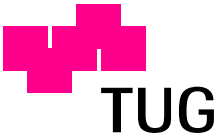
Seminar Paper

J2ME's Communication Facilities in Theory and Practice

Matthias Kerstner
matthias.kerstner@student.tugraz.at

Graz University of Technology



Adviser: DI Dr. Victor Manuel García-Barrios

Graz, September 2008

Abstract:

Java™has become a very powerful platform-independent programming language. Java Micro Edition (J2ME) is finally a version that offers a small enough footprint able to run on very resource limited mobile devices, such as mobile phones or PDAs. This paper discusses J2ME's technical specifications and limitations compared to Java Standard Edition (J2SE) on one hand, as well as its built-in communication facilities, provided by the Generic Connection Framework (GCF) on the other. By providing reference solutions, which target known issues for network execution code, Nautiport's flexible connection manager class will be presented.

Keywords Java™, Java Micro Edition, J2ME, Generic Connection Framework, Mobile Devices, Mobile Communication

Contents

1 Introduction **3**

2 Java Micro Edition And Connectivity Provided **5**
2.1 J2ME Overview 6
2.1.1 Target Devices 6
2.1.2 Architecture 7
2.1.3 Configuration vs. Profile 8
2.2 J2ME Advantages and Limitations 10
2.2.1 Configuration, Profiles and Modularity 11
2.2.2 Limitations 14
2.3 Generic Connection Framework 16
2.3.1 General Overview 16
2.3.2 URL Scheme 17
2.3.3 Mobile Network and TCP 18
2.3.4 HTTPS 18
2.4 Summary 19

3 J2ME In Practice **20**
3.1 A Multi-Device Dive Data Managing Solution 20
3.1.1 Goals and Requirements 22
3.1.2 Architecture 23
3.2 CommunicationManager 26
3.2.1 Supported Communication Endpoints 26
3.2.2 Connectivity 27
3.2.3 Concurrency 29
3.2.4 Synchronization 31
3.2.5 Security 32

4 Conclusions and Outlook **34**

Chapter 1

Introduction

> "They are such great gadgets – with a cell phone, you can talk to anyone on the planet from just about anywhere!" [LBT08]

The remarkable growth of the sector of mobile devices, such as PDAs, handhelds, smartphones and mobile phones, has been deserving great attention over the last years. Due to the ongoing technological advances concerning miniaturization and wireless network capacities/capabilities, mobile devices have become a viable alternative to standard PCs [YN01]. Whereas for instance only 11 million mobile phones existed back in 1990, an astonishing amount of 2.5 billion devices are registered in 2008 [Eco07].

Simultaneously with their increasing amount, also their technical capabilities evolved. For instance, mobile devices are now capable of connecting to different networks, such as "mobile cellular-" (GSM, UMTS) on one hand, as well as "wireless data networks" (IEEE 802.11[1]) on the other. With the continuously increasing variety in hard- and software for mobile devices, the mobile market segment developed a high rate of heterogeneity over the time [MTH+05]. Consequently, mobile device application developers are confronted with the problem of choosing which platforms to support, while always trying to keep portability as high as possible.

This paper addresses the aforementioned problem by introducing Java's Micro Edition (**J2ME**), that was specifically targeted at mobile devices, such as handhelds, PDAs, smart- and mobile-phones. The main aim of this paper is to show why J2ME is especially suitable for mobile devices. Thereby a two step methodology has been chosen. Firstly, the next chapter deals with the main general issues about J2ME itself, i.e. Chapter 2 gives a general overview of J2ME, including target devices, its architecture and J2ME's foundations (e.g. the **configuration** and the overlying **profile**). Based on this overview, this papers elaborates on J2ME's advantages, while also providing a distinct set of limitations compared to Java's Standard Edition. And second, Chapter 3 presents a reference implementation of a simple communication class (**MIDlet** Suite).

From the technical point of view, this paper focuses on the communication aspects of mobile devices, using standard out-of-the-box means offered by J2ME in form of the Generic Connection Framework (**GCF**). A general description of the GCF will be presented

[1] see `http://ieee802.org/11/`

in Section 2.3, including aspects such as its underlying URL scheme, mobile network communication specific limitations as well as security considerations when dealing with network communication.

Through the presentation of a concrete sample implementation, called **Nautiport** (see Chapter 3), this paper tries to demonstrate how the problem of heterogeneity can be solved using the **GCF**. On the one hand, the proposed solution within Nautiport, enables mobile devices accessing the Web through a general communication class. On the other hand, a mechanism to communicate with other local devices over a serial, infrared (IrDa) or Bluetooth™(BT) connection is provided through the **Communication Manager** class from **Nautiport**[2]. Upon explaining Nautiport's general **goals**, **requirements** and **architecture**, this paper further elaborates on the Communication Manager class in more detail. Thus, the focus is set on three distinct topics, including an overview of the Communication Manager's **synchronization/upload mechanisms** on the one hand, dealing with **concurrency** issues on the other and finally, explaining its means of **connectivity**, which can be separated into "local device-to-device-" and "remote device-to-Web" connections. Furthermore, also data- and communication **security** is discussed.

The last chapter of this paper summarizes J2ME's advantages as a platform for mobile devices, describes results gathered from implementing the Communication Manager class, as well as gives an outlook of future work needed to further improve it.

[2]http://www.kerstner.at/nautiport/

Chapter 2

Java Micro Edition And Connectivity Provided

Although capacities, as well as technological capabilities of mobile devices have tremendously increased over the last years, they still provide limited hard- and software resources, as compared to standard PCs. Therefore, when dealing with these resource limited mobile devices, it is crucial to keep the applications' footprint as small as possible, while still trying to maintain a high level of usability. Due to hard- and software limitations, application developers are often confronted with a very limited amount of APIs to choose from. For example, whereas Java's Standard Edition by default consists of 8 fundamental packages[1], the much lighter Micro Edition only covers 4 distinct types[2] [SM06b] [SM06a]. In addition to this problem, mobile device producers often tend to provide proprietary APIs, thus drastically restricting portability. Furthermore, due to existing heterogeneity[3] in the mobile device segment, developing applications that are to be supported by a broad range of devices, such as mobile phones, PDAs and smartphones, becomes a tedious task.

With Java's Micro Edition (**J2ME**), an approach was started to provide mobile device application developers with a hardware-independent platform. Just like Java's Standard Edition, J2ME represents a virtual machine providing platform-independent APIs that are specifically targeted at resource limited mobile devices.

This chapter presents an overview of J2ME's cornerstones. These include target devices, its architecture and underlying foundation. In addition, J2ME's advantages and limitations will be shown, particularly compared with Java Standard Edition (J2SE). In order to focus on J2ME's means of protocol-independent communication, the last part of this chapter is dedicated to the Generic Connection Framework (**GCF**). Thereby, its underlying flexible architecture as well as the protocol-dependent URL scheme are discussed. Based upon the information provided by this chapter, an approach towards a communication class will be presented in Chapter 3.

[1] i.e. base, graphics, IO, network/Web, text, components, data/SQL and XML
[2] i.e. base, IO, generic connections and utilities
[3] i.e. underlying hardware, operating systems used, etc.

2.1 J2ME Overview

Sun Microsystems[4] defines **J2ME** as

> "a highly optimized Java run-time environment targeting a wide range of consumer products, including pagers, cellular phones, screen-phones, digital set-top boxes and car navigation systems." [JG01]

Since its initial release in 1996 the Java™platform has tremendously increased in functionality. As a consequence, it also increased in its source code size and hardware demands. Due to this fact, Sun Microsystems announced in June 1999 to split its platform into three distinct editions [Sch07]:

- Java 2 Enterprise Edition (**J2EE**)[5]
- Java 2 Standard Edition (**J2SE**)[6]
- Java 2 Micro Edition (**J2ME**)[7]

Each edition targets a special audience. Whereas the Enterprise Edition (J2EE) is meant for server-side applications, the Standard Edition (J2SE) is suitable for desktop solutions. In contrast to these two editions, the Java Micro Edition (J2ME) exclusively targets (very) resource-limited mobile devices.

Consequently, much effort was spent to keep J2ME as light-weighted as possible, while still trying to keep it powerful, compatible to other editions and last but not least, easy to use. The result is an edition with a very small footprint and the least amount of APIs compared to the other two more potent editions.

2.1.1 Target Devices

According to [Sch07], there exist two categories of J2ME-supported devices that can be distinguished by their type of usage, technological capabilities and capacities:

- Personal, mobile, networked
- Shared, stationary, networked devices

Whereas the second category covers devices that offer more advanced UIs and memory capacities (like set-to-top boxes, video-telephony or navigation systems), the first one includes mobile telephones, PDAs or even smart-phones. These are very limited concerning their memory (RAM and ROM), UI and often also their networking support. Therefore, successful middleware systems or applications targeting mobile devices are often kept very "slim", as discussed in Section 2.1.3.

Figure 2.1 shows target device categories for J2ME by comparing it to the other editions. In accordance to the figure, it is highly relevant to highlight at this point that the devices

[4] http://www.sun.com
[5] http://java.sun.com/javaee/
[6] http://java.sun.com/javase/
[7] http://java.sun.com/javame/

targeted by J2ME depend on the underlying **configuration**[8], which is going to be discussed in more detail in Section 2.1.2.

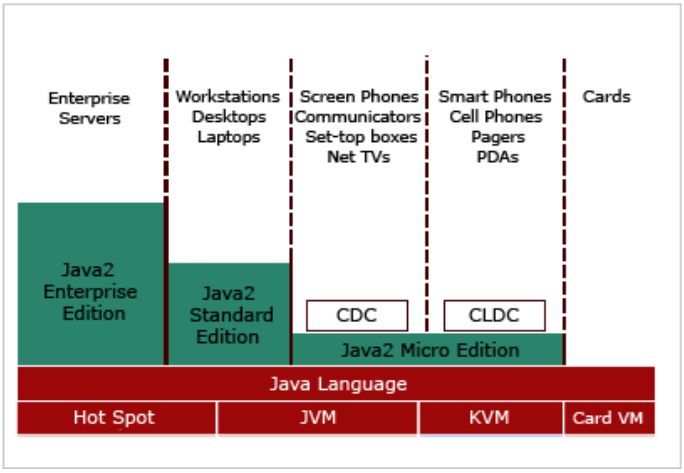


Figure 2.1: Target Device Audience For Java Editions Available [Roe08]

2.1.2 Architecture

> "A platform for mobile phones, PDAs, set-top boxes, and vehicle telematics systems." [SM08]

Figure 2.2 on the next page gives an overview of the typical component relationship in the software stack of mobile devices. Starting at the lowest layer is the native operating system (OS), such as Microsoft Windows Mobile[9], or Symbian[10], providing an abstract interface to the device. The so-called "Connect Device Configuration" (**CDC**), which represents J2ME's software basis, resides directly on the native OS layer. The CDC and its respectively more compact version "Connected Limited Device Configuration" (**CLDC**) are usually referred to as the **configuration**. On top of the CDC/CLDC lies the "Mobile Information Device Profile" (**MIDP**) - or simply referred to as the **profile**, as well as further mandatory and optional packages and classes. Finally, there are several so-called **MIDlet**-Suites and OEM applications running on top.

Thus, Figure 2.2 shows that J2ME consists of three basic layers [Sch07]:

1. Configuration: CDC/CLDC
2. Profile: MIDP, including optional packages and classes
3. MIDlet-Suites and OEM applications

[8] illustrated as "CDC" and "CLDC" respectively in Figure 2.1
[9] http://www.microsoft.com/Windowsmobile/default.mspx
[10] http://www.symbian.com/

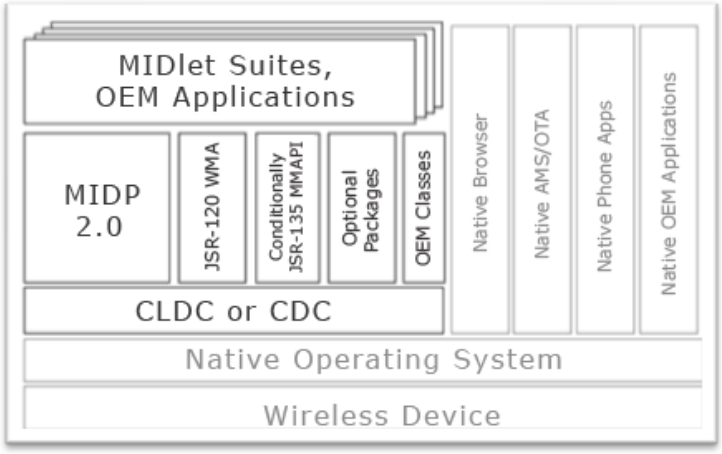


Figure 2.2: Typical Software Stack in Today's Mobile Devices [SM03]

The ensuing section explains the first two layers in more detail, thereby providing a basis for a presentation of a sample MIDlet implementation later on in Chapter 3.

2.1.3 Configuration vs. Profile

> "Recognizing that one size does not fit all, J2ME has been carefully designed to strike a balance between portability and usability." [YL02]

As mentioned in Section 2.1.2, J2ME basically consists of a **three-layer architecture**. On the very bottom is the **configuration**, represented by the **CDC** and **CLDC** respectively, that provides the upper two layers with a virtual machine (VM) and the core class libraries, to support a wide range of consumer products [SM03].

Depending on whether the CDC or CLDC is used, the class library and the VM used will be more or less powerful, as shown in Figure 2.1. Whereas the CDC uses the complete original Java Virtual Machine (JVM), the CLDC uses the much smaller, API-limited Kilobyte Virtual Machine (**KVM**). Therefore, the CLDC incorporates the most basic set of libraries and Java VM features required for J2ME implementations on highly resource limited devices. CDC on the other hand has far more advanced security, mathematical, and I/O functions [YL02]. On top of the configuration there is the so-called **profile** that defines the application's features by adding domain and device specific API libraries [JG01].

In summary, J2ME uses configurations and profiles to customize the underlying runtime environment by providing applications with a virtual machine, as well as core and additional libraries.

CDC vs. CLDC

As previously mentioned in this section, J2ME poses some basic hardware requirements that must be met by devices in order to be able to run applications. These requirements are

basically set by J2ME's foundation - the CDC and CLDC respectively. Figure 2.3 shows, that the more advanced CDC requires at least 2MB of memory and a 32-Bit processor, whereas the CLDC still runs on devices with only 128-512KB memory and a 16-Bit CPU.

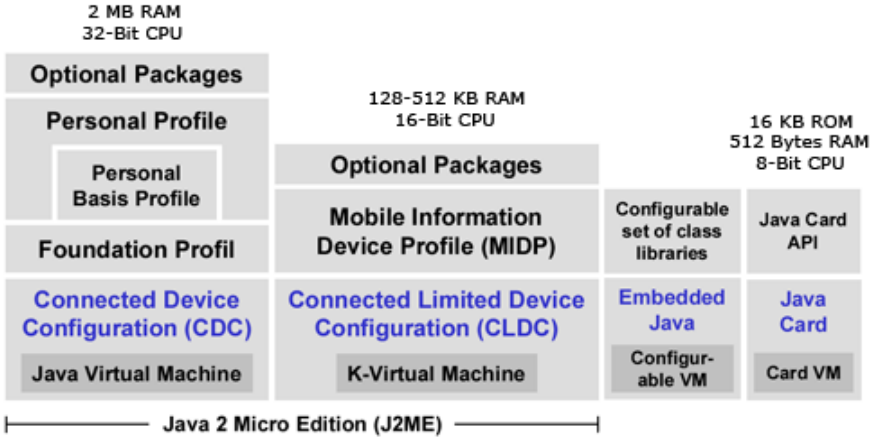


Figure 2.3: J2ME Hardware Requirements [Roe08]

According to [Sch07], hardware requirements can be split into the following categories:

- Memory capacity (RAM/ROM)
- Display
- Input Capabilities
- Network Connectivity
- Multimedia options

Furthermore, [JG01] states that target devices for J2ME applications developed using CLDC generally have the following hardware characteristics:

- 160 to 512 KB of total memory available for the Java™platform
- Limited power, often battery powered
- Network connectivity, often with a wireless, inconsistent connection and with limited bandwidth
- User interfaces with varying degrees of sophistication, sometimes with no interface at all

Compared to CDC, devices must additionally meet these requirements [JG01]:

- 32-bit processor

- 2MB or more of total memory available for the Java platform
- Full functionality of the Java 2 "Blue Book" VM

Furthermore, hardware requirements change depending on the configuration and profile version used. Note that 128KB ROM are enough for MIDP 1.0; version 2.0 already requires the double amount. Therefore, it is crucial for application developers to know the target device's hardware specifications in advance.

Profile

Profiles pose J2ME's second of the three layer architecture and can be seen as a complement of CDC/CLDC's core class libraries [Sch07]. Besides the PDA profile, the MIDP is the most important profile for CLDC, and thus, is implemented on the majority of mobile phones [YL02]. At the time of this writing, there are two different versions of the MIDP available, each compatible with the underlying CLDC 1.0 and 1.1 respectively. On the other hand, the Foundation and the Personal profile are the most significant profiles for the CDC, as shown in Figure 2.3.

Together with the configuration, profiles provide developers with a rather extensive API. The MIDP for example has according to [Sch07] its main focus on the following categories:

- Controlling the application's life-time-cycle
- UI
- 2D games
- Media Processing
- Communication
- Administration of persistent data

The functionalities for these categories are provided in form of the well-known Java packages, such as *java.lang*, *java.util*, *javax.microedition.io*, *javax.microedition.lcdui* or *javax.microedition.rms*. Whereas some of Java's core packages, such as *java.lang* and *java.util* are available out-of-the-box, others are missing due to the lack of processing power and memory of targeted devices. Nevertheless, MIDP does not only define new classes, it also extends existing ones. For instance, MIDP adds the *IllegalStateException* to the *java.lang* package since *java.util.Timer* can throw it.

2.2 J2ME Advantages and Limitations

Apart from the fact that J2ME was specifically designed for resource limited mobile devices, while still trying to maintain the flexibility and portability of the "classic" J2SE, what makes J2ME so powerful is its three layer architecture, consisting of **configuration**, **profile**, and applications running on top, is . Nevertheless, in order to provide a platform with an ultimate small footprint, various cutbacks had to be made. This section describes J2ME's main advantages on the one hand, as well as its limitations compared to J2SE on the other.

2.2.1 Configuration, Profiles and Modularity

J2ME's foundation - CDC and CLDC respectively - represents one of the most crucial parts in J2ME and has been developed and standardized during the Java Community Process (JCP[11]). The idea behind the JCP is to provide all participating members with

> "a process to develop and revise Java™ technology specifications, reference implementations, and test suites"[Pro08b]

Consequently, packages contained in the CDC/CLDC have been thoroughly tested and underwent the rather long-lasting JCP timeline, as shown in Figure 2.4. Furthermore, since a broad range of today's most successful mobile-phone producers and carriers alike are members of the JCP, much effort has been spent to find **common denominators**, in order to provide developers with device-independent APIs, thereby greatly enriching portability.

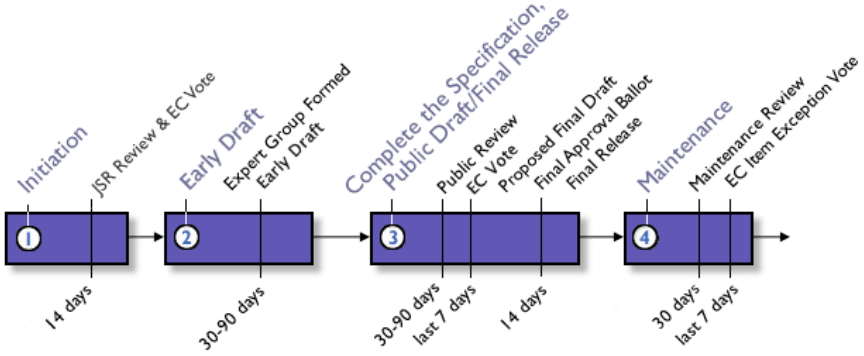


Figure 2.4: JCP Timeline [Pro08b]

Profile

As already mentioned in Section 2.1.2, profiles are built on top of the configuration. They extend the configuration's core classes by providing additional domain-specific packages and/or extending available ones. The idea behind profiles is to aim at a range of similar devices. Consequently, J2ME applications written for specific profiles, such as the MIDP, can be easily ported across all devices supporting this particular profile. The ultimate goal for profiles is to provide developers with a common, vendor-independent and industry-standard platform. For instance, according to [JG01] the MIDP is already a complete and well supported basis for developing mobile applications.

Modular Architecture

Due to J2ME's modular software architecture, it is actually very easy to define extra functionality in form of additional optional packages. As shown in Figure 2.5, J2ME offers a

[11] `http://www.jcp.org`

wide range of extra packages, thereby enhancing its base functionality by providing further domain-specific APIs. Depending on the configuration and profile used, a rather extensive list is available to choose from. Table 2.1 for instance gives a quick overview of packages optionally available using the CLDC/MIDP version displayed.

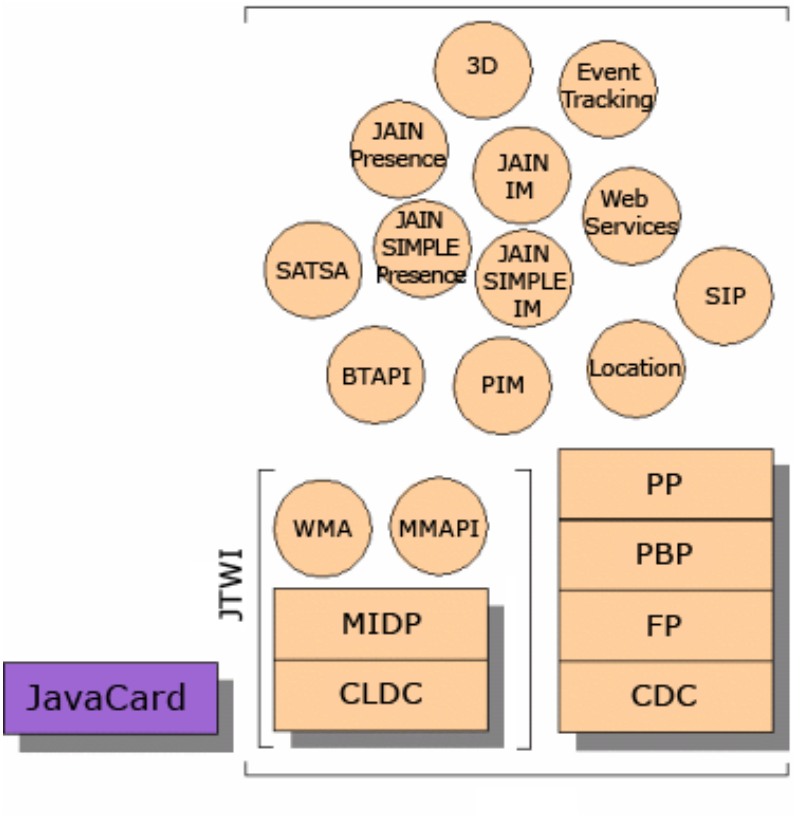


Figure 2.5: J2ME Modular Architecture [Roe08]

Since these optional APIs are designed through the JCP by a consortium of well-known end device producers[12], carriers[13], as well as platform developers[14] alike[15] software engineers are provided with consistent and powerful interfaces [Pro08a]. Although packages' APIs are ratified by JCP's participating members, implementing them still lies in the responsibility of numerous independent device producers. Consequently, the problem of **fragmentation** arises, since not all vendors will implement all packages at the same time, endangering J2ME's most important pillar - portability. Section 2.2.1 in the following elaborates on measures taken to fight fragmentation.

[12]e.g. Nokia, Motorola, Sony Ericsson, etc.
[13]e.g. Vodafone, T-Mobile, etc.
[14]e.g. Esmertec, Aplix, etc.
[15]i.e. "JCP Expert Group"

Package	CLDC	MIDP
Mobile Media API (JSR 135)	1.0	1.0
Bluetooth/OBEX for J2ME (JSR 82)	1.0	1.0
JAXP XML Parser (JSR 172)	1.0	1.0
Location API for J2ME (JSR 179)	1.1	1.0
SATSA-JCRMI (JSR 177)	1.0	1.0
SATSA-CRYPTO (JSR 177)	1.0	1.0
Mobile Internationalization API (JSR 238)	1.1	2.0
SIP API (JSR 180)	1.0	1.0
Scalable 2D Vector Graphics API (JSR 226) *	1.1	1.0
Wireless Messaging API 1.1 / 2.0 (JSR 120/205)	1.0	1.0
PDA Profile for J2ME (JSR 75)	1.0	1.0
Web Services API (JSR 172) *	1.0	1.0
Mobile 3D Graphics for J2ME (JSR 184)	1.1	1.0
SATSA-APDU (JSR 177)	1.0	1.0
SATSA-PKI (JSR 177)	1.0	1.0
Content Handler API (JSR 211)	1.1	2.0
Payment API (JSR 229)	1.1	2.0
Advanced Multimedia Supplements (JSR 234)	1.0	1.0
Java Binding for OpenGL ES (JSR 239)	1.1	2.0

Table 2.1: CLDC/MIDP Optional Packages Overview

Mobile Service Architecture

In first place, the Mobile Service Architecture (**MSA**[16]) represents JCP's approach to fight fragmentation caused by a wide spread of optional packages. Generally speaking, the MSA follows three underlying design goals [GR04]:

1. Minimize fragmentation by providing developers with a predictable as well as highly interoperable applicant and service environment
2. Broaden J2ME's usage on different markets as well as customer segments by introducing two different platform definitions: MSA and MSA Subset
3. Ensure highest possible level of consistency in MSA/MSA Subset's and MSA Advanced environment's definition

As it turns out, what originally caused a problem, lead to clearer package specifications, by minimizing the room for interpretations. Furthermore, due to MSA's mandatory components, such as the CLDC 1.1, MIDP 2.1, JTWI[17] 1.0 and various optional packages, application developers now have a much broader range of programming interfaces to choose from. Although MSA's final public version dates back to 2003, the first device to fully support MSA - Sony-Ericsson's Z750 - was not released until 2007 [SEMC08].

[16] see `http://jcp.org/en/jsr/detail?id=248`

[17] Java Technology for the Wireless Industry (JTWI), JSR 185, `http://java.sun.com/products/jtwi/`

2.2.2 Limitations

Following J2ME's general overview and advantages over other platforms, time has come to explain existing limitations. In the course of this section, feature restrictions will be discussed by comparing J2ME's two types of configurations - CDC and CLDC respectively.

Whereas the CDC incorporates a fully functional Java virtual machine, the CLDC has to cope with the feature limited **KVM**[18]. Therefore there are no particular limitations posed on the CDC. The KVM on the other hand basically is a limited version of the original Java VM (JVM). It fulfills all JVM's specifications[19] except those listed in the CLDC reference. Consequently, the CLDC differs from the CDC through a set of limitations.

The following list by [Sch07] enumerates all limitations posed on the KVM using CLDC 1.0:

- No floating-point support
  - CLDC devices often don't have floating point hardware and in most cases software emulation would be too memory and CPU intense
- No Java Native Interface (JNI) support
- No user-defined class loader
  - KVM has a built-in class loader
- No support for reflection
  - Including serialization and RMI
- No support for thread-groups or daemon-threads
  - Must be implemented using single threads
- No finalization
  - Garbage collection exists though
- No weak-references
- Limited exception handling
  - The CLDC defines that errors must be handled by the application or exceptions

Apart from the above listed limitations, application developers have to cope with further restrictions. These are going to be discussed in the following sub sections.

[18] refer to 2.1.3

[19] see `http://java.sun.com/docs/books/jvms/`

No Shared Libraries

Another of CLDC's crucial restriction is the lacking support for shared libraries. Whereas MIDlets of the same MIDlet-Suite can share classes, MIDP's specification prohibits sharing among other suites, unless they are provided by the OEM and already reside on the platform [Cor04]. Consequently, possibly shareable code that would normally be swapped out to libraries must be individually deployed in MIDlet-Suites, thus enlarging them.

Minimal Control over High-level UI Layout

In order to provide a maximum level of interoperability, visual alignment of high-level components cannot be controlled programmatically [Cor04]. Hereby, visual components are restricted to be vertically aligned on the display, each always placed in a new line. Therefore it is impossible to create table-like layouts, thus severely narrowing down support for designing more complex UIs. As a consequence, application developers are forced to use low-level UI components, in order to design custom layouts.

No Shared Data

Although support for persistent data is provided by J2ME's RecordManagementSystem (RMS), using MIDP 1.0 stored data can only be shared among MIDlets contained in a specific MIDlet-Suite. With the advent of MIDP 2.0, this restriction partly became obsolete, as MIDlets are now able to access stores of foreign suites, in case proper rights have been previously assigned [Sch07]. Nevertheless, since there still exist mobile devices that do not support MIDP 2.0 applications, targeting a broad range of devices might be forced to fall back to other solutions, in order to share data.

Compiler Asymmetry

Due to the above mentioned limitations, byte code produced by J2SE's *javac* may not run inside the KVM. There are two options to choose from:

1. Create a separate CLDC compiler that produces "valid" byte-code by rejecting all restricted commands from the source code
2. Use the default J2SE compiler *javac* and cope with any extra (invalid) byte-code

In practice, developers normally go for the second option by accepting the asymmetry between the powerful J2SE compiler and the feature limited KVM [Sch07]. Any additional, restricted byte-code produced by *javac* is then rejected by J2ME's byte-code verificator during the two-phase byte-code verification process.

The remainder of this chapter is dedicated to network communication using J2ME's Generic Connection Framework (**GCF**). Section 2.3 will discuss the GCF in detail by elaborating on its architecture and built-in means for network related programming. This will provide a solid foundation, in order to present an approach towards a communication class in Chapter 3.

2.3 Generic Connection Framework

J2ME was originally designed to be used in a broad range of mobile devices ranging from simple mobile phones to high-end Pocket PCs and smart-phones. Nonetheless, due to their heterogeneity these devices generally differ in their hardware equipment, as well as their scale of functionality. According to [Sch07], they all share two essential characteristics. Firstly, they all are by definition designed for mobile usage, and secondly, they enable (network) communication with other mobile devices, or even full-blown computer systems, through what by now is a feasible set of network protocols.

Over the time and due to ongoing research for faster and smaller hardware components, support for protocols of the TCP/IP family increased. This trend becomes obvious when looking at J2ME's specifications. What seemed a bit utopist some time ago is now already an integral part of the **GCF - network protocol independent APIs**.

2.3.1 General Overview

[Ort03] defines the **GCF** as

> "a straightforward hierarchy of interfaces and classes to create connections (such as HTTP, datagram, or streams) and perform I/O."

Although J2ME's primary reference is J2SE, a closer look at the GCF reveals that when it comes to network communication a break in this tradition has been made. According to [Sch07], in the course of designing the CLDC and MIDP, developers realized that J2SE's network APIs in *java.net* were too extensive to be used on resource limited mobile devices. Consequently, a different approach had to be taken, resulting in the GCF.

The basic idea behind the GCF is to provide a **generic approach to connectivity** by using a common, device-independent foundation API [Ort03]. Hereby **three pillars** had to be implemented:

1. An extensible interface hierarchy
2. A connection factory
3. Standard Uniform Resource Locators indicating the type of connection to use

As illustrated in Figure 2.6, GCF's base is made up of the *Connection*-interface. It provides the primary interface for a transmission protocol independent connection. There exists a separate interface, extending *Connection*, for each protocol. In addition to this connection interface hierarchy, the GCF provides the *Connector*-class and the *ConnectionNotFoundException*. The Connector-class serves as a factory to generate connection-objects, hence the name "*Connection-Factory*".

One of GCF's main benefits is its design as a **lowest-common-denominator framework** [Ort03]. That mentioned, it is to say that due to its flexibility and extensibility the GCF is used across J2ME profiles, as well as optional packages and now even on the J2SE platform itself. It is therefore also possible to define new connection types. [Ort03] lists three tasks to do so:

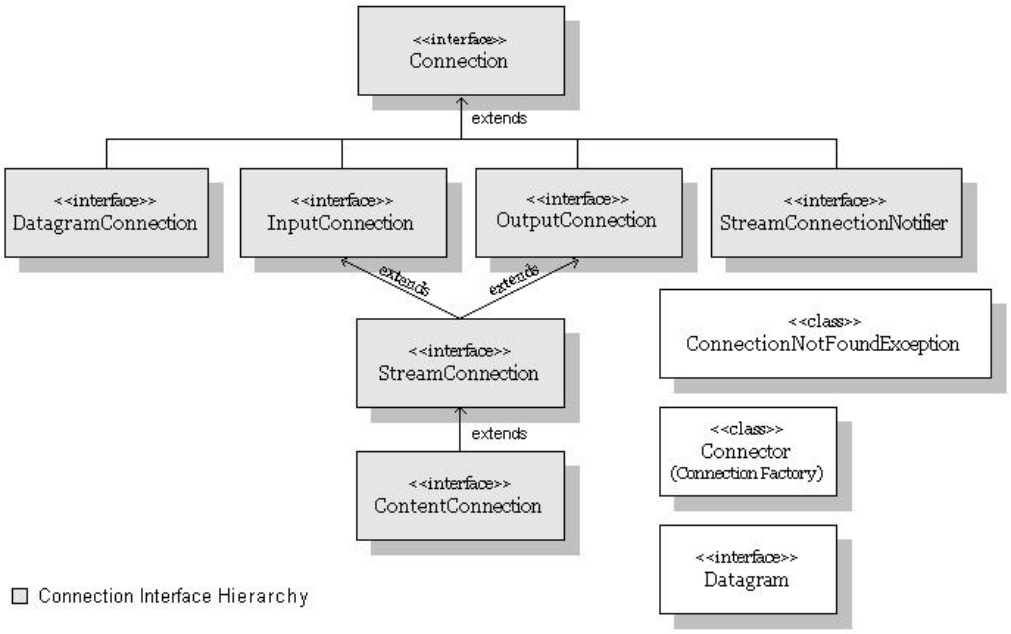


Figure 2.6: GCF Connection Interface Hierarchy and Related Classes [Ort03]

1. Sub-type the Connection-interface and providing the required support classes
2. Provide the Connector factory class to generate the connection-object
3. Define new URL scheme that identifies the new connection type

2.3.2 URL Scheme

Connection types are determined by their URL schemes and generally have the following form:

```
scheme:target[;parameters]
```

"Scheme" defines the protocol, "target" the protocol-dependent network address and "parameters" any optional connection-specific settings using the form

```
name1=value1;name2=value2;...
```

Table 2.2 lists all network protocols with their corresponding URL scheme that are currently available using MIDP 2.0. Thereby, it is again remarkable to see that the GCF provides an API that can be applied to all connections, independent from the underlying communication protocol.

Url Scheme	Protocol
socket://host:port	TCP
socket://port	TCP
ssl://host:port	SSL/TLS
datagram://host:port	UDP
http://host:port/file?query	HTTP
https://host:port/file?query	HTTPS
comm://port;parameters	Serial

Table 2.2: MIDP 2.0 Protocol URL Schemes

2.3.3 Mobile Network and TCP

Due to the fact that mobile networks have a different architecture compared to classic computer networks, connections using the TCP protocol, such as HTTP connections, are inefficient [Sch07]. TCP's cornerstones like flow-control, three-way handshake, error handling and delay timing, naturally conflict with the comparatively high error rates and latency in mobile networks [Wil01], [Hol02].

Consequently, numerous mobile phone models don't even have a TCP/IP stack implemented. Instead, they rely on protocols that were specifically designed for mobile networks, such as the Wireless Session Protocol (WSP) [Sch07]. Even though numerous implementations were built upon TCP, HTTP for instance is independent from underlying transport protocols. Therefore HTTP connections can also be transported over WSP, for example. In mobile networks HTTP-communication is managed by intermediate WAP-gateways that translate WAP-specific protocol commands to target TCP/IP systems.

Although TCP's counterpart UDP is contained in MIDP 2.0's specification (JSR-118[20]), it is marked as optional [Pro02]. Consequently, support for UDP may not be implemented on all mobile devices using MIDP as a profile. As it will be shown in Chapter 3, HTTP connections can be easily established using J2ME's out-of-the-box means provided by the GCF. On the other hand, due to restrictions posed on UDP support in MIDP 2.0, applications relying on this protocol might be forced to use external libraries, thereby raising the problem of MIDP 1.0's lacking support for shared libraries, as previously discussed in Section 2.2.2.

2.3.4 HTTPS

When it comes to network communication, security is always a hot topic. Starting with MIDP 2.0, several additional requirements concerning communication security have been introduced. Whereas Transport Layer Security (TLS) and Secure Socket Layer (SSL) are optional, the corresponding JSR-118[21] defines that support for HTTPS is mandatory.

This might seem a bit strange since HTTPS connections either use TLS or SSL to secure communication channels. But as already mentioned, intermediate WAP-gateways act as translators between devices lacking a TCP/IP stack and fully functional network devices, such as Web servers. Hereby, Chapter 3 will show how secure HTTPS communication be

[20] see `http://jcp.org/en/jsr/detail?id=118`
[21] see Section 2.3.3

established by using GCF's built-in mechanisms.

2.4 Summary

This chapter commenced by giving a general overview of J2ME's target device audience and of its **three-layer architecture**, as well as an in-depth comparison of the underlying **configuration** and **profile**. Thereby, J2ME's flexible and modular three-layer architecture has proven to perfectly fit today's mobile devices' software-stack. It was shown that the configuration represents the bottom-most of the three layers, that provides the upper two with a virtual machine, core libraries and classes. The profile on the other hand enriches the already existing set of APIs by providing domain-specific ones. Additional functionality is available through a broad range of optional packages. By elaborating on J2ME's **advantages**, also its **limitations** compared to the J2SE have been discussed. Hereby, topics such as **fragmentation** and measures taken to fight it have been dealt with.

The second part of this chapter was dedicated to J2ME's Generic Connection Framework (**GFC**), that represents a well-designed and powerful interface for dealing with different types of connections, such as HTTP, datagrams or even streams. By enumerating GCF's **three pillars**, its interface hierarchy and related classes were discussed. Furthermore, it was briefly shown how additional connection types can be implemented, using GCF's flexible and extensible architecture. Additionally, GCF's **URL scheme** was presented to give an overview of the broad range of possible connection types. Moreover, currently existing limitations concerning TCP communication were discussed and possible solutions in form of WAP gateways specified. Finally, also means of secure communication were briefly discussed by defining MIDP 2.0's additional security constraints. Concluding, this chapter provided the required background information for a better understanding of the topics addressed in the remainder of this paper. Chapter 3 will present an approach towards a communication class using J2ME's built-in facilities.

Chapter 3

J2ME In Practice

Chapter 2 outlined **J2ME**'s technical specifications, thus providing a solid understanding of its inner workings. On the one hand, its target device audience and hardware requirements were discussed, and on the other an in-depth analysis of its advantages and limitations, especially compared to J2SE, was carried out. Thus, based on the findings of the previous chapter, this one specializes on presenting a practical solution built with J2ME, namely "**Nautiport**". Starting with a general description of Nautiport in Section 3.1, including goals, requirements, as well as its architecture, the second part of this chapter elaborates on Nautiport's **Communication manager** class. Targeting at the main context of this paper - i.e. heterogeneity in mobile applications - the focus is set on critical technical aspects of this communication-oriented solution: data **synchronization**, **concurrency**, **connectivity** and **security**.

3.1 A Multi-Device Dive Data Managing Solution

Examining dive data after dive sessions is a crucial task. Whereas hobby divers use collected data (among others) to better plan future sessions, more advanced divers try for example to optimize their dive routes in order to maximize dive ranges. Although many divers still use hand written dive logs, using for example the form by PADI[1] shown in Figure 3.1, numerous digital dive computer models exist, that eliminate the tedious process of manual record keeping during dive sessions. Data logged by these dive computers, such as diving depth, tank volume or the logging interval, can usually be specified in advance, thus giving divers the freedom to further personalize information to be gathered.

Although special PC software is normally provided by dive computer vendors[2], bulky PCs or even laptops are usually too much of a burden to carry around at dive spots, left alone the fact that power plugs are not always available. As a consequence, divers are often forced to manually import data from paper-based dive logs into software applications, in order to be able to efficiently analyze dive sessions.

[1] see `http://www.padi.com`

[2] e.g. Suunto Dive Manager, `http://www.suunto.com`

Furthermore, the process of **mutual signing** between buddies after dive sessions can also become quite complicated in larger groups. Ultimately, members of dive sessions might want to share raw data or further information about experiences gathered. A possibility of divers capable to add the entire staff of buddies from dive sessions would not only save organizational time afterwards, it would also provide sophisticated means of **effectively sharing data** gathered amongst them.

At this point, compact mobile devices come into play. Modern mobile devices, ranging from simple mobile phones over smart phones to PDAs, have become quite powerful over the years. As discussed in Chapter 2, with **J2ME** a widespread platform is available that is especially targeted at these resource limited devices. Therefore, one of the simplest ways to exchange and examine data on dive computers is to connect them to mobile devices running some special software.

Figure 3.1: PADI Paper Dive Log [Cra07]

Nautiport represents an approach for such a software, which retrieves dive data stored on dive computers and imports it into mobile devices. There it can be further examined or

even exported to online services, thus providing a platform for all buddies that participated in a dive session. The next sections will present Nautiport's features in more detail. Starting with an overview of Nautiport's goals and requirements in Section 3.1.1, its architecture will be discussed in Section 3.1.2. Based on this information, Nautiport's core means of communication, the **Communication Manager** class, will be presented later in Section 3.2.

3.1.1 Goals and Requirements

Nautiport is a **MIDlet**-suite written in Java, especially optimized for very resource limited devices, using the **CLDC**[3]. It represents a sophisticated but easy to use solution for **retrieving and managing dives "off-line" on mobile devices** [Ker08]. Hereby, Nautiport enables users to quickly add, update, delete, import and export dives directly from their mobile device. Additionally, it provides an interface to easily upload any existing dives to a **Web platform**[4], where they can be shared with buddies "all over the globe".

Due to its target device audience, Nautiport was intentionally kept very simple and straight forward. Great attention was drawn to device **compatibility** and **portability**. Although J2ME offers a broad range of optional packages, as previously discussed in Chapter 2, in practice numerous device vendors do not provide support for them, thus limiting compatibility for applications using them. Consequently, in order to maximize compatibility for the target device audience, Nautiport currently only uses a single optional package, the so-called Java APIs for Bluetooth (**JABTW**[5]), for Bluetooth™communication.

Furthermore, also user interface issues may arise when using J2ME's low level components. Consequently, Nautiport's UI entirely consists of so-called high-level components of the lowest common denominator user interface (**LCDUI**). Its menu navigation was basically designed using the **zoom-on-demand** principle. It is structured in cascading menus, starting from the main screen and going in deeper, as the user requests more details. Figure 3.2 on the next page shows Nautiport's main screen, which has been built with Sun's Java Wireless Toolkit[6].

Unfortunately, one drawback when using LCDUI components is their very **limited layout control**, as previously discussed in Section 2.2.2. Nevertheless, due to the fact that Nautiport's UI was intentionally kept very simple, LCDUI's present means of controlling existing components are sufficient. Nonetheless, as Nautiport's development continues future versions might still offer a more advanced UI, as improvements always can be made.

In order to provide a device independent storage of dive data, as well as personal settings, Nautiport uses an adaption of J2ME's built-in Record Management System (**RMS**). Since Nautiport is a self-contained MIDlet-suite and does not share data among other suites, known issues addressed in Section 2.2.2, can be left aside. Although, Nautiport supports the storage of data through XML files on mobile devices, this feature is currently not used since manipulating files and directories using J2ME requires an additional optional package - the FileConnection package (JSR-75[7]).

[3] see Section 3.1.2
[4] see Chapter "The Web Platform" in [Ker08]
[5] see `http://jcp.org/en/jsr/detail?id=82`
[6] see `http://java.sun.com/products/sjwtoolkit/`
[7] see `http://jcp.org/en/jsr/detail?id=75`

Figure 3.2: Nautiport's Main Screen

As already mentioned, Nautiport targets at resource limited mobile devices using the CLDC. Concretely speaking, Nautiport runs on all mobile devices supporting **CLDC** 1.0 and **MIDP** 2.0[8]. Although it does not pose any special hardware requirements to be able to run[9], dive data retrieval from dive computers requires the mobile device to provide some sort of communication endpoint. Possible choices presently available are serial, infrared (**IrDa**) or Bluetooth™(**BT**) connections. Furthermore, in order to be able to upload locally stored dive data to remote Web services[10], mobile devices must additionally be able to access the internet in some way. As these hardware requirements are subject to further investigation, Section 3.2 will discuss choices made concerning currently supported communication endpoints in more detail.

The next section will present an overview of Nautiport's architecture, thus providing the basis for a discussion on the Communication Manager later on in Section 3.2 of this chapter.

3.1.2 Architecture

Prior to explaining Nautiport's communication facilities, this section first elaborates on its architecture. Nautiport follows the model-view-controller (**MVC**) design pattern ([Bur87]), by strictly separating content from design and control. For demonstration purposes, Fig-

[8] see Section 3.1.2
[9] see Section 2.1.3 for basic hardware requirements
[10] see Section 3.2

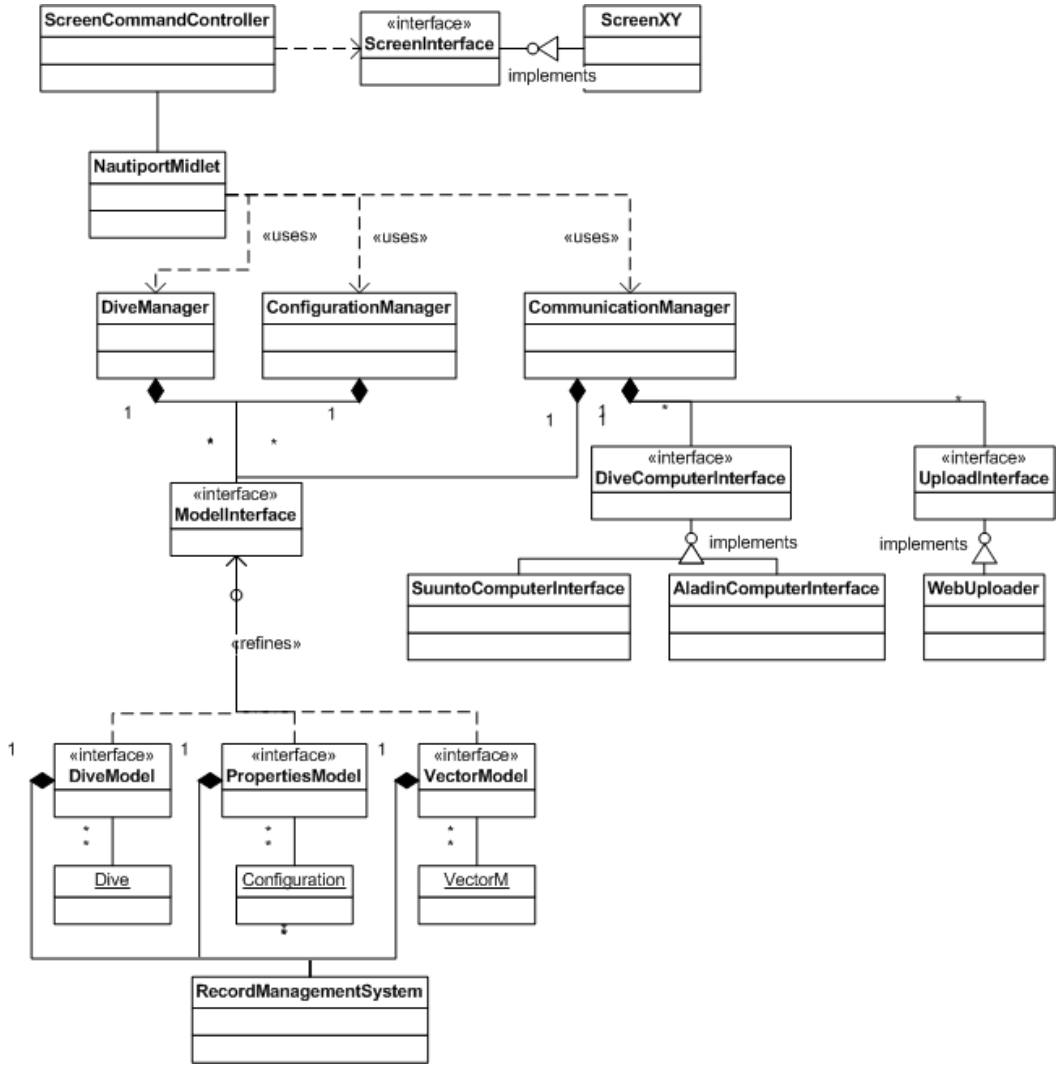


Figure 3.3: Nautiport UML Diagram [Ker08]

ure 3.3 shows Nautiport's architecture as an Unified Modeling Language (UML) diagram. Hereby, its core is represented by the MIDlet's entry point *NautiportMidlet*. This class implements MIDlet-mandatory methods, such as *startApp()*, *pauseApp()* and *destroyApp()*. As discussed later in Section 3.2.3, these methods serve as the application management system's (**AMS**) primary interface for controlling Nautiport's current state, such as pausing it due to an incoming voice call, or suspending it on user interaction.

As a matter of fact, in order to optimize code execution time, Nautiport's code was designed to be kept as small as possible. Apart from the fact that classes with small footprints generally execute faster[Kwo05], using compact classes leads to even less byte-code produced during obfuscation. Consequently, Nautiport's architecture profits from the **synergy** between small code-footprints and faster execution times.

The *ScreenCommandController* acts as **controller** for managing UI actions. In order to provide a flexible way to add and alter screens (i.e. **views**), Nautiport introduces a new base class: *ScreenInterface*. Doing so, J2ME's LCDUI components, such as Forms and TextBoxes, can be easily integrated into a screen, that can then be registered in the ScreenCommandController. This approach, enables a flexible and consistent way of handling different types of views.

Likewise views, also **models** used by Nautiport must conform to a consistent interface: the *ModelInterface*. Using this design, it is possible to quickly exchange model types. Ultimately, it is mandatory for models to implement the *set-* and *getBytes()* methods. Consequently, Nautiport is capable of exchanging data as pure Byte arrays, independent of the model used. This makes it quite powerful when it comes to dive data retrieval, or remote data synchronization, as discussed later in Section 3.2.

Currently there exist three types of models:

1. Dive model
2. Properties model
3. Vector model

From the programmatic point of view, Dive- and the Properties model are Nautiport-specific classes, whereas the *VectorModel* on the other hand extends Java's built-in Vector class, by implementing *ModelInterface*'s additional methods. Dive model represents a single dive, including basic dive data, such as depths and tank volume measured during a dive on one side, and any additional dive-related data, such as images or videos. It is to note though, that Nautiport does only keep references to the latter mentioned data, in order to minimize memory requirements on the mobile device. The Properties model on the other hand represents the current settings, including the categories general-, user- and communication.

As already mentioned in Section 3.1.1, persistent **storage** is provided through J2ME's built-in Record Management System (RMS), represented by Nautiport's *RecordManagementSystem* class. In addition to minimizing code size, in order to optimize execution times, [Kwo05] also advices to use **static** qualifiers whenever possible. Therefore, Nautiport's managing code was designed as compact, static classes. These managing classes can be divided into three types, depending on their scope of functionality:

1. Dive Manager

2. Configuration Manager
3. Communication Manager

Device specific settings are managed by the *ConfigurationManager*, whereas dive-related functionality is handled by the *DiveManager*. Finally, the *CommunicationManager* is in charge of providing access to other devices, that can be split into two categories:

1. Local, device-to-device communication over serial, infrared (IrDa) or Bluetooth™(BT) connections
2. Remote, Web platforms

The remainder of this chapter deals with Nautiport's communication facilities, represented by the *CommunicationManager*.

3.2 CommunicationManager

As previously mentioned in Section 3.1.1, during the process of designing Nautiport's software architecture great attention was laid on portability and compatibility. In order to maintain a high level of device compatibility, tradeoffs concerning supported communication endpoints had to be made. As a consequence, prior to explaining **connectivity-**, **concurrency-**, as well as **synchronization** issues dealt with when designing Nautiport's Communication Manager, this section will first elaborate on decisions made for currently **support communication endpoints**.

3.2.1 Supported Communication Endpoints

Today's mobile devices generally support a broad range of communication endpoints. Whereas the first mobile phones were limited to serial connections through special RS232 cables[11], modern phones additionally usually provide users with IrDa and BT support. Unfortunately, presently available popular dive computers, such as Suunto's current product line[12] on the other hand are often still limited to serial and (optionally) IrDa connectors. Therefore, tradeoffs have to be made in order to find a **common denominator**, for deciding which communication endpoints Nautiport will support.

Although numerous of today's more modern mobile devices offer various means of communication, support for traditional serial connections is often limited when using J2ME. As a matter of fact, some vendors purposely prohibit access to the corresponding interface through special restrictions[13]. A possible solution to this problem is to implement device-specific interface controls through middle-layer applications, written for example in the C-language. Nevertheless, using this approach portability issues arise, since these interface middle-layers must then be adapted for each and every device supported by target applications, such as Nautiport.

[11] e.g. Nokia's DLR-3P Serial Cable
[12] see `http://www.suunto.com`
[13] e.g. various Nokia mobile phones, such as the Nokia 6230

Consequently, IrDa and BT connections remain as logical choices. Apart from the rather low maximum transmission range, constant intervisibility is required in order to establish and maintain IrDa connections, thus drastically limiting its usability. Devices communicating through BT connections on the other hand, do not require intervisibility and therefore greatly enhance the possibilities of communication.

In the process of deciding which communication endpoint to support, various scenarios have been tested. As already mentioned, serial connections on mobile devices can be problematic, when trying to maximize compatibility. Although IrDa on the other hand is a potential choice, BT is a more up-to-date technology, providing a richer set of possibilities to establish device communication. Furthermore, J2ME provides a ready-to-use API for BT communication - the **JABTW**[14]. Using JABTW's serial port emulation mechanism **RFCOMM**, it is additionally possible to establish BT connections that can be handled like old-fashioned serial connections.

Based on these assumptions, Nautiport currently only supports **BT connections** to retrieve dive data. In order to arrange dive computers with BT support, numerous serial-to-BT adapters exist[15].

Future Supported Communication Endpoints

Apart from the fact that Nautiport currently only supports BT connections to exchange data between mobile devices and dive computers, the Communication Manager was designed to also provide an interface to serial and IrDa communication. Consequently, legacy devices that are only capable of serial or IrDa communication and meet the basic requirements discussed in Sections 2.1.3 and 3.1.1 respectively, will also be supported by Nautiport in future versions. For a detailed description of Nautiport's communication facilities please refer to Section 3.2.2.

3.2.2 Connectivity

As previously mentioned in Section 3.1.2, the Communication Manager class covers two distinct types of connections: **local device-to-device** and **remote internet-based connections**. Figure 3.4 shows the Communication Manager class as an UML diagram.

Using Figure 3.4, one can differentiate between these two types of connection. Local device-to-device communication is established through the *DiveComputerInterface*, whereas remote Web-based connections are represented by the *UploadInterface*. The remainder of this section is dedicated to discussing these communication facilities in more detail.

Device to Device Communication

Local device-to-device communication between mobile devices and dive computers is the first of the two possible connection types. It is basically represented by the **DiveComputerInterface**, that acts as an interface to underlying dive computer specific **communication protocols**. For each dive computer model there exists a specific implementation of the DiveComputerInterface, such as the *SuuntoComputerInterface*.

[14] see Section3.1.1

[15] e.g. IOGears's GBS301, see `http://www.iogear.com/product/GBS301/`

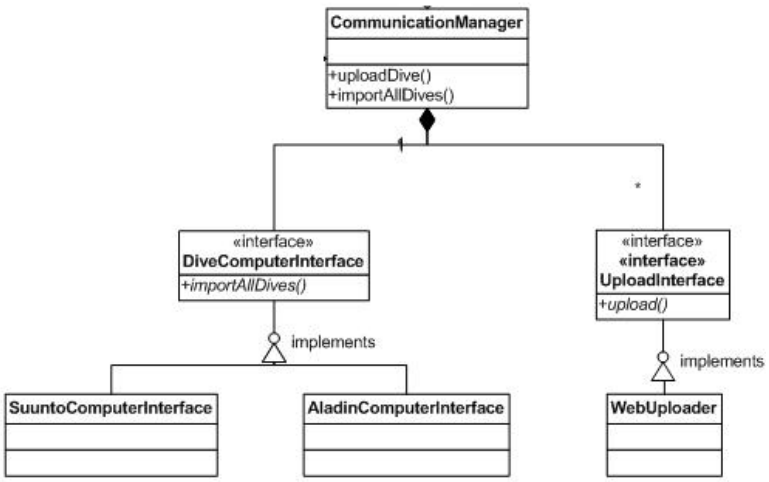


Figure 3.4: Communication Manager UML Diagram

Furthermore, depending on the dive computer model chosen, varying communication endpoints exist. For instance, the SuuntoComputerInterface currently supports serial, IrDa and BT connections alike. This support is represented by the specifically implemented classes SuuntoComputerInterface*Serial*, SuuntoComputerInterface*Infrared* and SuuntoComputerInterface*Bluetooth* respectively. Depending on the current settings made through Nautiport's UI, a specific interface class will be loaded, thus providing access to the dive computer.

The Communication Manager hereby acts as a **delegator** that simply loads the interface, corresponding to the current user's settings and delegates control to it. Looking at Figure 3.4, one can see that the DiveComputerInterface only defines a single method that must be implemented by dive computer specific interfaces: *importAllDives()*. As described later in Section , data retrieval from dive computers using importAllDives must be managed in a separate thread to not block any concurrent user actions.

Dive data retrieved during communication between mobile devices and dive computers must be saved locally. This is achieved by the previously mentioned Record Management System[16]. In order to maintain **data consistency**, dives already present on the mobile device must not be altered, unless the dives' date differ and the depth is greater than zero. Furthermore, dive data saved on dive computers will not be deleted during data retrieval through Nautiport. This ensures that users still have the possibility to export dive data to other devices after using Nautiport.

[16] see Section 3.1.1

As discussed in Section 3.2.2, by using Nautiport's second communication facility, users have the possibility to export locally saved dives to remote Web services, where they can be optionally shared with other people.

Device to Web Server Communication

Apart from local device-to-device communication to import dive data from dive computers to mobile devices, Nautiport's Communication Manager offers another interesting feature. Using the **UploadInterface**, as shown in Figure 3.4, it is possible to upload locally saved dive data to remote Web services, where they can be further processed and optionally shared with other divers all around the globe.

Like the DiveComputerInterface, the UploadInterface also only offers a single method to be called: *upload()*. Hereby, the UploadInterface serves as an interface to a specific Web service. Currently Nautiport's Communication Manager offers the *WebUploader* to upload saved dives to remote Web servers. Additional Web services can be easily integrated by implementing the UploadInterface.

Using this approach, a multitude of possibilities to transfer dive data to remote devices can be realized. As for example, if users want to transfer locally saved dives from mobile devices to Web servers via FTP, Nautiport must simply be extended with a FTPUploader class.

Another very interesting feature to mention is the fact, that using Web communication through the UploadInterface, it is also possible to **synchronize** dive data between mobile devices and any remote storage. Since this creates additional opportunities for enhancing Nautiport this topic will be discussed in more detail later in Section 3.2.4.

3.2.3 Concurrency

In order to keep applications responsive, time consuming tasks, such as retrieving remote data are usually carried out in separate threads. Especially, IO-operations are often swapped out in order to not block the UI, ensuring that users still have the opportunity to carry out other tasks. This section covers concurrency issues when using network communication on mobile devices, dealt with when designing Nautiport. In order to understand J2ME's handling of concurrent actions, this section will first briefly discuss the life-cycle of MIDlets in general. Afterwards, a description of the Communication Manager's approach towards non-blocking device communication will be presented.

Application Management System

In J2ME a MIDlet's life-cycle is controlled through the device's application management system (**AMS**) [Ort04]. As Figure 3.5 shows, it hereby can be in one of the following three states: *paused*, *active* or *destroyed*. Fundamentally seen, according to [Sch07] the AMS is in charge of two actions:

1. managing MIDlets' life-cycle
2. managing MIDlets' resources

Whereas the first task handles actions, such as pausing currently active MIDlets due to an incoming voice call, the second one covers topics like freeing allocated resources of paused MIDlets in order to prohibit bottlenecks.

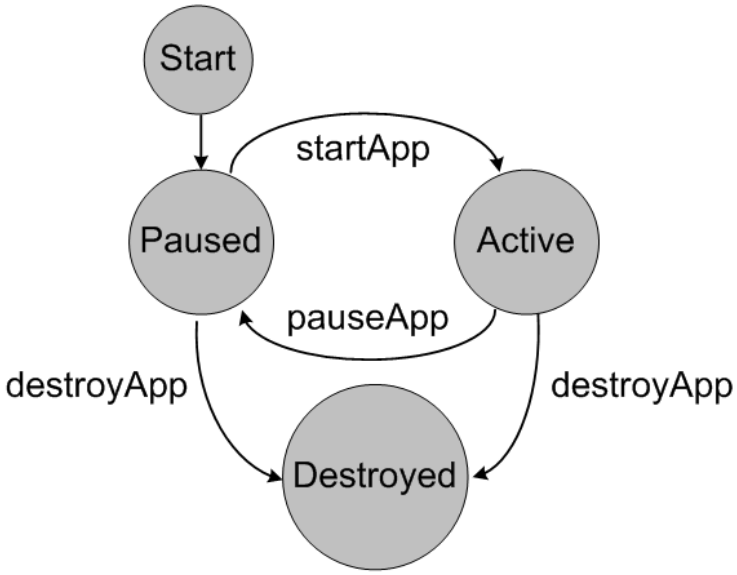


Figure 3.5: Application Management System (ASM) [Ort04]

Therefore, when designing MIDlets one has to always be aware that context switches can occur. As for instance, voice calls always take priority over any other running application, MIDlets are forced to pause, independent of currently active tasks. Consequently, any opened data connections have to be (temporarily) closed and important data written to temporary buffers to ensure consistency.

MIDlets are started by *system threads*, that are controlled by the AMS[Gig04]. Therefore, a MIDlets state can be altered by the AMS, through its respective system thread. MIDlets on the other hand cannot access the system thread they are currently running in. Nevertheless, they are allowed to create an arbitrary amount of application specific threads, as for example to swap out network execution code.

Whenever the AMS triggers the event to dispatch a currently active MIDlet, the corresponding system thread's status switches from *active* to *paused*, as depicted in Figure 3.5. Likewise, if users decide to quit running MIDlets their status switches from *active* to *destroyed.*

Generally speaking, concurrency can be divided into two categories:

1. Device-specific concurrency
2. MIDlet-specific concurrency

Whereas device-specific concurrency deals with AMS's actions of switching between different running applications, MIDlet-specific concurrency is about a MIDlet's inner han-

dling of isochronous actions, such as transferring data and refreshing the UI. The second type therefore is limited to the scope of a single MIDlet, without effecting other applications. The first one on the other hand entirely falls in the AMS's field of activity. This paper only deals with the second type of concurrency, by elaborating on Nautiport's approach towards non-blocking data transfer.

MIDlet Specific Concurrency

Applications that block on operations that could possibly be carried out in the background, such as saving data or upload content, are often conceived as badly designed and lead to a bad user experience. When using possibly blocking IO-operations, sophisticated, multi-threaded solutions generally win over applications running in a single thread.

Due to the fact that Nautiport targets very resource limited, CLDC-based mobile devices, using too many concurrent threads might lead to efficiency problems. According to [Kwo05], using threads should not be exaggerated, as context switches are expensive performance-wise. Therefore, Nautiport was designed to wisely use threads, in order to keep the responsiveness-performance ratio as high as possible. As a consequence, the following operations will run in separate threads:

1. RMS[17]-specific IO-operations
2. dive data retrieval through DiveComputerInterface
3. remote Web-based data exchange through UploadInterface

Although, Nautiport's UI events could have been designed to be handled by a special event-dispatcher-thread, they are delivered to the ScreenCommandController[18] for further processing.

As previously discussed in Section 3.2.5, during dive data retrieval nothing is altered on the dive computer itself. Using this approach, it is possible to always ensure data consistency, even though MIDlets might be paused or even destroyed during retrieval.

With remote data upload/synchronization, using the UploadInterface, consistency must be ensured through a **two-way handshake protocol**, divided into an authorization- and a data transfer related part. For instance, when using Nautiport's WebUploader the first step is to request authorization on the remote Web service, using locally stored user credentials. Once permitted, Nautiport will then send a list of dives to be uploaded, thus updating remote data if necessary.

3.2.4 Synchronization

One of the benefits when using Nautiport's UploadInterface, is the possibility to synchronize local with remotely stored dive data. As previously discussed in Section 3.2.2, additional communication classes can be implemented, that enable Nautiport to transfer dive data using specific Web services, such as FTP.

[17] see Section 3.1.1
[18] see Section 3.1.2

Originally, Nautiport's UploadInterface was meant for uploading and synchronizing dive data between mobile devices and a Web-based social network platform. Using this platform, divers could not only synchronize their dives, but also plan future dives forming groups, as well as exchange opinions about diving spots and equipment.

The synchronization process is primarily based on comparing modification dates in the first place. In order to minimize the amount of data transferred, Nautiport will first compare local with remote modifications dates. In case of variations, Nautiport uploads dives that are merged on the server side. Once completed, the Web service will response with changes made, so that mobile devices also can update locally stored data. It is to mention, that due to the limited computing power of mobile devices, resource consuming tasks are designed to be executed on the server side. Doing so, time needed for the synchronization process can be minimized, thus relieving the mobile device.

As mentioned in Section 3.2.2, due to Communication Manager's flexible architecture, remote communication is not only limited to the currently implemented WebUploader. As Nautiport's user community evolves, further communication classes will be made available.

3.2.5 Security

Apart from **data security** discussed in Section 3.2.2, **communication security** is another important topic to deal with. Speaking of data security, one has to further differentiate between security concerning possible loss of data through inconsistency, as well as prohibiting unauthorized access. Whereas the first issue is mostly disposed by Nautiport's non-data-altering policy during dive data transmission, as elaborated in Section 3.2.2, local data security will be discussed later in Section 3.2.5. In the following, security measures taken for Web communication will be presented.

Web Communication Security

As discussed in Section 3.2.2, dive data can be transferred over Web connections using the UploadInterface. Consequently, it is a crucial task to secure these communication channels to not expose confidential data to strangers. Section 2.3.4 already elaborated on theoretical measures available to be taken in order to establish secure HTTP connections, using TLS/SSL encryption in J2ME.

Practically seen, starting with MIDP 2.0, J2ME's built-in GCF[19] provides native support for HTTPS connections. Due to the fact that Nautiport targets mobile devices using MIDP 2.0, Web communication through the secure HTTPS protocol does not pose additional requirements, as it is supported out-of-the-box. Consequently, Nautiport establishes secure HTTPS connections when transferring data to remote Web services.

Local Data Security

Aside from securing Web communication, confidential data locally stored on mobile devices, such as passwords must also be protected from unrestricted access. Since mobile

[19] see Section 2.3

devices can be stolen, thus theoretically enabling thieves to read stored credentials they should be stored in an **encrypted** manner.

As writing own encryption code usually is not the recommended approach, due to the complexity often involved implementing selected algorithms, existing solutions come in handy. Nautiport incorporates one of the more popular encryption APIs - the **Bouncy Castle Crypto APIs**[20]. Since according to [BC08], this lightweight API supports all Java editions, ranging from J2ME to JDK 1.6, it makes it an ideal choice for Nautiport. Consequently, sensitive and confidential data is encrypted before being stored in the RMS, thus making it useless for data thieves.

[20] see `http://www.bouncycastle.org/`

Chapter 4

Conclusions and Outlook

This paper started with an in-depth analysis of **J2ME** - Sun's smallest and most feature-scarce platform, that was especially targeted at mobile devices. Based on the findings of this analysis, the second part of this paper presented a concrete solution built using J2ME. This MIDlet based solution, called **Nautiport**, represents an approach to provide divers with an easy-to-use, yet powerful mobile application to retrieve dive data from dive computers for further examination as well as for exportation to remote Web services. Hereby, J2ME has proven to be a sophisticated choice for a device-independent platform. Furthermore, J2ME's built-in communication framework **GCF** provided a solid foundation for Nautiport's **Communication Manager**. By using an adapted version of J2ME's **RMS**, Nautiport was equipped with facilities for persistent storage.

Apart from J2ME's technical description, Chapter 2 of this paper focussed on the Communication Manager. Topics, such as **supported communication endpoints** as well as **connectivity**-, **concurrency**- and **security issues** have been discussed. Furthermore, Nautiport's **synchronization** feature has been briefly presented. Thereby, the Communication Manager's flexible architecture was discussed. Among others, the flexibility of this architecture enables the binding of Web services. Hence, as Nautiport's user community evolves, support for further Web services, such as FTP-uploads or exports to foreign Web platforms, can be provided.

Although the context of this paper elaborates on J2ME strengths, its weaknesses and **limitations** have been discussed as well. In general, although J2ME represents a throughout planned platform, there still exist some minor flaws. As discussed in Section 2.2.2, lacking support for **shared libraries and data**, as well as **minimal control over LCDUI components** pose significant limitations. With the advent of MIDP 2.0, sharing of data between different MIDlet-suites became possible, nonetheless support for shared libraries is still to come. In the meanwhile, developers are forced to include possibly reusable code separately in every MIDlet-suite, thus unnecessarily enlarging them. To overcome this limitation, a possible practical solution would be to use the same access control mechanism, as with sharing data through the RMS. Prior to deployment, MIDlet-suites could then define which embedded libraries are allowed be used by foreign suites.

The minimal control over high-level UI components in favor of compatibility poses another significant limitation. Although it is comprehensible to constrain programmatical

layout control to a certain extent, fundamental properties, such as horizontal component alignment are crucial during UI design. Future J2ME versions should therefore consider the compatibility-feature ratio once again.

In sum, J2ME provides a solid (but still limited) foundation for building mobile applications. As a practical demonstration, Nautiport has been developed. It represents a small but sophisticated application, specifically targeted at resource limited mobile devices, for managing dive data. Due to its flexible architecture, support for additional Web services is provided, thus enlarging the number of potential future users.

As the mobile technology segment advances, software developers are challenged to develop even more sophisticated applications, in order to fully utilize capacities as well as technological capabilities of up-to-date mobile devices. Furthermore, interaction between mobile devices and standard PCs is gaining more and more importance. Keeping up-to-date by synchronizing calendars, tasks and contacts between mobile phones and PCs is only one of a variety of examples.

Nautiport has proven to be another valuable example of a solution for the main problems concerning heterogenous interaction. Chances are likely that more powerful future versions of mobile devices will ultimately be the only useful solution for ”multi-purpose mobile PCs”, making even more benefits from their high mobility in our daily lives possible.

Bibliography

[BC08] BOUNCY CASTLE, The L. t.: *The Legion of the Bouncy Castle*. Available online at http://www.bouncycastle.org/java.html, July 2008. – Last visit: 22.09.2008, Last update: July 2008 33

[Bur87] BURBECK, Steve: *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*. Available online at http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, January 1987. – Last visit: 22.09.2008, Last update: January 1992 23

[Cor04] CORDREY, Glen: *Pushing the Limits*. Available online at http://www2.sys-con.com/ITSG/virtualcd/Java/archives/0612/cordrey/index.html, February 2004. – Last visit: 22.09.2008, Last update: February 2004 15

[Cra07] CRAZYSCUBA.COM: *PADI Adventure Dive Log Refill Pages*. Available online at http://www.crazyscuba.com/prod_images_blowup/logrefill_5001.jpg, February 2007. – Last visit: 22.09.2008, Last update: February 2007 21, 39

[Eco07] ECONOMIST.COM: *Mobile telecommunications*. Available online at http://www.economist.com/research/backgrounders/displaybackgrounder.cfm?bg=1258181, October 2007. – Last visit: 22.09.2008, Last update: October 2007 3

[Gig04] GIGUERE, Eric: *Understanding MIDP System Threads*. Available online at http://developers.sun.com/mobility/midp/ttips/threading3/index.html, December 2004. – Last visit: 22.09.2008, Last update: June 2008 30

[GR04] GLAHN, Kay ; RYSA, Erkki: *JSR-000248 Mobile Service Architecture*. Available online at http://jcp.org/aboutJava/communityprocess/final/jsr248/index.html, February 2004. – Last visit: 22.09.2008, Last update: February 2004 13

[Hol02] HOLTKAMP, Heiko: *Einführung in TCP/IP*. Available online at http://www.rvs.uni-bielefeld.de/~heiko/tcpip/tcpip.pdf, February 2002. – Last visit: 22.09.2008, Last update: February 2002 18

[JG01] JONES, Shari ; GOULD, Steve: *J2ME: Step by step*. Available online at http://www.stevengould.org/portfolio/developerWorks/j2me/

j2metutorial/j-j2me/j-j2me-ltr.pdf, June 2001. – Last visit: 22.09.2008, Last update: June 2001 6, 8, 9, 11

[Ker08] KERSTNER, Matthias: *Nautiport Manual*. Available online at http://www.kerstner.at/nautiport/manual.pdf, May 2008. – Last visit: 22.09.2008, Last update: May 2008 22, 24, 39

[Kwo05] KWON, JungHyuk: *Effective J2ME Programming*. Available online at http://kr.sun.com/developers/event/data/Effective.pdf, June 2005. – Last visit: 22.09.2008, Last update: June 2005 25, 31

[LBT08] LAYTON, Julia ; BRAIN, Marshall ; TYSON, Jeff: *How Cell Phones Work*. Available online at http://electronics.howstuffworks.com/cell-phone.htm/printable, June 2008. – Last visit: 22.09.2008, Last update: June 2008 3

[MTH+05] MUELLER, Jochen ; THORSTEN, Lenhart ; HENRICI, Dirk ; HILLENBRAND, Markus ; MUELLER, Paul: Developing Web Applications for Mobile Devices. In: *Proceedings of the First International Conference on Distributed Frameworks for Multimedia Applications*, 2005, S. 346 – 350 3

[Ort03] ORTIZ, Enrique C.: *The Generic Connection Framework*. Available online at http://developers.sun.com/mobility/midp/articles/genericframework/, August 2003. – Last visit: 22.09.2008, Last update: August 2003 16, 17, 39

[Ort04] ORTIZ, C. E.: *Managing the MIDlet Life-Cycle with a Finite State Machine*. Available online at http://developers.sun.com/mobility/midp/articles/fsm/, August 2004. – Last visit: 22.09.2008, Last update: June 2008 29, 30, 39

[Pro02] PROCESS, Java C.: *Mobile Information Device Profile for Java™2 Micro Edition*. Available online at http://jcp.org/aboutJava/communityprocess/mrel/jsr118/index.html, November 2002. – Last visit: 22.09.2008, Last update: November 2002 18

[Pro08a] PROCESS, Java C.: *Introducing the Java Community Process (JCP) Program, Version 2.6*. Available online at http://www.jcp.org/files/whitepaper.JCP26Whitepaper.pdf, January 2008. – Last visit: 22.09.2008, Last update: January 2008 12

[Pro08b] PROCESS, Java C.: *The Java Community Process(SM) Program - Introduction - FAQ*. Available online at http://jcp.org/en/introduction/faq, March 2008. – Last visit: 22.09.2008, Last update: March 2008 11, 39

[Roe08] ROEHRIG, Christof: *Mobile Computing*. Available online at http://www.inf.fh-dortmund.de/personen/professoren/roehrig/SS08/tk/tk02.pdf, May 2008. – Last visit: 22.09.2008, Last update: May 2008 7, 9, 12, 39

[Sch07] SCHMATZ, Klaus-Dieter: *Java Micro Edition*. dpunkt.verlag, 2007 6, 7, 9, 10, 14, 15, 16, 18, 29

[SEMC08] SONY ERICSSON MOBILE COMMUNICATIONS, AB.: *Z750: first Java Platform 8 (JP-8) phone supporting next-generation Mobile Services Architecture*. Available online at `http://developer.sonyericsson.com/site/global/newsandevents/latestnews/newsmarch07/p_z750_firstjp8phone_msa.jsp`, March 2008. – Last visit: 22.09.2008, Last update: March 2008 13

[SM03] SUN MICROSYSTEMS, Inc.: *Java™ Technology for the Wireless*. January 2003 8, 39

[SM06a] SUN MICROSYSTEMS, Inc.: *CLDC Library API Specification 1.0*. Available online at `http://java.sun.com/javame/reference/apis/jsr030/`, May 2006. – Last visit: 22.09.2008, Last update: December 2007 5

[SM06b] SUN MICROSYSTEMS, Inc.: *Java™ Platform, Standard Edition 6 API Specification*. Available online at `http://java.sun.com/javase/6/docs/api/`, May 2006. – Last visit: 22.09.2008, Last update: December 2007 5

[SM08] SUN MICROSYSTEMS, Inc.: *Java Micro Edition - Homepage*. Available online at `http://java.sun.com/javame/index.jsp`, May 2008. – Last visit: 22.09.2008, Last update: September 2008 7

[Wil01] WILLIAMS, Rob: *Computer Systems Architecture*. Addson-Wesley, 2001 18

[YL02] YUAN, Michael J. ; LONG, Ju: *Securing Wireless J2ME*. Available online at `http://www.ibm.com/developerworks/wireless/library/wi-secj2me.html`, June 2002. – Last visit: 22.09.2008, Last update: January 2007 8, 10

[YN01] YEE, Wai G. ; NAVATHE, Shamkant B.: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems, 2001, S. 78 – 89 3

List of Figures

2.1 Target Device Audience For Java Editions Available [Roe08] 7
2.2 Typical Software Stack in Today's Mobile Devices [SM03] 8
2.3 J2ME Hardware Requirements [Roe08] 9
2.4 JCP Timeline [Pro08b] 11
2.5 J2ME Modular Architecture [Roe08] 12
2.6 GCF Connection Interface Hierarchy and Related Classes [Ort03] 17

3.1 PADI Paper Dive Log [Cra07] 21
3.2 Nautiport's Main Screen 23
3.3 Nautiport UML Diagram [Ker08] 24
3.4 Communication Manager UML Diagram [Ker08] 28
3.5 Application Management System (ASM) [Ort04] 30

List of Tables

2.1 CLDC/MIDP Optional Packages Overview 13
2.2 MIDP 2.0 Protocol URL Schemes 18