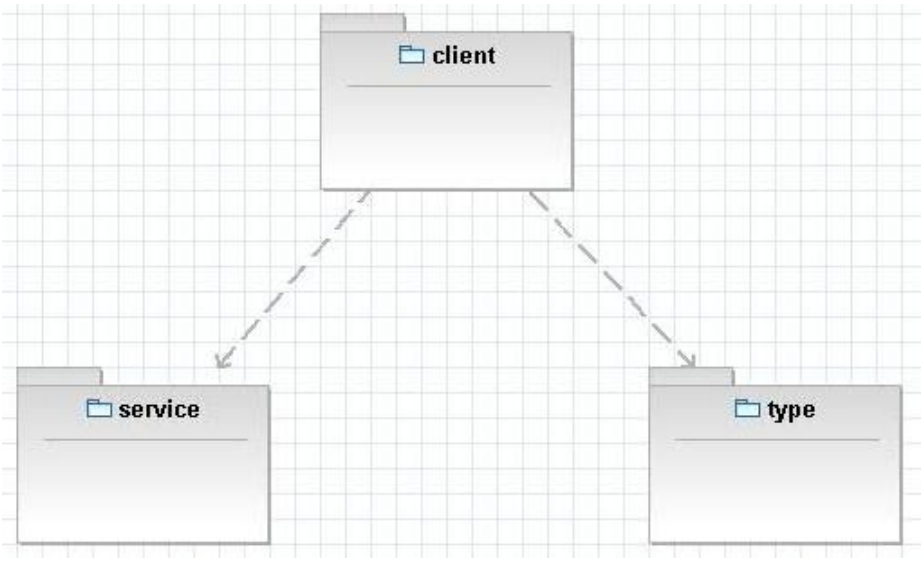

Figure 4.14

4.8 Implementation

In this section we will present the source code of some interesting classes included in the application we developed to give a more accurate idea of how an Android application is made and of course to show how the design solutions described before were translated into working code. An example of implementation of the MVC pattern is shown in paragraph 4.8.3. We leave out other classes where the same pattern is repeated. The code of the restructured stub has been already presented, in comparison with the original one, in paragraph 4.5.2.

4.8.1 The Manifest and the main Activity

In the Manifest we note that a node (signalled by a tag) is present for every Activity in the application. No Activity can work without being included in the Manifest. An Intent Filter for every Activity is also set. The uses-permission tag has been set to get access to the Internet in order to connect to the ICAAS server.

With regard to the DroidICAASMobile activity, managing the main screen of the application, we highlighted in light blue the code responsible for the GUI handling: View objects definitions (in this case Buttons) and onClickListener methods to respond to user

inputs. In red, the setContentView method for building the GUI from the layout xml file associated with this Activity. In yellow, the code responsible for starting a new Activity through an Intent. In the case of the Settings Activity the Intent is also loaded with some input parameters by a Bundle object. The startAcitivityForResult method is used instead of startActivity to get some output parameters back from the called Activity. The code managing the return parameters is highlighted in green.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="vitodaniele.DroidICAASMobile"
      android:versionCode="1"
      android:versionName="1.0.0">
      <uses-permission android:name="android.permission.INTERNET" />
    <application android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:name=".DroidICAASMobile"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"
/>
            </intent-filter>
        </activity>
        <activity android:name=".Settings"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT"
/>
            </intent-filter>
        </activity>
        <activity android:name=".SensorsList"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT"
/>
            </intent-filter>
        </activity>
        <activity android:name=".SingleSensor"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT"
/>
            </intent-filter>
        </activity>
        <activity android:name=".PropertyDetail"
                  android:label="@string/app_name">
            <intent-filter>
```

```
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT"
/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

DroidICAASMobile.java

```
package vitodaniele.DroidICAASMobile;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;


public class DroidICAASMobile extends Activity {

     //UI Variables

     private Button help;
     private Button settings;
     private Button login;
     private Button exit;

     //Login Variables

     private String server = "http://192.168.1.20:8080/axis/services/";
     private String username = "EnteC";
     private String password = "passwordc";

     //Subactivities Codes

     private static final int SHOW_SETTINGS = 1;
     private static final int SHOW_SENSORS = 2;

     /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        help = (Button) this.findViewById(R.id.help);
        login = (Button) this.findViewById(R.id.login);
        settings = (Button) this.findViewById(R.id.settings);
        exit = (Button) this.findViewById(R.id.exit);


        settings.setOnClickListener(new OnClickListener(){
           public void onClick(View v)    {

                Intent i = new Intent(
                DroidICAASMobile.this,
                Settings.class);
```

```
                Bundle b = new Bundle();
                b.putString("server",server);
                b.putString("username",username);
                b.putString("password",password);
                i.putExtras(b);
                startActivityForResult(i,SHOW_SETTINGS);

            }

        });


        exit.setOnClickListener(new OnClickListener(){
            public void onClick(View v)    {

                DroidICAASMobile.this.finish();


            }

        });


        login.setOnClickListener(new OnClickListener(){
            public void onClick(View v)    {
                Intent i = new Intent(
                        DroidICAASMobile.this,
                        SensorsList.class);
                        Bundle b = new Bundle();
                        b.putString("server",server);
                        b.putString("username",username);
                        b.putString("password",password);
                        i.putExtras(b);
                        startActivity(i);

            }

        });


        }

    protected void onActivityResult(int requestCode, int resultCode,
Intent data)  {

        if (requestCode == SHOW_SETTINGS) {
                if (resultCode == RESULT_OK) {
                    // Settings were modified
                    Bundle b = data.getExtras();
                    server = b.getString("server");
                    username = b.getString("username");
                    password = b.getString("password");
                }
        }

    }
    }
```

4.8.2 Changing the Settings

Leaving out the parts of code analogous to the ones already commented in the previous section, we remark the lines, in yellow, for passing the modified connection parameters to the calling Activity, that will receive it through the onActivityResult method, as seen.

Settings.java

```
package vitodaniele.DroidICAASMobile;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

public class Settings extends Activity {

     //UI Variables

     private EditText server;
     private EditText username;
     private EditText password;
     private Button back;
     private Button change;


    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.settings);
        Bundle bundle = getIntent().getExtras();
        String url = bundle.getString("server");
        String usr = bundle.getString("username");
        String pwd = bundle.getString("password");

        back = (Button) this.findViewById(R.id.back);
        change = (Button) this.findViewById(R.id.change);
        server = (EditText) this.findViewById(R.id.server);
        username = (EditText) this.findViewById(R.id.username);
        password = (EditText) this.findViewById(R.id.password);
        server.setText(url);
        username.setText(usr);
        password.setText(pwd);

        back.setOnClickListener(new OnClickListener(){
           public void onClick(View v)   {

                 Settings.this.finish();

           }
```

```
        });


        change.setOnClickListener(new OnClickListener(){
           public void onClick(View v)   {
           String server = Settings.this.server.getText().toString();
           String username = Settings.this.username.getText().toString();
           String password = Settings.this.password.getText().toString();

           Intent i = new Intent(Settings.this,DroidICAASMobile.class);

           Bundle b = new Bundle();
           b.putString("server",server);
           b.putString("username",username);
           b.putString("password",password);
           i.putExtras(b);
           Settings.this.setResult(RESULT_OK,i);

           }

        });



        }

     public void onDestroy() {
      super.onDestroy();
      }
    }
```

4.8.3 Managing the sensors list: the AVA solution coded

The SensorsList Activity is responsible for the screen displaying the list of the sensors received from the ICAAS server through the ClientThread (that uses the Stub for invoking Web Services). This Thread is not shown, as it has been inherited from the original application and kept almost unchanged. SensorsList also creates the Adapter (SensorModel) that manages the data (modified by ClientThread) underlying the screen, in addition to building the View from the Sensors.xml layout file. Row.xml works instead as a layout model for a single row of the list. In green, the code for building the ListView and associating it to a SensorModel Adapter. In yellow, the lines that create and start the ClientThread. In light blue, the methods for creating the Menu and responding to user's clicks on it. In grey, the Handler that allows the Activity to update the View, following the events triggered by the ClientThread by proper messages.

The SensorModel class reuses the original class implementing the Model in the MVC

pattern, with some changes needed for converting it into an Adapter (inheriting from BaseAdapter, as highlighted in yellow). In light blue, the getView method that allows the SensorsList Activity to build and update the screen depending on the data managed by the Adapter (basically a Vector of items of the Sensor custom type). It is also interesting to remark the use of the R.java file for indexing the resources of the application. In green the method for adding sensors to the list, used by ClientThread.

SensorsList.java

```
package vitodaniele.DroidICAASMobile;

import java.io.IOException;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Vector;
import org.ksoap2.serialization.SoapObject;
import android.app.Activity;
import android.app.ListActivity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.os.Parcelable;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;

public class SensorsList extends Activity {

    boolean running=false;

    //MenuItem end;

    String url;
    String usr;
    String pwd;

    //Menu variables
    private static final int START = 1;
    private static final int END = 2;
    private static final int STOP = 3;
    private static final int BACK= 4;
    private static final int LOG_VIEW = 5;
```

```
	private static final int SET_PERIOD = 6;
	private static final int EXIT = 7;

	private static final int SHOW_SENSOR = 1;

	ClientThread client;
	ListView SensorsListView;
	SensorModel adapter;
	Vector <Sensor> sensors;
	TextView loading;


	public Handler listhandler = new Handler(){
		//@Override
		public void handleMessage(Message msg){
			if(msg.what==-1){
				loading.setText(null);
				adapter.notifyDataSetChanged();
			}
			else if(msg.what==1){
				loading.setText("thread stopped: click start to
view the sensors list");
				adapter.notifyDataSetChanged();
				}

			}
		};


	public void onCreate(Bundle icicle){
		super.onCreate(icicle);

		setContentView(R.layout.sensors);

		loading = (TextView)this.findViewById(R.id.loading);

		SensorsListView =
		(ListView)this.findViewById(R.id.myListView);
		sensors=new Vector <Sensor>();

		int layoutID = R.layout.row;

		adapter = new SensorModel(this,layoutID,sensors);

		SensorsListView.setAdapter(adapter);

		Bundle bundle = getIntent().getExtras();
		url = bundle.getString("server");
		usr = bundle.getString("username");
		pwd = bundle.getString("password");


		client = new ClientThread (url, usr, pwd, adapter,
listhandler);
	client.start();

	SensorsListView.setOnItemClickListener(new OnItemClickListener(){

		public void onItemClick(AdapterView parentView, View
childView, int position, long id) {
			Intent i = new Intent(
```

```
                        SensorsList.this,
                        SingleSensor.class);

                ((Sensor) adapter.getItem(position)).setChange(false);


                  Bundle b = new Bundle();
                  b.putSerializable("sensor", (Sensor)
adapter.getItem(position));
                  i.putExtras(b);
                  startActivityForResult(i, SHOW_SENSOR);

                }



            });

            }

    protected void onActivityResult(int requestCode, int resultCode,
Intent data)  {

    if (requestCode == SHOW_SENSOR) {
           if (resultCode == RESULT_OK) {
               // Settings were modified
                adapter.notifyDataSetChanged();
                }
           }

       }

    public boolean onCreateOptionsMenu(Menu menu){
          menu.add(0,START,0,"Start");
          menu.add(0,STOP,0,"Stop");
          menu.add(0,SET_PERIOD,0,"Set Period");
          menu.add(0,LOG_VIEW,0,"Log View");
          menu.add(0,BACK,0,"Back");


          return true;
    }



    public boolean onOptionsItemSelected(MenuItem item){
          switch(item.getItemId()){

          case START:


              client = new ClientThread (url, usr, pwd, adapter,
listhandler);
              client.start();

                  running = true;

                  return true;
```

```
                case STOP:

                      client.stop();
                      client.setEnd(true);
                      running = false;

                      return true;

                case SET_PERIOD:

                      client.setPeriod(0);

                      return true;

                case LOG_VIEW:

                      return true;

                case BACK:

                      SensorsList.this.finish();

                      return true;


                }

                return false;
          }


}
```

SensorModel.java

```
package vitodaniele.DroidICAASMobile;

import java.util.ArrayList;
import java.util.Vector;
import android.content.Context;
import android.graphics.Typeface;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.ImageView;
import android.widget.TextView;

public class SensorModel extends BaseAdapter {

      private Vector <Sensor> sensors = new Vector <Sensor>();
      private int layoutID;
      private Context context;
      private long time;
      private ArrayList <String> descriptions = new ArrayList <String> ();


      public SensorModel(Context activity,int resource,Vector <Sensor>
items){
            context=activity;
```

```
            layoutID=resource;
            sensors=items;
      }


      public Vector <Sensor> getSensors(){
            return sensors;
      }

      public int getCount() {
              return sensors.size();
            }

      public long getItemId(int i){
              return i;
      }

      public Object getItem(int i){
              return sensors.get(i);
      }


      public View getView(int position, View convertview, ViewGroup
parent){

            View row=convertview;

                  if (row==null) {

                    LayoutInflater inflater =
(LayoutInflater)context.getSystemService(Context.LAYOUT_INFLATER_SERVICE)
;
                        row=inflater.inflate(layoutID,null);

                }
                  TextView label=(TextView) row.findViewById(R.id.label);
                  label.setText(descriptions.get(position));
                  ImageView icon=(ImageView)row.findViewById(R.id.icon);

                  if(sensors.get(position).isChange()){
                    icon.setImageResource(R.drawable.sensorred24);
                    label.setTypeface(Typeface.DEFAULT_BOLD);
                    }

                  else{
                    icon.setImageResource(R.drawable.sensorgreen24);
                    label.setTypeface(Typeface.DEFAULT);
                  }

            return (row);
      }

      public synchronized void addElement(Sensor sensorRec) {

            if(!sensors.contains(sensorRec)) {

                  sensors.addElement(sensorRec);
                  descriptions.add(sensorRec.getDescription());

                  if (sensorRec.getTime()>time)
                        time=sensorRec.getTime();
```

```
            } else {

                  Sensor
sensor=(Sensor)sensors.elementAt(sensors.indexOf(sensorRec));

                        sensor.setGroups(sensorRec.getGroups());
                        sensor.setLevel(sensorRec.getLevel());

                        Vector propertiesRec=sensorRec.getProperties();

                  for (int j=0;j<propertiesRec.size();j++) {
                              Property propertyRec=(Property)
propertiesRec.elementAt(j);
                              Vector properties=sensor.getProperties();
                              if(properties.contains(propertyRec)) {
                                  Property property=(Property)
properties.elementAt(properties.indexOf(propertyRec));

property.setValue(propertyRec.getValue());
                                  property.setTime(propertyRec.getTime());

property.setLevel(propertyRec.getLevel());
                              } else{
                                  properties.addElement(propertiesRec);
                              }
                              sensor.setProperties(properties);
                        }



                  sensor.setChange(true);

                  if (sensor.getTime()>time)
                        time=sensor.getTime();

                  sensors.setElementAt(sensor,sensors.indexOf(sensorRec));
            }


      }

      public synchronized long getTime() {
            return time;
      }

      public synchronized Sensor elementAt(int arg0) {
            return (Sensor) sensors.elementAt(arg0);
      }

      public synchronized int size() {
            return sensors.size();
      }

      public synchronized void removeAllElements() {
            sensors.removeAllElements();
            time=0;

      }

}
```

Sensors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
android:id="@+id/widget38"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
xmlns:android="http://schemas.android.com/apk/res/android"
>
<TextView
android:id="@+id/listtitle"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="All Sensors"
android:textStyle="bold"
android:layout_alignParentTop="true"
android:layout_alignParentLeft="true"
>
</TextView>
<TextView
android:id="@+id/loading"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="loading: please wait a few seconds..."
android:layout_below="@+id/listtitle"
android:layout_alignParentLeft="true"
>
</TextView>
<ListView
android:id="@+id/myListView"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_below="@+id/loading"
android:layout_alignParentLeft="true"
>
</ListView>
</RelativeLayout>
```

row.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
android:id="@+id/widget38"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
xmlns:android="http://schemas.android.com/apk/res/android"
>
<ImageView
android:id="@+id/icon"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
>
</ImageView>
<TextView
android:id="@+id/label"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="TextView"
```

```
>
</TextView>
</LinearLayout>
```

Chapter 5

DroidICAASMobile working: examples of use and scenarios

In this chapter we will present some examples of use and scenarios by showing several screens from the Android client for accessing the ICAAS framework we developed: DroidICAASMobile. Most of them will clearly match some functional requirements identified during the analysis phase.

5.1 The Main Screen

As we can see in the picture below, after starting the application, the user can check or change the Settings or directly login to the ICAAS server.

The screenshot on the left shows instead the emulated Android device menu, containing a launchable icon of the DroidICAASMobile application.

Figure 5.1

5.2 Scenario 1: Changing the Settings

Figure 5.2

In this screen the user can simply view the current settings concerning server URL, username and password or change them by typing in the proper text fields and then go back to the main screen.

5.3 Scenario 2: Login and View of the Sensors List

Figure 5.3

Soon after the user clicks on the login button in the main screen, a loading message is showed while the data are being received from the server. In a few seconds the list of the sensors in the current network (in this case Solfatara) is displayed. At the beginning they are all green as none of them has got its data updated.

As soon as the data of a sensor are updated, its icon becomes red to signal this event to the user, as we can see in the following figure.

Figure 5.4

An updated sensor stays red till the user click on it to view its details (as we will see below). Instead in the following figure is shown the drop-down menu letting the user start and stop the thread that receives data from the server.

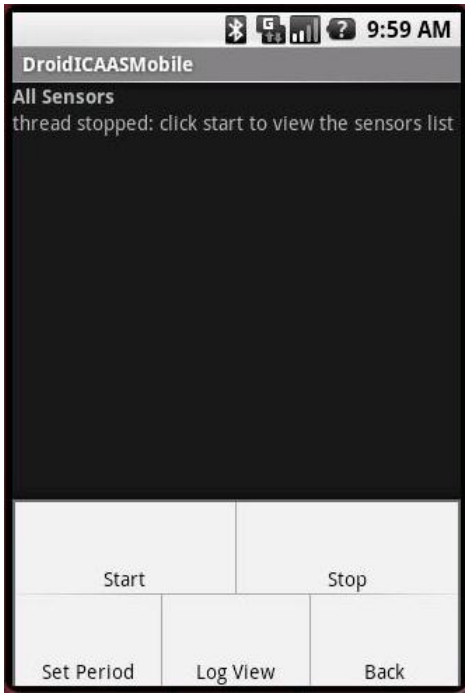

Figure 5.5

In this case the thread has been stopped and a message warning about this is displayed. The application can be restarted using the proper button so that the list of the sensors can be viewed again. This way the application is in fact reset and data are forced to be updated.

5.4 Scenario 3: Selection of a Sensor of the List

When the user clicks on a Sensor in the list previously seen, its main properties with their values are displayed in a new screen, as shown below for a sensor of the Vesuvio network. As already mentioned, if the clicked sensor was red before viewing this screen, after going back to the list of the sensors, it will be green. The sensor will become red again only the next time its data will be updated.

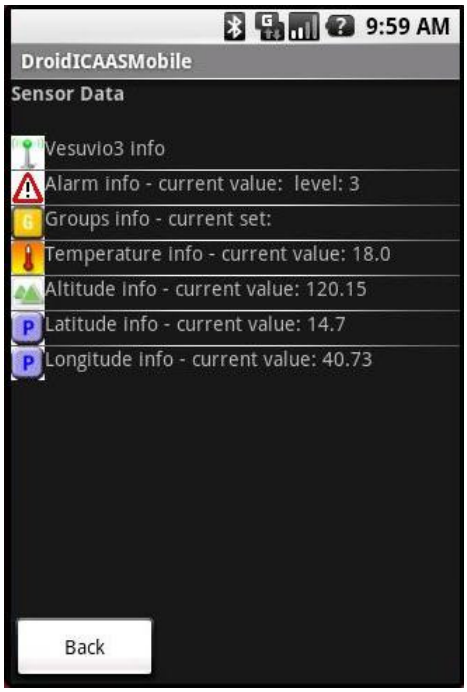

Figure 5.6

5.5 Scenario 4: View of a Property of the current sensor

Clicking on a property of the current sensor will then load a new screen giving a detailed view of the property. We show here the screenshots concerning four different properties: general info, temperature, latitude and altitude.

Figure 5.7

Chapter 6

The Android Experience: a comparative critical analysis

In this final chapter we will present some critical considerations originating from our experience with the Android mobile platform.

Firstly we will try to analyse it in a developer's perspective. Then we will make a comparison with the other platform involved in the porting we performed: J2ME. Finally we will briefly consider some other competing mobile technologies.

6.1 Getting started: prerequisites, initial problems and learning curve

As all Android applications are written in Java, the first obvious background a developer needs to start working with the Google mobile platform, is a good experience with the SUN's programming language.

A useful (although not strictly essential) requisite concerns moreover the Eclipse Development Environment, as a proper plugin allows to integrate in it the Android SDK. Recently an analogous plugin for NetBeans has been released but, when we started this work, Eclipse was in fact the only Android-ready IDE. Of course one could do without any IDE but his task would be much harder than needed.

However, taken for granted this basic knowledge, Android has a number of special features, quite different from any other platform, that a developer has to learn.

First, Android imports some Java standard APIs but also has its own libraries: many things need to be done using them instead of Java's. Furthermore learning to use these special

APIs requires understanding, at least at a basic level, the overall architecture of the platform and above all the main components of a typical Android application and their functions, as well as their lifecycle. The point is that Android "is" Java to a certain extent: the most common misunderstanding people fall into is about this.

The second main issue you have to deal with is the AndroidManifest.xml file: what is it and how does it work? We discussed it in Chapter 2.

Then you need to follow the Android way to GUIs: xml resource files, their relationship with activities and with the "magic" R.java file (generated and updated by the SDK). It is important to highlight that this is anything but trivial, as a Java developer would naturally tend to bypass this constraint and keep on coding the user interface, so distorting the Android philosophy and narrowing the platform capabilities. With regard to this, a good support can be provided by a drawing tool automatically generating the xml code for the GUI so that the developer has not to write it manually (we used one called DroidDraw).

One of the major difficulties we found during this learning process was due to the lack of full and clear documentation: although definitely improved after the SDK launch period, it hasn't reached yet the level of support that a Java developer is used to have at his disposal. Luckily a good number of Android developers' communities were created very soon over the Internet for several reasons: the spread of Java, the open philosophy of the platform, the popularity of Google and – why not? – the spur of the Android Challenge.

They really helped us to overcome many obstacles.

That being so, we can say that the learning curve of the Android development, although the prerequisites are very common, is quite steep at the beginning, as an experienced developer needs a sort of mind-changing when getting acquainted with some key features. Moreover he has to study the architecture and the applications structure and lifecycle before being able to actually develop. Nevertheless, after the first phase, the learning becomes definitely faster and developing for Android could turn out to be easier than developing for other mobile platforms (particularly flexible and powerful is the GUI design technique).

6.2 Android and Java: which is the relationship?

One of the first things people usually wonder when approaching to Android is: "Where is Java?". Or better: "Which role does Java play within it?". I mean, all applications are written in Java, some – not all – the APIs are standard Java libraries, but in spite of this Android is something completely different from a simple Java framework and from the Java mobile version (J2ME). So where is the trick?

The point is that the number and quality of available applications determines the success of a mobile platform, and Google, entering a competitive market like the mobile one, had to strongly consider this. That is why, for an open source platform like Android, widely relying on the developers' community for improvements and new applications, Java was the natural choice: one of the largest developers' community, a mature set of development tools and a well-known bytecode with its specifications, allowing to modify and optimize it at will. What could one want more? In fact every Java developer is a potential Android developer.

Besides this, a further motivation to start developing for Android was the mentioned Android Challenge and then the introduction of the Android Market, where anyone can sell or even donate his applications for a reasonable fee. Moreover no license is required for getting the SDK, unlike other platforms.

However, as seen in Chapter 2, Android, although relying on Java as a programming language, does not run Java bytecode but something achievable from it through a proper tool generating its own optimized executable format, explicitly thought for the Dalvik Virtual Machine, that is not a simple Java Virtual Machine implementation, as already discussed. So Java applications can run on Android only provided that they have been transformed into Android applications.

One could anyway wonder why Google, though deciding to use Java, did not rely on J2ME when thinking of a new OS for mobiles. The first reason is that J2ME was not as

successful as J2SE and JEE, because it did not completely keep the same promise: ”Write once, run everywhere”. Mobile devices are indeed too different from one another (for size, resolution, input mode, technologies and connections supported) to run all the same software. They should rather be divided into classes (as occurred with Symbian devices for example) with analogous features. Moreover J2ME was initially conceived as a subset of Java SE, leaving out the APIs not fit for a constrained environment like a mobile devices. Rather than designing a new mobile platform the goal was selecting the set of Java APIs working in a mobile environment too and then adding to them some proper mobile libraries. As a consequence most of J2ME APIs are not optimized for mobiles because not originally thought for them. Therefore Google and OHA preferred to create a new mobile-specific optimized platform *ex novo* (except programming language, Linux kernel and some libraries) rather than trying to fit J2ME to their purposes.

6.3 J2ME vs Android: motivations for changing over

As mentioned before, J2ME (Java 2 Platform Micro Edition) is a specification of a subset of the Java platform aimed at providing a certified collection of Java APIs for the development of software for tiny, small and resource-constrained devices.

In this paragraph we will try to analyze it, under a developer’s point of view, in comparison with Android, the target platform of the porting we performed from J2ME.

The first difference to be highlighted is that, unlike Android, J2ME is not a complete software stack, as it does not include an Operating System, but needs one to run (several mobile OS support it, included Symbian). The Configuration Level, based on a Virtual Machine, is of two types: CDC (Connected Device Configuration), running on a standard JVM, for powerful devices, and CLDC (Connected Limited Device Configuration), running on a special lightweight Virtual Machine known as KVM, for resource-constrained devices.

On CLDC also lies a Profile Level, MIDP (Mobile Information Device Profile), including

a set of APIs that facilitates the creation of applications (GUIs in particular). An overall scheme of this software stack is shown below.

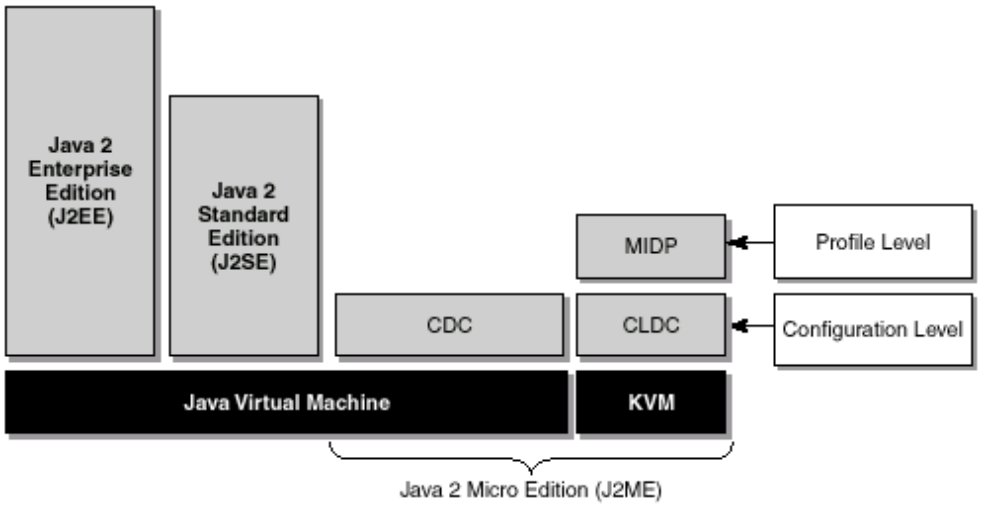

Figure 6.1

The original client we ported to Android was developed exploiting the MIDP profile level. Such an application is also called *midlet*.

The J2ME SDK providing the APIs described and the SUN Wireless Toolkit needed for emulating a J2ME-compatible device and testing applications are available for both Eclipse and NetBeans. The IDEs are therefore the same available for Android.

Considering that the other key prerequisite is also common (Java, obviously), we could say that the learning curve for the two platforms is comparable. We have already discussed our experience with Android. About J2ME we note that an initial effort is required to learn its specific APIs (mainly concerning the GUI), although, apart from this, an application can be developed following a common Java approach, without any need for delving into the software stack and other special features (like in Android). Proper tools can also be used to automatically create the GUI of the application, although this is not advisable for code readability and maintainability reasons.

We can say that the learning curve in this case is less steep than the Android one at the beginning but after the first training period they are quite similar. In the middle term developing for Android could become easier and faster than for J2ME. These considerations are about a Java developer approaching to J2ME.

Of course the case of a J2ME experienced developer approaching to Android is analogous to the case of a Java developer regarded before.

So why should one develop for Android rather than for Java Micro Edition?

I believe that, leaving out the learning curve, a good reason for changing over is about the GUI design. The xml-defined non Java-coded approach, typical of Android, grants the key benefit of natively separating visual look and business logic, so improving the application portability as well as its maintainability. This way, indeed, changing GUI only implies changing some xml code (possibly using an automatic tool) and more than one GUI could be written for the same application in order to suit it to several devices (with different screen size, resolution and so on).

Besides this, another advantage Android has over J2ME is the underlying Linux kernel, enabling it to run, at least in principle, on every Linux-compatible device.

On the contrary J2ME, to work on a device, needs to be supported by its OS and at the same time to support it with a specific JVM (or KVM) version. Moreover a J2ME application can hardly reach the same level of access to the core libraries of the OS on which is running: Android lets all applications access the same APIs used for developing the native ones, so giving an excellent control on the device.

A lack of full portability is one of the main flaws of J2ME, so that some handset producers have been forced to add proprietary extensions to make it work properly on their specific devices. Furthermore porting a J2ME application from a device to a different one can mean in some cases almost completely rewriting it.

Android looks very promising under this aspect, although we should wait for more devices (the only two available are made by the same company, HTC) to draw any conclusion about this.

For developers, another essential aspect, still impossible to accurately evaluate, is obviously the market share, as J2ME owns a very significant one, while Google Android has just entered the mobile world.

6.4 Android and the others: Symbian, iPhone and Windows Mobile

In this paragraph we will describe some development-related features of the Android's main competitors in the mobile market.

Symbian is currently the leading Operating System for "smart phones", developed by Symbian Ltd and running on ARM processors (can be emulated on x86). It is a proprietary OS, although Nokia, that has recently acquired the company, created the Symbian foundation in order to make it open source. Anyway parts of the source code were already provided to mobile manufacturers.

Its native programming language is a non standard version of C++, while the SDK depends on the specific Symbian platform. Indeed there are several platform based on Symbian OS and thought for different devices or families of devices (UIQ and S60 are the most common). Each platform can have a specific SDK or SDK extension. Anyway Symbian plugins for Eclipse and Visual Studio as well as for Borland are available (basic versions are free).

Symbian C++ programming has a steep learning curve, as Symbian requires the use of special techniques such as descriptors and the cleanup stack. This can make even relatively simple programs harder to implement than in other environments.

J2ME programming is also supported as mentioned.

The Apple iPhone is based on a mobile version of Mac OS X. Its native programming language is Objective-C, a special object-oriented version of C already known to Mac developers but not very spread outside the Apple world: this definitely makes the learning curve steeper for non Mac-experienced developers.

Developing for iPhone also requires a specific IDE and emulator, downloadable after

subscribing the Apple Developer Connection membership and installable only on Mac OS. For selling application on the Apple Store, besides a fee, a permission is required: applications duplicating existing functionalities are not accepted.

Windows Mobile is a compact operating system combined with a suite of basic applications for mobile devices based on the Microsoft Win32 API. Developing for it requires writing native code with Visual C++, Managed code that works with the .NET Compact Framework, or Server-side code that can be deployed using Internet Explorer Mobile or a mobile client on the user's device.

Microsoft typically releases Windows Mobile SDKs that work with Visual Studio development environments and include emulator images for developers to test and debug their applications.

Therefore Windows Mobile predictably looks ideal for Microsoft-oriented developers.

In comparison with the three mobile technologies here presented, the Android platform, though immature and with no significant market share yet, seems to have some key advantages: a well known development environment and programming language with a huge potential developers' community worldwide; an average learning curve (as discussed before); an open philosophy that also influences some industry strategies (e.g. Android Market vs Apple Store, more device manufacturers rather than only one).

Conclusions

This work was originally aimed at studying the new Google Android Mobile Platform, with a special focus on its capability of consuming Web Services. Anyway, after starting on our case study, many more issues and interesting topics emerged.

First, we learnt about the ICAAS framework, which turned out to be an ideal test-bed for our goals, thanks to the SOA access module provided, and then we had to deal with the compelling problem of porting a J2ME application to Android.

The outcome of all this was a working Android client for accessing the ICAAS framework, deriving from the original J2ME one through a porting process, and able to use the functions exposed as Web Services by the framework itself.

The first result to be highlighted is the successful adoption of the KSOAP client-side solution for Web Services on Android: this was not granted at all and allowed us to achieve the main goal of the thesis. It is important to note that the KSOAP solution perfectly works once imported to Android, despite all doubts raised by many developers about the need for rewriting some classes of the API and the overhead introduced in a resource-constraint environment. However, with regard to the latter aspect, we should point out that we did not have the chance to test our application on a real Android device but only on the emulator included in the SDK so that no performance analysis was possible. Moreover we want to remark that our result does not prevent us from expecting a native solution for invoking Web Services on Android very soon.

The second aspect we would like to consider is the porting performed from J2ME to

Android. Although the Google mobile platform is in a sense Java-based, it has its own virtual machine executing something different from bytecode and besides this, it does not include all Java standard APIs and least of all Java Micro Edition APIs. Furthermore Android applications have their own special structure and key features that make them quite different from J2ME applications.

Nevertheless we achieved another important result: porting an application from J2ME to Android turned out to be feasible reusing most of the original code (of course not the J2ME-specific one).

In addition, when studying the porting strategy to be adopted, we had to face a further issue: how to port the Model-View-Controller design pattern to Android.

As already discussed, the Android approach to designing GUIs is peculiar as Views are not Java-coded but xml-defined. Moreover its APIs do not allow to directly import the MVC pattern without any change. The main hurdle was understanding which Android classes (or concepts) could substitute the MVC ones and which role a declarative view could play in this context. The outcome of this analytical process was in fact a new design pattern, explicitly thought for Android, that we called AVA from the names of its components: Adapter-View-Activity. This result goes beyond the particular porting we performed, providing a potentially reusable model for designing Android GUIs according to the same idea that inspired the classic MVC pattern: separating user interface from business logic and data access.

Finally we outlined some critical considerations based on our development experience with Android, concerning the learning curve for a developer and a comparison with some features of other popular mobile platforms, first J2ME, involved in our porting process.

Future Works

With regard to the Android application developed, a future work could involve its improvement under both functional and graphic aspect. In particular, under the functional aspect, a dynamic view of the sensors updates events (Log View, already present in the user interface) could be implemented, and new requirements could be added and met.

For instance the client could be provided with configuration facilities (besides the monitoring ones) such as users' registration.

The client should also be tested on real Android devices for analyzing its performance and actual portability features.

Other client-side solutions for accessing Web Services (e.g kXML-RPC) could also be tested on Android and their performance compared with the one of KSOAP while waiting for a native API. A Performance comparison with other mobile platforms would be as well an interesting work.

Another potential field of study might concern porting existing applications from other mobile platforms (e.g. Symbian, iPhone or Windows Mobile native applications) to Android and adapting to it other design and programming patterns, as we made with MVC.

Appendix A

How to install the ICAAS framework

Here are the key steps:

Download and setup mySQL DBMS (at http://dev.mysql.com/downloads/)

Run the iCAASdb_creation.sql sql script for creating the icaasdb database; to make this easier download the MySQL Query Browser tool

Download and setup the Apache Tomcat (http://tomcat.apache.org/download-60.cgi)

Download the SOAP Engine Apache Axis and install it within Tomcat (http://ws.apache.org/axis/)

Copy the libraries needed to run Hibernate into the /lib directory of Tomcat (antlr.jar cglib.jar asm.jar asm-attrs.jars commons-collections.jar, commons-logging.jar hibernate3.jar jta.jar dom4j.jar log4j.jar)

Copy the libraries containing the framework classes, iCAAS-1.0,jar (core classes) and iCAAS-1.0-ws.jar (web services classes) into the /lib directory of Tomcat.

Start Tomcat

Include the Axis libraries in the global classpath

Run from shell the following command for deploying web services, being in the directory containing iCAASService-deploy.wsdd:

java org.apache.axis.client.AdminClient iCAASService-deploy.wsdd

Now the ICAAS server is ready for responding to remote clients' requests: it can be started and stopped at http://localhost:8080/axis/iCAASServer.

Failing a real Sensor Network, to actually test a mobile client consuming ICAAS Web Services, you also need to import into your IDE and run the ICAASNetSimulator Java application, that emulates a WSN sending data to the ICAAS Server.

Appendix B

The easiest way to develop an Android application is using the plug-in for Eclipse provided by Google. For this purpose you need first to download the Android SDK (http://developer.android.com/) and unpack it in a directory you have full access rights to (on Linux or Mac it may be the home directory).

After this you have to download the Eclipse plug-in ADT and install it using the Eclipse update manager, as shown in Figure A.1.

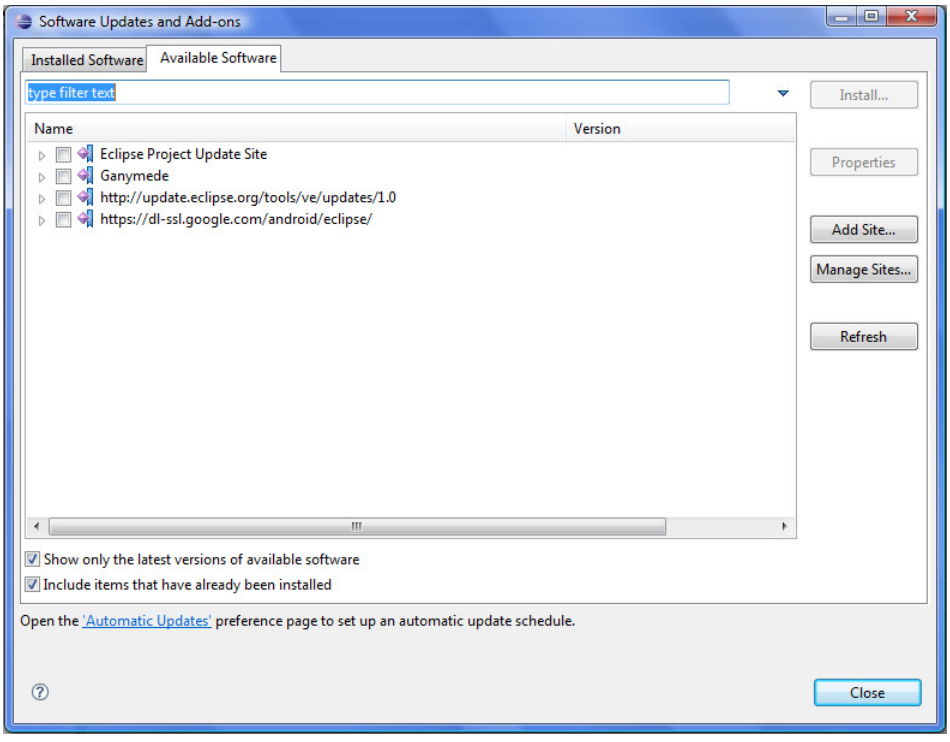

Figure B.1 – The Eclipse update manager

To get the plug-in you have to add the site: https://dl-ssl.google.com/android/eclipse/.

As an alternative (in case of problems with the remote site) you can download the ADT archive from http://developer.android.com/guide/developing/tools/adt.html and add its location as local directory in the update manager.

To configure the plug-in (specifying the SDK location) you have to select Window > Preferences and click on Android in the left panel.

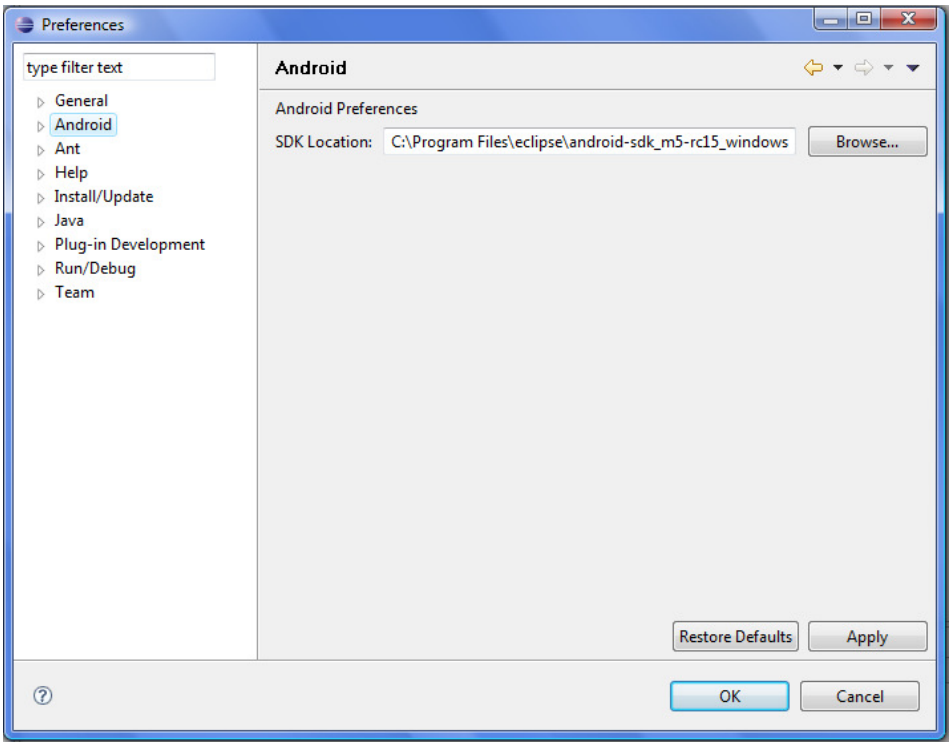

Figure B.2 – The Android configuration panel

Appendix C

We will now explain how to create an Android Application.

The first step is creating a new Android Project using the Eclipse wizard, as shown below.

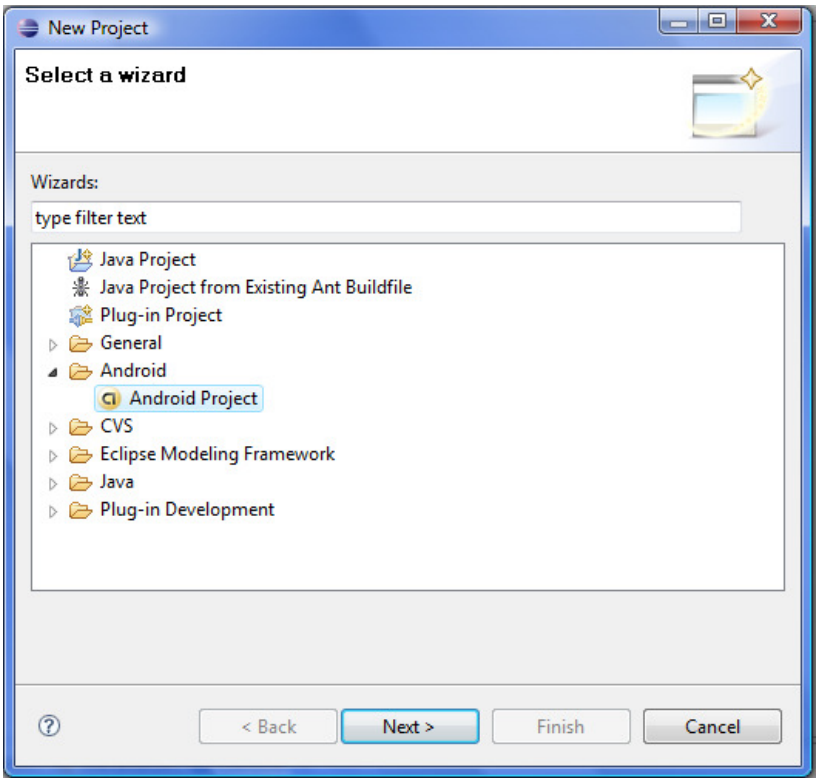

Figure C.1 - Creating a new Android Project

After this you can configure the basic parameters of the project using the proper

Configuration Window (following figure).

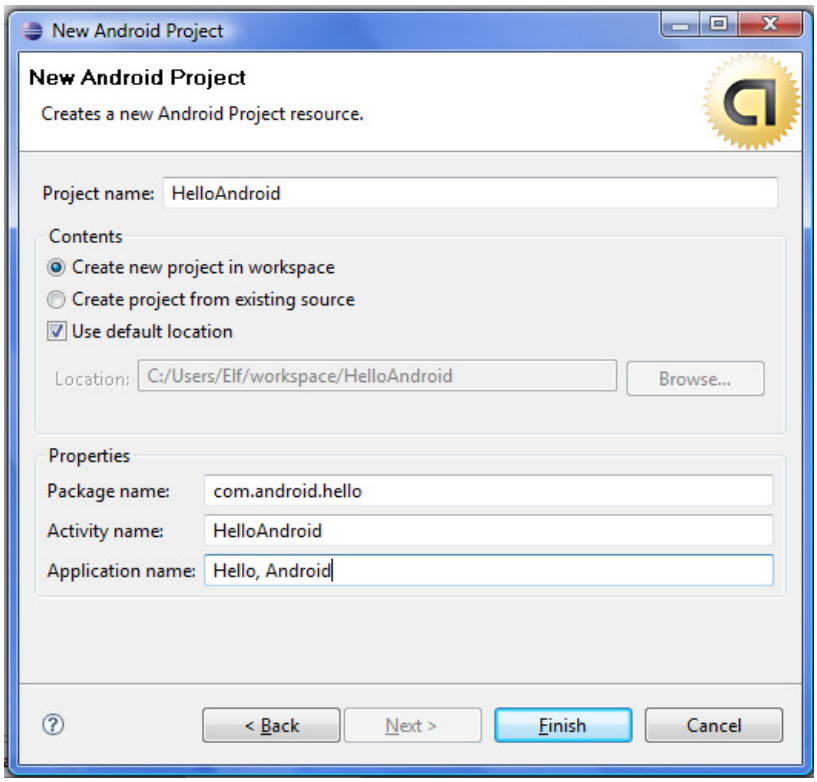

Figure C.2 – Project Configuration Window

The package name follows the typical Java style and must be unique on the system, while the Application name can be whichever. Every Android project needs an Activity class (like every Java project needs a main) although this is not the only entry point of the application (like the main in Java), as there is a really modular philosophy behind.

The code automatically generated for the activity will be like this.

```
package com.google.android;

import android.app.Activity;
import android.os.Bundle;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
    }
}
```

This Activity, of course, does not do anything and does not display any GUI. However it is registered by default in the AndroidManifest.xml file automatically included in the project. The R.java file for indexing external resources is also created.

To associate a GUI to the Activity, the standard technique is creating an xml file in the /res/layout folder of the project and using its identifier (the related attribute generated in R.java, e.g. *R.layout.main*) as input parameter of the setContentView(int layoutResID) method.

For avoiding to manually write the xml code for the GUI you could use a drawing tool that automatically generates the xml code for the layout you draw and then copy it in a proper file of the /res/layout folder.

We show here a screenshot from such a tool, DroidDraw (http://www.droiddraw.org/).

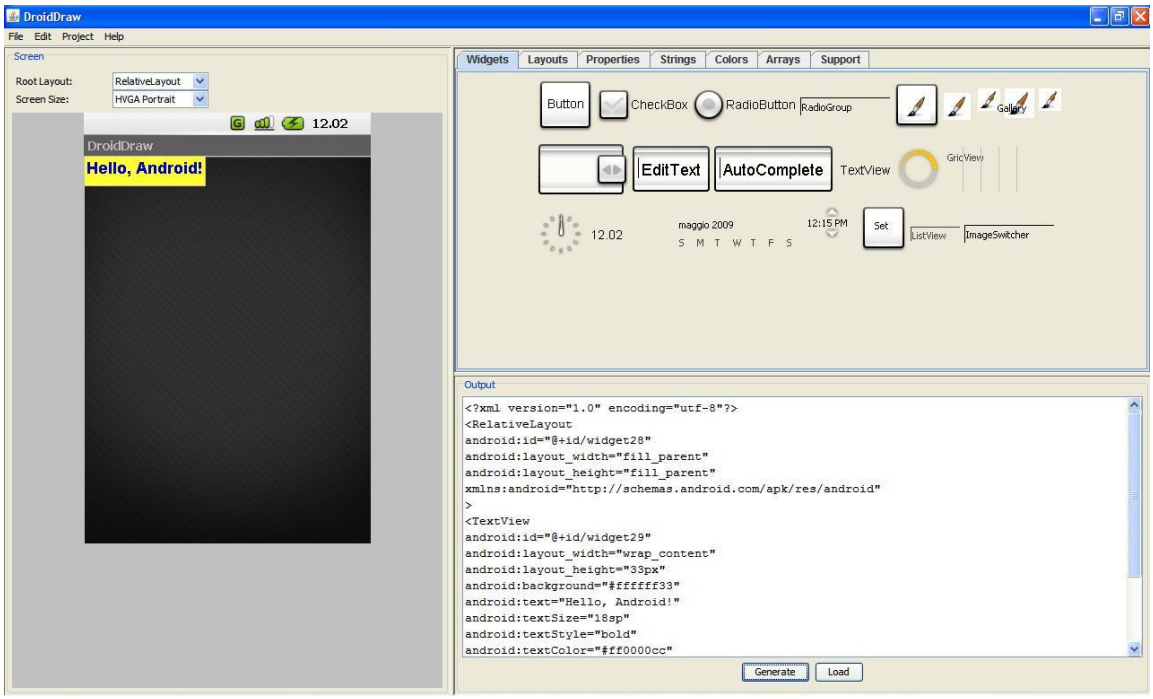

Figure C.3 – DroidDraw tool for automatic GUI generation

This is instead the Run Configuration window for running a project.

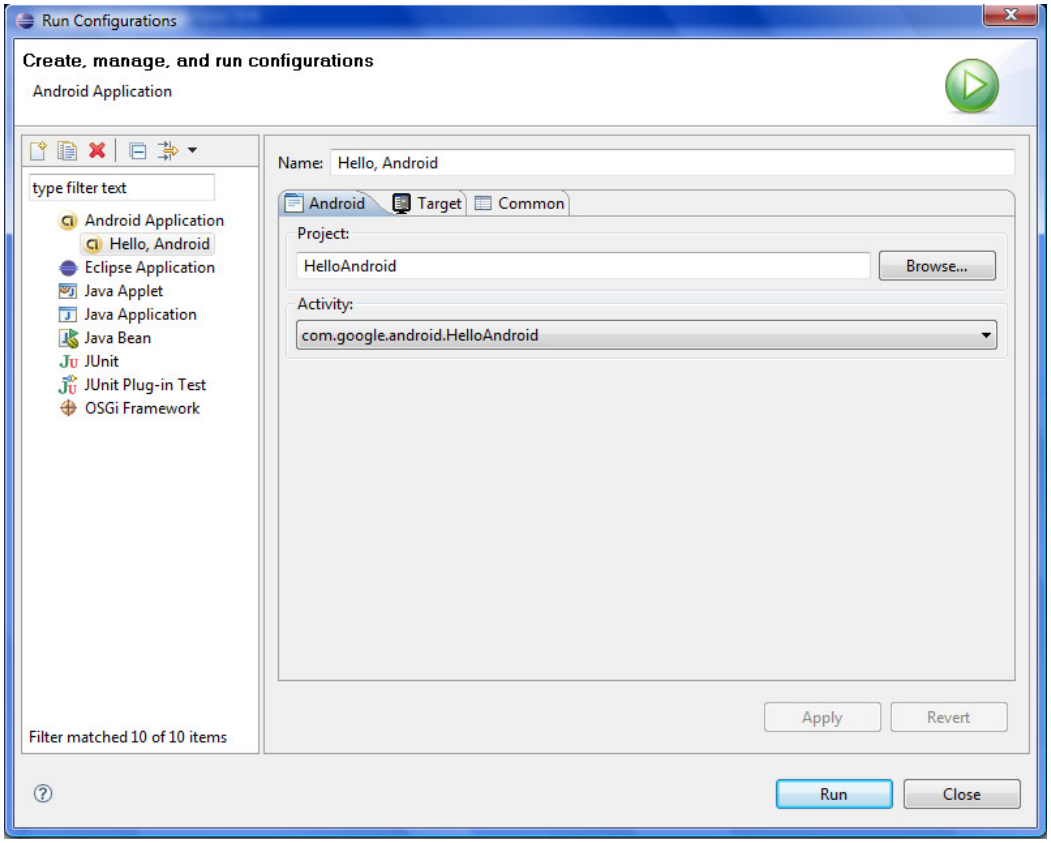

Figure C.4 – Run Configuration Window

With regard to the device emulator included in the SDK, we do not show it here as we have already seen its overall look and some screens in Chapter 5.

References

[1] Coulouris, Dollimore, Kindberg – Addison Wesley 2005 - "Distributed Systems Concepts and Design"

[2] Russo, Savy, Cotroneo, Sergio – Mc-Graw-Hill 2002 - "Introduzione a CORBA"

[3] Khoshafian – Auerbach 2007 – "Service Oriented Enterprises"

[4] Nagappan, Skoczylas, Sriganesh – Wiley 2003 - "Developing Java Web Services"

[5] Meier – Wiley 2009 – "Professional Android Application Development"

[6] Murphy – Commonsware 2009 – "The Busy Coder's Guide to Android Development"

[7] DiMarzio – McGraw-Hill 2008 – "Android, a Programmer's Guide"

[8] Haseman – Apress 2008 – "Android Essentials"

[9] Gramlich – http://andbook.anddev.org 2008 – "andbook – Android Programming"

[10] http://developer.android.com – official website for Android developers

[11] http://code.google.com/android - official Google page about Android

[12] http://groups.google.com/group/android-developers - Google group for Android developers

[13] http://www.anddev.org/ - community of Android developers

[14] Bornstein – Google Presentation 2008 – "Dalvik VM Internals"

[15] Panzuto – Master Thesis 2008 – "Progetto e sviluppo di un'architettura interoperabile e configurabile per l'accesso a reti di sensori"

[16] http://remoam.consorzio-cini.it/ - official website of the REMOAM research project

[17] Mindfire Solutions – White Paper 2001 – "Porting: a Development Primer"

[18] Pressman – McGraw-Hill 2000 – "Software Engineering, a Practitioner's Approach"

Acknowledgements

I would like to thank my supervisor, Prof. Stefano Russo, and Eng. Marcello Cinque, who followed and supported me with great competence and helpfulness during this work.

Thank you also to Prof. Domenico Cotroneo, who put me in contact with my supervisor in Glasgow and then with the Mobilab Research Group, letting me begin this thesis.

Thanks to Prof. Sotirios Terzis and Prof. Duncan Smeed of the University of Strathclyde in Glasgow, where I studied the Android platform during my Erasmus exchange period: an amazing experience that I will never forget.