Application Development using J2ME – Architecture for Device Independence

By

Terje Eggum

A thesis submitted for the degree of Master of Science in Information and Communication Technology

Agder University College - Faculty of Engineering and Science and University of New South Wales - Faculty of Engineering

Grimstad, 18th of July 2005

ABSTRACT

The operating system Symbian OS and the programming language Java have existed in a symbiosis since the first version of Symbian OS arrived on the mobile scene. This thesis explores important aspects of the mobile version of Java, namely the Java 2 Micro Edition, on Symbian OS based mobile phones.

Part one of the thesis reviews the structure and evolution of Java 2 Micro Edition and the Symbian OS, and the symbiosis between them. This is done through a thorough theoretical investigation of the programming interfaces offered to the developer. Particularly certain problem areas such as hardware control, wireless messaging, network services and file access have been investigated. To evaluate the maturity and feature richness of the platform, a test application has been made which incorporates features depending on all these areas.

We found that Java 2 Micro Edition platform was quite easy to use when implementing features like camera recording, HTTP/Servlet communication and Graphical User Interface programming. However, we also experienced that the platform is lacking some advanced options in each of the mentioned features.

In the second part of the thesis, the device independent aspects of Java2 Micro Edition have been examined. The idea was to evaluate the portability offered by this development platform, and consider the feasibility of creating device independent applications that offer an even higher degree of portability. By reviewing Java2 Micro Editions built in portability and studying relevant projects, two frameworks, built on top of Java2 Micro Edition, have been suggested as possible tools for development of better device independent applications.

PREFACE

This Master thesis is the final work in order to achieve the Master of Science degree in Information and Communication Technology at Agder University College (AUC), Faculty of Engineering and Science. The thesis is written in collaboration with AUC and the University of New South Wales (UNSW).

The work has been done in Sydney, Australia and Grimstad, Norway between February and July 2005.

The thesis consist two parts, one joint part and one individual part. The first and main part is written in collaboration with fellow student Håvar Lunberg, the second part is written exclusively by me, Terje Eggum.

We would like to thank our supervisor, assistant professor Lars Line (AUC) for valuable guidance throughout the thesis. We would also like to thank our assistant supervisor, Fritjof Boger Engelhardtsen, for useful feedback.

Grimstad 18th of July 2005

Terje Eggum

Håvar Lundberg

TABLE OF CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

ABB	Audio Building Block
AMS	Application Management Software
CDMA	Code Division Multiple Access
CLDC	Connected Limited Device Configuration
EMS	Enhanced Messaging Service
eSCO	extended Synchronous Connection Oriented
FC	File Connection
FP	Foundation Profile
GCF	General Connection Framework
GPRS	General Packet Radio Service
GUI	Graphical User Interface
IPSEC	IP Security
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Editon
J2SE	Java 2 Standard Edition
JAD	Java Application Descriptor
JAR	Java Archive
JVM	Java Virtual Machine
KVM	K-Virtual Machine (Kauai VM)
MAMSAPI	Advanced Multimedia Supplements
MID	Mobile Information Devices
MIDP	Mobile Information Device Profile
MIDP	Mobile Information Device Profile
MMAPI	Mobile Media API
MMS	Multimedia Messaging Service
NDS	Nokia Developer Suite
OMA	Open Mobile Alliance
OTA	Over The Air
PAN	Personal Area Network
PBP	Personal Basis Profile
PDP	Packet Data Protocol
PIM	Personal Information Management
PP	Personal Profile
RDS	Radio Data System
RTP	Realtime Transfer Protocol
SMS	Short Messaging Service
SyncML	Synchronization Markup Language
UDP	User Datagram Protocol
WCDMA	Wideband Code Division Multiple Access
WMA	Wirless Messaging API
WTK	Wireless Toolkits
BICA	Built In Context Adaptability
DOC	Device Optimized Components
CAL	Context Adaption Layer
BODF	Build On Demand Framework
ACE	Automated Code Editor

1 Introduction

1.1 Background

It is a well known fact that computer technology evolving fast in a more and more mobile environment. Professional users carry laptops and advanced smartphones with them in order to be able to do useful work when and where it might please them. Whether this is a good thing is a question for others to answer, but since we are heading down this mobile path at least we should have decent tools to work with. Since application development on mobile technology is a relatively young subject, and the devices themselves are rapidly getting more advanced, it is important to periodically evaluate development platforms in order to see whether or not they are using the available technology to the full extent.

When Sun decided to divide Java into three branches, Java 2 Second Edition (J2SE), Java 2 Enterprise Edition (J2EE) and Java2 Micro Edition (J2ME), the mobile Java lost some functionality. There were many reactions to this; some developers even predicted that J2ME would be only temporary. However, current statistics tell us otherwise: *"Globally there are more than 708 million J2ME capable mobile devices worldwide, according to Ovum, and more than 140 operators that have deployed Java technology-based services, according to Nokia. Java technology-based devices are expected to reach 1.5 billion consumers by 2007 according to some analysts, and the overall revenue from services enabled by Java technologies is forecast to reach $15 billion by 2008."* [1]. The accuracy of this statement is hard to test, but it clearly states that J2ME is still here. So, the question explored in part one of this thesis is whether J2ME has eradicated these childhood diseases, or if there still is a substantial lack in its functionality.

With so many devices out there and J2ME applications being fronted as portable and we still have MIDlets specified for the different types of devices new questions arises. What kind of obstacles is the J2ME platform facing in the struggle for device independence, and what can be done to improve? These questions are explored in part two of the thesis.

1.2 Problem specification

The first part of the problem specification states: *"The assignment assumes that the student has good skills in object-oriented java development, but no explicit experience with J2ME on Symbian OS phones. The first part of the assignment is to explore this development environment and evaluate maturity and features. Possibilities for initiating network services and controlling local devices like camera and audio recording must be included in the evaluation. The first part of the assignment can be done in cooperation with other students."*

The second and individual part of this thesis can be described in the following: "*A core idea with Java is "develop once and run anywhere". Experience shows that many applications still are tailor made for specific brands and models of mobile phones. The reason for this can be differences in features like screen resolution, input devices or other characteristics. The intention of this thesis is to explore to what extent it is feasible to develop good device independent applications. A framework for characterisation of device features and a solution for how this should be handled by the application shall also be explored."*

1.3 Delimitations – Part One

In the evaluation part of the thesis, we do not have the time to examine all parts of the J2ME/Symbian relationship. We have therefore made these delimitations:

1.3.1 Focus Areas – Features and Maturity

As stated in the problem specification there are certain areas of the J2ME platform that are more relevant than others and it is in these topics we will conduct our most thorough research and testing.

Hardware control: We will implement and test photo and audio recording functions. This requires API's to control hardware extensions such as camera and microphone.
File access**:** File access is essential since we need to store image and audio files in order to make a decent application.
Network services: We are going to implement and test Multimedia Messaging Service (MMS) functions and other ways of transferring the gathered files and information from the device to the server.

Besides these three focus areas we will only make brief investigations regarding general programming issues such as Graphical User Interface (GUI) programming and general maturity.

1.3.2 Platform

Although we will conduct some research on all the old versions of J2ME and Symbian OS, this is merely to see where the evolution is heading. The real focus will be on Symbian OS version 8.0 and J2ME (Connected Limited Device Configuration (CLDC) 1.1, Mobile Information Device Profile (MIDP) 2.0) since these are currently the newest and most richly featured versions on the market. These are the only platforms we will do any development on.

1.3.3 Testing

The only Symbian OS based mobile phone available to us is the Nokia 6630, with Symbian version 8.0. This will therefore be the only "real" test platform for our application. The reason for choosing this particular phone was that at the time it had the newest version of Symbian OS and it had all the hardware extensions needed for the thesis.

1.4 Delimitations – Part Two

The second part of this thesis is approached theoretically, i.e. there will only be conducted partial testing. Solutions will be based on literature and my own ideas, and the implementation of these solutions will be left to future work.

1.5 Thesis overview

This thesis is divided into two separate parts. The first part is co-written with Håvar Lundberg and concerns itself with evaluation of maturity and features of the J2ME/Symbian OS development platform

Chapter 2 is a technical review of the Symbian OS and the J2ME development language. The operating system is examined historically and architecturally. This is also the case with the research on J2ME, but here we also go into the tools available and look more specifically at API's we can use in the development process. The interaction between Symbian and Java is also examined.

Chapter 3 is where we present our research on the platform. We give the scope and the method for our investigation and we present a demonstrator application made to illuminate the areas mentioned in the problem specification. Our experiences on each of these subjects are thoroughly discussed in chapter *3.3 Test Results*. A conclusion based on this chapter and the previous is made in chapter 3.5.

Chapter 4 will contain a discussion of our experience with the platform, and a conclusion regarding maturity will finish of the first part of this thesis.

In the second part of the thesis I take on the task of considering device independent application development with J2ME

Chapter 5 reviews the device independence issue regarding MIDP based applications. Sun's approach to device diversity and other interesting projects will be studied here.

Chapter 6 is where I present my proposed solutions. I present two frameworks with some similarities and differences.

Chapter 7 will contain a discussion on the topic of device independence, and a comparison between the two proposed frameworks. A conclusion is also made on the feasibility of extending MIDlets' portability with a framework.

PART ONE – MATURITY AND FEATURES OF J2ME ON SYMBIAN OS

2 Technical review

2.1 The Symbian OS

A few years ago the mobile phones had very few features and most manufactures used their own operating system in their products. The phones nowadays are much more complex and require an advanced operation system to provide a reliable and versatile platform for third party software. In 1998 some of the leading companies in wireless communication (Sony, Ericsson, Nokia, Motorola and Psion) formed the company Symbian 0. Symbian developed the Symbian OS which is an advanced, open standard operating system for data enabled phones written in C++. The Symbian OS is by far the most used OS for smartphones and it holds a 61% market share world wide [3].

2.1.1 Symbian - A mobile OS

The Symbian OS is made entirely for the mobile market and its particular needs. Certain issues that are common only for mobile phones have to be addressed. The Symbian OS was created because it was more adequate to develop a particular mobile OS to meet these needs rather than to redefine already existing desktop or server OS. Many unfortunate compromises would have had to be made in order to make this possible.

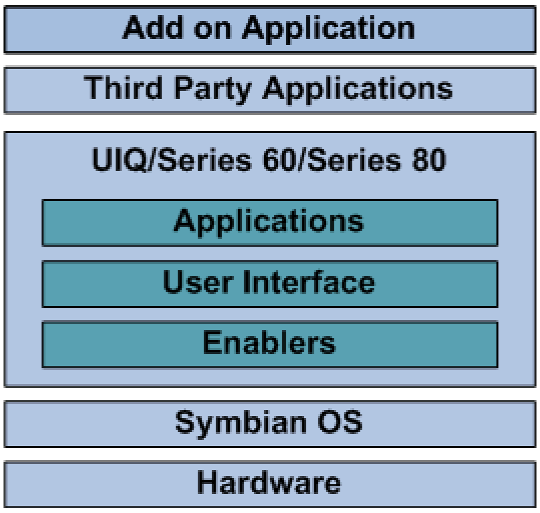


Figure 2.1 Symbian mobile phone configuration

Some of the important issues that have to be addressed in a mobile environment are memory footprint and processor power. Depending on type and model, most mobile phones only have a small amount of memory available, and this issue has to be coped with in order for the phone to work in a satisfying manner. For example, if a user frequently experience that his phone is hanging or has to be restarted due to lack of memory or processor power, he will most certainly get frustrated and probably change mobile phone manufacturer the next time he buys a phone. This issue is rather common on desktops and most users are accepting that these incidents occur once in a while. The mobile phones however, have to work flawless, thus the OS have to have a very effective memory handling and an effective use of available processor power.

The OS has to provide built-in power management features in order for the phone to work in a practical manner. These features turn of battery draining functions and applications when they are not in use. In addition to this, Symbian phones are provided with flash memory to avoid loss of data in case of a shutdown.

A mobile OS have to cope with the networking use and capabilities that are common for a mobile phone. There are principally three different states a mobile phone operates in; connected to the operator network, connected to a local network or operating in offline mode. In order to transfer data there has to be some kind of connectivity, either using a wide area network or a local area network. The wide area network can be based on different technologies, e.g. GSM, General Packet Radio Service (GPRS) or Wideband Code Division Multiple Access (WCDMA), and the local area network can be based on e.g. Bluetooth or Infrared connectivity. In any case, the phone has to handle fade outs and one can not always assume that the phone is connected due to incomplete coverage. The phone has to function as an advanced client and these issues have to be handled in a way that is transparent to the user.

Other important issues that have to be dealt with are different types of keyboard input and different screen types. Mobile phones come in different shapes and sizes and some are very sophisticated, others are very primitive. A phone can be equipped with a large screen, a small screen, a keyboard, a pen input or a perhaps a keypad. Regardless of phone design and technical solutions from different vendors, the OS have to handle these variations.

2.1.2 The Symbian OS Architecture

Symbian OS is an open standard operating system licensed by some of the worlds leading mobile manufacturers. It is designed to meet the requirements of data-enabled 2G, 2.5G and 3G mobile phones. The OS includes a multitasking kernel, integrated telephony support, communications protocols, data-management, advanced graphics support, a low-level graphical user interface framework and a variety of application engines.

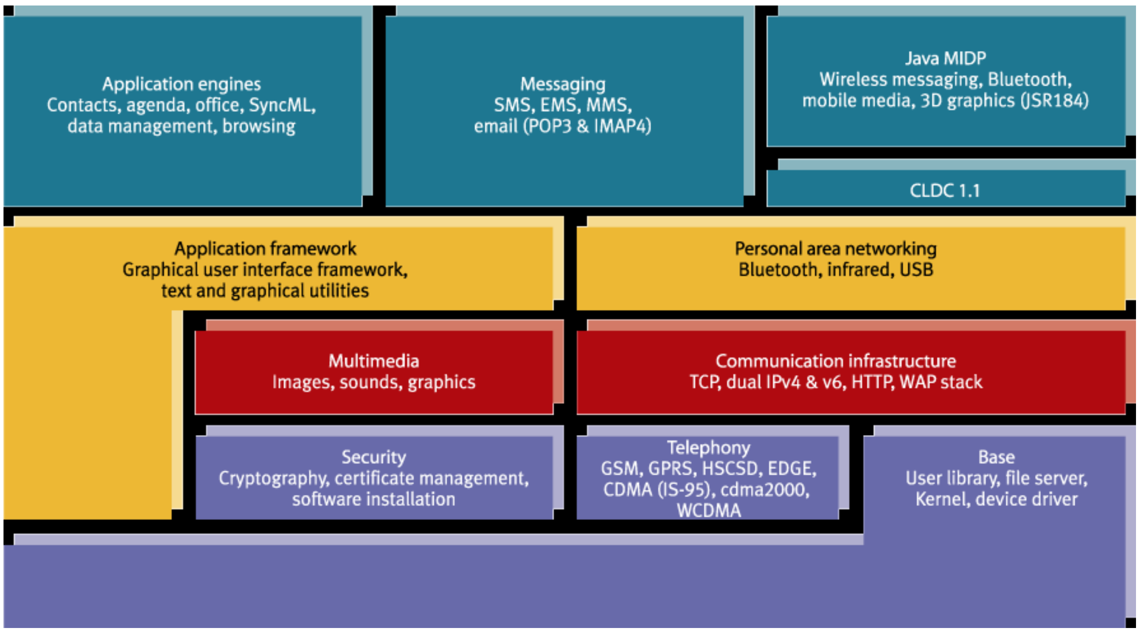

Application engines
Contacts, agenda, office, SyncML, data management, browsing
Messaging
SMS, EMS, MMS, email (POP3 & IMAP4)
Java MIDP
Wireless messaging, Bluetooth, mobile media, 3D graphics (JSR184)
CLDC 1.1
Application framework
Graphical user interface framework, text and graphical utilities
Personal area networking
Bluetooth, infrared, USB
Multimedia
Images, sounds, graphics
Communication infrastructure
TCP, dual IPv4 & v6, HTTP, WAP stack
Security
Cryptography, certificate management, software installation
Telephony
GSM, GPRS, HSCSD, EDGE, CDMA (IS-95), cdma2000, WCDMA
Base
User library, file server, Kernel, device driver

Figure 2.2 Functional overview of Symbian OS v8.0 [4]

The architecture of Symbian OS can be divided into two different parts, the main kernel that handles protocol stacks and network resources, and the graphical user interface platform which can be altered by the different phone vendors. The graphical user interface has been divided into four different platforms in order to handle different screen sizes and keyboard inputs. These are UIQ, Series 60, Series 80 and others.

UIQ

UIQ is designed for smart phones and the newest version is v3.0 and is based on Symbian v9.1. In contrast to its predecessors it supports one-handed use with softkeys, in addition to pen-based input. Other UI designs can easily be implemented by the mobile phone manufacturer on this platform. Sony Ericsson P910, Motorola A1010 and BenQ P30 are all typical UIQ phones.

Figure 2.3 Sony Ericsson with the UIQ platform

Series 60

The Series 60 platform is created by Nokia and it is designed for smart phones. It supports single-hand operated mobile phones and it is designed for voice communication, multimedia messaging, content browsing and application downloading. Series 60 2ndedition has existed since 2003 and was last implemented on Symbian v8.1. The newest version is the Series 60 3rdedition and it runs on Symbian v9.1. Nokia N91 is announced as the first mobile phone that is based on Series 60 3rdedition [5]. Both the 2nd and the 3rd edition have a scalable UI's and support the following screen sizes: 176 x 208, 240 x 320 (QVGA) and 352 x 416. Nokia 6620, Nokia 6630, Nokia 6680 and Panasonic X700 are examples of Series 60 mobile phones. This platform is distributed as Symbian's official Graphical User Interface (GUI).

Figure 2.4 Nokia 6630 with the Series 60 platform

Series 80

The Series 80 is also created by Nokia and it is designed for enterprise devices with large horizontal screens (640 x 200 pixels) and keyboard-based input. The series 60 is based on Symbian v7.0s. Nokia 9500 and Nokia 9300 are examples of Series 80 mobile devices.

Other GUI

Not all Symbian mobile phones fall into the above mentioned categories such as the mobile phones developed by Fujitsu for the FOMA network.

2.2 The J2ME standard

In 1999 Sun realized that the idea of one Java platform for all purposes was perhaps not yet feasible. The Java2 platform consequently divided into three distinct parts, each with a complete runtime environment for Java applications. J2EE targets the enterprise market, the J2SE focuses on desktop applications and J2ME handles the wireless environments [6].

The world of wireless platforms is arguably the most diverse of the three target areas, and to manage this diversity J2ME have different approaches to different groups of devices. It is possible to “tailor” the J2ME setup with a mix of configurations, profiles and optional packages. Figure 2.5 shows the different layers that comprise the J2ME platform, from hardware to application.

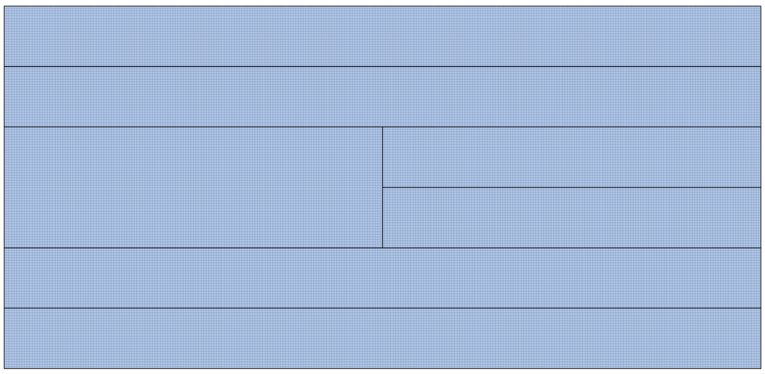


Figure 2.6 J2ME related to the OS and the device

In this chapter we will list the most common configurations and profiles that make up the J2ME platform [10]. We start with configurations, after a short virtual machine history, as they are the foundation on which all the other parts build upon. Not all will be described at the same level of detail, but the Connected Limited Device Configuration and the Mobile Information Device Profile will be emphasized as they are the most relevant for this project.

2.2.1 The Virtual Machine

As in all Java platforms J2ME applications run on a virtual machine. Due to limited resources on the devices they can not use the standard Java Virtual Machine (JVM) used on stationary computers. So, in 1999 the K-Virtual Machine (KVM) for mobile devices based on CLDC/MIDP was introduced by Sun Microsystems. The K was put there instead of the J because the KVM was the result of the project "Kauai”, and not because its size is measured in kilobytes instead of the megabytes in the standard JVM [7]. The KVM was a lot slower than the JVM and ran at about 30% to 80% of JDK1.1.x desktop speed performance [8].

With the release of J2SE 1.3.x, Sun Microsystems introduced the Java HotSpot Virtual Machine technology to the java developers community. The introduction of HotSpot Optimized JVM technology to CLDC/MIDP devices occurred in 2001 [CLDC HI Whitepaper] CLDC HotSpot Implementation Virtual Machine. The HotSpot Java Virtual Machine for CDC/J2ME Platform devices was introduced in 2004. This largely improved performance of the mobile virtual machine [9].

2.2.2 Connected Device Configuration (CDC)

“*The J2ME CDC provides the basis of the Java 2 Platform, Micro Edition in devices characterized as follows:*

- *512K minimum ROM available*
- *256K minimum RAM available*
- *Connectivity to some type of network.*
- *Supporting a complete implementation of the Java Virtual Machine as defined in the Java Virtual Machine Specification, 2nd Edition.*

User interfaces with varying degrees of sophistication down to and including none may be supported by this configuration specification. TV set-top boxes, web enabled phones, and car entertainment/navigation systems are some, but not all, of the devices that may be supported by this configuration specification.” **Error! Reference source not found.**

The J2ME CDC will define the minimum required complement of Java Technology components and API's for connected devices. Supported APIs, application life-cycle, security model, and code installation are the primary topics to be addressed by this specification.

The core APIs of CDC are almost identical to the ones found in J2SE.

2.2.3 Connected Limited Device Configuration (CLDC)

The CLDC was developed to be used in devices where CDC is too large to meet the strict memory footprint requirements that are characteristic of CLDC target devices. Two versions of the CLDC have been defined, version 1.0 and version 1.1. CLDC 1.1 adds a few new features over CLDC 1.0. Floating point support is the most important feature added, but several minor bug fixes have also been added. CLDC 1.1 is the configuration we will use for development in this project, and it is intended to be backwards compatible with version 1.0.

The CLDC provides these packages to the developer [10]:

- **java.io:**
  Provides classes for input and output through data streams.
- **java.lang:**
  Provides classes that are fundamental to the Java programming language.
- **java.lang.ref:**
  Provides support for weak references.
- **java.util:**
  Contains the collection classes, and the date and time facilities.
- **javax.microedition.io:**
  Classes for the Generic Connection Framework (GCF).

As we can see there are no GUI classes provided by the CLDC. This is up to the profiles to provide.

The CLDC is intended to work on devices with intermittent network connections, small processors and limited memory. Devices that support CLDC typically include 192 to 512 KB total memory available for the Java platform and a 16-bit or 32-bit processor. Within this group of devices, the variety of features is immense, and to make a standard Java platform suiting them all is difficult. Therefore the CLDC makes a minimum of assumptions about the environment it exists within.

2.2.4 Foundation Profile (FP)

FP is a set of Java APIs that support resource-constrained devices without a standards-based GUI system. Combined with the CDC, FP provides a complete J2ME application environment for consumer products and embedded devices. FP is the most basic of the CDC family of profiles.

2.2.5 Personal Profile (PP)

J2ME PP is a set of Java APIs that supports resource-constrained devices with a GUI toolkit based on AWT. Combined with the CDC, J2ME Personal Profile provides a complete J2ME application environment for consumer products and embedded devices.

2.2.6 Personal Basis Profile (PBP)

J2ME PBP is a set of Java APIs that support resource-constrained devices with a standards-based GUI framework. Combined with the CDC, J2ME PBP provides a complete J2ME application environment for consumer products and embedded devices. J2ME PBP includes all of the APIs in Foundation Profile.

2.2.7 Mobile Information Device Profile 1.0 (MIDP 1.0)

The MIDP target Mobile Information Devices (MID). To be classified as a MID a device should have the following minimum characteristics:

- Display:
    - Pixels: 96x54
    - Display depth: 1-bit
    - Pixel shape (aspect ratio): approximately 1:1
- Input
    - One- or two-handed keyboard or touch screen
- Memory:
    - 128 KB of non-volatile memory for the MIDP components
    - 8 KB of non-volatile memory for application-created persistent data
    - 32 KB of volatile memory for the Java runtime

- Networking:
  - Two-way, wireless, possibly intermittent, with limited bandwidth

We will not go into packages provided by MIDP 1.0 since we will be using MIDP 2.0 in development, and the packages there are an extension of MIDP 1.0.

2.2.8 Mobile Information Device Profile 2.0 (MIDP 2.0)

Requirements for display, input and networking are the same as for MIDP 1.0. Memory requirements have been raised in the MIDP 2.0 specification. There must be 256 KB of non-volatile memory for the MIDP implementation, beyond what's required for the CLDC and 128 KB of volatile memory for the Java runtime. Requirements for sound have been added. The ability to play tones is now made a requirement.

MIDP 2.0 is backwards compatible with MIDP 1.0, hence it provides all functionality defined in the MIDP 1.0 specification. In addition it provides OTA provisioning. This feature was left to Original Equipment Manufacturers (OEM) to provide in the MIDP 1.0 specification.

These are the packages that MIDP 2.0 provides the developer with:

- **javax.microedition.lcd**
  The UI API provides a set of features for implementation of user interfaces for MIDP applications.
- **javax.microedition.lcdui.game**
  The Game API package provides a series of classes that enable the development of rich gaming content for wireless devices.
- **javax.microedition.midlet**
  The MIDlet package defines MIDP applications and the interactions between the application and the environment in which the application runs.
- **javax.microedition.rms**
  The MIDP provides a mechanism for MIDlets to persistently store data and later retrieve it.
- **javax.microedition.io**
  MIDP includes networking support based on the Generic Connection Framework from the CLDC.

- **javax.microedition.pki**
  Certificates are used to authenticate information for secure Connections.
- **javax.microedition.media**
  The MIDP 2.0 Media API is a directly compatible building block of the MMA (JSR-135) specification.
- **javax.microedition.media.control**
  This package defines the specific Control types that can be used with a Player.
  Core Packages
- **java.lang**
  MIDP Language Classes included from J2SE.
- **java.util**
  MID Profile Utility Classes included from J2SE.

As we can see, this is a much more extensive library to work with than what the CLDC alone provides. An enhanced user interface has been defined, making applications more interactive and easier to use. Media support has been added through the Audio Building Block (ABB), giving developers the ability to add tones, tone sequences and WAV files even if the MMAPI optional package is not available.

Game developers now have access to a Game API providing a standard foundation for building games. This API takes advantage of native device graphic capabilities.

MIDP 2.0 adds support for HTTPS, datagram, sockets, server sockets and serial port communication.

Push architecture is introduced in MIDP 2.0. This makes it possible to activate a MIDlet when the device receives information from a server. Hence, developers may develop event driven applications utilizing carrier networks.

End-to-end security is provided through the HTTPS standard. The ability to set up secure connections is a leap forward for MIDP programming. A wide range of application models require encryption of data and may now utilize the security model of MIDP 2.0 based on open standards.

2.3 Optional Packages

An optional package is a set of APIs, but unlike a profile, it does not define a complete application environment. An optional package is always used in conjunction with a configuration or a profile. It extends the runtime environment to support device capabilities that are not universal enough to be defined as part of a profile or that need to be shared by different profiles.

The Optional Packages mentioned in this chapter are the ones that are relevant to the problem specification of thesis. There are, of course, more APIs available but they have been excluded from this paper for lack of relevance.

2.3.1 JSR 75: PDA Optional Package

This specification will define two independent optional packages that will extend and enhance the "J2ME CLDC" JSR-000030. These packages separately represent important features found on many PDAs and other mobile devices. The optional packages are:

- Personal Information Management (PIM) - This package gives J2ME devices access to personal information management data that resides natively on mobile devices. Information to be accessed are contained in address books, calendars, and to-do lists residing in many mobile devices.
- FileConnection - This package gives J2ME devices access to file systems residing on mobile devices. The primary use of this API is to allow access to removable storage devices, such as memory cards that many of today's devices support.

The PDA Optional Package is placed on top of the CLDC and provides optional APIs common to PDAs and handsets. For example, the PIM functionality in JavaPhone makes its re-introduction into J2ME Platform devices within this optional package. FileConnection API is added to allow General Connection Framework (GCF) to access removable media storage.

2.3.2 JSR 120: Wireless Messaging API (WMA 1.0)

"*The messaging API is based on the GCF, which is defined in the CLDC 1.0 specification. The package javax.microedition.io defines the framework and supports input/output and networking functionality in J2ME profiles. It provides a coherent way to access and organize data in a resource-constrained environment. The design of the messaging functionality is similar to the datagram functionality that is used for the User Datagram Protocol (UDP) in the GCF. Like the datagram functionality, messaging provides the notion of opening a connection based on a string address and that the connection can be opened in either client or server mode. However, there are differences between messages and datagrams, so messaging interfaces do not inherit from datagram. It might also be confusing to use the same interfaces for messages and datagrams. The interfaces for the messaging API have been defined in the javax.wireless.messaging package*" [10].

WMA provides a common API for sending and receiving text and binary messages, typically SMS messages. WMA was first defined in JSR 120 and revised in JSR 205, which introduced support for multi-part messages and the Multimedia Message Service (MMS). This revision is not supported by our test mobile Nokia 6630. However, there are ways to overcome this obstacle, and we will describe this further in chapter *3.3.4*.

WMA is based on GCF and depends on CLDC as its lowest common denominator, meaning that it can be implemented along with both CLDC- and CDC-based profiles. It targets cell phones and other devices that can send and receive wireless messages.

2.3.3 JSR 205: Wireless Messaging API 2.0 (WMA 2.0)

"*With the WMA 2.0 it will be possible for Java applications to compose and send messages, which can contain text, images and sound. This technology allows a richer possibility for messaging on mobile devices. For the realisation the framework of JSR 120 will be used.*[10]"

With the WMA 2.0 it will be possible for Java applications to compose and send messages, which can contain text, images and sound. This technology allows a richer possibility for messaging on mobile devices. For the realisation the framework of JSR 120 will be used.

2.3.4 JSR 135: Mobile Media API (MMAPI)

”The API is targeted to fulfill the needs for the control and simple manipulation of sound and multimedia for applications in mobile devices, with scalability to other J2ME devices. Mobile devices may feature a great variety of multimedia capabilities. Some of the target devices may only be able to produce single monophonic sounds while others may feature both sampled, synthetic audio and other media types. The API should also be able to support the control of time-based multimedia formats. This causes special consideration for the API design. The main requirements for the API are:

- *Enable the use of the basic sound generation routines with simple controls.*
- *Do not provide too much hard coded functionality that is obsolete on the basic devices.*
- *Provide methods to access more sophisticated audio features if they exist.*
- *Address media synchronization issues*
- *Be able to extend support to other media types*
- *Maintain low footprint*

These requirements are fulfilled by a design where the API provides direct support for basic features such as simple generation and playback of sound, and playback of multimedia. A control interface is proposed to enable the management and control of different multimedia formats and extended functionalities. This design enables the supported features to vary according to the platform and the corresponding implementation of the MMAPI.”

MMAPI provides a generic but flexible foundation for multimedia processing for devices with advanced sound and multimedia capabilities. This optional package was introduced by JSR 135. MMAPI depends on the CLDC as its lowest common denominator, so it too can be used with CDC-based profiles. The only requirement is that the implementation includes IllegalStateException, which is not present in CLDC 1.0.” [10]

The MMAPI splits media processing into two main concepts: data source handlers, media protocols specified by an URL, and content handlers, media controls and players. In addition, a media manager provides a factory of resources such as players, as well as methods to query for supported content types and protocols. The manager also includes a simple tone player.

MMAPI 1.0 defines protocols, controls, and players for a number of media types, such as MIDIControl, VideoControl, ToneControl, and VolumeControl. The specification does not mandate any particular one, allowing implementers to subset the MMAPI as appropriate. The only requirement is that implementations must guarantee support of at least one media type and protocol.

2.4 APIs in development

Here we will briefly go through some interesting API's that are currently being developed in the Java Community Process. Specifically we look at API's that will improve the platforms features for general development. All this information is gathered from the JCP web site [10].

2.4.1 JSR 234 Advanced Multimedia Supplements (MAMSAPI)

This specification will define an optional package for advanced multimedia functionality which is targeted to run as a supplement in connection with MMAPI (JSR-135) in J2ME/CLDC environment.

Java equipped terminals are evolving into general multimedia and entertainment platforms. Features like camera and radio which have traditionally belonged into different device categories are now integrated into same terminals. The increase in the processing power of modern mobile phones allows more sophisticated media processing capabilities. Displays will remain relatively small due physical limitations but rich aural experience can be achieved without adding the physical size of the terminals.

The purpose of this API is to give access to multimedia functionality of the modern mobile terminals. Specifically, better support for camera and radio and access to advanced audio processing will be introduced but it's possible to add other functionality as well.

This specification will bring the following capabilities to the mobile terminals with J2ME/CLDC support:

- Access for camera specific controls like visual settings (brightness, contrast), flashlights, lighting modes and zooming.
- Proper access to radio and other channel/frequency based media sources including RDS (radio data system)
- Access to advanced audio processing capabilities like equalizer, audio effects, artificial reverberation and positional 3D audio. Dynamically changing audio resources are addressed as well.
- Media output direction. For example, the ability to choose whether the audio is played out from speaker of from headset.

This specification had its final release the 20th of June this year.

2.4.2 JSR 238: Mobile Internationalization API

This JSR defines an API that provides culturally correct data formatting, sorting of text strings and application resource processing for J2ME MIDlets running in MIDP over CLDC.

This specification will provide a common API for the internationalization of MIDP applications, delivered and licensed as an optional package. It will provide the means to isolate localizable application resources from program source code and an API for accessing those resources at runtime, selecting the correct resources for the user's/device's locale. The specification will also define an API for supporting cultural conventions in applications, e.g. for formatting dates, times, numbers, and currencies, and sorting text strings correctly for the user's locale. The API needs to be memory-efficient to run on resource-constrained devices such as mobile phones.

The need for this API arises from the fact that mobile devices are personal by nature, and users expect them to conform to the cultural conventions they are accustomed to. Users want to be able to interact with the device in their own native language and see data rendered as in their everyday environment.

This API had its final release the 21st of April this year.

2.4.3 JSR 230: Data Sync API

This JSR will be a J2ME optional package that can be used with the J2ME configurations CLDC and CDC. It enables applications to synchronize their application specific data stored in the terminal with corresponding data stored on a server, replicating any changes made to either instance of the data. It should provide a generic interface to the data synchronization device implementation, to enable data synchronization via underlying implementations of data synchronization protocols. One example of the data synchronization protocols to be accessed from Java applications will be SyncML / OMA Data Synchronization.

The API should be a high level API, which provides a common set of synchronization commands.

2.5 The MIDlet

A MIDlet is a MIDP application that runs on a device with CLDC configuration and MIDP profile, and it is built upon the MIDlet class. This class provides programmatic interfaces for invoking, pausing, restarting and terminating the MIDlet application. For instance, the application manager can pause a MIDlet to allow the user to answer an incoming phone call, and a MIDlet can also make a request to be paused and later restarted.

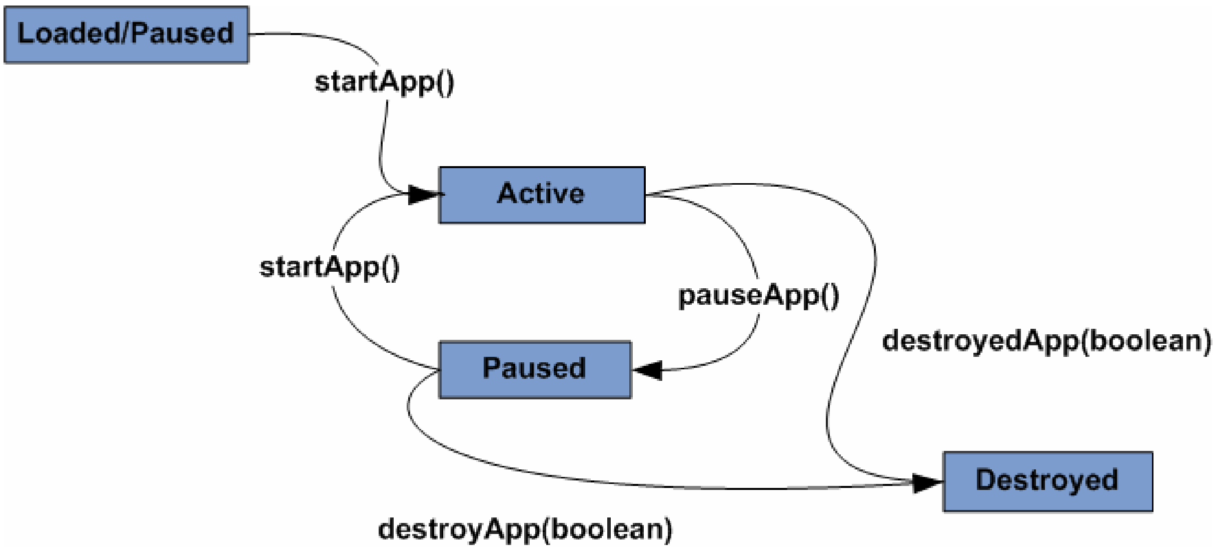


Figure 2.7 The lifecycle of a MIDlet [11]

Since today’s mobile phones seem to favor this CLDC/MIDP setup, this is the type of application this thesis will prioritize.

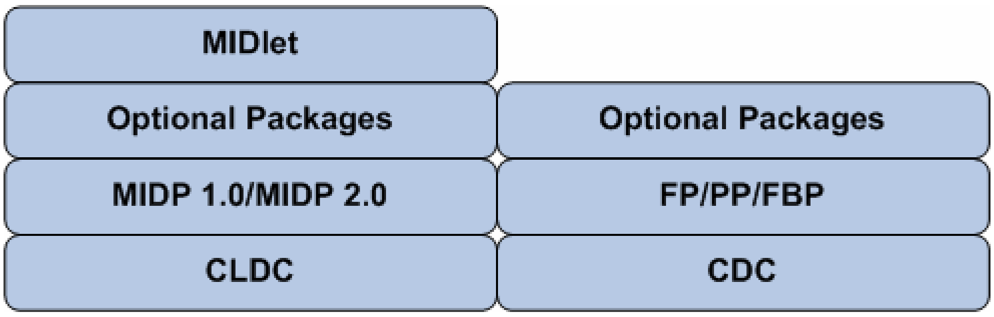


Figure 2.8 An illustration of the MIDlet on top of the MIDP/CLDC structure

Instead of executing like an ordinary Java application, MIDlets are stored in a jar-file called a MIDlet suite. Then this suite is put onto a MIDP device which contains Application Management Software (AMS), which again opens and launches the MIDlet on the device. *Figure 2.7* shows how a MIDlet fits in the J2ME universe.

2.6 Generic Connection Framework

To handle the communication with the servers we used the Generic Connection Framework (GCF). Below the structure of the GCF is displayed. As we can see it is a straightforward hierarchy of interfaces and classes used to create various sorts of connections.

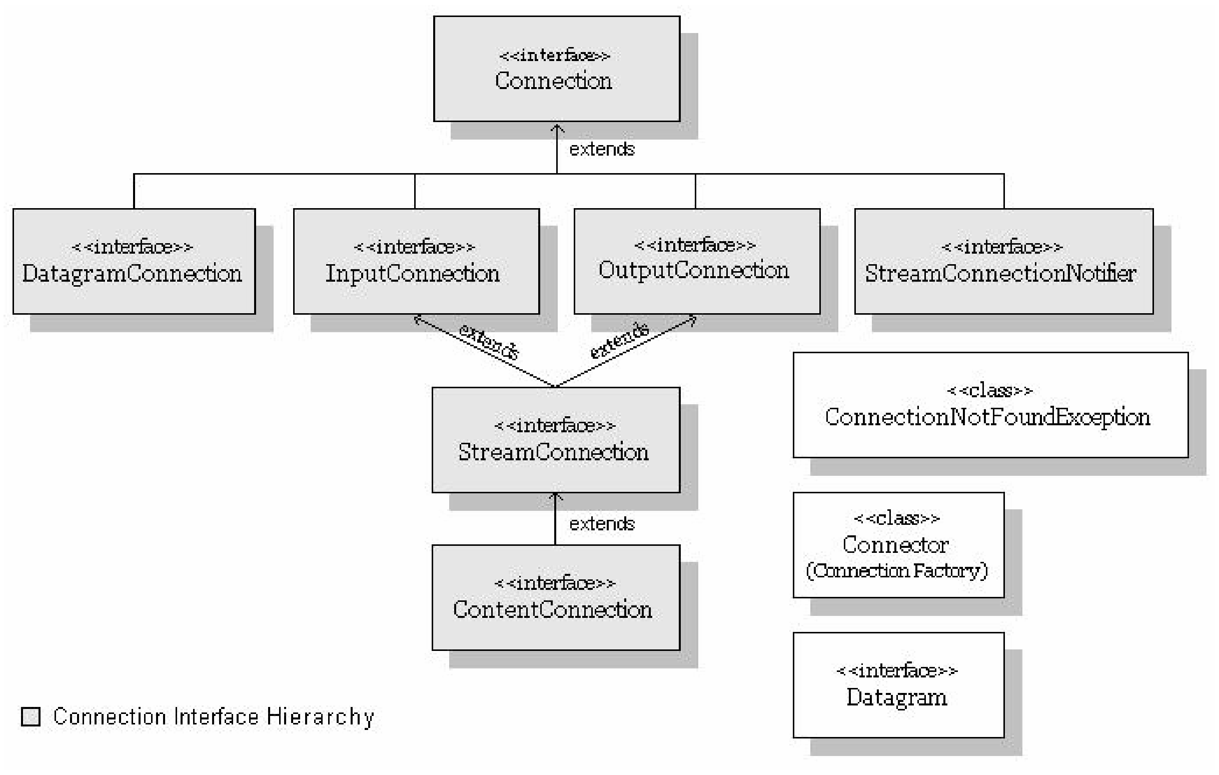


Figure 2.9 GCF overview[12]

The GCF is very flexible and it is easy to extend it when needed. New connection types, which are defined and standardized via the Java Community Process (JCP), can be added by defining a new Connection subtype and supporting classes, providing a Connector factory class that supports the newly defined connection type, and defining a new URL scheme that identifies the new connection type. *Figure 3.7* illustrates how the GCF could be extended by a profile or an optional package.

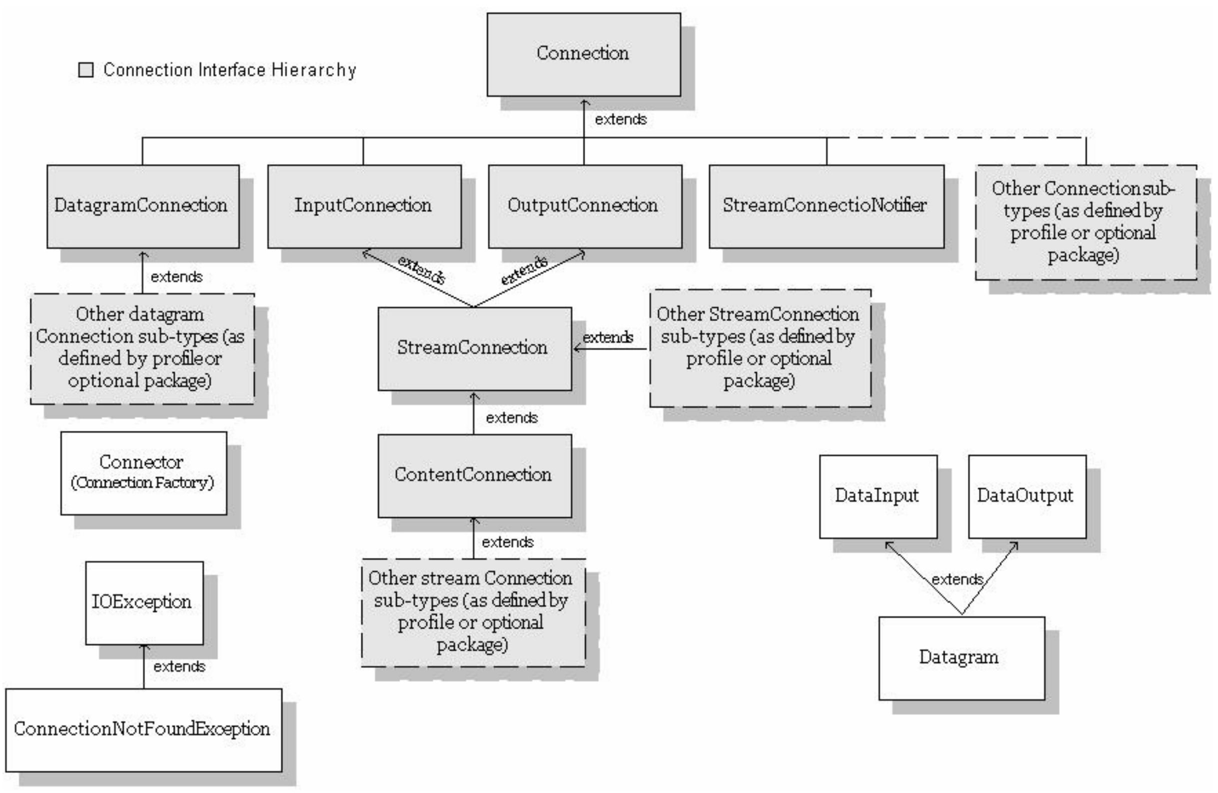

Connection Interface Hierarchy
Connection
extends
DatagramConnection
InputConnection
OutputConnection
StreamConnectioNotifier
Other Connection sub-types (as defined by profile or optional package)
extends
Other datagram Connection sub-types (as defined by profile or optional package)
StreamConnection
extends
Other StreamConnection sub-types (as defined by profile or optional package)
Connector (Connection Factory)
ContentConnection
Other stream Connection sub-types (as defined by profile or optional package)
IOException
ConnectionNotFoundException
DataInput
DataOutput
Datagram

Figure 2.10 Extended version of the GCF [12].

The GCF provides a whole range of connection types for the developer. One of the best features from the GCF is the way it standardizes the connection syntax. All connections are opened with a standard URL like this: *scheme://user:password@host:port/url-path;parameters*, where the different parts are [12]:

- ***scheme*** specifies the access method or protocol, such as FTP or HTTPS. In the GCF, it describes the connection type to use, which maps to an underlying connection or I/O protocol.
- ***user*** is an optional user name.
- ***password*** is an optional password.
- ***host*** is the fully qualified name or the IP address of the host where the resource is located.
- ***port*** is an optional port to use. Its interpretation depends on the scheme.
- ***url-path*** is the "path" to the resource. Its format and interpretation depend on the scheme. The url-path may define optional parameters.

Below, the currently available GCF connections are listed

URL Scheme	Connectivity	GCF Connection Type	Defined By
btl2cap	**Bluetooth**	**L2CAPConnection**	**JSR 82. Support is optional**
datagram	**Datagram**	**DatagramConnection**	**All CLDC- and CDC-based profiles, such as MIDP, Foundation and related profiles, and with JSR 197, J2SE support is optional.**
File	**File Access**	**FilleConnection Input Connection**	**JSR 75. Support is optional.**
http	**HyperText Transport Protocol**	**Httpconnection**	**MIDP 1.0, MIDP 2.0, Foundation Profile, J2SE (JSR 197). Support is required.**
https	**Secure HTTP**	**HttpsConnection**	**MIDP 2.0 support is required.**
comm	**Serial I/O**	**CommConnection**	**MIDP 2.0 support is optional**
sms	**Short Messaging Service**	**MessageConnection**	**JSR 120, JSR 205. Support is optional.**
mms	**Multimedia Messaging Service**		
cbs	**Cell Broadcast SMS**		
apdu jcrmi	**Security Element**	**APDUConnection JavaCardRMIConnection**	**JSR 177. Support is optional.**
socket serverSocket	**Socket**	**SocketConnection ServerSocketvonnection**	**JSR 118 (MIDP 2.0). Support is optional**
datagram	**UDP Datagram**	**UDPDatagramConnection**	**JSR 118 (MIDP2.0). Support is optional.**

Table 2.1 GCF connections [12]

2.7 J2ME on Symbian

2.7.1 History [13]

Symbian's first Java implementation, based on Sun's JDK 1.1.4, was released as a part of Symbian OS v5 in 1999.

Symbian OS v5.0 was released in 1999 and was the first Symbian OS with Java support and it was based on Sun's JDK 1.1.4. The next Symbian release, Symbian v6.0, based its Java support on the PersonalJava 1.1.1 specification and was released in 2000. PersonalJava, which was based on JDK 1.1.6, had the advantage of reduced memory footprint. This Symbian release also implemented Sun's JavaPhone API, which is a vertical extension to the PersonalJava platform. Because of this extension, it was now possible to access telephony functionality, send and receive datagrams and manipulate address book and calendar information.

The Micro Edition was designed for a range of consumer and embedded electronic devices with little resources. It was clear that J2ME MIDP was highly suitable for mass market mobile phones and it became very popular among phone manufacturers because of its lightweight configuration. Symbian included this standard whit its Symbian v7.0 release and also back-ported it to earlier releases. Even tough this standard was foreseen to have ha great future, it was also apparent that MIDP 1.0 had its limitations due to the limited MIDP 1.0 specification.. Because of this, both J2ME and PersonalJava lived side by side on Symbian phones until the release of Symbian v8.0, where PersonalJava was no longer supported.

J2ME has progressed a lot since the release of MIDP1.0, and in 2002 MIDP 2.0 was released as a part of the Java specification Request (JSR 118). In addition to this a range of optional packages were released, also part of the Java Community Process. The optional packages enhance the MIDlet functionality, giving support to range of features.

Symbian version 7.0s was released in 2003 and was the first Symbian OS with MIDP 2.0 support. It introduced a lot of new features and APIs like the new security model, new game and audio APIs, enhanced UI API, the Push Registry, Bluetooth and SMS support. In addition to this Symbian gave support for Sun's high performance CLDC HI VM.

Nokia has used Symbian OS v7.0s for Version 2.0 of their Series 60 Developer Platform. The Series 60 2nd edition supplements the functionality that comes standard in Version 7.0s with an implementation of the Mobile Media API (MMA, JSR 135) providing Java support for video playback, tone generation and photo capture, adding to the audio API that comes as part of MIDP 2.0.

Symbian 8.0 was announced in 2004 and enhanced the J2ME CLDC/MIDP implementation adding the following optional packages to Symbian OS: Mobile Media API (JSR 125), Mobile 3D Graphics (JSR 184), File GCF (part of JSR 75) all running on top of Sun's CLDC HI 1.1 VM. In addition, the Java implementation is now fully compliant with the Java Technology for the Wireless Industry specification (JTWI, JSR 185). The JTWI is an initiative defined via the JCP to specify a minimum set of APIs and behaviour that a compliant phone should support. By targeting the JTWI, ISVs and 3rd party developers can know that their applications will run on the largest possible number of phones. Release 1 of the specification mandates MIDP 2.0, CLDC 1.0 and WMA as a minimum API set with the

MMA also required if multimedia functionality is exposed to Java. Symbian OS v8.0 also integrates support for the Universal Emulator Interface (UEI) allowing Symbian MIDP emulators to fully integrate with standard tools such as Sun’s Wireless Toolkit and IDEs such as JBuilder and Sun One Studio.

2.7.2 MIDP 2.0 on Symbian OS phones

Nokia 6600 was the first MIDP 2.0 enabled Symbian phone on the market. This phone was based on v7.0s, which was the first Symbian version with MIDP 2.0 support. This support has also been back-ported to UIQ 2.1 phones based on Symbian v7.0. Symbian v9 is currently the newest OS and supports the UIQ 3 and the Series 60 UI platforms. Nokia N91, which is the first v9 mobile phone, will be available in 3Q or 4Q 2005.

Model	**SE P910**	**Nokia N91**	**Nokia 6680**	**Nokia 6630**	**Nokia 9300**	**BenQ P30**	**Panasonic X700**	**Motorola A1010**
OS	V7	v9	V8	V8	V7s	V7	V7s	V7
UI	UIQ	Series 60	Series 60	Series 60	Series 80 Series 40	UIQ	Series 60	UIQ
Screen	208 x 320	176 x208	176x208	176 x 208	640x200 128x128	208x320	132x176	208x320

Table 2.2 Some MIDP 2.0 enabled Symbian phones

2.7.3 How to use native Symbian services with J2ME [14]

Even though the MIDP/CLDC together with optional packages typically provides the developer with a rich API set there are bound to be things that only a C++ application with access to native services can do. This poses a big problem if a key feature in a MIDlet depends on a service that simply is not accessible through conventional methods. There is however, a way to circumvent these limitations. This requires more than the regular Java skills to do, but to the experienced Symbian developer it is a reality.

MIDlets handle socket communication with other hosts, and the same way they can handle communication with sockets listening on the local loopback address 127.0.0.1. This means that we can actually have a MIDlet communicating with a native C++ application running on the same device. Since the native application has the whole spectrum of native services available, this means that even the MIDlet can reach them indirectly through socket communication.

What you need to have on the native side is a so-called daemon. This will be an EXE program, always resident and ready to process requests from the MIDlet. Just implement the desired native functions into the daemon, and you have access to Symbians, for J2ME developers, hidden features. Of course, this breaches the perimeters of the sandbox, but it can sure be useful to a capable C++ programmer.

2.7.4 Benefits of J2ME on Symbian

Symbian and J2ME are two fast growing technologies that enhance the mobile environment. They both have great value separately and when joined together they produce a very reliable environment for mobile applications. The J2ME implementation on Symbian is very robust and it is running on the very stable Symbian OS kernel. Its implementation has a small footprint which takes advantage of Symbian OS’s compact and effective philosophy, both for MIDP 1.0 and 2.0 applications. The Java UI components directly mapping to the native UI components is very efficient and allows the applications to work at a faster rate. J2ME on Symbian also have the advantage of the JCP. They frequently provide new optional Java APIs, which again leads to that MIDlets to get more and more functionality and features. With the performance and capabilities of J2ME on Symbian OS continually improving it now offers third party developers a viable developer environment. It's likely to believe that this environment will approach the mass market in an even greater extent.

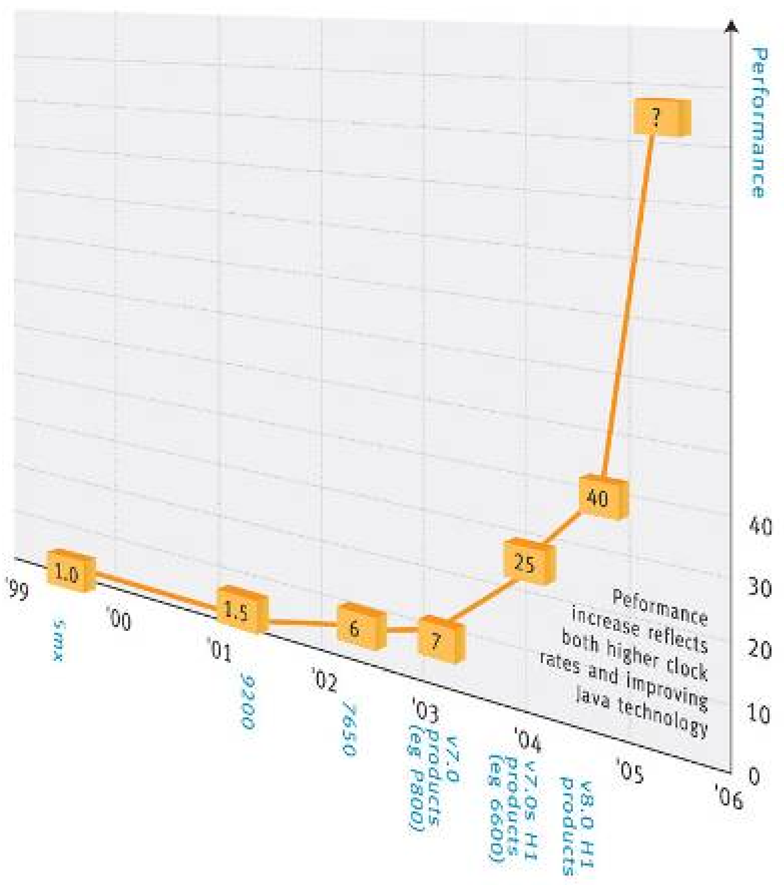


Figure 2.11 Estimated performance of J2ME on Symbian

3 Evaluation of J2ME on Symbian

In this chapter we will go through our evaluation of the J2ME/Symbian platform. We start with defining the tasks scope and method, and go through the development of an application used for testing of the platform. The results/experiences from this development are presented in its own chapter *3.3*, and here we will go through the core issues of the process. Following this will be a brief discussion of these experiences and a conclusion. We also include a short look to the future at the end because J2ME/Symbian is a constantly evolving symbiosis.

3.1 Scope and Method

3.1.1 Scope

In this first part of the thesis we will evaluate the development platform J2ME on the operating system Symbian OS. It is especially the areas of initiating network services, hardware control, and file access that will undergo a thorough evaluation. GUI and general development issues will also be explored, but in this thesis these topics will not have the same priority as the previously mentioned focus-areas. The idea is to examine the maturity level of this development platform, and the richness of the features it provides.

3.1.2 Method

Research is an essential part of making an evaluation. Without theoretical knowledge, there is little to base conclusions on. Therefore the technical review we did in the previous chapter is the foundation on which we build this evaluation. In the research for the technical review we gained an extensive knowledge about both J2ME and the Symbian OS, and the relationship between them. To evaluate the maturity and feature-level of this development platform, a practical approach is taken. We intend to develop an application where the all the elements of interest are incorporated. This way the maturity and features are examined from both a theoretical and a practical angle, and this will give us the background we need to draw a well well-considered and well-tested conclusion. There are four questions on which we will base our conclusion:

1. How well does MIDP's hardware control fit the underlying technology?

2. Is MIDP a good networking profile?
3. Can you easily develop good GUI’s with MIDP?
4. Is the general development process reasonably hassle-free?

3.1.3 Choice of Tools

The tools we chose for the development of our test application were chosen merely on theoretical grounds. The need for advanced enterprise features was not there, and we could have done just as well with just a text editor and Suns Wireless Toolkit. However when using the professional tools, you get a certain sense of how much effort the industry is putting into the platform, and this can help us in the evaluation process.
We landed on Borland JBuilder as the choice of IDE as this is the leading IDE for Java development on the market now. This comes as a 30 day trial version and we felt that this was enough time for us to test the features we needed, and to make a good test application. We also used Sun Wireless Toolkit 2.2 as a testing base for our application. Since we can add any desired emulator to this toolkit, we got to test the application on several different emulated devices.

3.2 Test Application

The best way to explore the capabilities of the different Optional Packages and the J2ME MIDP/CLDC platform in general, is to put them to use. This is what we aimed to do with this application which will be deployed and tested on a Nokia 6630 with integrated photo and recording hardware and the operating system Symbian OS v8.0

3.2.1 Use Case

To make this the following use case has been defined: *An inspector, e.g. a foreman at a construction site, wishes to report a detail in the construction back to a central computer. He takes a picture, classifies the detail, adds some measurements and records some additional audio comments. These recorded data will then be assembled into one message by the application and sent back to the computer where it will be analysed for further actions.*

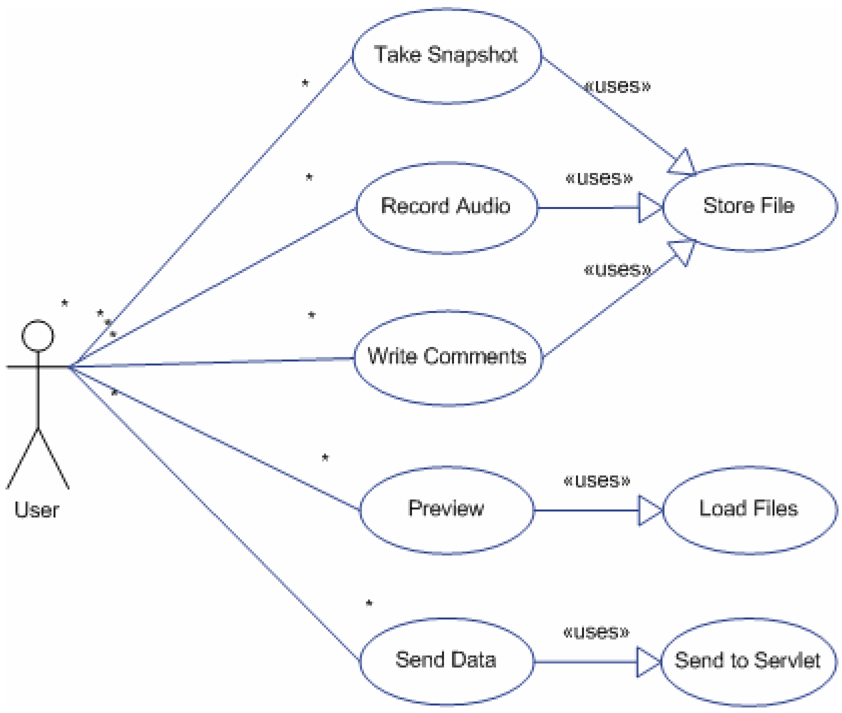


Figure 3.1 Use Case of the registration MIDlet RegApp

3.2.2 User Interface

As we have seen in previous chapters J2ME/MIDP has limited standard GUI components. Therefore the design for our test-applications user interface is a minimalistic one. To make it we used the following mix of low-level and high-level MIDP UI classes:

Screen

This is the common superclass of all high-level user interface classes.

Form

A Form is a Screen that contains an arbitrary mixture of items: images, read-only text fields, editable text fields, editable date fields, gauges, and choice groups.
We use this to contain and organize where there are several elements like TextBoxes and Strings.

List

The List class is a Screen containing list of choices. It is ideal for simple menus, where all menu items are of the same class.

TextBox

The TextBox class is a Screen that allows the user to enter and edit text. We used this to typically take notes from the user, or to specify addresses and such.

Canvas

The Canvas class is a base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display.
This we used to contain the VideoController we needed to implement the camera function. Since it is low-level we have control over the location and size of the elements we put into it.

Navigation

The main menu is simply a List object with several elements which functions as a menu. The user has several choices:

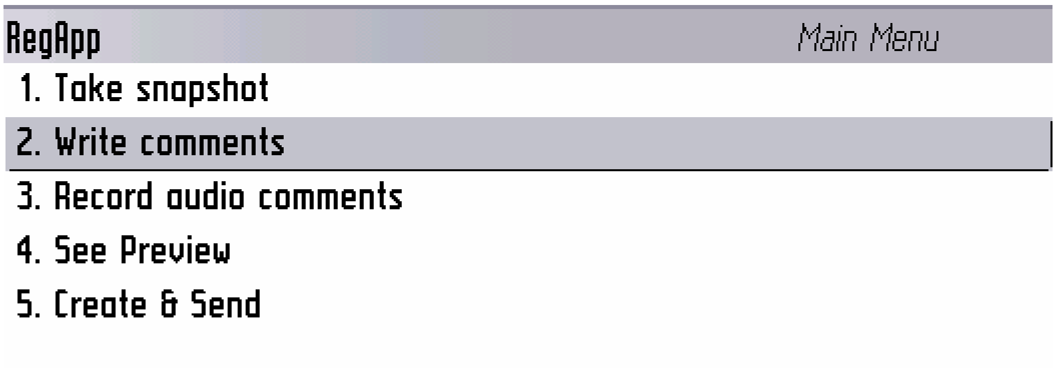


Figure 3.2 Available features in RegApp

Each of the choices leads to a new screen and you can at any time move back to the main menu.

3.2.3 Functionality

The whole application consists of the five choices in the menu. Each of them described here:

1. Take Snapshot: Selecting this menu-item displays a low-level GUI for taking snapshots. It contains simple functionality; simply take snapshot, and go back to main menu. Snapshots are stored in files for later use.

2. Record Audio Comments: This displays a simple start-stop audio recording interface. Contains start and stop functions. Audio is stored in files for later use.

3. Write Comment: This displays a big textbox for writing additional comments.

4. Preview: To be sure what you want to send is actually what you send, preview functionality is added.

This reads files from and displays the message in an orderly way.

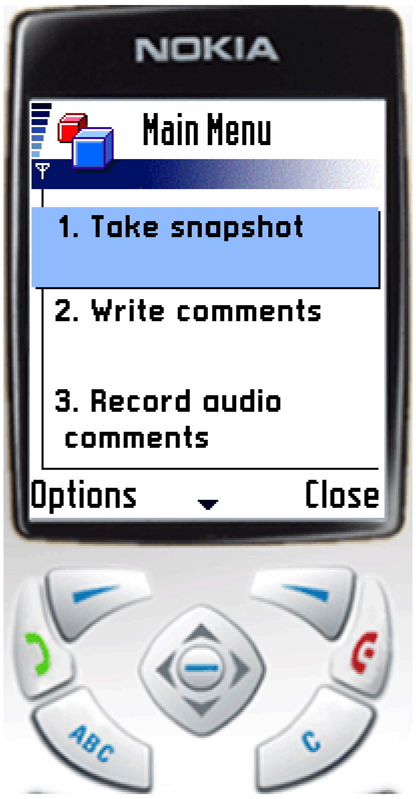


Figure 3.3 Main menu on RegApp shown on the Series 60 emulator

5. Send to Server: Displays different kinds of sending options. We have implemented MMS, Servlet, Socket and Datagram communication, but only Servlet communication will appear in the final application because of support issues.

3.2.4 Implementation issues

As the API support in the IDE and WTK differ from the support in the actual device, there were bound be some problems. Even though we had the newest version of Symbian OS available, it did not have support for the revised version of Wireless Messaging API (JSR205). We found out that this is because this Optional Package is not supported until Series 60 3rdedition, which are only implemented on Symbian OS v9. The first phone shipped with this OS is expected on the market 3Q or 4Q 2005. As a result the MMS implementation that worked just fine in the emulator made the application crash in the actual device.

3.3 Test Application Experiences

As a relatively young platform J2ME is constantly undergoing huge improvements. When reviewing the maturity of the platform one has to make a decision whether to review the version currently dominating the market or the latest version that is not yet available in any handsets. For instance, on the java-enabled phones people use today there are two dominating stacks. We have the original MIDP stack with CLDC 1.0 and MIDP 1.0 as figure *3.4* illustrates.

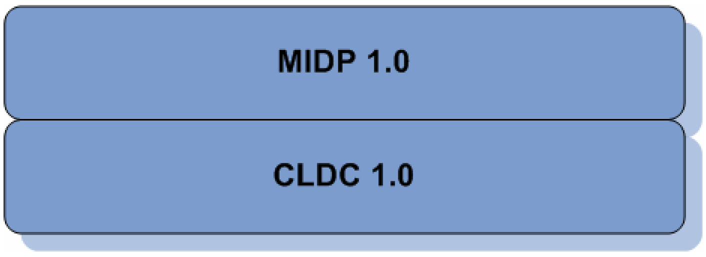


Figure 3.4 MIDP 1.0 on top of CLDC 1.0 [15]

Although this combination has been wildly successful since its release in September 2000, it is clearly just a start and not a mature platform for software development. It offers rather basic environment for general application development. Vendors had to make a lot of device specific APIs to make up for the lack of functionality, and this led to quite a fragmented platform for developers to use.

Then we have the JSR 185 stack, as illustrated below, which provides a wireless Java application environment that tries to reduce the fragmentation effect and improves portability.

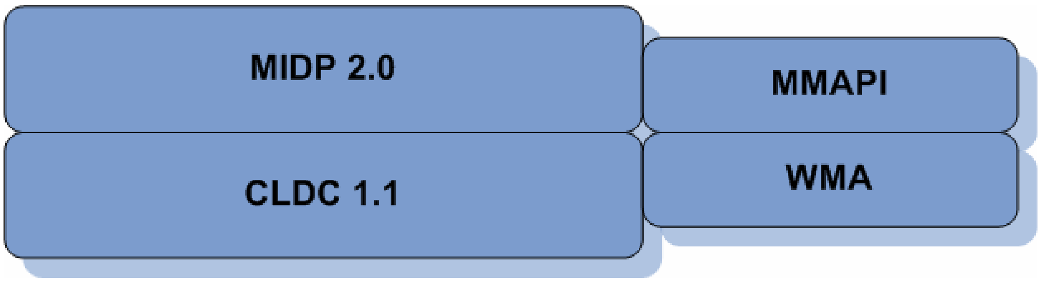


Figure 3.5 The JSR 185 stack [15]

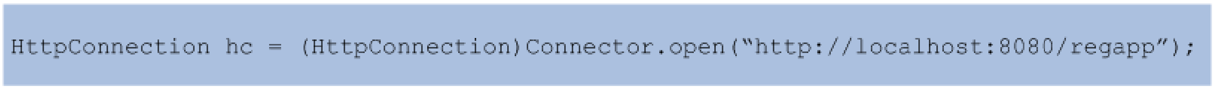
Fragmentation is addressed by providing many crucial capabilities in one standard application environment. Interoperability is addressed by clarifications to existing specifications and an exhaustive suite of compliance tests.

Our review of the J2ME platform will therefore only focus on the latter of the two versions mentioned above. To do this we made an application in which we incorporated a lot of features to explore the maturity of the J2ME APIs. The analysis of the platform is presented here, and the application itself is described in detail in *Appendix C*.

3.3.1 Using GFC

In our test application we used GCF for connecting to sockets, UDP-servers and a Servlet. We also used it to perform I/O operations on the file system on the mobile phone

Using the GCF is very simple. To create a connection you use the Connector factory class and a URL. To close it, you use the created Connection subtype object. Here is one code example to illustrate a connection made from a MIDlet to a Servlet:



```
HttpConnection hc = (HttpConnection)Connector.open("http://localhost:8080/regapp");
```

Figure 3.6 Connecting MIDlet to Servlet with HttpConnection (from test application).

All the connections made in the test application were created the same way. Needles to say, this makes the developers job a whole lot easier than if he had to use a new procedure on each of the different connection types. Of course there are differences when using the different connection types, because each connection type has its own peculiarities.

3.3.2 Networking capabilities

In the test application we chose to implement several ways of MIDlet/Server communication. We created three different servers: A simple servlet, a simple TCP server and a simple UDP server. The emphasis in this thesis is on the MIDlet-side of the system, and the servers were given one task only; just reassemble the message received from the MIDlet and display it.

3.3.2.1 Datagrams and Sockets

Using datagrams as means of communication has the advantage that they are rather lightweight when compared to TCP-based connections such as sockets. When programming applications for wireless devices with limited network capacity this is clearly a thing to consider. In the process of making the test application we tested the UDP and TCP protocols as means to send a composite message from a MIDlet to a server.

```
Connection conn= Connector.open(“socket://localhost:6789”);
Connection conn= Connector.open(“datagram://localhost:9876”);
```

Figure 3.7 An example of socket and datagram connections

It is no problem to send data from a MIDlet to a server using these two protocols. We just converted the data files to byte arrays and sent them over an OutputStream object. However, none of these protocols are mandatory implementations in the MIDP platform; it is entirely up to the handset manufacturers and network operators to deploy these capabilities [16]. We chose therefore just to test them out, but not make them part of the final application.

3.3.2.2 Http communication

As mentioned, sockets and datagram communications are network dependent. And some networks may implement only one of these, and not the other. This clearly makes any datagram or socket based application less portable. Because HTTP support is mandatory in MIDP devices and HTTP is a high-level, standard network-independent protocol, this gives wireless applications developed using HttpConnection a very high level of portability. HTTP communication also makes it easier to deal with issues such as network security and firewalls, because the HTTP's well-known port 80 is the least likely port blocked by firewalls.

In the test application we use HttpConnection to communicate with a servlet and to send messages containing pictures and audio. Below is an example of how to send data to a Servlet from a MIDlet using the HTTP POST request.

```
private void sendToServlet() {
        try {
            String url = servUrl.getString(); // a URL
            HttpConnection conn = (HttpConnection) Connector.open(url);
            conn.setRequestMethod(HttpConnection.POST);
            byte[] data = new byte[10]; // the data to send
            for (int i = 0; i < data.length; i++) {
                data[i] = (byte) i;
            }
            conn.setRequestProperty("Content-Length",
                                    Integer.toString(data.length));
            OutputStream os = conn.openOutputStream();
            os.write(data);
            os.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
```

Figure 3.8 Example on sending data to a Servlet from a MIDlet using the HTTP POST request

Figure 3.9 is a sample from an early version of the test application. A more complex *sendToServlet()* method is found in the final version.

To send multiple files as we did in the test application, we found that the easiest way to do this was to convert all the files to byte arrays and implement a small protocol. First we send a String message, indicating the file type arriving in the succeeding stream of bytes, and then the payload is sent. This is repeated for each file.

3.3.3 File access

File access for MIDlets has been an issue since Sun decided to move away from Personal Java and JavaPhone and to put their efforts into J2ME instead. With the FileConnection API however, this important hurdle has been overcome. The API is very simple containing just one class, two interfaces, and two exceptions. As a part of the GCF, the FileConnection interface extends the Connection interface and gives access to directories and individual files.

Implementations of FileConnection are created using the *Connector.open()* method. The argument of the *open()* method is an URL with the format *file://<host>/<root>/<directory>/<directory>/.../<name>*, and a parameter to decide if read and write rights will be given.

The host element may be empty, and it often will be, when the string refers to a file on the local host. The root directory corresponds to a logical mount point for a particular storage unit. Root names are device-specific. The following table provides some examples of root values and how to open them:

Root Value	How to Open a FileConnection
`CFCard/`	`FileConnection fc = (FileConnection) Connector.open("file:///CFCard/");`
`SDCard/`	`FileConnection fc = (FileConnection) Connector.open("file:///SDCard/");`
`MemoryStick/`	`FileConnection fc = (FileConnection) Connector.open("file:///MemoryStick/");`
`C:/`	`FileConnection fc = (FileConnection) Connector.open("file:///C:/");`
`/`	`FileConnection fc = (FileConnection) Connector.open("file:////");`

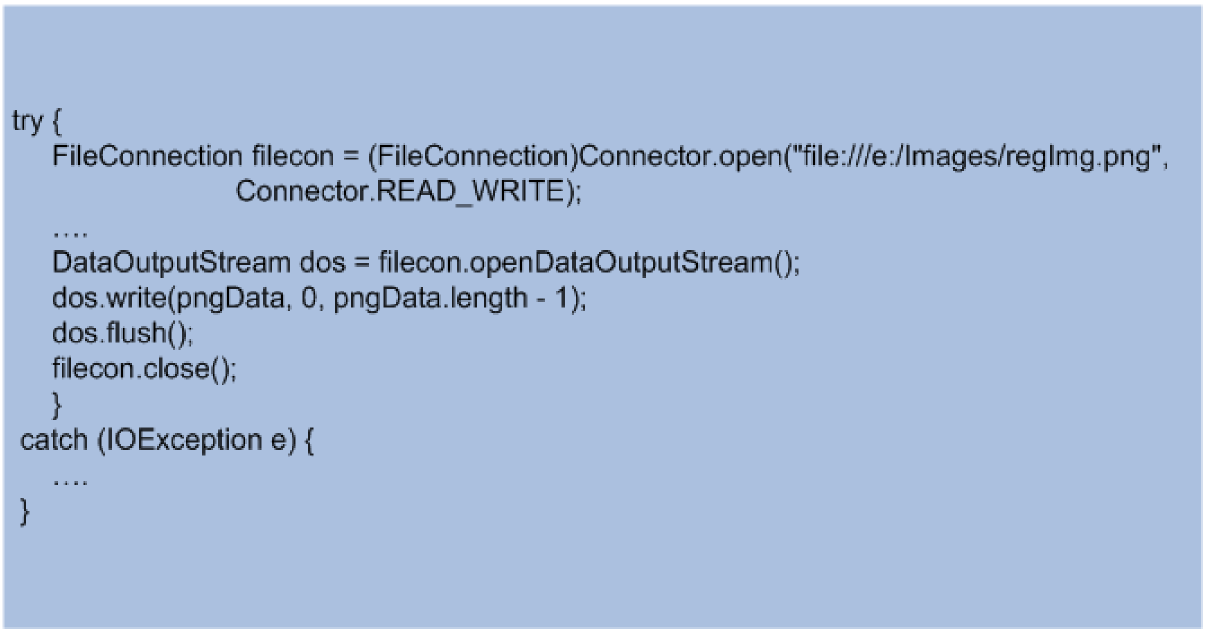
Figure 3.9 Some GCF root values and how they could be opened[17]

When a connection to the file system is established, there are several kinds of operations that can be performed. FileConnection includes amongst others [17]:

- Get a filtered list of files and directories using the method *list(String filter, boolean includeHidden)*. In the filter parameter you can use * as a wildcard to specify zero or more occurrences of any character. The *includeHidden* parameter specifies whether you want to list only visible files or hidden files as well.
- Discover whether a file or directory exists using *exists()*.
- Discover whether a file or directory is hidden using *isHidden()*.
- Create or delete a file or directory using *create()*, *mkdir()*, or *delete()*.

For a list of all the valid root values in a device, call the *listRoots()* method of FileSystemRegistry.

FileConnection behaves differently from other Generic Connection Framework connections in one important way: The *Connector.open()* method can return successfully without referring to an existing file or directory. This capability enables you to create new files and directories. Here is a segment of code that creates a new file; assume SDCard is a valid file-system root:



```
try {
    FileConnection filecon = (FileConnection)Connector.open("file:///e:/Images/regImg.png",
                      Connector.READ_WRITE);
    ....
    DataOutputStream dos = filecon.openDataOutputStream();
    dos.write(pngData, 0, pngData.length - 1);
    dos.flush();
    filecon.close();
    }
 catch (IOException e) {
    ....
 }
```

Figure 3.10 Example on how to create a file

In the test application we wanted to have persistent storage of the registration data, and therefore we used the FileConnection to read and write files. It works smoothly as soon as you get to know the file system.

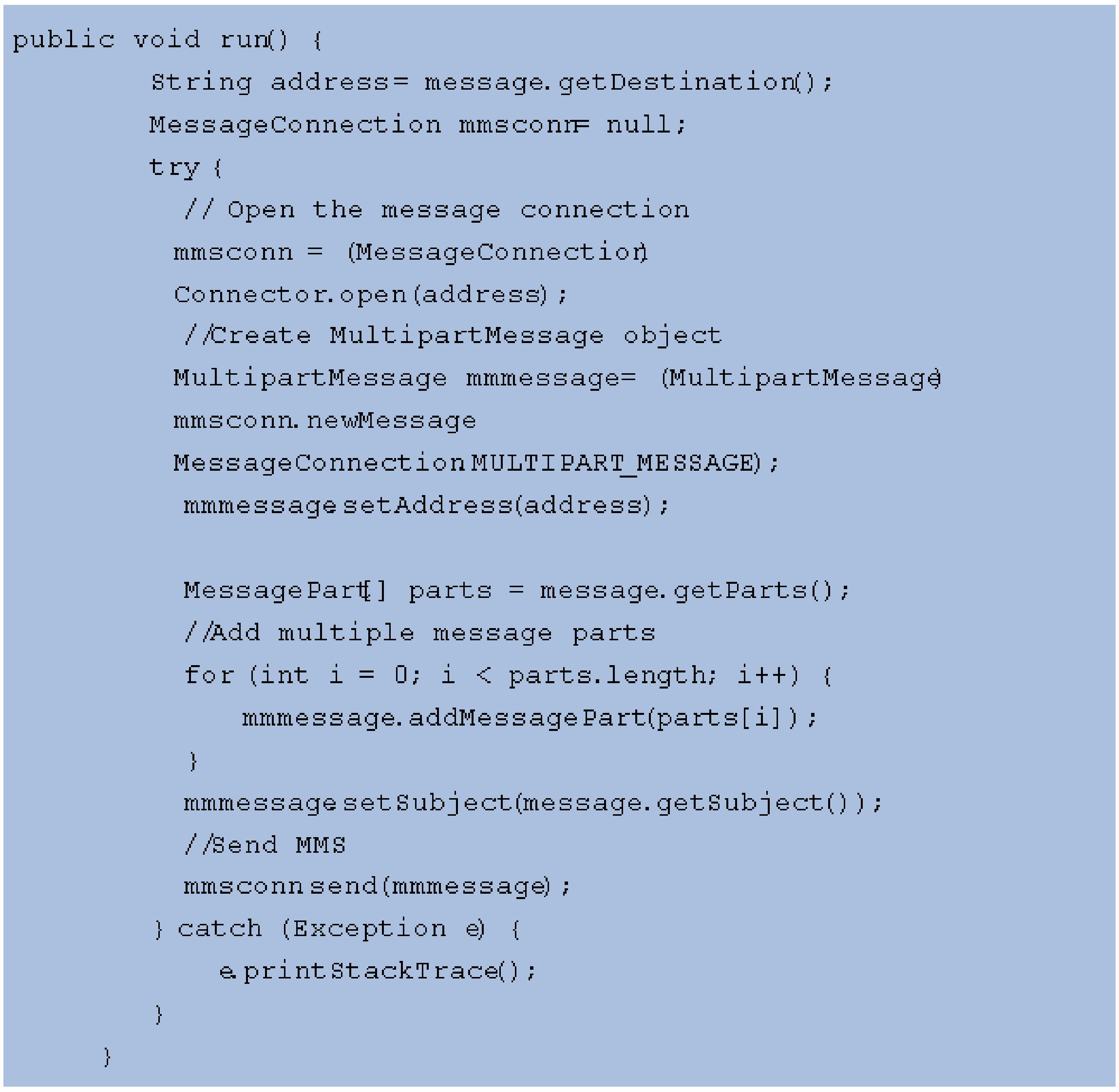
3.3.4 Wireless Messaging

Sending messages with the Wireless Message API is really not a problem. For sending a text message we just have to create a MessageConnection object and pass it a parameter to say it will send text messages. Then we create a TextMessage object and use the *setAddress()* method to set receiver address, and *setPayloadText()* to fill it with a String message. The Message object is sent with the MessageConection's *send()* method. A sample from the application is displayed below.

```
public void sendTextMessage(MessageConnection mc, String msg, String url) {
    try {
        TextMessage tmsg =
            (TextMessage)mc.newMessage(MessageConnection.TEXT_MESSAGE);
        if (url!= null)
            tmsg.setAddress(url);
        tmsg.setPayloadText(msg);
        int segcount = mc.numberOfSegments(tmsg);
        if (segcount == 0) {
            alertUser();
        }
        else
            mc.send(tmsg);
    }
    catch(Exception e) {
        //  Handle the exception...
        System.out.println("sendTextMessage " + e);
    }
}
```

Figure 3.11 Example on creating and sending SMS

To send an MMS is not much worse, the difference is that we have to create a MultipartMessage object which can, as the name implies, contain multiple message parts. These can be files such as images or video, and also plain text messages. This feature arrived first with the revision of the WMA, the JSR205, and at the time of writing the application, no phone supported this. It worked without hassle in the emulator, and there is no reason it should not work in a device which supports the revised WMA. Below is part of the MMS implementation. It's not included in the final application, as it will not work on the actual device. It is merely included here to show how it is done.



```
public void run() {
        String address= message.getDestination();
        MessageConnection mmsconn= null;
        try {
          // Open the message connection
         mmsconn = (MessageConnection)
         Connector.open(address);
          //Create MultipartMessage object
         MultipartMessage mmmessage= (MultipartMessage)
         mmsconn.newMessage
         MessageConnection MULTIPART_MESSAGE);
          mmmessage setAddress(address);

          MessagePart[] parts = message.getParts();
          //Add multiple message parts
          for (int i = 0; i < parts.length; i++) {
              mmmessage.addMessagePart(parts[i]);
          }
          mmmessage setSubject(message.getSubject());
          //Send MMS
          mmsconn send(mmmessage);
        } catch (Exception e) {
            e printStackTrace();
        }
    }
```

Figure 3.12 Creating and sending MMS Example from RegApp

There is a third option in the WMA, namely binary messages. This allows the developer to convert the entire message into bytes and send it as a byte-stream. We figure this can be an alternative to MMS for devices without JSR205 support.

Sending a binary message is no worse than sending a text message, as we can see in this code sample.

```
public void sendBinaryMessage(MessageConnection mc, byte[]msg, String url) {
    try {
        BinaryMessage bmsg=
            (BinaryMessage)mc.newMessage(MessageConnection.BINARY_MESSAGE);
        if (url!= null)
            bmsg.setAddress(url);
        bmsg.setPayloadData(msg);
        mc.send(bmsg);
    }
    catch(Exception e) {
        //  Handle the exception..
        System.out.println("sendBinaryMessage " + e);
    }
}
```

Figure 3.13 Example on creating and sending a binary message

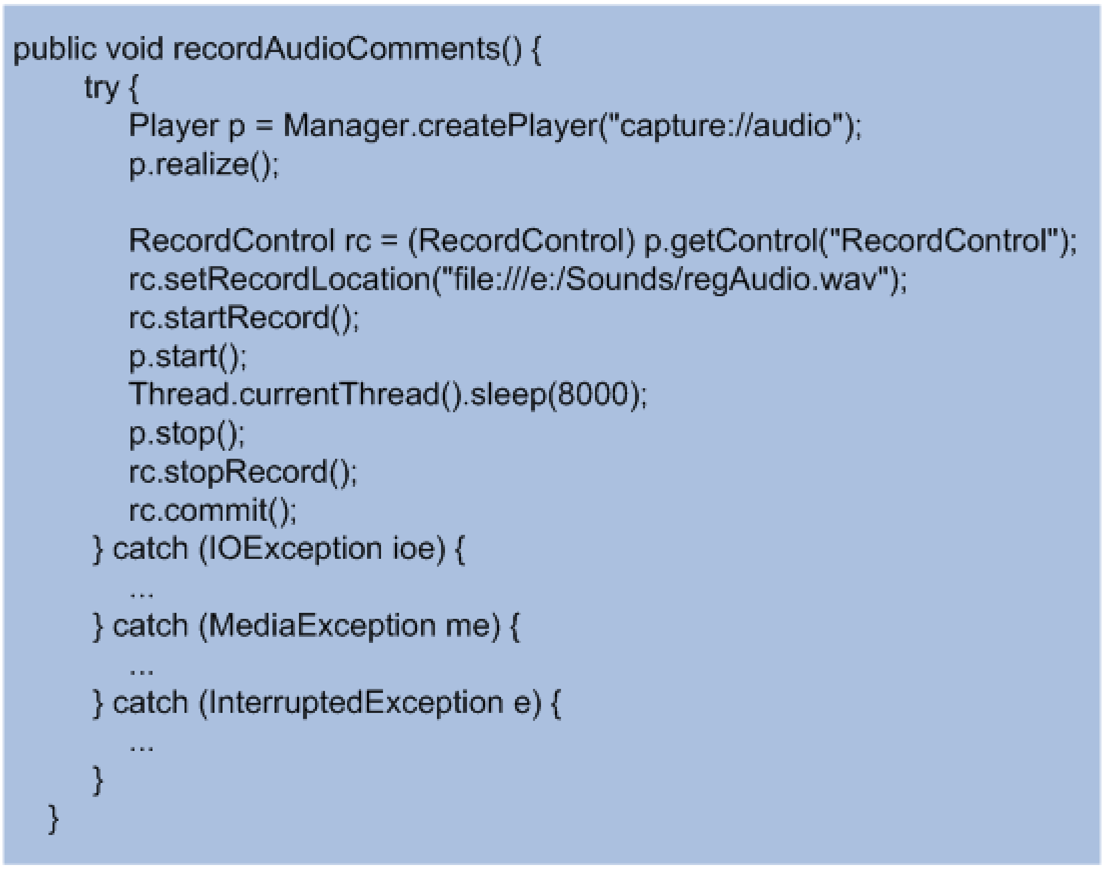
3.3.5 Hardware control

In our application we instantiate a video capture player object by passing the URI locator "*capture://video*" to the *Manager.createPlayer()* factory method. Then we display the resulting video in a canvas and we are able to grab a snapshot from this by calling the *VideoControl.getSnapshot()*. We can pass arguments to this method to adjust the type and the dimensions of the resultant image. Below is a code sample from the application, which shows how we implemented this feature with help from Forum Nokia.

```
CameraCanvas(RegAppMIDlet midlet) {
        .......
        try {
            player = Manager.createPlayer
            player.realize();
            //Grab the video control and s
            videoControl = (VideoControl)
                    "VideoControl"));
            if (videoControl == null) {
                discardPlayer();
                message1 = "Unsupported:"
                message2 = "Can'tgetvideoc
            } else {
                videoControl.initDisplayMo
                ...
               //Code for placement of the
                 ...
            }
        } catch (IOException ioe) {
            discardPlayer();
            ...
        } catch (MediaException me) {
            discardPlayer();
            ...
        } catch (SecurityException se) {
            discardPlayer();
```

Figure 3.14 Creating a visible video controller and taking a snapshot in the test application

The procedure for implementing audio recording is somewhat simpler, since it does not require a display, such as the Video display we used for the snapshot function. But it is similar in the way that we use the *Manager.createPlayer()* method.



```
public void recordAudioComments() {
        try {
            Player p = Manager.createPlayer("capture://audio");
            p.realize();

            RecordControl rc = (RecordControl) p.getControl("RecordControl");
            rc.setRecordLocation("file:///e:/Sounds/regAudio.wav");
            rc.startRecord();
            p.start();
            Thread.currentThread().sleep(8000);
            p.stop();
            rc.stopRecord();
            rc.commit();
        } catch (IOException ioe) {
            ...
        } catch (MediaException me) {
            ...
        } catch (InterruptedException e) {
            ...
        }
   }
```

Figure 3.15 Creating and using AudioRecorder in the test application.

Even though recording audio did not pose a problem, the replay did. Loading the recorded wav file into the memory took so much time it was not any point including the replay function in the final version of the application.

3.3.6 GUI

In order to show something on a MIDP device, you will need to obtain the device's display, which is represented by the *Display* class. This class is the one and only display manager that is instantiated for each active MIDlet and provides methods to retrieve information about the device's display capabilities.

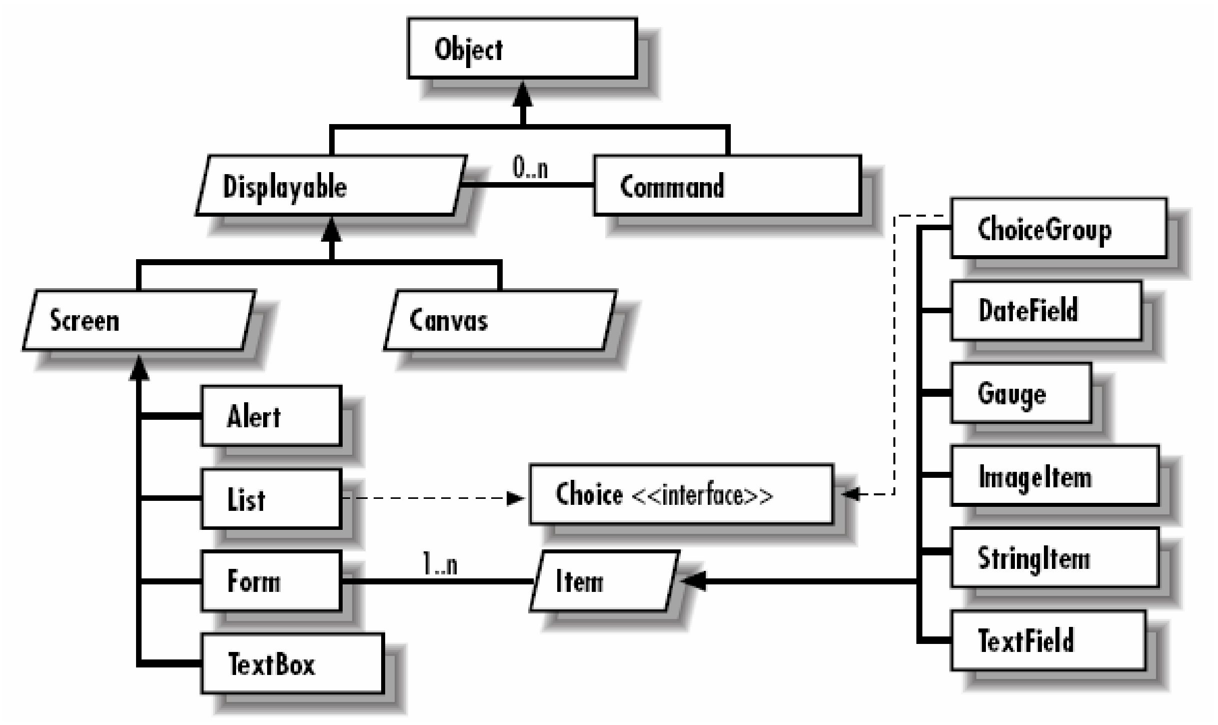


Figure 3.16 Overview of available GUI components in J2ME [18]

To make something useful for the user, you have to go further down the *javax.microedition.lcdui* tree to the level of the Screen class and the Canvas class.

We used three types of Screen implementations in our application; List, Form and TextBox. They are all straight forward and easy to use, but perhaps not as flexible as you would want. Not much creativity allowed, since the underlying implementation takes care of most of the placement and size issues.

In order to directly draw lines, text, and shapes on the screen, you must use the Canvas class. The Canvas class provides a blank screen on which a MIDlet can draw. We used this to display the VideoController output in the snapshot function.

For an application like the one we have made in this project you can do fine with a mix of low-level and high-level MIDP UI API’s.

3.3.7 General programming Issues

Due to the nature of the targeted devices, J2ME and MIDP are understandably limited. Here we will go through some of the general limitations you experience when moving from J2SE development to the mobile world of MIDP [19].

Serialization

Serialization of objects comes in handy when data classes such as the SessionData class in our test application. This is a class whose only job is to store images, audio and text, and serializing this object would make it much easier to transfer these data over a byte stream. Since MIDP does not support serialization this process becomes quite cumbersome.

Exception Handling

Exception handling is resource-expensive and is therefore limited in J2ME. For instance, CLDC only defines three error classes: *java.lang.Error*, *java.lang.OutOfMemoryError*, and *java.lang.VirtualMachineError*. This imposes extra care in coding and testing for the developer.

Finalization

And you can not do finalization in J2ME. It is unwise to rely too much on this even when using J2SE, but at least you have the possibility to do so if you wish. In J2ME this possibility has been removed.

Threading

There are no thread groups or daemon threads in J2ME, however MIDP supports multithreading. Thread groups can be created at the application level by using a collection to store the tread objects.

GUI

Large UI APIs such as Swing and AWT are not suited to be used on a small device, and therefore MIDP implements its own set of UI APIs that fits the smaller screen size and minimal resources. Divided into high-level and low-level UI, this provides the developer with easy to use UI components and the ability to draw on the display. The high-level components leave a lot of the GUI design up to MIDP, and thereby limit developers' freedom. And the low-level UI is a bit too low level to be used in fast development of applications.

4 Discussion and Conclusion

When J2ME was first introduced to the Symbian platform it was as a kind of second class citizen. It did not have the features to compete with the native C++, but it had something that we believe has contributed to its continuous existence; portability. However J2ME needs to prove itself in more ways than this to defend its place as a prioritized language on the Symbian platform. Through this paper we have examined some specific problem areas and we have looked at the evolution of J2ME and Symbians co-existence.

4.1 Discussing maturity and features

4.1.1 Focus Areas

The first area we examined was the hardware control. This had long been a weakness for J2ME on Symbian. What we experienced currently the situation was rather the opposite; the arrival of the MMAPI has made developing hardware controlling software such as camera and audio recording apps not only possible, but easy as well. Of course, the features are not very advanced; it is basically just record and play functions that are implemented. For instance, neither zooming nor filtering is implemented.

As mentioned in chapter *2.4.1* a new JSR had its final release the 20th of June this year called JSR 235 ASMAPI. This will greatly enhance the developers' capabilities to control cameras and other recording equipment. It will actually take a step closer to the features of the specialized devices out there. One can ask oneself why this hasn't been done before but, the answer is most likely that API development is just taking the steps one at the time. The MMAPI was designed to be easily extensible, and this pays of now as ASMAPI utilizes this framework by introducing control for advanced multimedia features.

File access have also been lacking in the J2ME/Symbian platform, and again we found that the problems have been mended. At least to the degree we needed to make the application without any problems.

Originally we intended to register data with the camera and microphone and send it as an MMS with the help of the WMA. However, the phone we used for this project does not support the revised WMA (JSR205), and we were therefore unable to send the data as MMS. Still, we did make an implementation that we only tested on the emulator.

When it comes to networking in J2ME we tried three different approaches: Socket, datagram and HTTP. All three are good ways to connect the application to servers but we found that an HTTP/Servlet solution is the best for mobile networking with J2ME. The HTTP protocol is more adaptable to the somewhat unstable network that mobile phones operate in.

We could have tested all these API's further, but this would be best to do as separate projects as they each would need to be studied at a much deeper level. To thoroughly review and suggest improvements needs the time and expertise at the level of JSR expert groups.

4.1.2 GUI

The general feeling we got from developing the GUI on the test application was that it was quite easy. But that is really just what we expected when we found out that MIDP 2.0 gave us very limited options.

The MIDP profile has too few GUI classes and this puts serious constrains on the developer if he wishes to develop a creative GUI solution. This is of course because MIDP is created to be the lowest common denominator for mobile devices, but when developing on advanced devices with operating systems like Symbian you want more control. J2SE GUI classes such as menus and drop down lists are sorely missed.

You can of course use the low-level GUI API to create your own hierarchy of components, but the moment you start to adapt the GUI components to the capabilities of each device, you instantly loose one of J2ME's major selling points; the portability. This is a challenge has to be overcome if J2ME applications are to be able to pass as first citizens in the Symbian environment.

4.1.3 Using Native Services

As described in the J2ME on Symbian chapter of the technical review, there is a way to break out of the sandbox and access native services, if necessary, via a daemon program on the local loopback address 127.0.0.1. This way of communicating gives the MIDP application indirect access to the full native API, and is a good solution if the project is depending on a few services that lie outside the reach of MIDP or when the device does not support a certain optional package.

This technique is not very common however, and this is probably due to the barrier of Symbian C++ programming. For a MIDP developer this can be a daunting task to take on, but the benefits are clearly there for grabs if one is willing to take the challenge.

One other downside is of course the portability issues that appear once you breach the sandbox, but this is the prize to pay for access to native functionality.

4.2 Conclusion

As mobile phone technology moves forward with an increasingly high pace, the software industry has a tough job keeping up. When looking at the variety of Symbian OS based phones on the market, it is easy to see that this progress is creating a very fragmented market for developers of mobile applications to work in. Even though all Symbian platforms support Java in one way or the other, this is not a uniform support. From the Symbian 6.X to 8.0 which are the platforms we have looked at in this project, the range in Java support stretches from PersonalJava/JavaPhone to J2ME MIDP2.0/CLDC1.1 and the differences here are substantial. Even within MIDP2.0/CLDC1.1 based devices there are differences in optional packages that make programs less portable. Therefore, the evaluation of J2ME is a difficult task.

We decided that we would focus on the platform on our chosen device, the Nokia 6630, as this had the newest version of Symbian OS and supported the most optional packages.

The general impression of this platform is that it is streamlined to develop simple applications fast. Being used to work in environments like J2SE and J2EE it is not hard to get into J2ME programming. The language itself is grammatically the same, but it requires a slight change in the way the developer thinks. The tools we used were mature in the way that they provided help in all parts of the process, so that we could focus on the coding.

GUI programming in J2ME is easy. Very few standard high-level UI elements help you to get the complete overview of possibilities. Unfortunately, this has a downside for the more creative developers as it limits GUI freedom.

The hardware control provided by the optional package MMAPI was also very good considering ease of use. However it lacks features to exploit the technical finesses of the hardware. However, there are improvements coming in the near future with a supplement package specified by the Java Community Process.

As far as networking goes there were little problems to find; at least in the developing process. We tested socket and datagram networking and HTTP/Servlet communication. How it works in works in real-life environments with the mobile networks is outside the scope of this project.

When reviewing the whole platform of J2ME on Symbian OS, we will have to say that there is still quite a way to go before it is fully matured. There is a lack of richness in the J2ME language that limits development of advanced applications. The fact that one can access native services through a C++ daemon application is of course helpful, but should be seen as a shortage of features and should not be considered as part of the J2ME features.

Our claim is that a development language is never more mature than the platform it will be used on. An application will always be limited by its environment. The lack of maturity and features is therefore not due to limitations in the J2ME itself, but rather in the willingness of the mobile device manufacturers to agree amongst them selves to implement standard APIs.

4.3 The future of J2ME on Symbian OS

The 2nd of February 2005, Symbian Limited today announces the launch of Symbian OS™ version 9, the latest evolution of the world's leading smartphone operating system. According to the executive vice president of marketing at Symbian, Marit Døving: "Symbian's strategic focus is to ensure that Symbian OS is the ideal choice for Symbian OS licensees' development of smaller, less expensive and more powerful smartphones," said Marit Døving, Executive Vice President, Marketing at Symbian [20].

Regarding J2ME support on Symbian OS, they are still supporting the newest configurations and profiles [21] in the coming versions of the OS and Symbians intention is clearly to make J2ME a first class citizen in the Symbian OS environment.

The fact that Symbian is now aiming for the masses, instead of just high-end mobile phones is a certain sign that the Symbian OS is expanding its territory. And with Symbians efforts to stay in front with J2ME technology, we believe J2ME will evolve and mature alongside with the Symbian OS.

PART TWO – DEVICE INDEPENDENCE

5 Device Independence

The idea that applications can be used across all platforms and computer hardware is a good one, but it seems unattainable when the enormous diversity is considered. The leap from a stationary PC to a small mobile phone is still just too great. But then again it is not very likely that you would want to use your mobile and PC for a lot of the same tasks. Of course, web browsing and similar tasks can be done with both, but the “heavy” computing jobs are still the PC’s domain. This is much because input and display makes working inconvenient on small devices, but also because computing power is still much greater on a PC. This thesis will therefore focus on device independent software development on a more homogeneous group of devices, namely mobile phones with J2ME/MIDP support.

The motivation behind device independence is first and foremost cost- and time efficiency, since the idea is to develop once and run/deploy anywhere. The aim is to determine whether it is feasible to develop applications that are fully portable across devices that support the Java platform, and to propose frameworks for such development.

5.1 Diversity issues

Even though mobile phones are used for many of the same tasks, the diversity amongst them is huge. As we see in the picture below, the designers really do not have many restrictions on their work.

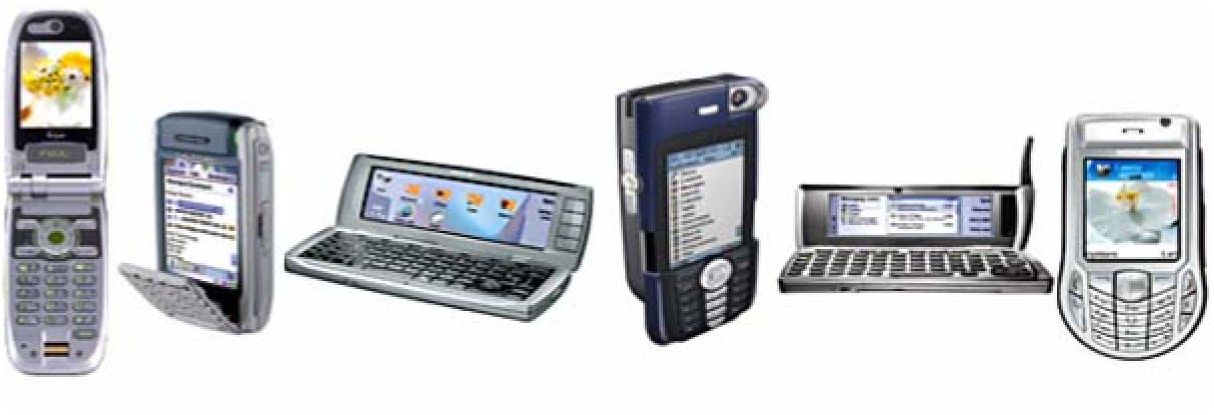


Figure 5.1 Diversity in design

For software developers this poses several different challenges, which are discussed in this chapter.

5.1.1 Graphical User Interface

GUI's differ a lot in the world of mobile devices, as we can see in the figure above. Although the technology moves towards bigger and better displays all the time, they range from the very small to almost small laptop sized ones. In addition to their variation in shape, they vary in resolution and range of colours. This, of course, poses a great challenge to any developer working with mobile technology. At least if the application is intended to be used by a group of consumers with different devices, as these groups rarely are homogenous. The developer will then need to know the features of each device before designing an application.

There are several ways of aligning GUI components such as menus and buttons on a screen, depending on the size of each different display. When considering the more cosmetic issues of a GUI, the screen resolution and colour depth possible on the device comes in to play. When developing for more than one type of phone it is easy to use the "lowest common denominator" approach. This usually results in a decent GUI for the low-end devices, but the features of the high-end devices will not be used to the full extent. Although the application might be useful it will most likely not have an optimal design.

5.1.2 Input Devices

As with displays, there are several different types of input devices that might be found on a MIDP platform. There are the standard numeric key-pads that most mobile phones come with, and there are more sophisticated solutions like full QWERTY key-pads, or touch screens. To make good applications input devices must be taken in to consideration during the design.

5.1.3 Platform Fragmentation

Another problem to consider when developing applications for mobile phones is the great diversity in platforms. It seems as if every single device has a different platform. This is much due to the fact that in the world of mobile technology business there is no dominant factor; at least not to the same extent as we see on operating systems for PC's, where Microsoft is able to act like a monopolist. There are a lot of negative aspects to monopolism, but one thing it has contributed to the PC market, is standardisation.

The lack of standards in mobile technology is a problem for any developer. At least it increases the workload, since every time a new platform is encountered the developer needs to thoroughly study its features in order to fully take advantage of the system.

The MIDP implementations on each device are often bundled by the device's manufacturer and are considered to be part of the system software, i.e. they provide the underlying implementations that interact with the operating system. This allows the implementation to access features that would normally be off-limits to a third-party implementation, and also ensures a good degree of integration with the other parts of the system. The down-side for a device independent developer is that this integration often results in a fragmentation of the whole mobile market. This means that we have a situation where different platforms allow certain functions to be accessible and some do not.

5.2 The J2ME Approach

As platform independence is Java's foundation stone and one of its best features, Sun faces some expectations in the device independence area. This chapter takes a closer look at what exactly is J2ME approach to this issue.

5.2.1 The Java Virtual Machine

Being the cornerstone of all Java platforms, the JVM is the component that makes Java portable across different hardware and operating systems. The job of the virtual machine is to be an abstraction of the machine it is running on. It interprets java class-files and makes calls to the underlying system based on these [22].

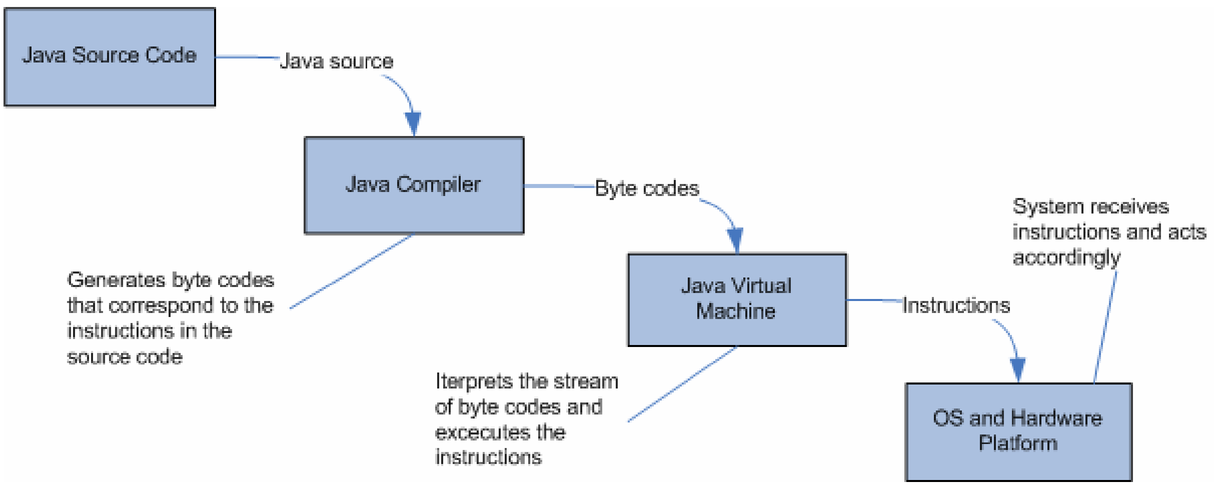

Java Source Code
Java source
Java Compiler
Byte codes
Generates byte codes that correspond to the instructions in the source code
Java Virtual Machine
Instructions
System receives instructions and acts accordingly
Iterprets the stream of byte codes and excecutes the instructions
OS and Hardware Platform

Figure 5.2 Java Program Execution

Each particular host operating system needs its own implementation of the JVM and runtime. The virtual machine itself is therefore not device independent, but it acts as an abstraction of the underlying system so that Java programs can run independently. The virtual machines interpret the byte code semantically the same way, but the actual implementation may differ from platform to platform. More complicated than just the emulation of byte code is compatible and efficient implementation of the J2ME core API which has to be mapped to each host operating system. This way J2ME applications can, in principal at least, be ported to any hardware running the java virtual machine. Although a certain level of device independence is reach this way, there are still the issues mentioned in chapter *5.1 Diversity Issues* that cannot be solved just by making code executable on each device.

5.2.2 MIDP portability

The high-level MIDP applications are portable across various all variations of MIDP enabled devices and they are usually designed for applications where portability across a range of handsets is desired. To achieve this portability, the APIs use a high level of abstraction from the underlying implementation provided by the device manufacturers.

When making GUI’s with J2ME, one has initially two approaches: High-level or low-level user interface API’s. The former is the one intended to provide the developer with portability while the latter is much more direct in the control of the screen.

High-Level GUI

The high-level user interface API provides the developer with a standard set of user interface components that stay rather consistently in functionality across different devices. However, their appearance and placement differs on different devices.

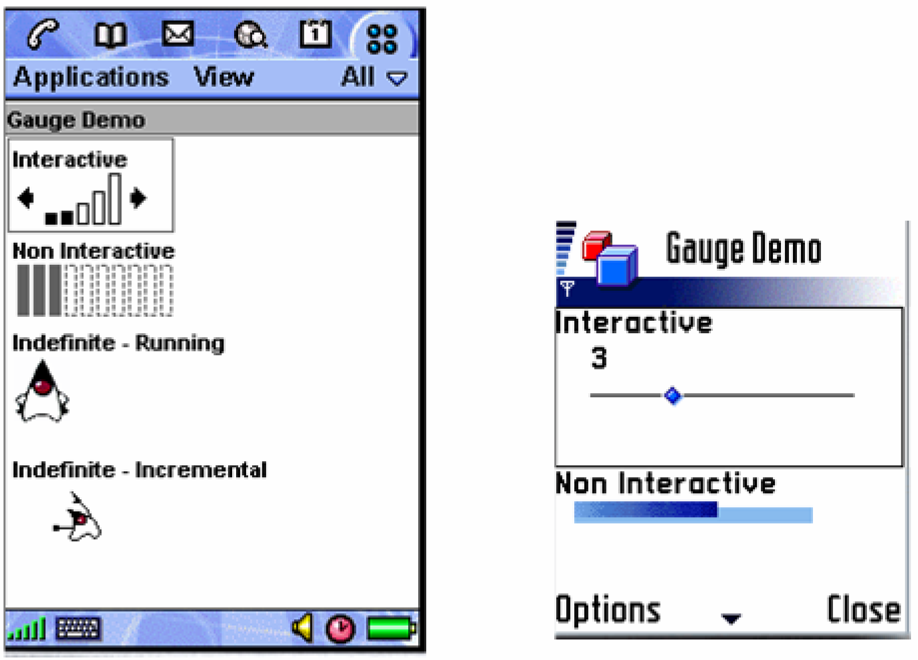


Figure 5.3 A high-level GUI sample (WTK2.2 sample). Same application running on Ericsson P910 (left) and Nokia series 60

This component behaviour is the result of a high level of abstraction in the user interface API, which actually uses interface components provided natively by the device. It creates a decent look to the application, but it leaves the developer with little control over the interface design.

Low-level GUI

Although the high-level API's provides a good set of UI-components, there is often a need for more detailed control. This is why MIDP provides the low-level UI API's, which are capable of almost direct control over the screen space allocated to the application.

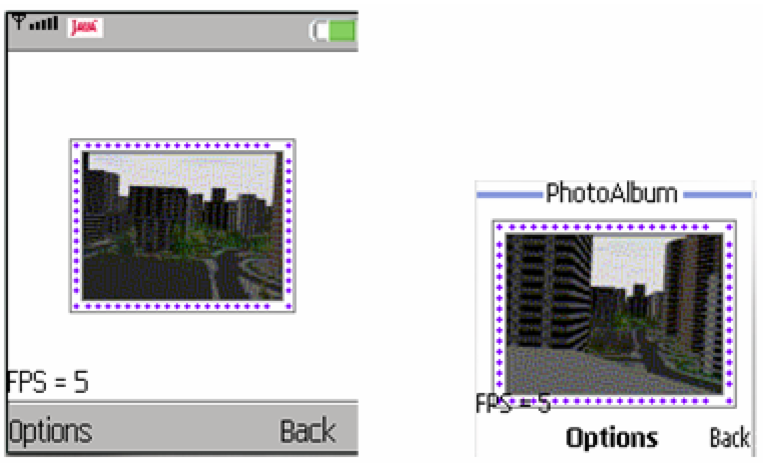


Figure 5.4 A low-level GUI sample (WTK2.2 Sample). Same application running on Ericsson K750 (left) and Nokia series 40

As we can see from the two pictures above, the difference is not that evident between low-level GUIs on different devices. This is because of the direct control the developer has through the low-level API's.

This direct control can easily become a problem when creating portable programs; however there are some tricks to use in order to minimize these problems. In chapter *6.3 General Techniques for portable J2ME Programming*, I will go through some of these.

5.2.3 Optional Packages

Problems arise when optional packages are added to a MIDP platform in order to improve exploitation of device features. These are strong contributors to the fragmentation of the platform, as the variations in supported optional packages tend to vary a great deal. On the other hand they are definitely a necessity as new device features emerge all the time. This fragmentation is one of the great challenges faced by developers of device independent MIDP applications.

5.3 Relevant projects

There are not exactly many projects on the field of making framework for development of portable J2ME applications. The most likely reason for this is that J2ME itself provides a certain level of portability, and there is constantly work being done to improve. Projects on device independence in general are more into making web-content device independent, or they go beyond frameworks and into the field of dynamic languages. Still, I managed to find some inspirational ideas from a few projects and they are described in this chapter.

5.3.1 Component Based Development

A component based development framework is an interesting way of solving portability issues. This is particularly the case with MIDlets as functions based on optional packages can put into components and be used when the platform allows this. Treating GUIs as components can also be advantageous.

In the paper, *”Migratable User Interface Descriptions in Component-Based Development”* [23], it is described how a component based approach can be combined with a UI description language to get more extendible and adaptable UIs for embedded systems and mobile computing devices. It envisions a new approach for building adaptable user interfaces for embedded systems, which can migrate from one device to another.

5.3.2 J2ME Polish

J2ME Polish [24] is a set of tools used to optimize J2ME applications for each device. It is not a tool for device independence as such, but it contains some ideas that can be useful.

In short J2ME Polish enables the developer to write code once and build it in device specific versions. Three ideas presented by the J2ME Polish documentation are especially interesting for this thesis.

Automated Code Optimization
The optimization is based on build-tools for creating application bundles, for multiple devices and multiple locales, out of one source project. This includes a code pre-processor which changes the code before compilation in order to optimize it for each device.

Device Characterization
J2ME Polish includes a device database in which the capabilities of known devices are defined. These capabilities are then incorporated into the code during pre-processing.

GUI Optimization

J2ME Polish includes an optional GUI, which can be designed using the web-standard Cascading Style Sheets (CSS). The GUI is compatible with the javax.microedition.ui classes; therefore no changes need to be made in the source code of the application. The GUI will be incorporated by the pre-processing mechanism automatically.

5.3.3 Content adaption

This is part of a field that is a bit on the side of this thesis. Content adaption is a technique used to make information viewable on multiple devices. The idea is to use this to build web pages and web applications that can be used on very limited devices as well as on stationary computers. Some members of the World Wide Web Consortium (W3C) have an ongoing project in this field. This group is called Device Independence Working Group (DIWG), and works on standardizing the World Wide Web in the hope of making it device independent [25].

Even though this particular group works on web page and web applications the ideas can be transferred to application development. It is possible to view the GUI in much the same way as web content, and thereby transpose principles from content adaption to GUI development.

6 Proposed Solutions

I have looked at two ways of approaching the problem of portability in MIDlets. They have similarities and differences, but the most important difference is that in the first one, portability is handled at runtime by the application on the device. In the other one the applications will not really portable in the same sense; it is more like a "portable idea" framework. Both the proposed frameworks ideas are extracted and modified from the projects mentioned in chapter *5.3 Relevant Projects,* and they exist only as propositions for solving device independence issues as they are not implemented and have not undergone full-scale testing.

6.1 Built In Context Adaptability (BICA)

As MIDP portability in itself is not an optimal solution because it is just capable of exploiting a small part of the available device features, there is a need for greater adaptability in applications. Here I propose a way of developing MIDlets with built in context adaptability.

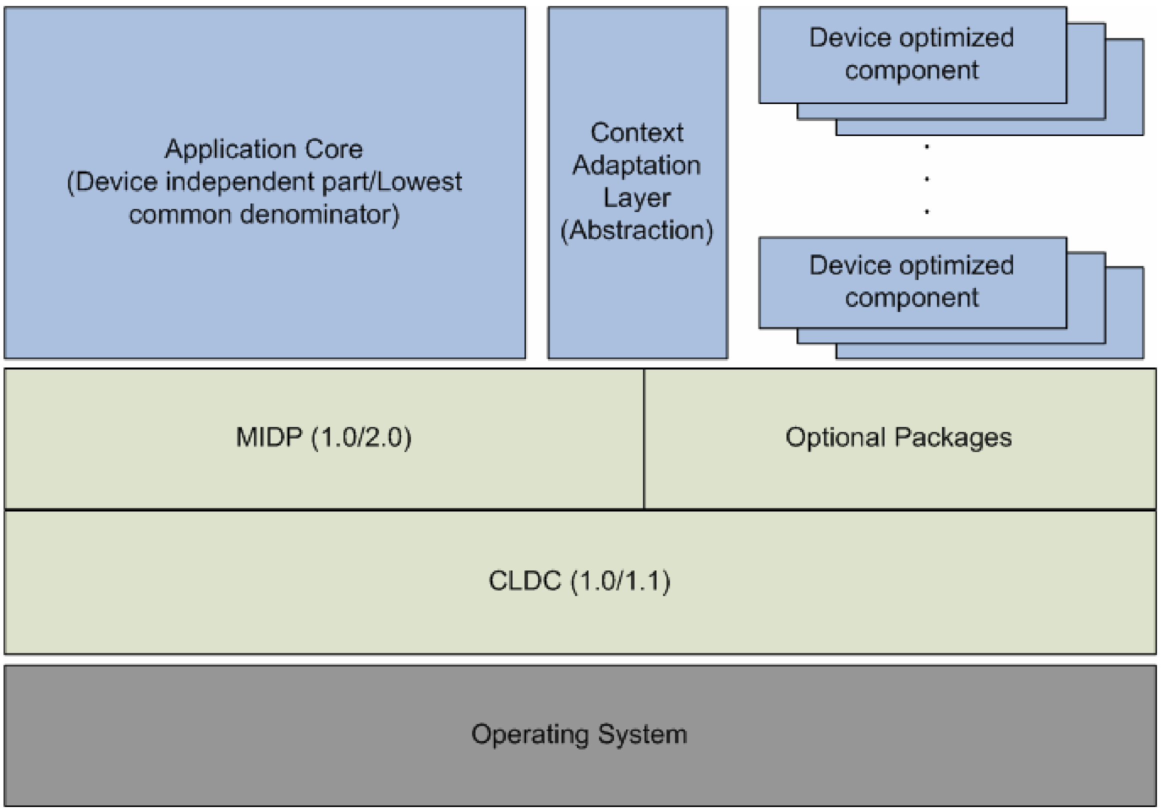

Application Core
(Device independent part/Lowest common denominator)
Context Adaptation Layer (Abstraction)
Device optimized component
Device optimized component
MIDP (1.0/2.0)
Optional Packages
CLDC (1.0/1.1)
Operating System

Figure 6.1 Illustration of the Built In Context Adaptability framework

The top blue area of the figure illustrates the application. It is split up into three major parts: The application core, device optimized components and a context adaption layer. These three are intended to replace the standard MIDlet structure by isolating the parts that causes portability problems.

6.1.1 Structure and Principles

On an abstract level this is the intended structure of an application developed using BICA:

Application Core:

This is where the main structure of the application will be located. No optional packages or device specific implementations are allowed here to avoid device and platform dependence.

Device Optimized Components (DOC):

These are program components which are optimized for different devices. For instance, if one platform does not support one optional package, a suitable backup component may be used, or in worst case, the feature may be excluded without this crashing the application.

ContextAdaptor:

In order to be able to fully adapt to different devices, the MIDlet needs to have knowledge about different devices features. It is the ContextAdaptor's task to read the system properties and use the correct DOC's according to these.

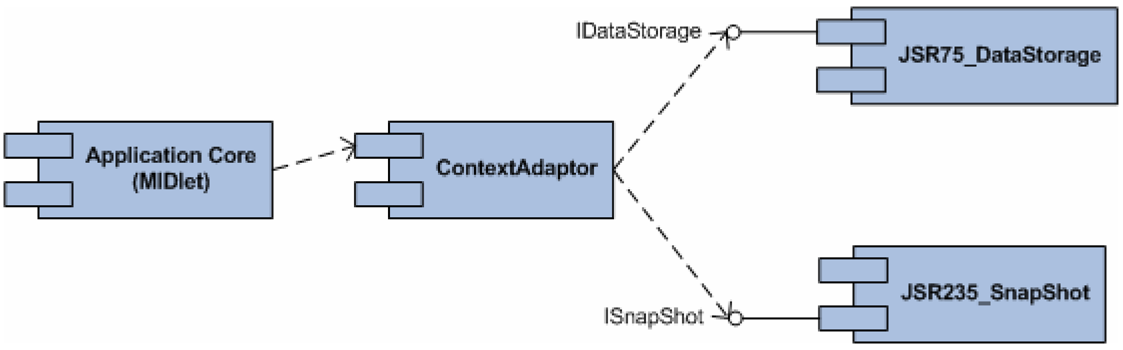


Figure 6.2 Component diagram showing component structure of an application created with BICA

The idea is that a MIDlet shall be aware of its environment at runtime. The core acts as the glue in the application, binding the different components together.

At start-up, the ContextAdaptor registers what kind of device it is running on. System properties are available to help determine the type of device on which the MIDlet suite is running. The core then consults the ContextAdaptor to find out which of the built in components the platform supports, and uses them accordingly.

Of course, this consultation may fail to advise the core of components being available if the platform has no such support. If this happens, the core will exclude the unsupported feature from the application.

6.1.2 Context Adaption in BICA

Portability usually means that some features will be weakened by too many constrictions in the platform that supports the least features. The BICA framework intends to solve this by including device optimized implementations of each feature in the MIDlet suite. If there are usable backup implementations possible for the features these will be used when the optimal solution is not present.

This structure is part of what is called the Context Adaption Layer (CAL). It works as an abstraction from the DOCs, allowing the core application to be unaware of the underlying implementations. Since each function is implemented differently for the different devices, the different versions of each DOC will all have to implement interface methods to make sure they act equally to the application. The core only uses the interface methods.

Device characterization is an important part of the ContextAdaptor. This process uses the system property names, defined by the various J2ME JSR documents. These can be queried at runtime, and provides two services [http://developers.sun.com/techtopics/mobility/midp/questions/properties/]:

- **To indicate the availability of an optional package:**
  For example, if the device supports the Location API for J2ME then the property *microedition.location.version* will be present. The value associated with it will be "1.0", to indicate compliance with JSR 179.

- **To provide platform-dependent configuration data**
  For instance, the property *microedition.commports* is present in the MIDP 2.0 specification. Its value is a comma-separated list of ports you can use to build a URL, which the Generic Connection Framework can in turn use to create a javax.microedition.io.CommConnection object.

One problem with this approach is that for some of the properties it is up to the devices MIDP implementations whether they are accessible or not. In this system the ContextAdaptor assumes that if the system properties are not accessible then the actual feature is not available either.

On start-up the MIDlet core creates an object of the ContextAdaptor which will be used throughout the MIDlets life cycle. At creation this object reads all system properties into a list and this list is what decides which DOCs will be used.

When a call to a DOC method is made, the ContextAdaptor directs the call to the correct DOC.

This is the intended structure of the ContextAdaptor class. It will obviously be more complex than this in practical use, but this illustrates the idea behind it.

```
public class ContextAdaptor{
      /*
      List of system properties, e.g. private String JSR75_File = "";
      */

      public ContextAdaptor(){
            readSystemProperties();
      }

      private readSystemProperties(){
            /*
            Reading all system properties, e.g.
            JSR75_File = System.getProperty
                  ("microedition.io.file.FileConnection.version");
            */
      }

      /*
      Various methods using correct components based on presence of
      system properties,
      e.g.

      public void storeData(String path, String name){
       IDataStore ds;
       if(JSR75_File == "1.0"){
            ds = new JSR75_DataStore();
            //Perform storage operation according to JSR75
       }
       else
            //Perform altarnative storage operation
      }
      */
}
```

Figure 6.3 ContextAdaptor class sample

A small test was conducted to check if the ContextAdaptor class worked as intended, and in this small-scale test with one MIDlet, the ContextAdaptor and two components it was successful. It showed however, that exception handling is important since the MIDlet suite has implemented functions for unsupported APIs as well as the ones the ContextAdaptor chooses to use.

6.1.3 Component Structure

The component structure is described below, through a small example. What we see there are two classes which implement an interface. The two classes are two versions of a component for storing persistent data. The idea is to have at least one device dependent and one device independent component under each interface, so that the application can provide one optimized solution, and one backup solution in case of lack of support in the device.

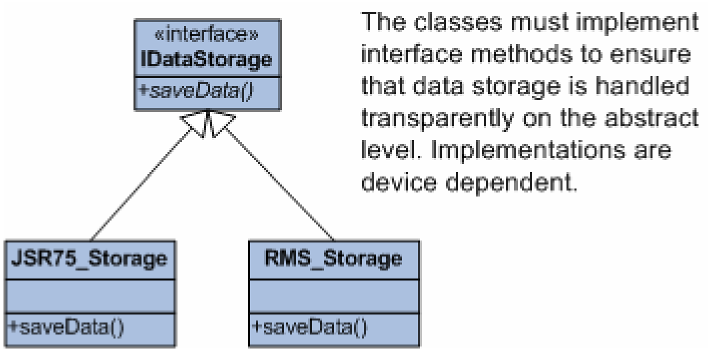

«interface»
IDataStorage
+saveData()
The classes must implement interface methods to ensure that data storage is handled transparently on the abstract level. Implementations are device dependent.
JSR75_Storage
+saveData()
RMS_Storage
+saveData()

Figure 6.4 Sample of component structure

6.1.4 Coding Guidelines

The components based on optional packages will be named according to the JSR numbering, and the components which are made from functions supported by all MIDP devices are named after the package they reside in.

6.1.5 Optimized Portable GUI

Regarding optimized portable GUIs, this is a subject that needs to be handled differently from the rest of the structure. To incorporate optimized portability into the GUI of applications developed with BICA, there will be a need for a new framework with more features than MIDP offers. Portable optimized GUI-components can be part of the frameworks component libraries, but layout poses a challenge. The number of elements and their alignment relative to each other is not quite as easy to handle.

In “Migratable User Interface Descriptions in Component-Based Development” [X], it is proposed that the portable GUI issue could be handled with a so-called render-component. In BICA this would be a component which reads GUI instructions from an XML-file and presents the user with its interpreted GUI; much like a browser interpreting HTML code. This feature has not been included in the proposed BICA framework because I did not have the time to implement and test such a component. However, it would be interesting to see how it would help complete the framework.

6.1.6 Example Scenario

Consider a MIDlet created to take a snapshot and store it in a file on a mobile phone before viewing it. It is a very simple application with no purpose but to illustrate the usefulness of Built In Context Adaptability. This application is targeted for two different mobiles; the Nokia 6630 and the Nokia 6230.

To implement this, two optional packages are needed: Mobile Media API and the FileConnection API. Nokia 6630 supports them both but Nokia 6230 supports only the Mobile Media API.

Both devices will receive the exact same implementation, and both will work. The difference will be that the application in 6630 will work 100% correctly, and the Nokia 6230, that cannot store files, will use MIDP Record Management System (RMS) instead. Using standard MIDlet programming would result in the application crashing in the Nokia 6230, but using BICA it will still be functional.

6.2 Build On Demand Framework (BODF)

The BODF does not make applications portable as such, but rather it recreates the application, optimized for a device, each time a user requests a download. It is a little bit outside the scope of portability but it targets many of the problems encountered when trying to create portable applications.

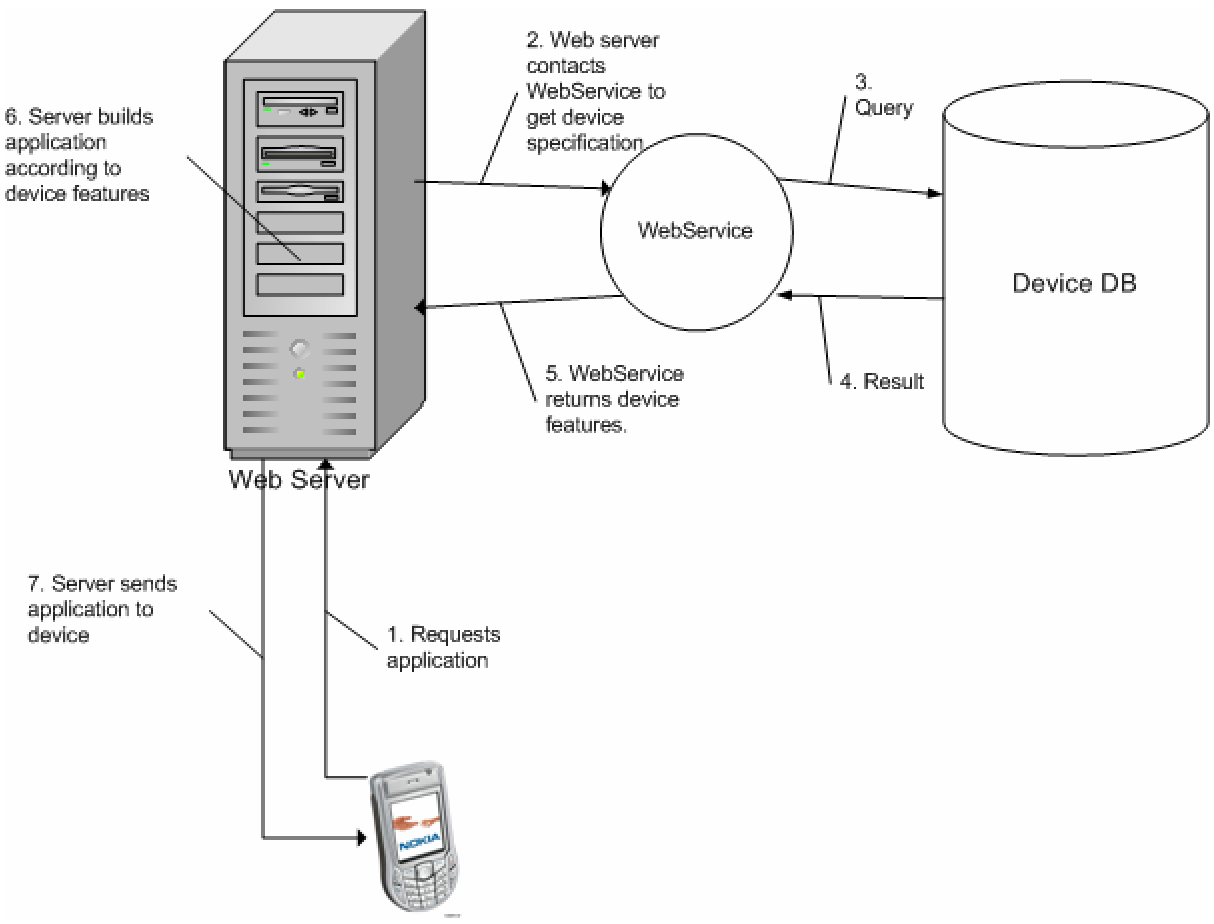

6. Server builds application according to device features
2. Web server contacts WebService to get device specification
3. Query
WebService
Device DB
5. WebService returns device features.
4. Result
Web Server
7. Server sends application to device
1. Requests application

Figure 6.5 Illustration of the BODF framework

6.2.1 Structure and Principles

In this framework the application is tailor made for each device centrally. The system consists of three elements on an abstract level:

Device Database:

The device database is storage for device specifications. It contains information about supported APIs and profiles.

WebService:

This handles requests from the Build and Deployment Server and queries the device database regarding the features of specific devices. This is presented to the Build and Deployment Server as a SOAP document.

Build and Deployment Server:
This handles download requests from mobile phones and builds the application according to information from the WebService.

The idea behind this structure is that if device-to-device portability is not needed, then the optimal solution is to automatically tailor the application upon download requests. Even though the resulting MIDlet will not be portable, important issues are taken care of. For instance, the application only needs to be developed once; the rest is automated by the framework.

When a mobile phone connects to the server using HttpConnection the server can register what mobile type is used, either automatically or from the user indicating what type of device is used. This is then passed on to a WebService which queries a device database about device features, and the results are used by the server to build the application.

As in BICA, the final MIDlet suite will consist of components, but in this case the components that are unsupported will be left out of the build to minimize the size.

6.2.2 Process Description

What enables the applications developed with BODF to be so well adapted to each device is that the building and compilation happens when a download is requested. This allows the system to access and manipulate to the MIDlets source code.

The BODF system can be explained through seven steps:

1. **Coding:** The developer has to conform to a certain style in order to make the system work. No mixing of the core application and device dependent functions is allowed. As far as naming goes, the classes and methods need to follow the BODF standard. When calling device independent functions from the core tagging must be applied.
2. **Server structure:** When the code is finished it will be placed on the server. Device dependent code will be compiled and put in libraries, but the core remains uncompiled until a download request is made. The libraries are open to all applications in the system, as reuse of code eases the job for the developers.

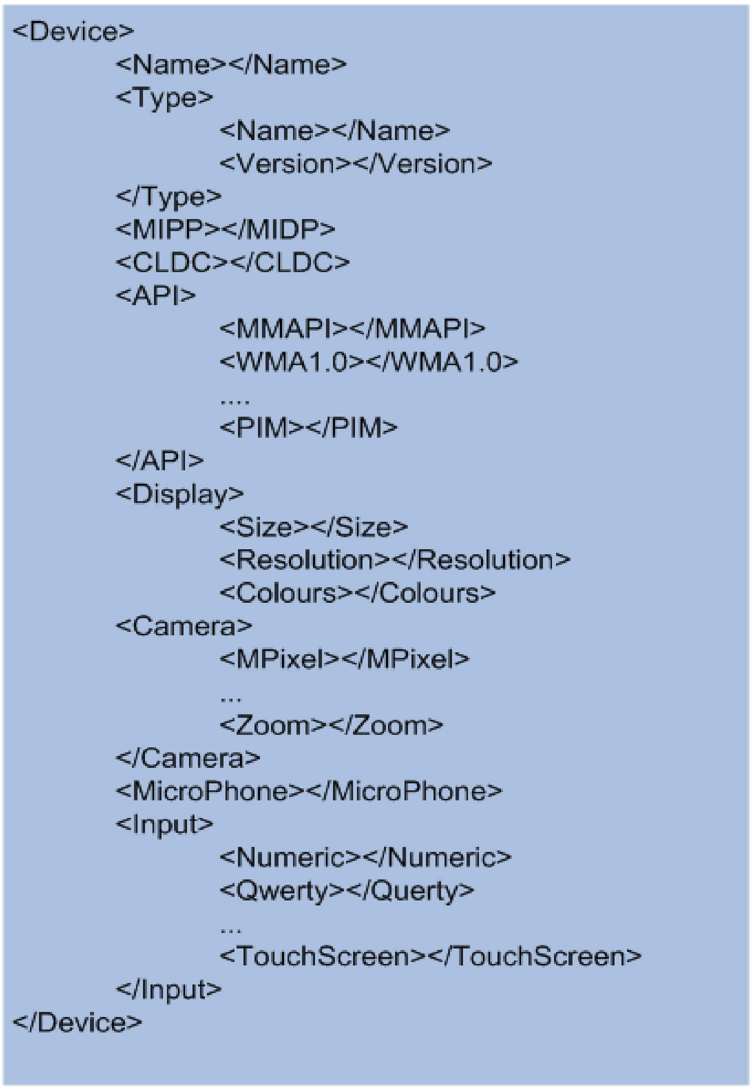
3. **Download request:** When a device requests a download, it gives the server its name, e.g. Nokia 6630. This name is passed on to the WebService which queries the device database to find required information about the device. This is stored as an XML file on the server.
4. **Automated Code Editor (ACE):** ACE is a simple program for traversing code and responding to tags put there by the programmer. If desired, more of the development tasks can be taken care of by extending this application. It reads an XML file in order to know what manipulations are needed when encountering tags in the code. When encountering tags it will manipulate code in accordance to the BODF standard.
5. **Compiler:** This is just a standard compiler. It is used to compile the manipulated code of the core.
6. **Packaging:** Taking care of packaging the application and all its necessary files in a JAR file.
7. **Deployment:** Returning the requested application, specified to the users device, as any other MIDlet

Following these seven steps will result in an application which is tailored to a specific device.

A small test was conducted to check if it was possible to use the automated code editor to manipulate the code when it was written according to the guidelines. In this small-scale test with one MIDlet, and two components it was successful.

6.2.3 Device Characterization

As mentioned, when the build/download server gets the name of the device it queries the device database through a web service. The result presented to the server is an XML-file like the one below.



```xml
<Device>
        <Name></Name>
        <Type>
                <Name></Name>
                <Version></Version>
        </Type>
        <MIPP></MIDP>
        <CLDC></CLDC>
        <API>
                <MMAPI></MMAPI>
                <WMA1.0></WMA1.0>
                ....
                <PIM></PIM>
        </API>
        <Display>
                <Size></Size>
                <Resolution></Resolution>
                <Colours></Colours>
        <Camera>
                <MPixel></MPixel>
                ...
                <Zoom></Zoom>
        </Camera>
        <MicroPhone></MicroPhone>
        <Input>
                <Numeric></Numeric>
                <Qwerty></Querty>
                ...
                <TouchScreen></TouchScreen>
        </Input>
</Device>
```

Figure 6.6 Sample XML document read by ACE

This is an empty XML file, but it will contain all the device specifications needed to tailor the core code. This way of organizing the device data makes the system easily extendible to new emerging devices.

6.2.4 Component Structure

The component structure shares some similarities with the one in BICA. The difference is that BODF uses one more abstraction level, the Component interface.

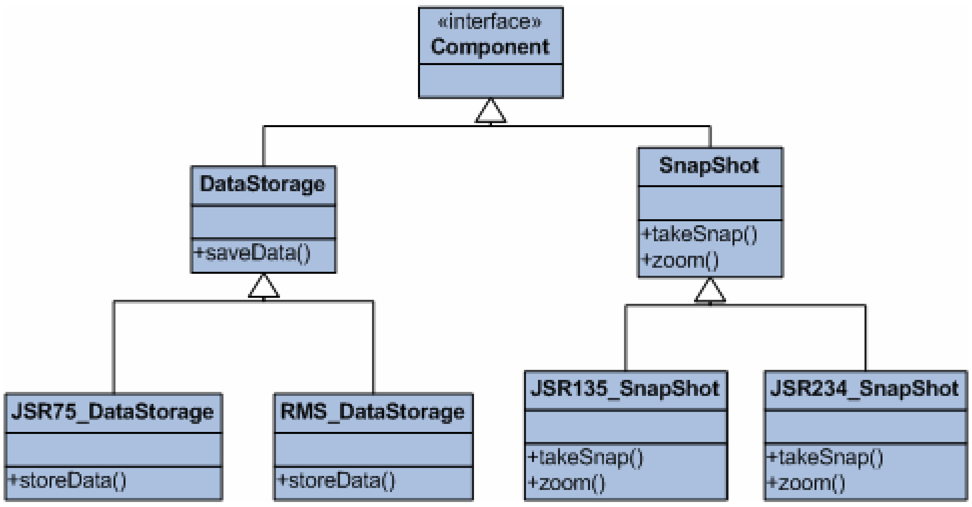


Figure 6.7 Sample component structure of the BODF.

What is illustrated in the figure above is a sample of the BODF component framework. The real structure will of course be a lot more complex and contain more classes with more attributes and methods. This illustration only serves the purpose of displaying the general idea.

6.2.5 Optimized GUI

The problems with optimized GUI construction is much the same in BODF as it is in BICA, but BODF has the advantage that it will be handled before the application is deployed. This opens for the use of the principles described in the J2ME Polish framework mentioned in chapter 5.3.2 [26].

6.2.6 Coding Guidelines

To make the BODF process work, the developers need to conform to a certain way of programming. This is because of the automated code editor that reads through the code and changes it to fit a certain mobile phone.

When using components it is suggested to use this syntax:

```
// #DataStorage#
DataStorage ds = new DataStorage();
ds.saveData("file:///e:/Sounds/", "sound.wav");
```

Figure 6.8 Pre-edited code

It is then easy for ACE to perform code optimization. It reads the *#DataStorage#* tag and gets the suitable component from reading the XML file containing all device info. This is the resulting code:

```
JSR75_DataStorage ds = new JSR75_DataStorage();
ds.saveData("file:///e:/Sounds/", "sound.wav");
```

Figure 6.9 Edited code

When using classes that will undergo device specification by ACE the developers should always use the super-class, which in the sample above is DataStorage. And the preceding tag should always be in the format *#<super-class>#*.

6.2.7 Example Scenario

We can use the same example application as the one used in the BICA sample scenario. The major difference is that with BODF the application will have only the optimized components for the specified device, and therefore it will be smaller. And it will not be portable in the same sense.

6.3 General Techniques for Portable J2ME programming

J2ME itself is rather portable, but still it is important to use proper programming techniques that will not limit portability.

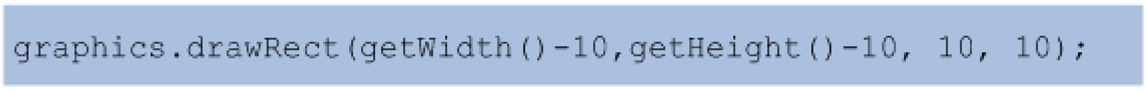
The high-level UI API is designed to be portable, but when using the low-level UI API there are some things to consider. For instance, when working on a canvas, always align elements proportionally to each other and the screen size. There are methods for getting screen size and other features from the system.

Instead of drawing a rectangle like this:

```
graphics.drawRect(118, 118, 10, 10);
```

Figure 6.10 Straight forward way of drawing a rectangle[x]

One should rather do it like this:



```
graphics.drawRect(getWidth()-10,getHeight()-10, 10, 10);
```

Figure 6.11 Portable way of drawing a rectangle[x]

When using media in MIDlets it important to use media that is widely supported by devices, or at least make the media files easy replaceable if needed. And any byte arrays used to store media content must be flexible in size, since the sizes of different formats differ substantially.

The hard coding of constants should also be avoided. Constants should rather be stored in separate configuration files.

7 Discussion and Conclusion

7.1 Discussion

Since MIDP already is portable technology and it is so because it uses the smallest common denominator approach, there is no point in making it more portable. A core MIDP2.0 application will work on any device that supports this platform. The main problem with the MIDP platform, or rather the portability on the platform, is the fact that it has been extended by different APIs on many different devices. With MIDP enabled devices being able to support two versions of CLDC, two versions of MIDP and a number of Optional Packages, a MIDlet might only work on the platform it was developed for. This great fragmentation is of course a necessary step in the process of growing into a mature platform for such a diverse group of devices, but as more and more devices gets the same technology this will probably change for the better. In the meantime, frameworks are needed to help J2ME along.

7.1.1 Proposed Frameworks

The two solutions proposed in this thesis both have the same goals: Cut development costs when developing for more than one platform, and exploit more of the technical features on each device than you would be able to with the “smallest common denominator” approach of MIDP.

BICA approach differs from BODF in the way that it builds the portability into the deployed MIDlet suite. This enables the application to be device-to-device portable, which is a feature that lacks in applications built with BODF. The question is then: when do we need device-to-device portability? For a software development company this may not be desired. After all, it would probably want to make money of the product, and maybe keep records of numbers of downloads. Of course, there are other ways of making copying hard, but it just shows that maybe this kind of portability is not needed. When using BODF, the MIDlet suite will be designed to fit only your type of device, but someone with another brand may also download and receive the same application tailor made to their needs.

As the mobile market is a market in motion, with new and extended technical possibilities constantly emerging, a framework for development of portable applications also needs to be able to extend accordingly. Both the suggested frameworks in this thesis extend by adding all developed components to a library. This extension of the libraries can however pose a problem for BICA as it needs to deploy implementations of all component versions in order to be portable to all devices. BODF has one advantage in the fact that it does not need a class to choose which component to use at runtime. This is handled centrally by ACE before compilation and deployment, and therefore it will not be a problem that libraries grow in size as only one version of the device-optimized components will be deployed with each application.

The issue of creating a better portable GUI than the one MIDP offers poses challenges to both of the proposed frameworks in this thesis. BICA needs an implemented GUI interpreter, but BODF can employ principles such as the ones used in the project J2ME Polish. Since this thesis is not about implementing a complete framework, only the feasibility of such principles is considered. It seems more than likely that BODF will be able to handle these challenges.

7.1.2 Feasibility and Further Work

The proposed frameworks have not undergone a full-scale implementation or testing, but indications from testing the core ideas in small applications gives reason to believe that a full-scale implementation is possible.

However, it is difficult to say exactly how complex applications will behave within these frameworks without actually testing, and therefore a full-scale framework is needed to make a conclusive decision on the feasibility issue. The structures suggested in this thesis should provide enough information to implement full-scale frameworks. Probably the BODF structure would be the easiest to make a full-scale framework from, since every step of the process is based on existing technology, only added a few structural guidelines. Being based on step-based process also makes it more flexible for structural changes.

7.2 Conclusion

This thesis explains how a portable technology like MIDlets needs help in order to achieve a higher level of portability. It is also clear that certain aspects cannot be solved by a programming framework, e.g. one device having a built-in camera and another has no camera at all or no J2ME support for it. However, within groups of mobile devices with similar technology, a framework for portable programming certainly can improve the development process and optimize the applications.

I will suggest that the most suitable framework would be one similar to the BODF proposed in this thesis, as this is an easily extendable framework type with its library of components. In addition it can create optimized GUIs which will greatly improve the users experience with MIDlets. With BODF, it is my claim that developers will experience that new devices emerging to the market will sometimes be able do download applications that will be automatically fitted without any new coding.

Although no complete framework has been implemented in this project it has been shown that, with a high degree of probability, an improvement of portability is feasible through frameworks and portability-conscious coding.

Bibliography

[1] 3G Americas, “Java Technical Recommendations for Handsets”, June 2005. Available: biz.yahoo.com/prnews/050623/sfth006.html

[2] Symbian, “History”. Available: symbian.com/about/history.html

[3] Canalys, " Global smart mobile device sales surge past 10 million in quarter”. April 2005. Available: canalys.com/pr/2005/r2005041.pdf

[4] Symbian, “Symbian OS version 8.0 product sheet”. Available: www.symbian.com/technology/symbianOSv8_ds_0204.pdf

[5] Symbian, “Symbian OS phones”. Available: symbian.com/phones/index.html

[6] Author: Martin de Jode, “Programming J2ME on Symbian”, chapter 1.1.1, 2004.

[7] Author: Qusay H. Mahmoud, ” J2ME Luminary Antero Taivalsaari”, January 2004. Available: developers.sun.com/techtopics/mobility/midp/luminaries/taivalsaari/

[8] Aouthor: Lauri Aarnio, ”Small scale Java virtual Machines”. Available: cs.helsinki.fi/u/campa/teaching/j2me/papers/Small.pdf

[9] Sun, ”The CLDC HotSpot Implementation Virtual Machine”. Available: java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf

[10] Java Community Process, Available: jcp.org

[11] Author: Michael Kroll, Stefan Haustein. “Java 2 Micro Edition Application Development”, chapter Introduction, June 2002

[12] Author: C. Enrique Ortiz. “The Generic Connection Framework”Available: developers.sun.com/techtopics/mobility/midp/articles/genericframework/

[13] Author: Martin de Jode, “Programming Java 2 Micro Edition on Symbian OS”, chapter 1.4, 2004

[14] Author: Arvind Gupta and Martin de Jode. “Extending the Reach of MIDlets: how MIDlets can access native services“, June 2005.

[15] Sun, "Mobility overview". Available: developers.sun.com/techtopics/mobility/overview.html

[16] Author: Qusay H. Mahmoud, "J2ME Low-Level Network Programming with MIDP 2.0", April 2003. Available: developers.sun.com/techtopics/mobility/overview.html

[17] Author: Qusay H. Mahmoud. "Getting Started with the FileConnection APIs", December 2004. Available: developers.sun.com/techtopics/mobility/overview.html

[18] Author: Qusay H. Mahmoud. "MIDP GUI Programming". Available: scmad.gayanb.com/tutorials/midp-gui-programming-part-1.php

[19] Forum Nokia. "What’s in MIDP 2.0: A Guide for Java™ Developers", September 2003. Available: forum.nokia.com

[20] Symbian Press Release. "Latest version of Symbian OS targets smartphones for mass market", February 2005. Available: symbian.com/news/pr/2005/pr20051892.html

[21] Symbian. "Symbian OS v9.1 functional description". Available: symbian.com/technology/symbos-v91-det.html

[22] Author: Bill Venners. “Inside the Java Virtual Machine”, chapter 1.

[23] Author: . “Migratable User Interface Descriptions in Component-Based Development”, 2002.

[24] J2ME Polish. Available: j2mepolish.org

[25] W3C. Available: http://www.w3.org/2001/di/

[26] J2ME Polish. “Specific Design Attributes”. Available: http://j2mepolish.org/docs/css-specific.html

Appendix A – The Sybmian OS Evolution

Version 7.0

Symbian OS v7.0 was released in 2002 Building on 2.5G GSM / GPRS support in previous versions, Symbian OS v7.0 includes support for multimode and 3G mobile phones, enabling manufacturers to bring out Symbian OS phones worldwide, across all networks, with the ability to reuse their application side software. Symbian OS v7.0 includes Enhanced Messaging Service (EMS) and MMS, providing key revenue generating services for network operators. More networking capabilities have been added, including both IPv6 and IP Security (IPSEC) technologies, extending the abilities of mobile phones to communicate securely with each other on a peer to peer basis. V7.0 incorporates Java MIDP, extending mobile phone capabilities to run the millions of Java applications and services designed specifically for mobile phones, and Synchronization Markup Language (SyncML), allowing convenient Over The Air (OTA) synchronisation of data.

Version 7.0s

Symbian OS v7.0s was released in 2003 and provides new functionality providing a fit-for-purpose platform for the 3G market and enabling the OS for 3GPP compliance, enabling the delivery of 3G services. It has Lightweight multi-threaded multimedia framework and support for Wideband Code Division Multiple Access (W-CDMA). More Java functionality has also been added like the Java MIDP 2.0, Bluetooth® 1.1 and Wireless Messaging API (WMAPI) 1.0 profiles. V7.0s has been given support for multiple primary/secondary Packet Data Protocol (PDP) contexts.

Version 8.0

Symbian OS v8.0 was released in the beginning of 2004 and has improved kernel architecture with hard realtime capabilities, and it introduces SyncML compliant device management framework. Significant support for Java has been added including CLDC 1.1, MobileMedia API (MMA), Mobile 3D Graphics API, Personal Information Management (PIM) and FileConnection (FC). Symbian OS v8.0 is provided in application compatible two variants. The first variant, v8.0a uses the legacy kernel (EKA1) as per Symbian OS v6.1, v7.0 and v7.0s. The second variant v8.0b adopts the new hard realtime kernel (EKA2). V8.0 also has the addition of the Media Device Framework (MDF) which provides a Hardware Abstraction Layer for multimedia hardware acceleration.

Version 8.1

Symbian OS v8.1 was released in 2004 and delivers extensions to CDMA IS95 / 1xRTT Telephony, Networking and SMS technology that are standard to all operators. It provides new customisation and configurability options with support for multiple displays and scalable user interfaces. It has continued alignment with standards including Java PIM, Bluetooth® 1.2, Bluetooth® Personal Area Network (PAN) and USB Mass Storage.

Version 9.1

Symbian OS v9.1 was released in the beginning of 2005 and is the newest contribution to the Symbian OS familly. V9.1 provides a native Realtime Transfer Protocol (RTP) stack. This stack can be used by licensee and 3rd party applications without the need for a separate RTP stack. Features which give network operators and enterprises new capabilities to manage phones in the field are also provided. This includes Open Mobile Alliance (OMA) Device Management 1.1.2 support and OMA Client provisioning 1.1. V9.1 continues to add Bluetooth innovations to the operating system. In this release support for Bluetooth extended Synchronous Connection Oriented (eSCO) and Bluetooth Stereo headset profiles are implemented. Symbian OS v9.1 is built using the ARM RVCT 2.1 compiler. This compiler is compliant with the ARM EABI standard. This allows compatibility with the latest ARM compliers and reduces the Symbian OS footprint while enhancing performance. Symbian OS v9.1 provides a proactive defence mechanism against malware. The platform security infrastructure uses a capability based model which ensures that sensitive operations can only be accessed by applications which have been certified by an appropriate signing authority. Data caging allows applications to have their own private data partition. This allows for applications to guarantee a secure data store. This can be used for e-commerce, location applications and others.

Appendix B - Other development platforms on Symbian

There are three main options regarding programming on Symbian OS based phones: C++, OPL and .NET [ref: symbian.com]

B.2 - C++ Native programming

C++ is the native language of Symbian OS. All non-privileged system facilities are directly accessible via C++ APIs available in the C++ Software Development Kit. C++ is suitable when high performance and comprehensive functionality is required.

Programs written in native C++ usually offers best performance in memory use and execution speed. In addition to offering good performance, certain types of applications have to be written using C++ because of restricted access to system resources. Instances of this type of applications are servers, certain type of plug-ins and device drivers. Such programs either manage system resources, extends existing Symbian OS framework or interacts with the kernel.

B.3 - Open Programming Language

Open Programming Language (OPL) is a simple, easy to learn programming language that allows developers to rapidly create powerful applications for Symbian OS phones. OPL is an interpreted language that requires a translation phase before execution so is made up of two major components. To allow users to run an OPL application, the OPL runtime environment needs to be installed on their Symbian OS phone.

B.3 - Visual Studio .NET

AppForge Crossfire enables Microsoft® Visual Studio® .NET developers to use their existing skills to create applications for Symbian OS phones. Crossfire integrates directly into Visual Studio .NET, so developers can jump right into mobile phone application development using the language, debugging tools and interface they already know. Crossfire is an integral part of the AppForge Enterprise Developer Suite (EDS) which is designed for enterprise organizations and system integrators who wish to leverage their Microsoft .NET and Visual Studio resources for mobile and wireless application development. Appforge Crossfire makes it possible to write applications with Visual Studio .Net using C++, C#, Visual Basic.

Appendix C – Test Application

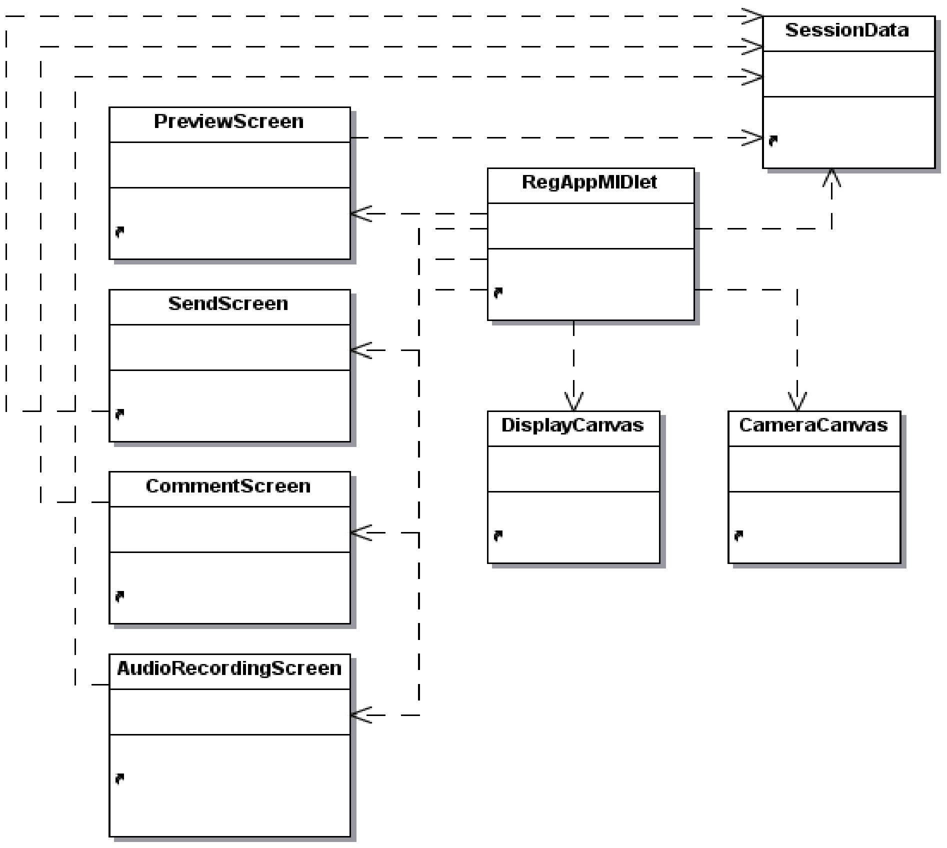


Figure 0.1 Class diagram for test application

RegAppMIDlet
This is the core class of the application. It displays the main menu, and organizes the application.

SessionData
Keeps track of the data registered by the user, i.e. image, audio and comments.

PreviewScreen
Displays the data registered by the user. Except the audio comments because this function was omitted.

SendScreen
Displays different options for sending the data, i.e. HTTP, TCP and UDP.

CommentScreen
Textbox where the user can write comments.

AudioRecordingScreen
Screen for managing audio recording. Includes start and stop recording.

DisplayCanvas
Canvas for displaying the snapshot taken by the user.

CameraCanvas
Canvas for displaying the video from which the snapshot is taken.

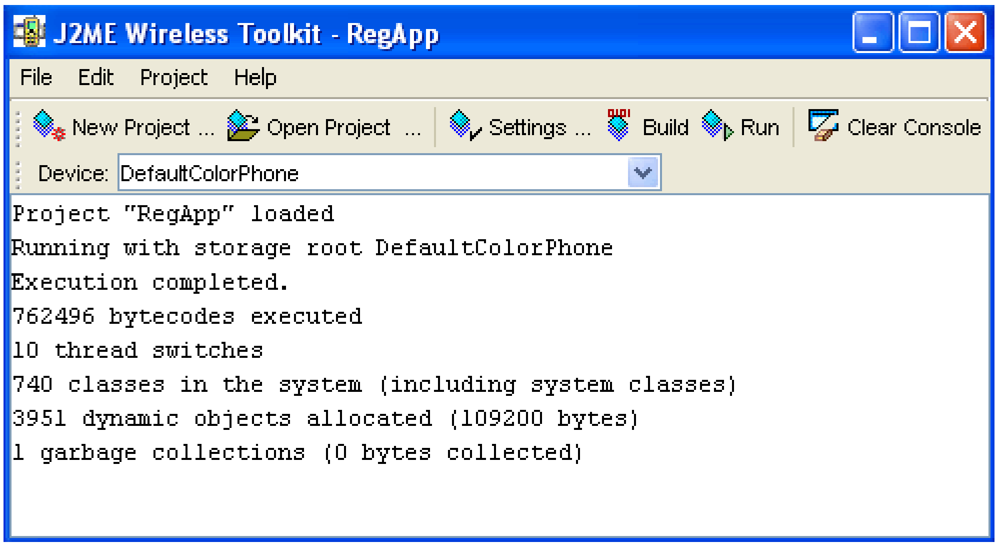
Appendix D – Development Tools

D.1 - Toolkits and emulators

J2ME applications must pass through a pre-verification process before being deployed on an actual device. Pre-verification allows the desktop compiler to verify that the compiled code can be run with J2ME's virtual machine. It is also helpful to do testing on emulators that will provide a reasonably real testing environment for a J2ME application. J2ME toolkits include tools that handle this, and they also often provide sample programs and documentation.

D.1.1 - Sun J2ME Wireless Toolkit 2.2

The J2ME Wireless Toolkit is a toolbox for developing wireless applications. It provides the basic tools needed for MIDP development, and for the time being it is free of charge. It does not provide the developer with a text editor or advanced debugging facilities, but it facilitates the process of compiling, pre-verifying and packaging of MIDlet suites. It also includes standard emulators for application testing.



Device: DefaultColorPhone

```
Project "RegApp" loaded
Running with storage root DefaultColorPhone
Execution completed.
762496 bytecodes executed
10 thread switches
740 classes in the system (including system classes)
3951 dynamic objects allocated (109200 bytes)
1 garbage collections (0 bytes collected)
```

Figure 0.1 The Sun Wireless Toolkit

The toolkit's emulator complies fully with the relevant API technology compatibility kits, ensuring that all the APIs are present and will react consistently with compliant implementations. In standalone mode, users can set individual preferences, build applications, create Java Archive (JAR) and Java Application Descriptor (JAD) files, and more, using either the toolkit's friendly KToolbar interface, or its command line. When integrated with an IDE, the toolkit's utilities and preferences appear in the IDE's menu selections, and also can be controlled from the IDE's command-line interface. When used with an IDE, the toolkit supports source-level debugging. []

WTKs friendly user interface lets the user choose what optional packages to include, what profile and configuration to use and many other useful features. The WTK also auto generates a JAD file when creating the project. This is very useful when building the project. Network and memory monitoring are two other very important features included. Because of the limited amount of resources on the mobile phones, a thorough examination of the memory use can be very handy when adjusting the application for optimal performance. In the same way an examination of the network traffic is useful in order to optimize the use of the limited available bandwidth. All in all the WTK is a very important tool in addition to an IDE when developing mobile applications. Its features is very useful when tuning, compiling, building and deploying applications.

D.1.2 - Sony Ericsson J2ME SDK 2.2.0

The Sony Ericsson J2ME SDK is a modified version of Sun Wireless Toolkit. In addition to the WTK, more features have been added in order for it to be custom made for Sony Ericsson and other UIQ products. This SDK supports all existing and newly announced mobile phones from Sony Ericsson, including the K600, K750, K300 and J300. And of course it includes all the APIs and emulators for two added JSR's, Java Bluetooth (JSR 82) and PDA Optional Package for J2ME Platform (JSR 75). A text editor is not included in SDK, but this is of less importance since it is primarily used in cooperation with an IDE.

D.1.3 - Nokia Developer's Suite 2.2 for J2ME™

As the Sony Ericsson SDK, the Nokia Developer Suite (NDS) is also created mainly to enhance IDEs such as Borland JBuilder and Sun Java Studio. NDS provides an audio conversion tool, application signing and features including application deployment to Nokia devices or FTP servers. Developers can create MIDlets based on the MIDP specifications that can be successfully implemented on Series 60 Nokia devices e.g. using the Series 60 MIDP SDK’s. There are many Nokia SDKs that comes in addition to the NDS to provide specific emulators, class libraries and documents targeted the different phone models.

[forum.nokia.com]

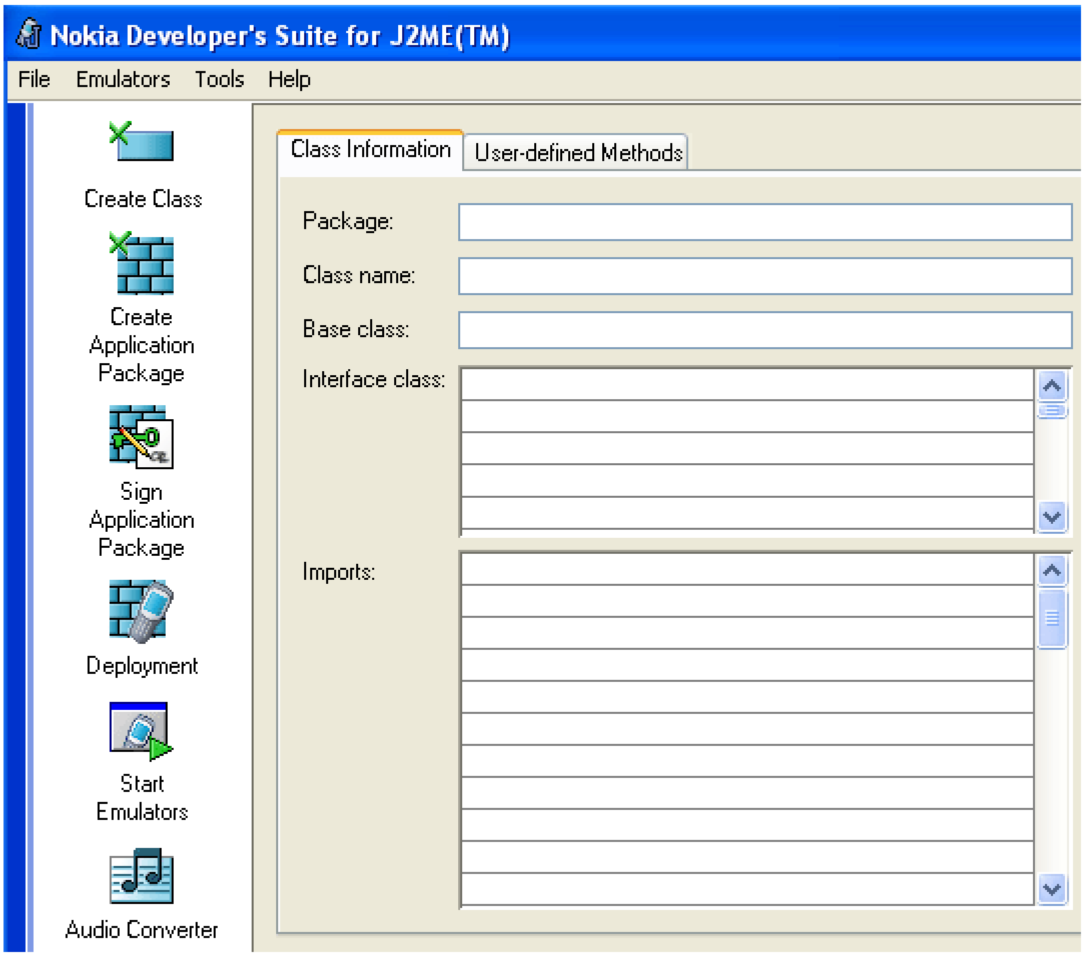


Figure 0.2 The Nokia Developer's Suite for J2ME

D.2 - Integrated Development Environments

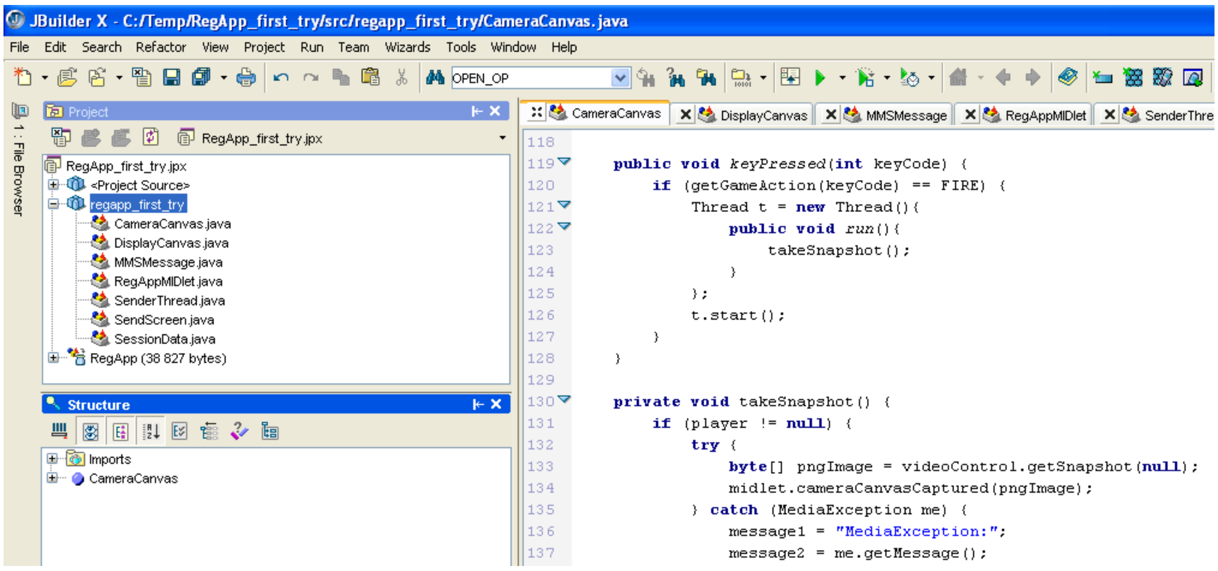
For a full-scale development of production quality applications it is practical to use a fully Integrated Development Environment (IDE). This thesis focuses on two of the most used and extensive IDE’s on the market, namely Borland JBuilder X Enterprise Edition and Sun Java Studio Standard 5.

D.2.1 - Borland JBuilder X Enterprise Edition

This JBuilder Enterprise version has integrated a lot of features for the Wireless environment and many wizards are provided to make development faster and easier. A Developer version and a Foundation version is also available, the latter is free of charge but does not include features for the Wireless environment. The Developer version does not include all of the wizards like the Enterprise version does, but it does contain features for the Wireless Environment.

JBuilder X Enterprise provides features like code obfuscation and integration of mobile applications with web services. Like all other IDE's, JBuilder also provides basic features like file editing, code completion, class and project browsing and easy-to-configure project properties.

In order to develop mobile applications the Wireless Toolkit has to be downloaded from the Sun web site. The Java Development Kit (JDK) path can easily be changed from the standard development kit to the Wireless Toolkit in the project properties. J2ME features will know be available as a wizard option when adding new elements to a project. A runable MIDlet can easily be constructed by the wizard without any code added by the user. The same javac compiler used for J2SE is used for compiling MIDlets. The only difference is the base Java classes that the compiler uses to compile the MIDlets against. All this however is transparent to the user. A built in emulator from the Wireless Toolkit or e.g. the Nokia Developer Suite will automatically pop up when running the MIDlet. The Tomcat server is also included and is very handy when developing MIDlets that is e.g. working against Servlets or JSP.



```
    public void keyPressed(int keyCode) {
        if (getGameAction(keyCode) == FIRE) {
            Thread t = new Thread(){
                public void run(){
                    takeSnapshot();
                }
            };
            t.start();
        }
    }

    private void takeSnapshot() {
        if (player != null) {
            try {
                byte[] pngImage = videoControl.getSnapshot(null);
                midlet.cameraCanvasCaptured(pngImage);
            } catch (MediaException me) {
                message1 = "MediaException:";
                message2 = me.getMessage();
```

Figure 0.3 Borland JBuilder X

D.2.2 - Sun Java Studio Standard 5

This IDE supports mobile application development features when installing the Mobile Editon modules from the Sun One Studio Update Center. This support comes in addition to enterprise and desktop application development features. It provides integration with the Sun J2ME Wireless Toolkit 2.2 for MIDlet development. In addition to this an implementation of the Tomcat server is provided to make communication with JSP and Servlets easier. Some wizards are also included to speed up and make the development easier.

Sun Java Studio Standard 5 provides full support for MIDP 1.0/2.0 development after installing the mobile modules.