Product Information Retrieval

over Cellular Networks

Author: John Delaney

A dissertation submitted to the University of Dublin,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

September 2006

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

John Delaney

11th September 2006

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

John Delaney

11th September 2006

Copyright

Abstract

Mobile phones have become commonplace in today's ever changing world. Their rapid growth in popularity is phenomenal. The following chapters detail how a typical camera enabled cellular telephone can now be transformed into a personal barcode reading device without any modification to the cellular phone's hardware. This can be achieved with a purely software based solution, using Sun Microsystems, Java 2 Micro Edition platform. Having captured a relatively low quality image using the MMAPI, it is possible to analyse and process a UPC barcode and retrieve relative information via a WAP / GPRS gateway.

This report provides a detailed technical description of the approaches taken along with the technologies; software and hardware utilised during the implementation of the application along with configurations used.

The documentation will also highlight the shortfalls, limitations and conclusions to the project, such as camera lense quality, software restrictions, processing requirements combined with other key effecting factors.

Acknowledgements

I have put a tremendous amount of time and effort into the following project in order to achieve the many goals that I can now stand proudly over. This would not have been possible without the kind support, advice and guidance of the following people to whom I would like to express my sincerest gratitude:

Dr. Mads Haar, for his advice, valuable points of view and helpful suggestions, with which gave me clear direction and guidance throughout the duration of the project. My good friend and Java guru, Jerry Kiely, who answered endless questions relating to java and also kindly proof read my final draft.

I would especially like to thank my parents for the much needed encouragement, reassurance and support, without whom, I could not have come this far.

Finally, I would like to express my gratitude to Lisa, for your tremendous support and patience all this time.

Thank you all!

John Delaney

Table of Contents

Introduction

Motivation

As consumers we all purchase items on impulse or on face value as we shop on a day to day basis. When we purchase any product for the first time, we usually know little or nothing about the actual product in question, i.e. its quality, reputation, reviews, any manufacturing details or even in the case of foods; their taste, sugar or sucrose content, even allergen substances that could be hidden in the ingredients. The same can be said for even the products that we are familiar with and use on a daily basis, we really know very little of the products or their origin. This is quite surprising considering the advancements in modern mobile technology and the rapidly growing area of ubiquitous systems.

Most mobile phone operators now provide access to the biggest information portal on the planet, the World Wide Web. Whether it is through the use of WAP or GPRS, most cell phones have the ability to connect to the outside world of data and all things internet related. Expanding this connectivity has also been made easier by the manufacturer's allowing developers do add on their own custom applications to the mobile phone devices, whether it be developed in the C/C++ language, or Java 2 Micro Edition.

Objective

Nearly every product sold in stores today will carry a unique identifier, a universal product code (UPC). This is essentially a number encoded in the form of a barcode, originally designed to help store owners speed up the checkout process and keep

stock of inventory. This product code can be used to relate a massive amount of information to any product.

With the rapidly developing technologies of the internet and mobile phone technology, the focus of this project is to explore and develop the methodologies suitable for creating an information portal for a consumer, yet at the same time, not adding a substantial cost for the benefit of this feature.

The concept being to use a cellular phone to photograph a barcode on a product and process this information using an implementation of a barcode reader which could be run from the handset itself. The cell phone could then use the translated UPC and launch a browser, linking to the information web site where the user could retrieve the information applicable to their product.

As an example, let's say the user would like to know where a specific pair of shoes they intend to purchase, are made. The user could photograph the barcode with their mobile device which automatically decodes this image into a UPC. The user can immediately find out valuable information regarding the shoes which would otherwise be unavailable to them. This can be applied in a variety of different scenarios, such as finding a review of a music CD, or even to allow a user to scan a CD of their own and find other albums for a particular group or groups of similar style or genre. The scale and usefulness of possible implementations are substantial. Other uses could include:

- Compare prices in retail stores
- Find concert dates from your own CD collection
- Acquire your medical history from a prescription bottle
- Check whether a product is friendly to the environment
- See if manufacturer is legitimate e.g. not involved in child labour etc.

To make this an application accessible and usable to an end user, certain constrains had to be considered in order to make it feasible:

- It must be able run on the majority of cell phones available on the market
- Should easily install for users without any technical knowledge
- Readily available – i.e. downloadable
- Easy to use and must return relevant data
- Performance should be relatively fast (work in real-time)

Technological Assessment

The following chapter will provide a brief introduction to the technologies assessed and implemented in this project, high-lighting why they were chosen, and their impacts or uses in the final product.

WAP

One of the technologies considered for this project was WAP (short for wireless application protocol) a specification which allows users to access information instantly via handheld wireless devices such as mobile phones, pagers, two-way radios, smart phones and communicators. WAP is supported by all operating systems which made it a strong candidate. WAP's that use displays and access the Internet run what are called micro browsers i.e. browsers with small file sizes that can accommodate the low memory constraints of handheld devices and the low-bandwidth constraints of a wireless-handheld network. WAP also supports HTML, XML and the WML language.

WAP currently uses the Wireless Transport Layer Security (WTLS) specification, which is now believed to have a significant amount of security issues. WTLS was specifically designed to conduct secure transactions over a low-bandwidth environment without requiring PC-level processing power or memory in the handset. For this to work, the WAP gateway acts as a translator between WTLS encryption and the internets current standard SSL (Secure Sockets Layer) security protocol.

A major problem occurs when the data is passed from WTLS layer to SSL layer. The data is decrypted and then re-encrypted, leaving the data venerable for a short period. Even though the data translation occurs within a secure data centre, it may be compromised for that split second.

Java 2 Platform, Micro Edition

The second technology chosen in this project was Sun's Java 2 Micro Edition (J2ME) which was developed specifically to address a huge market demand for small devices, ranging from smart cards to pagers. J2ME is a collection of specifications that define a set of platforms, which are suitable for a subset of these small consumer devices. The subset of the full Java programming environment for a particular device is defined by one or more profiles, which extend the basic capabilities of a configuration. The configuration and profiles for a device depend on both its hardware and the type of device, i.e. cell phone, PDA, etc.

One of the great features of J2ME, and indeed all java applications, is that they are restricted by placing untrusted code in a sandbox, where it runs safely without doing any damage to its environment. For example, when a MIDlet (an application written for MIDP) is running in the sandbox, there are a number of restrictions on what it can do. This means that it has no access to the local file system whatsoever or system resources without specifically giving permission. Another benefit of using J2ME is that it can support a range of internet protocols allowing communication with back-end servers which is a specific criteria required by this project. J2ME is widely support by the majority of 3G/Smartphones and also the lesser 2.5G phones.

Configurations

A configuration consists of a combination of a virtual machine and a minimal set of class libraries. These libraries are specifically designed to provide a base functionality for a set of devices with similar characteristics, such as memory, processing power, power supply or even network connectivity.

There are two configurations currently available:

Connected Device Configuration (CDC)

This more powerful configuration is intended for systems with around 2MB of memory such as smart phones and includes 13 packages. CDC is not used in connection with this project.

Connected Limited Device Configuration (CLDC)

CLDC is intended for systems with memory resources in the range of 160 - 512KB such as cell phones and low end PDA's. There are two versions - CLDC 1.0 and CLDC 1.1. The 1.0 version involves only four packages:

- java.io
- java.lang
- java.util
- javax.microedition.io

Many of the processors used in the target platforms for CLDC did not have floating point hardware; therefore the virtual machine was not required to support floating point operations.

CLDC version 1.1 added the package java.lang.ref and also added floating-point operations and a few other enhancements. However, it needs a minimum of 192KB versus 160KB for the 1.0 version. In this project, the chosen cell phone device uses CLDC version 1.0, rather than version 1.1. This created a few obstacles in the semantics of the program which shall be discussed further, in the analysis section of the report.

Profiles

A configuration provides a minimal set of class libraries which can be quite restrictive. The profile adds an additional layer on top of the configuration providing APIs for a specific class of device. This creates the ability for each configuration to

be adapted for each particular targeted device. While devices may appear to have a similar functionality, they do in fact have different requirements in terms of the available API's and interfaces to suit specific hardware.

Mobile Information Device Profile (MIDP)

Each different make and type of cell phone can have different memory, processor etc. Therefore a profile adds classes to the CLDC to provide for graphical user interfaces, networking, and other features. It is aimed at wireless systems, especially cell phones, and allows users to download MIDlet's. MIDlet's are small applications similar to applets, but will download and run on mobile devices that are limited to the CLDC configurations. There are also two versions – MIDP 1.0 and MIDP 2.0. MIDP version 2.0 adds enhancements over version 1.0, particularly to multimedia, game programming and HTTPS. For example, the only image format supported in MIDP 1.0 is the PNG format. However this has been expanded with MIDP 2.0.

Some of the MIDP packages include:

- javax.microedition.io (I/O connectivity)
- javax.microedition.lcdui (GUI)
- java.microedition.media (Multimedia)
- javax.microedition.midlet (MIDlet's)
- java.microedition.pki (Security)
- javax.microedition.rms (Persistent storage)

The cell phone is this project was capable of running MIDP version 2.0 and was therefore the version used.

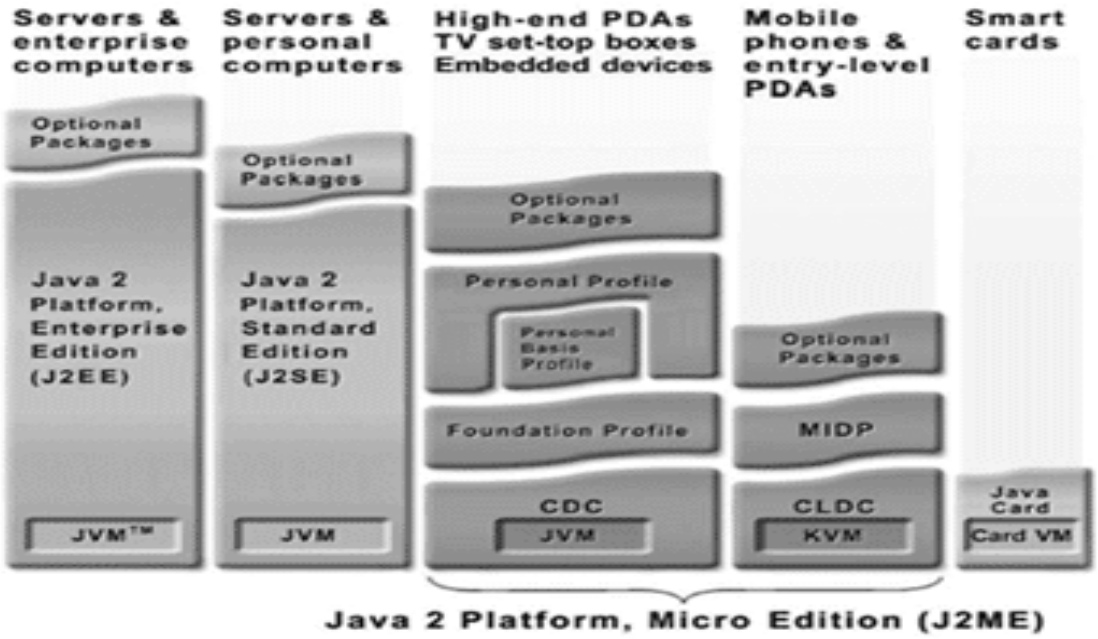


Figure 1 J2ME Platform

MIDlet's

When an application is written for MIDP, it must be packaged in order to transfer onto a device that will support J2ME. This is done by creating a Java Archive file (JAR). This JAR file is simply a file which is composed of the java class files and any other resource files used in the application, e.g. sound or image files.
Another file which must be included in every JAR file is the manifest. The manifest contains vital information required by the mobile device.

There are five required attributes:

- MIDlet Name - Name of the MIDlet suite
- MIDlet Version - Version number of the MIDlet
- MIDlet Vendor - Name of vendor that created the MIDlet
- Micro Edition Profile - What profile is required by the MIDlet
- Micro Edition Configuration - What configuration is required by the MIDlet

There are also a number of optional attributes which are trivial and will not be referenced further in this project.

Another file used while deploying a MIDlet, is the Java Application Descriptor file (JAD). The JAD file is optional and not specifically required to deploy a MIDlet. It does however, provide useful information if your device is being installed in a MIDlet suite (simply a JAR file containing one or more MIDlet's). The JAD provides information to the application manager which can then determine if the MIDlet can run on the device. For example, the JAD file contains the Micro Edition Configuration details. Therefore the application manager can check whether the cell phone is running CLDC version 1.0 or version 2.0 and what version JAR file conforms to. If the version on the device is lower than required, the application manager informs at installation time, and aborts. The following is an example of the actual JAD file from this project:

```
MIDlet-Name: BarcodeScannerMidlet
MIDlet-Version: 1.0
MIDlet-Vendor: John Delaney
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-Jar-URL: BarcodeScannerMidlet.jar
MIDlet-Jar-Size: 17555
MIDlet-1: BarcodeScannerMidlet,, jd.midlet.BarcodeScannerMidlet
```

It is a requirement that MIDlet Name, Version and Vendor all contain the same values as those in the manifest. Other values may be different between the manifest and the JAD file. If this is the case, then the values in the JAD file take precedence over those of the manifest:

```
MIDlet-Name: BarcodeScannerMidlet
MIDlet-Version: 1.0
MIDlet-Vendor: John Delaney
MIDlet-Description: BarcodeScannerMidlet
MIDlet-Icon:
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: BarcodeScannerMidlet, , jd.midlet.BarcodeScannerMidlet
```

Barcodes and Related Work

There are many different types of barcode readers currently on the market, most of which are the typical laser readers as seen in local stores. There are also quite a few software based barcode readers written in languages such as C++ and Java. However, there are very few J2ME based barcode readers available to date. This may be due to the limitations imposed by J2ME's characteristic restrictiveness and lack of extensive libraries.

There are a limited number of companies that do provide a J2ME barcode reading solution to run on mobile devices such as PDA's and cell phones. Some of these applications are for one dimensional barcodes, others for two dimensional and consist of hardware and / or software solutions.

Along with there being many different types of barcode readers, there are also many different types of barcodes, often referred to as barcode symbologies. Each of which have evolved in different areas of application and uses, but ultimately remain the same in terms of purpose; to encode a string of characters as a set of bars and spaces. The barcode needed ultimately depending on what data needs to be encoded and where the encoding is to be placed. As there are many different types of barcodes, and also many sub-variations of some of these barcodes, I will only cover the three most relevant to this project, which are:

- One dimensional universal product codes (UPC / EAN) – in detail
- Two dimensional quick response codes (QR Code) – brief introduction
- Two dimensional data matrix – brief introduction

Universal Product Code (UPC)

The Universal Product Code was the first barcode symbology adopted for product marking in the early 1970's throughout the United States and Canada. It was shortly

afterwards implemented in Europe as the EAN barcode. There have since been many adoptions of this barcode.

Figure 2: UPC Barcodes

Reading UPC Barcodes

Each UPC barcode encodes twelve decimal digits using seven bits to represent each digit. The white spaces in between each bar are also part of the code. Notice each barcode also begins and ends with three bars; one black, then one white followed by another black, which is represented with the bit pattern 101. These are called guard bars, whose purpose is to indicate the beginning and end of a barcode. The centre of the barcode also contains guard bars, but have the bit pattern 01010 to represent the midpoint of the code.

Therefore a barcode contains:

- 3 bits at the beginning representing the start of the code (101)
- Then 6 decimal digits represented by 7 bits each
- 5 bits denoting the centre of the barcode (01010)
- A further 6 decimal digits represent by 7 bits each
- 3 bits at the end to indicate the barcode is complete (101)

This gives a total of 95 bits per barcode. Each decimal and its representation can be seen in Table 1.

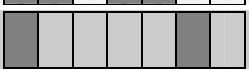
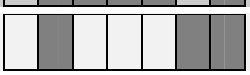
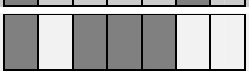
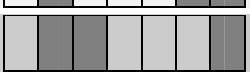
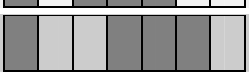
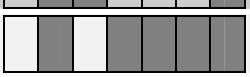
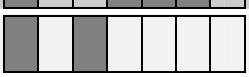
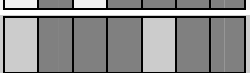
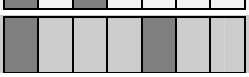
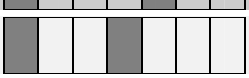
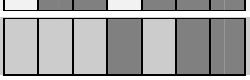
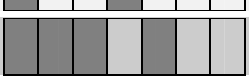
Decimal	Binary (Odd)	Visual (Odd Parity)	Binary (Even)	Visual (Even Parity)	Code Representation
0	0001101		1110010		3-2-1-1
1	0011001		1100110		2-2-2-1
2	0010011		1101100		2-1-2-2
3	0111101		1000010		1-4-1-1
4	0100011		1011100		1-1-3-2
5	0110001		1001110		1-2-3-1
6	0101111		1010000		1-1-1-4
7	0111011		1000100		1-3-1-2
8	0110111		1001000		1-2-1-3
9	0001011		1110100		3-1-1-2

Table 1: Barcode Decimal Representations

In the above table, we can see that there are no patterns in the data for the numerical representation with one exception; all decimals on the left side of the barcode begin with a zero (white space) and end with a one (black space). This is reversed on the right side of the barcode where every number begins with a one and ends with a zero. The purpose of this is so that one can scan the barcode without having to consider which side of the barcode faces upwards.

There are no other patterns in a barcode. In fact, all numerical representations are specifically chosen in order to make the numbers with as much variation in the code as possible. The codes are specifically designed so that there are never more than four ones or four zeros in order. All of this is to ensure ease of scanning and reliability.

If we take a barcode with number 7 18037 11184 1, the binary equivalent including the guard bars would be calculated as follows:

Decimal	Binary
Start	101
7	0111011
1	0011001
8	0110111
0	0001101
3	0111101
7	0111011
Centre	01010
1	1100110
1	1100110
1	1100110
8	1001000
4	1011100
1	1100110
End	101

Table 2: Conversion

Error Checking

The UPC system also allows for error checking ensuring a correct scan. This is done by calculating the following:

1. Add all the digits in the odd numbered positions together
2. Multiplying the result by 3 giving *n*
3. Add all the digits in the even positions together giving *m*
4. Add *m* and *n* together giving *q*
5. Subtract *q* from next highest multiple of 10 to give check digit

Again, taking the code 7 18037 11184 1, we can verify its integrity by performing the simple calculation following the above guidelines:

Step	Calculation	Result
1	7 + 8 + 3 + 1 + 1 + 4	24
2	24 * 3	72
3	1 + 0 + 7 + 1 + 8	17
4	72 + 17	99
5	100 – 99	1
Check Digit		**1**

Table 3: Check Digit Calculation

Quick Response Code (QR Code)

The quick response code is a two dimensional matrix barcode created in Japan by the Denso-wave Corporation. It was designed in order to provide a code whose contents could be decoded at a high speed. The main difference between a UPC barcode and a QR Code is that the QR Code contains information in both the vertical and horizontal directions. This allows the encoded of considerably more data. The QR Code however, is not to be confused with the smaller, more efficient Data Matrix.

Figure 3: Quick Response Code

Data Matrix

The data matrix is also a two dimensional matrix bar-coding system. The data matrix can store up to 2,335 alphanumeric characters and is made up of an even number of rows and columns, up to approximately 144 x 144 modules. The minimum size for a QR Code is 21 x 21 modules compared to that of the Data Matrix which is 77% smaller with a minimum size of 10 x 10 modules.

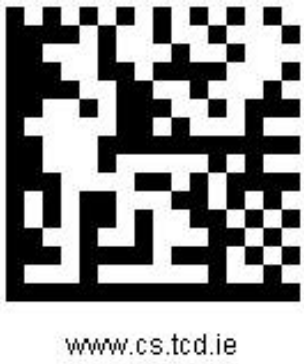


Figure 4: Example Data Matrix Barcode (Encoding a URL)

An independent committee from the Consumer Electronics Association's R9 Automatic Data Capture group submitted a comparison for the purpose of developing the IEC 62090 specification. They compared Data Matrix to QR Code to find that the data matrix proved to be the most space efficient of all the two dimensional symbologies [3].

	QR Code	Data Matrix	Space Saving
Example 1	42 x 42	24 x 24	67%
Example 2	25 x 25	18 x 18	48%
Example 3	29 x 29	20 x 20	52%
Example 4*	29 x 29	26 x 26	20%

Table 4: QR Code versus Data Matrix Comparison

*Note in the above, Example 4 encoded Japanese Kana characters. QR code was designed specifically to encode these characters efficiently. Even so, the Data Matrix is still more efficient.

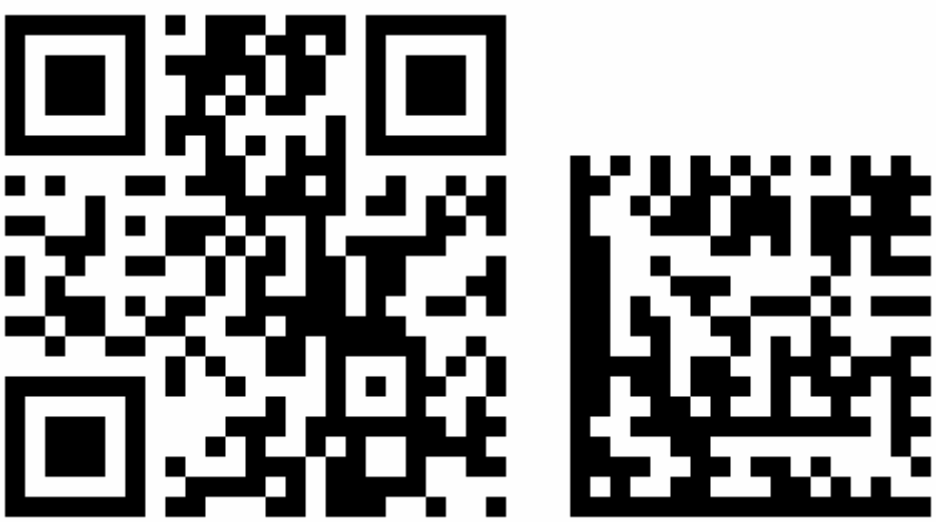


Figure 5: URL Encoding Comparison

In the above example, both data matrices are encoded with the URL "http://google.com". The module size for both is equal at 1.41 millimetres; however the data matrix code is 61% smaller.

Scanbuy

Scanbuy [4] are a North American based company that provide a range of barcode solutions for mobile phones. The company currently has four products available:

- Scanbuy shopper
- Scanbuy media
- Scanbuy coupons & tickets
- Scanbuy decoder

Scanbuy Shopper

Scanbuy shopper is a downloadable application which runs on most cell phones. It is designed to allow a user find various information about a product, or compare retail versus online pricing, based on the products barcode. This product does **not** scan the barcode from the mobile device camera. The user simply types in the barcode manually and is returned the information to their cell phone via the browser.

Figure 6 Scanbuy Shopper

Scanbuy Media

Scanbuy media is a system based around the larger two dimensional data matrix barcodes. It operates by simply pointing the camera enabled device at the two dimensional matrix. Once translated, the consumers are redirected to the encoded URL to retrieve more data. As seen from figure 3, these barcodes can be of significant size, quite some orders of magnitude larger than one dimensional barcodes found on in-store products.

Figure 7 Scanbuy Media

Scanbuy Coupons and Tickets

This is a simple, yet quite effective software solution to provide a consumer with a means of displaying coupons or tickets on the screen of their mobile phone. These codes can then be read by the specific reading devices for admission or discounts to a consumer. This system is based around two dimensional and also one dimensional barcodes.

Figure 8 Scanbuy Coupons and Tickets

Scanbuy Decoder

Finally, Scanbuy offer a solution to barcode generation, simply called Decoder. This is essentially a set of libraries combining image enhancing and barcode decoding algorithms with process an image frame to produce a decoded barcode value. This seems to be quite a similar product to this project. It is used to translate the two dimensional barcodes in the other software solutions, however, it is not implemented on the Scanbuy shopper one dimensional barcode reading software.

Semacode

Semacode [5] are a private Canadian company that provide a two dimensional barcode reader and SDK also based on the Data Matrix barcode. Using their SDK, the user can create visual tags for objects and contexts, and then read them using the camera on their mobile phone. When these tags are decoded, the user can obtain a web site address which can be accessed via the phones web browser.

SINTEF

In 2001, SINTEF [6] developed a hardware and software based solution combined to read Data Matrix barcodes. Their solution targets the more advanced devices such as PDA’s, but have suggested targeting the smart-phone sector of the market in the future.

Figure 9: SINTEF's 2D Barcode Reader

There are many companies that currently proved barcode reading solutions on the market. Most of these companies utilise the two dimensional data matrix barcodes, especially those designed to work on mobile phones. There are a limited number of companies that do provide solutions to reading one dimensional barcodes, such as Tasman barcode readers. These systems are typically written in the full Java specification rather than J2ME and target hardware devices with strong processing power.

System Design and Architecture

System Overview

This system is composed of two main areas; the barcode scanning software which runs on the users' mobile phone, and the HTTP server which awaits http get and post requests from the users.

The system was designed with scalability in mind, therefore all barcode processing is done on the handset, rather than passing large amounts of barcode data back and forth between the device and web server. This allows the system to be scaled using load balancing across multiple servers if required.

There are several stages to the system, from the point of taking the photograph, to the returning of data relevant to the barcode captured. These stages can be broken down and simplified into the following steps:

- Initialise the camera on the mobile phone including displays etc.
- Capture the snapshot of the targeted barcode
- Display this snapshot to the user
- Allow the user to proceed or retake the snapshot
- Convert the image taken, to a monochrome image for better processing
- Analyse the image, calculating the produce code encoded in the barcode
- Perform check digit calculation
- Create a connection to the network provider
- Connect to the web service providing the details
- Submit the product code to the web service
- Web service queries the product database
- Web service returns all or any relevant information
- Client terminates the connection
- Client display information regarding the barcode, if any

The above steps will be explored and describe in dept in the implementation section of the report.

Minimum requirements for the running of this application are as follows:

- Mobile phone capable of deploying and running J2ME applications
- 1 Mega-pixel camera
- 16kb local storage
- CLDC 1.0
- MIDP 2.0
- WAP / GPRS

Mobile Device Used in Implementation

There are a number of devices on which this application can run, as long as they meet the minimum requirements stated earlier. The mobile phone device which was used for testing and implementation of this application is detailed in the following table:

Device Manufacturer	**Nokia Corporation**
Device Model	7610
Operating System	Symbian OS v7.0s, Series 60 v2.0
Camera	1 Mega Pixel (1152 x 864)
Display Resolution	176 x 208 Pixels, 35 x 41 mm
Browser	WAP 2.0 / XHTML, HTML
Java Support	MIDP 2.0
	CLDC 1.0

Table 5 Nokia 7610 Specification

Java Classes Overview

During the coding of this application, there were numerous java classes and methods, used for debugging and testing. However, as the project progressed, the classes grew quite large in their numbers and in size.

In order to optimise performance of the J2ME classes and keep the JAR files size to a minimum, unnecessary methods and classes were removed or code was re-written. In certain cases, long cumbersome methods were completely re-written to a few lines of optimised code.

The final classes in the application are:

- BarcodeScannerMidlet.java – Main MIDlet class
- CameraCanvas.java – Canvas class for capturing images on cell phone
- DisplayCanvas.java – Canvas for displaying captured image
- ImageDecoder.java – Class for converting RGB image to monochrome
- BarcodeDecoder.java – Takes monochrome barcode and converts to UPC
- BarcodeCanvas.java – Canvas which displays a monochrome representation of the captured barcode
- DisplayForm.java – Form to display information relating to the UPC code
- HttpConnector.java – A thread to connect to a http server and retrieve data

The HTTP Servlet also contained the following class:

- BarcodeServlet.java – A HTTP Servlet which listens for connection requests and polls the database for information.

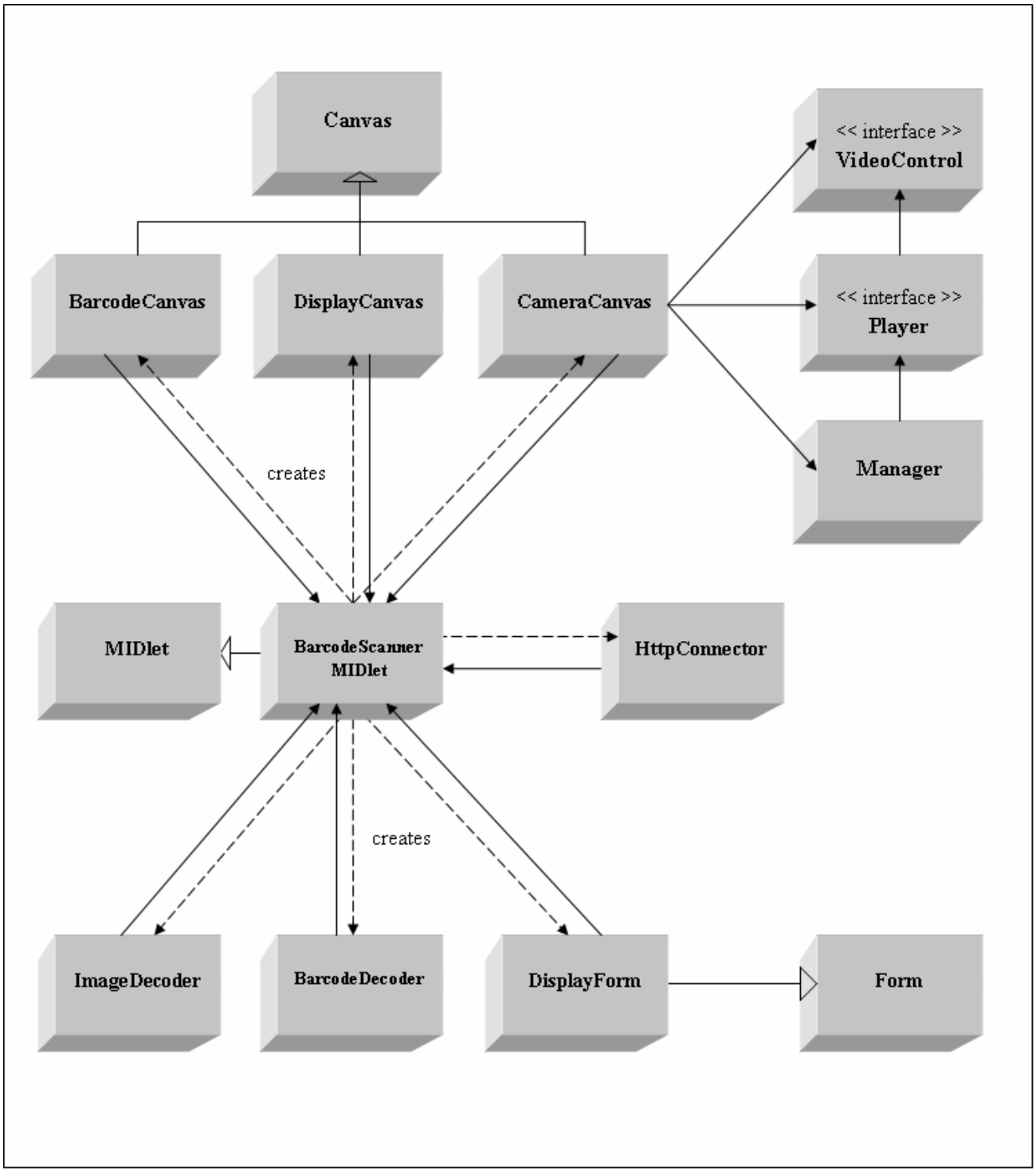


Figure 10: Cell Phone Software Design

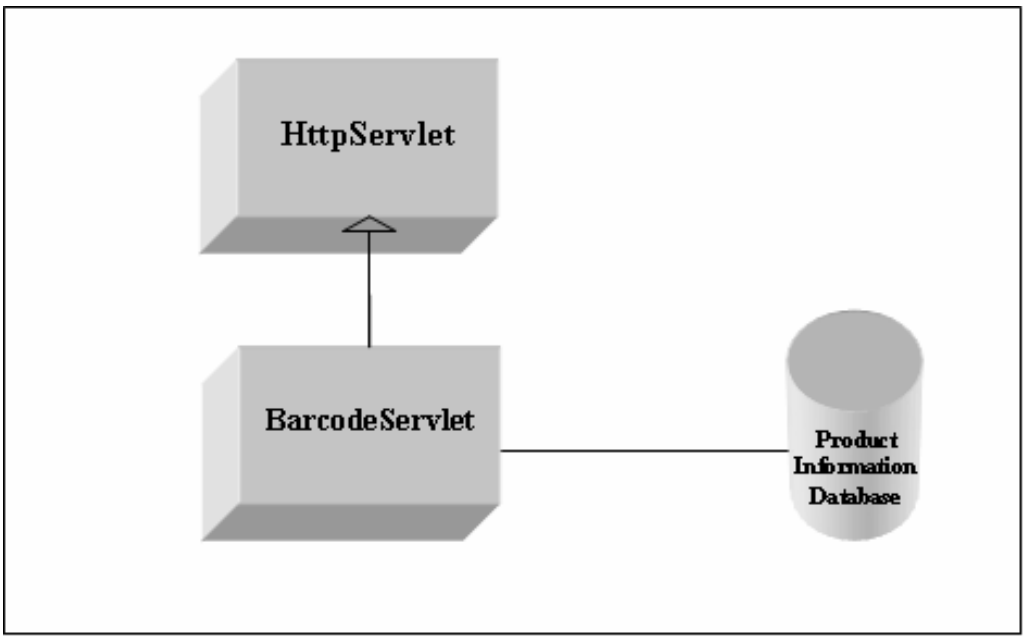
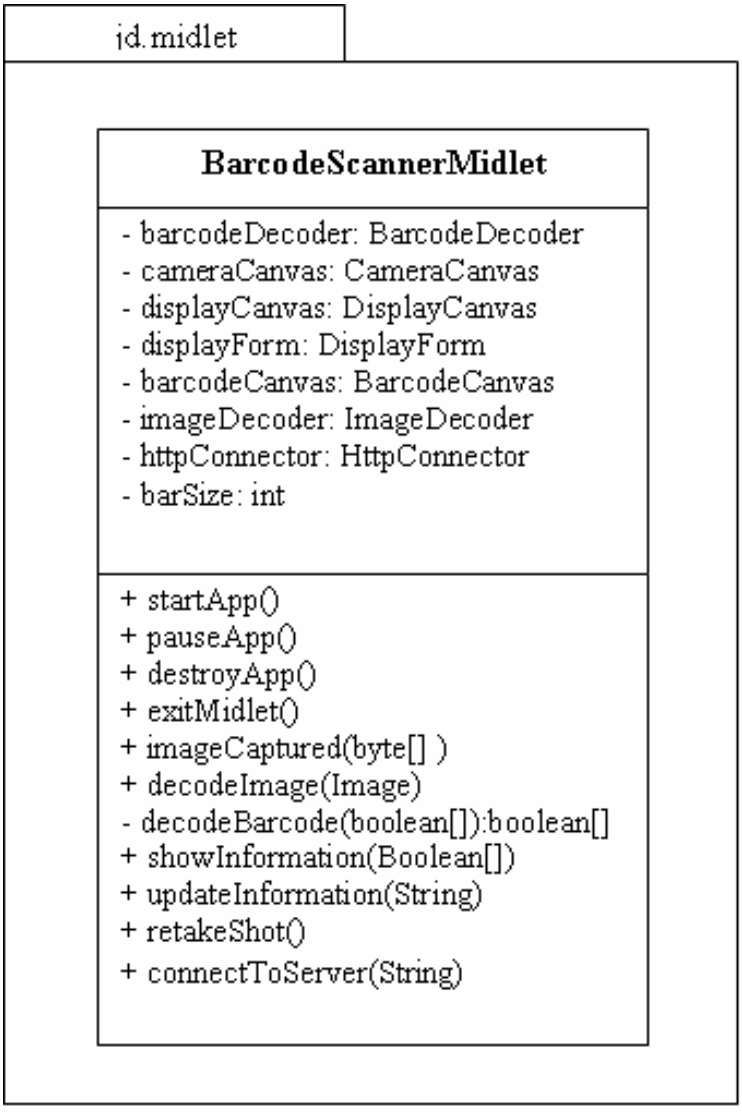
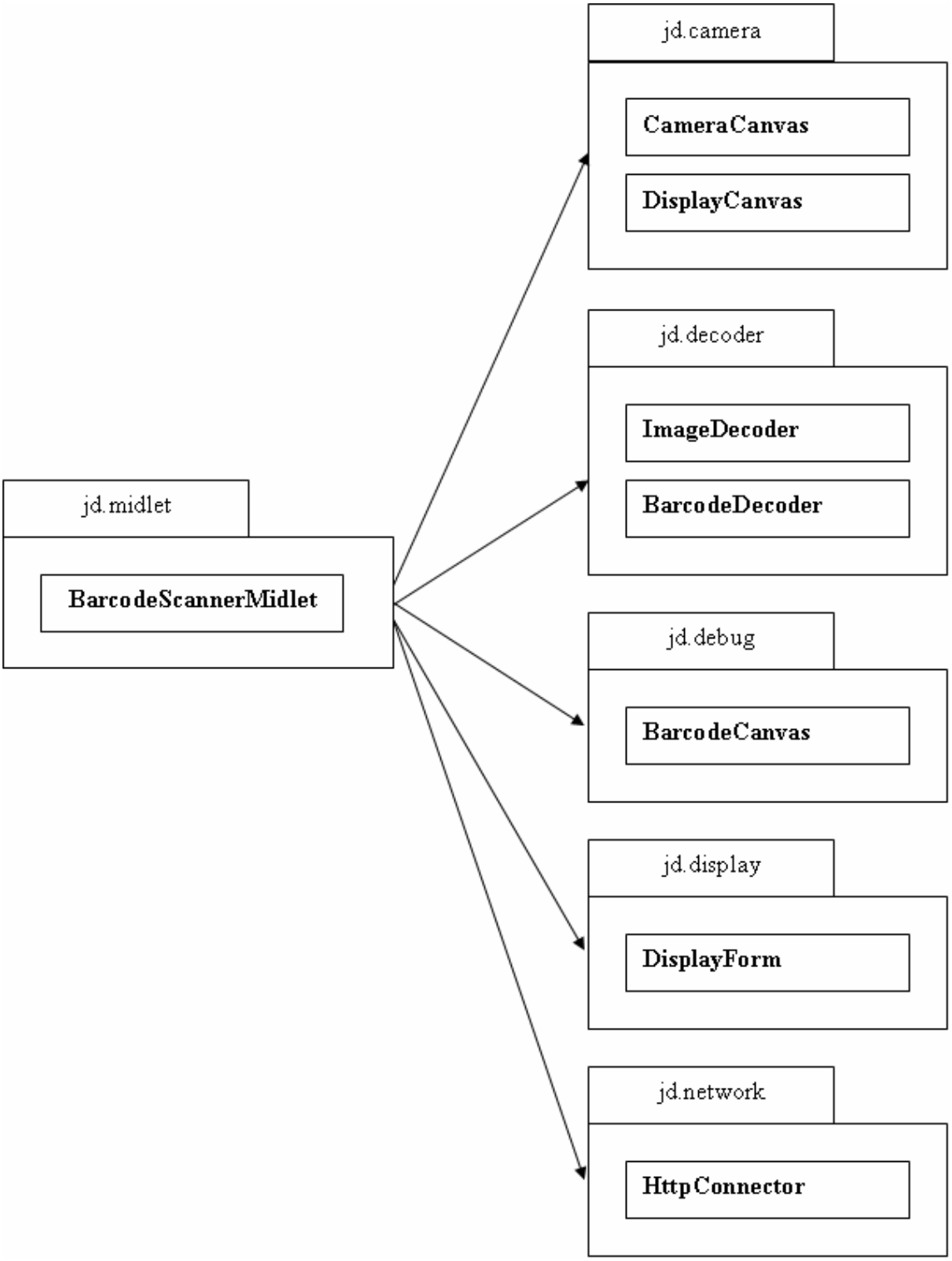
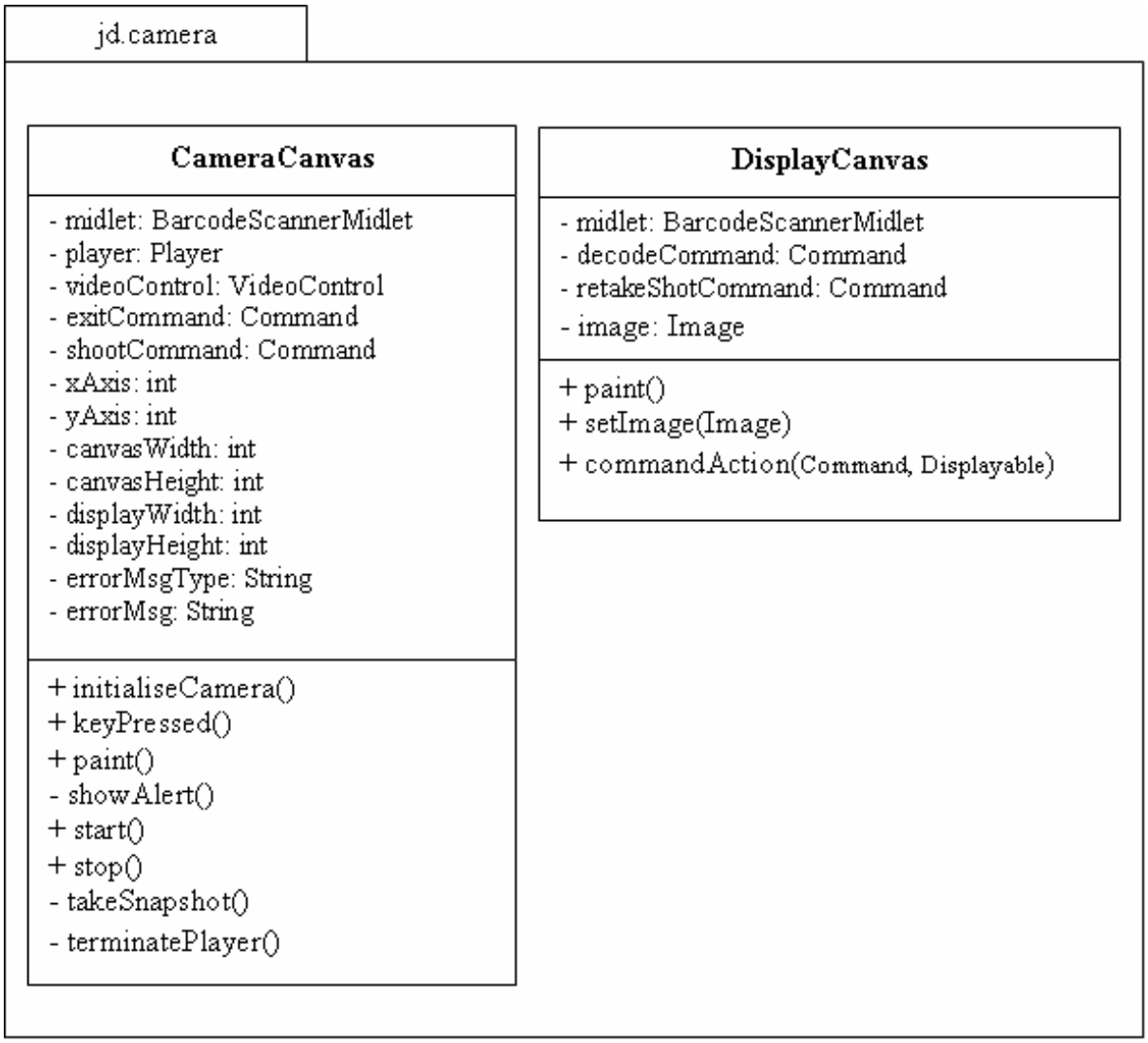


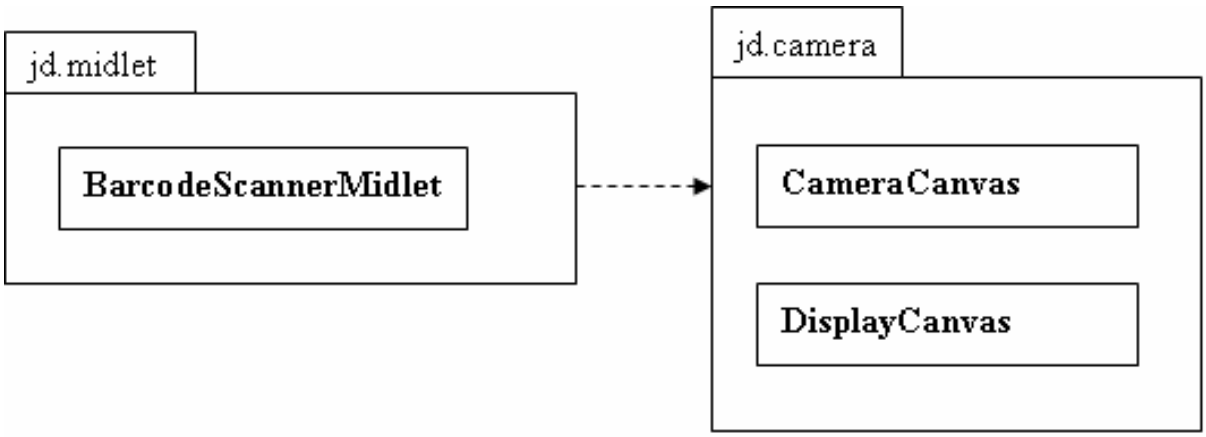
Figure 11 Server Side Software Design

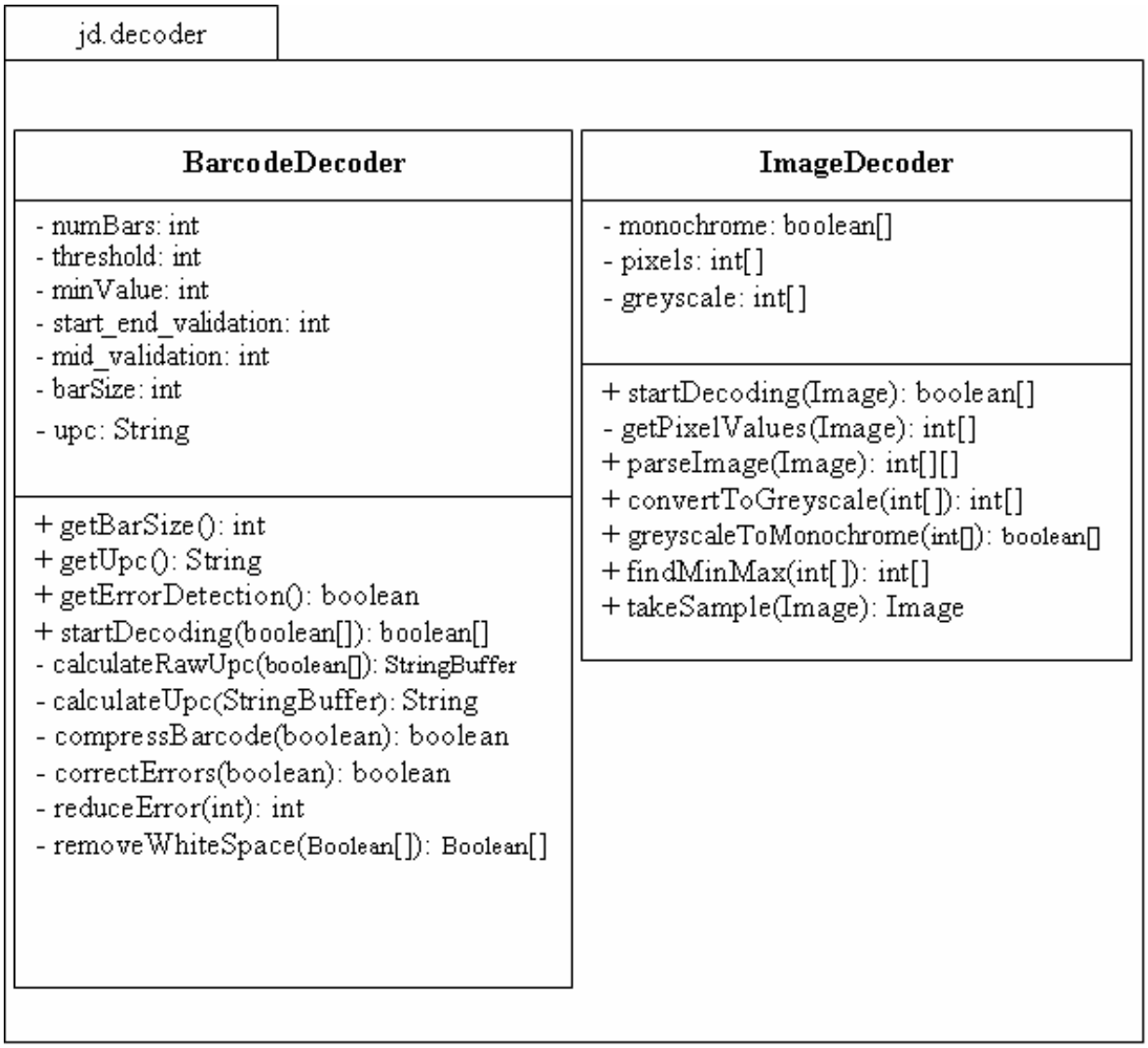
Detailed Package Diagrams (Client)

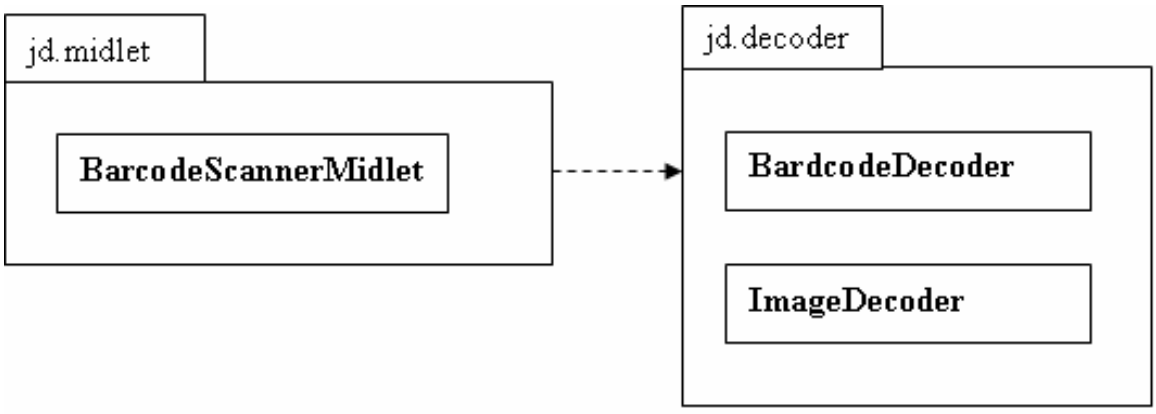


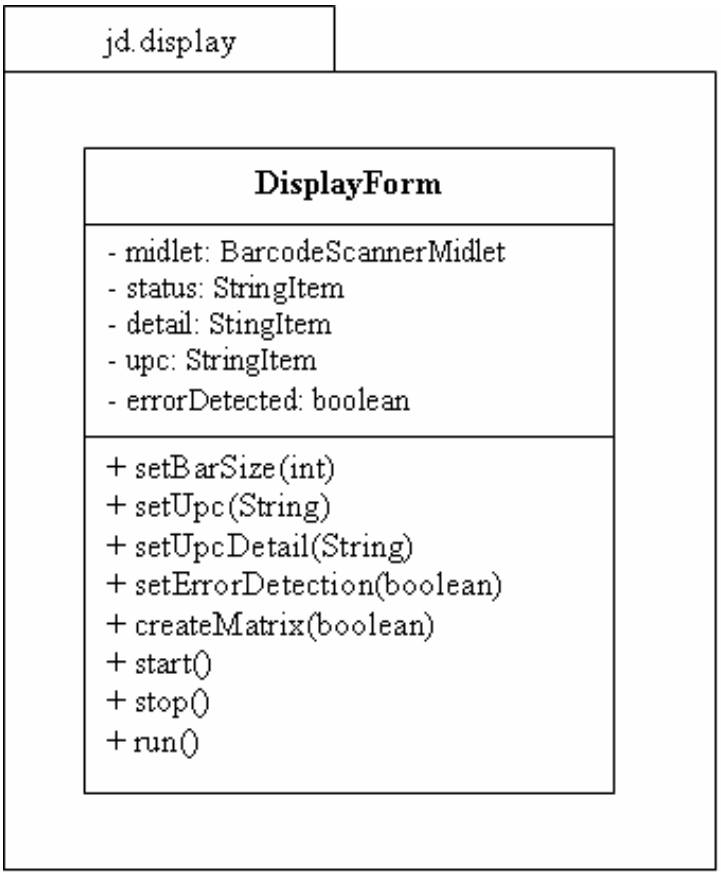

jd.camera
CameraCanvas
DisplayCanvas
jd.decoder
ImageDecoder
BarcodeDecoder
jd.midlet
BarcodeScannerMidlet
jd.debug
BarcodeCanvas
jd.display
DisplayForm
jd.network
HttpConnector

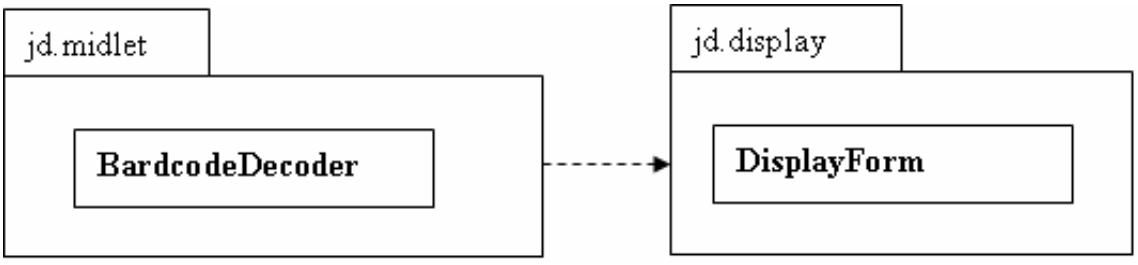

jd.camera
CameraCanvas
- midlet: BarcodeScannerMidlet
- player: Player
- videoControl: VideoControl
- exitCommand: Command
- shootCommand: Command
- xAxis: int
- yAxis: int
- canvasWidth: int
- canvasHeight: int
- displayWidth: int
- displayHeight: int
- errorMsgType: String
- errorMsg: String
+ initialiseCamera()
+ keyPressed()
+ paint()
- showAlert()
+ start()
+ stop()
- takeSnapshot()
- terminatePlayer()
DisplayCanvas
- midlet: BarcodeScannerMidlet
- decodeCommand: Command
- retakeShotCommand: Command
- image: Image
+ paint()
+ setImage(Image)
+ commandAction(Command, Displayable)

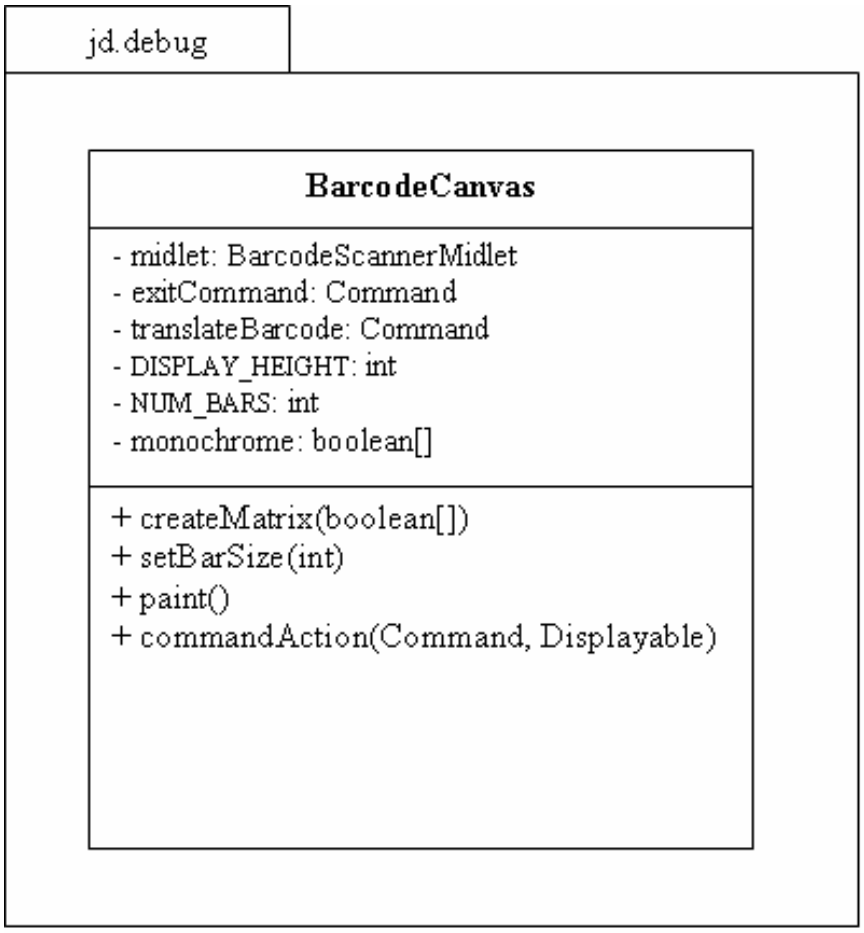

jd.midlet
BarcodeScannerMidlet
jd.camera
CameraCanvas
DisplayCanvas

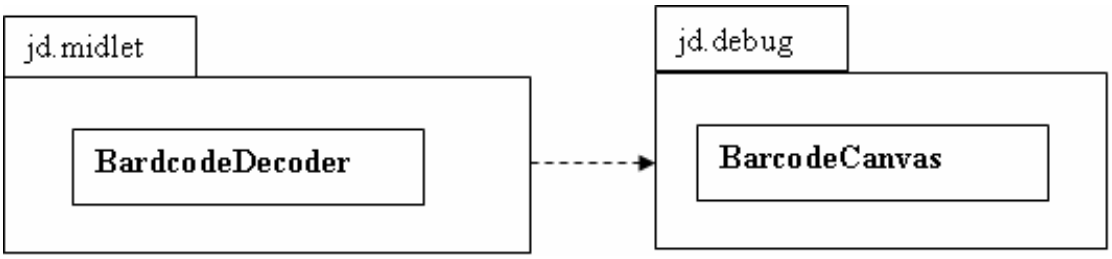

jd.decoder
BarcodeDecoder
- numBars: int
- threshold: int
- minValue: int
- start_end_validation: int
- mid_validation: int
- barSize: int
- upc: String
+ getBarSize(): int
+ getUpc(): String
+ getErrorDetection(): boolean
+ startDecoding(boolean[]): boolean[]
- calculateRawUpc(boolean[]): StringBuffer
- calculateUpc(StringBuffer): String
- compressBarcode(boolean): boolean
- correctErrors(boolean): boolean
- reduceError(int): int
- removeWhiteSpace(Boolean[]): Boolean[]
ImageDecoder
- monochrome: boolean[]
- pixels: int[]
- greyscale: int[]
+ startDecoding(Image): boolean[]
- getPixelValues(Image): int[]
+ parseImage(Image): int[][]
+ convertToGreyscale(int[]): int[]
+ greyscaleToMonochrome(int[]): boolean[]
+ findMinMax(int[]): int[]
+ takeSample(Image): Image

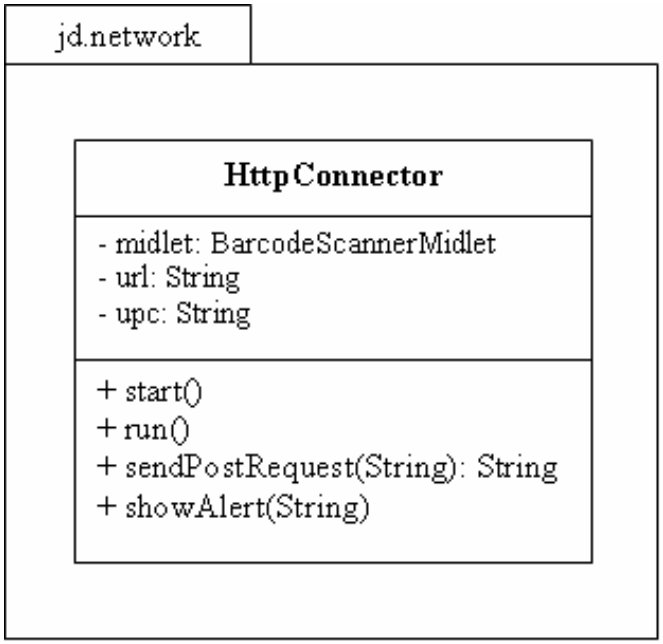

jd.midlet
BarcodeScannerMidlet
jd.decoder
BardcodeDecoder
ImageDecoder

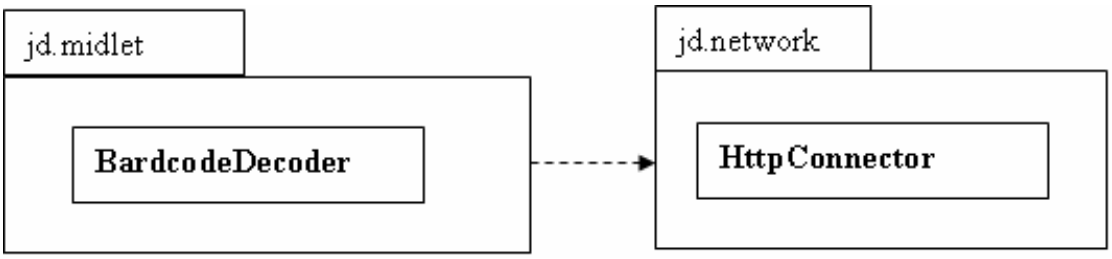

jd.display
DisplayForm
- midlet: BarcodeScannerMidlet
- status: StringItem
- detail: StingItem
- upc: StringItem
- errorDetected: boolean
+ setBarSize(int)
+ setUpc(String)
+ setUpcDetail(String)
+ setErrorDetection(boolean)
+ createMatrix(boolean)
+ start()
+ stop()
+ run()


jd.midlet
BardcodeDecoder
jd.display
DisplayForm


jd.debug
BarcodeCanvas
- midlet: BarcodeScannerMidlet
- exitCommand: Command
- translateBarcode: Command
- DISPLAY_HEIGHT: int
- NUM_BARS: int
- monochrome: boolean[]
+ createMatrix(boolean[])
+ setBarSize(int)
+ paint()
+ commandAction(Command, Displayable)


jd.midlet
BardcodeDecoder
jd.debug
BarcodeCanvas


jd.network
HttpConnector
- midlet: BarcodeScannerMidlet
- url: String
- upc: String
+ start()
+ run()
+ sendPostRequest(String): String
+ showAlert(String)


jd.midlet
BardcodeDecoder
jd.network
HttpConnector

Detailed Package Diagrams (Server)

Development Software

The following table contains a list of third party software used in the construction of this application. Versioning was of vital importance due to compatibility issues between different software versions. An example of this was the Eclipse SDK. During the project, I encountered a problem with the computer I was using. When I re-installed all of the development programs I was using on a new system, my code no longer executed and gave various run-time errors.
After considerable scrutiny, I later discovered that there was in fact, no problem with my code whatsoever. This was an issue which was caused from installing a newer version of Eclipse, version 3.2, on the new system. This version of Eclipse was incapable of running Carbide version 1.5. I consequently had to downgrade Eclipse to version 3.1.2.

	Software	**Version**
Java Software Development	Eclipse SDK	3.1.2
	Netbeans IDE	5.0
J2ME Software Development	Carbide.j	1.5
	J2ME Wireless Toolkit	2.2
Cell Phone Connectivity	Nokia PC Suite	6.80
	Nokia Connectivity Framework (via Bluetooth)	
Product Information Database	MySQL	4.0
Database Manager	MySQL Admin	
Database Connectivity (JDBC)	Connector/J	3.1
HTTP Server	Apache Tomcat	5.0
Build System	ANT	1.6.5
Unit Testing	JUnit	4.1

Table 6 Software and Versioning

Java Software Development

Both Netbeans and Eclipse were used in the development of this project. The reasons for this were for a technical and also personal preference reasons. Carbide provides a plug-in feature to allow it to be integrated with Eclipse, or can also be run as a stand alone. Being very comfortable with Eclipse, I opted for the plug-in configuration.

I decided to use Netbeans for the servlet development. From past experiences, I have found Netbeans to be better than Eclipse for any type of web development and also performs and integrates better with Tomcat which is essential for debugging and testing.

J2ME Software Development

Most of the J2ME code was written and emulated in Nokia's Carbide run from within Eclipse. Certain tests were also run on Sun's Wireless Toolkit, which was also used for development when the bug-filled Carbide acted irrationally.

Cell Phone Connectivity

In order to deploy any of the software written in J2ME to the Nokia handset, it was necessary to have installed and configured Nokia's PC Suite, Connectivity Framework (NCF) and PC Sync. Once configured correctly, any applications written in Eclipse could be compiled, run or emulated using Carbide, then deployed directly to the cell phone using these applications. The protocol used for deploying the software was via Bluetooth for convenience.

Product Information Database

MySQL was used for the storage of the product information. I briefly used MySQL admin to do minor administration work, however prefer using command line arguments to administer, purely due to being more familiar with this method. All tables and fields etc. were constructed manually through the command line. The database consisted of a few tables, containing a small sample of records.

Apache Tomcat

The information portal was deployed on Apache's Tomcat web server, sitting on a Windows based machine, using a static IP address over a 2mb broadband network. MySQL was front-ended and accessed by a servlet running on the Tomcat server, acting as a web service. All connectivity was achieved using HTTP connections on port 8080. Minor configurations had to be made to the firewall in order to allow this connection. Port forwarding also had to be set up, so that the router could forward the HTTP requests to the correct machine on the local network.

Mobile Device Implementation

Some of the drawbacks or restrictions with J2ME, include the limited processing power available to the programmer, coupled with a very limited library of API's available. This makes for very difficult programming, as useful library's are not always available, and class sizes need to be kept to a minimum in order to have an efficiently running application.

With this in mind, the classes written had to be separated carefully to ensure that each class had separate concerns to the next. The classes used in the implementation for the mobile device were as follows:

- BarcodeScannerMidlet.java
- CameraCanvas.java
- DisplayCanvas.java
- ImageDecoder.java
- BarcodeDecoder.java
- BarcodeCanvas.java
- DisplayForm.java
- HttpConnector.java

As we can see, each class has a descriptive name, which defines its functionality clearly, with once exception; BarcodeScannerMidlet. This is the main MIDlet class which is in essence, the main class for execution in the application. Its purpose is not to perform any calculations or actions as such. It is rather a control point for the application. The code contained in the MIDlet is merely a driver for the other objects, e.g. create objects or implement Runnable etc.

The BarcodeScannerMidlet class is responsible for the following actions:

- Starting the application
- Instantiating each individual class
- Controlling the sequence of events in the application
- Coordinating the user display between each canvas class
- Correct termination of the application

My decision to not allow the MIDlet to coordinate the keypad listener, simplified the integration of further classes. Any classes which needed a key listener simply had the short code embedded in the class. This design decision was due to each different display screen, required a different menu system. For example, the camera canvas requires a shoot button, while the display canvas requires a re-take shot button, and a decode barcode button.

This also allowed for the most part of the program to be single threaded. This was a design decision which I think was suitable for various reasons. Firstly, most of the procedures in the application could not be run simultaneously. It was necessary to decode the barcode in a particular sequence of events, which did not require multiple threads.

However, when it came to setting up a connection and communicating across the network, it was necessary to use multithreading. My decision to implement threads here was because without multithreading, a MIDlet that requests a connection to a network, blocks while it awaits a response from the network. The application is still required to run and be responsive even while setting up a connection, therefore multiple threads were required at this point in the application.

Breakdown of Classes

The first significant method to examine, is the method *startApp()* which is called when the MIDlet is launched. This method initialises all the classes required at the beginning of the program. Three canvas classes and an image decoding class. Highlighted in bold below, we will first look at the classes which are used to capture and display the images from the mobile phone's camera. These are the CameraCanvas class, and the DisplayCanvas class.

```
public void startApp() {
        currentDisplay = getDisplay().getCurrent();
        if (currentDisplay == null) {
                cameraCanvas = new CameraCanvas(this);
                displayCanvas = new DisplayCanvas(this);
                barcodeCanvas = new BarcodeCanvas(this);
                imageDecoder = new ImageDecoder();
                getDisplay().setCurrent(cameraCanvas);
                cameraCanvas.start();
        } else {
                if (currentDisplay == cameraCanvas) {
                cameraCanvas.start();
        }
                getDisplay().setCurrent(currentDisplay);
        }
}
```

Capturing Images using MMAPI

The CameraCanvas class implements the Mobile Media API (MMAPI) to capture images via the mobile devices camera. The first method to examine is the *initialiseCamera()* method, which does as its name suggests and performs all the

necessary steps to initialise the camera on the cell phone. This method is called from the constructor, as I felt the constructor method was over crowded and needed to contain tidier, more readable coding, therefore put all relating code in its own method:

```
public void initialiseCamera() {
        try {
                player = Manager.createPlayer("capture://video");
                player.realize();
                videoControl = (VideoControl) player.getControl("VideoControl");
        if (videoControl == null) {
                terminatePlayer();
                errorMsgType = "Video Error: ";
                errorMsg = "Video Control not found!";
        } else {
                videoControl.initDisplayMode(VideoControl.USE_DIRECT_VIDEO,
                this);
                •
                •
                •
}
```

Looking at the above code, the first step in the procedure is to create a Player in order to take input from the camera. Once this Player is created, it must then be initialised. After initialising the Player, we must also provide a viewing window so that the user can see what they are pointing the camera towards, i.e. a live viewfinder. This is achieved through the use of VideoControl.

Video control has two ways of displaying imagery using MIDP. It can be added as an item to a high level user interface form, or it can also be drawn using a low level canvas object. For the purpose of this project, all video displays are done using the canvas approach. Forms are also used for different display purposes in this application, but not in relation to video capture. Having initialised all the camera settings, the MIDlet then calls the method *cameraCanvas.start()*.

```
public void start() {
        if ((player != null) && !active) {
                try {
                        player.start();
                        videoControl.setVisible(true);
                .
                .
                .
}
```

This method starts the Player and sets the video control to visible, enabling the user to see where the camera is pointing. The camera is now ready to take a snapshot.

As mentioned earlier, the MIDlet is responsible for coordination of the user display between each of the canvas classes, and indeed the form class also. Therefore, when the MIDlet sets the focus of the current display to the camera canvas using method call *getDisplay().setCurrent(currentDisplay)*, the command listener for key presses in the camera canvas take precedence. There are two separate methods for handling key presses; one for specific key presses which have already been defined in the class, the other for capturing the “FIRE” button command.

```
public void commandAction(Command cmd, Displayable displayable) {

        if (cmd == exitCommand) {
                midlet.exitMidlet();
        }
        if (cmd == shootCommand) {
                takeSnapshot();
        }
}
```

There are only two possible actions from a selection of three options; the “FIRE” key and the menu item “Shoot” call the method *takeSnapshot()*, whilst the menu item

“Exit” informs the MIDlet the user wishes to terminate the application. Looking at the *takeSnapshot()* method, we see the line of code to capture a snapshot.

The parameters for getSnapshot is a String which states the required format of the data you wish to capture, i.e. JPEG, GIF etc. The default parameter for this is null, which means the data will be in the PNG format which is supported by all devices.

Images taken with MMAPI are of significantly lesser quality than using the C or C++ language on the Symbian operating system. I believe this to be either a limitation of the particular device used, or a bad implementation of MIDP on device, however further research would need to be done in order to clarify the precise reason for this.

```
private void takeSnapshot() {
        if (player != null) {
                try {
                        byte[] pngImage = videoControl.getSnapshot(
                        "encoding=png&width=1152&height=50");

                        midlet.imageCaptured(pngImage);
                } catch (MediaException mediaException) {
                        •
                        •
                        •
                }
        }
}
```

The parameters used in this project were “encoding=png&width=1152&height=50”, i.e. a PNG image of width 1152 pixels and height of 50 pixels. The maximum resolution supported by the Nokia 7610 device was 1152 pixels width and 864 pixels in height. However, the camera quality is exceptionally poor using MMAPI and also extremely slow. Placing the camera setting at full resolution causes the device to stop responding in excess of 30 seconds while it was processing the image. For this reason, the resolution had to be reduced, as any movement of the camera within this time

would blur the image or capture the image at the wrong moment in time, resulting in a different picture than required.

Analysis of Image Quality

If we take a look at the original barcode quality and compare this to the full size resolution of the Nokia camera, we can see that there is quite some blurring and distortion. It is quite difficult even with the human eye to determine exactly where one bar ends and another begins. What is even more difficult to determine, is the actual width of the bars, taking into consideration that the white bars are also part of the coding. The thinner bars bordering on impossible to visually calculate the widths.

Figure 12 Barcode Print Quality versus Photograph Quality Comparison

From the above photograph image, we can see there is slight lense refraction towards the upper right side of the image. This would not be of great importance had the full resolution of the camera been available. Several sample reads across the barcode could help determine any offset in the image. Instead, the image was captured with a height of a mere 50 pixels as seen below.

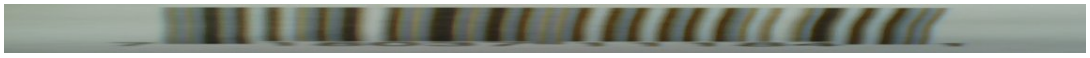


Figure 13 Actual Image Capture Quality

As seen from this image, the quality suffers even further when taking a smaller dimension. Rather than taking a slice of the image of height 50 pixels, the camera automatically squeezes the entire image into this small area, hence causing a much greater distortion throughout the barcode, particularly where light refraction occurs towards the edge of the camera lense. Once this image is captured, we then inform and return the image to the MIDlet for further processing.

Displaying Images on the Canvas

The MIDlet calls on the DisplayCanvas class to display the captured image. This class simply extends the Canvas class. The first thing the MIDlet must do is to pass the image to the display canvas. The canvas class cannot display the image in the current PNG format, therefore it is converted into an image object using the *createImage(byte[], int, int)* method:

```
public void setPngImage(byte[] pngImage) {
        image = Image.createImage(pngImage, 0, pngImage.length);
    }
```

After doing that, the MIDlet simply displays the canvas, which shows the image using the *paint()* method. This paints the image as show in Figure 13, an image of dimensions 1152 x 50 pixels.

```
public void paint(Graphics graphics) {
    graphics.setColor(0x00000000);
    graphics.fillRect(0, 0, getWidth(), getHeight());
    if (image != null) {
        graphics.drawImage(image, getWidth() / 2, getHeight() /
        2, Graphics.VCENTER | Graphics.HCENTER);
    }
}
```

The display on the Nokia 7610 is only capable of displaying 176 pixels in width. Rather than displaying a compressed image across this small display area, I took the design decision to display the centre 176 pixels of the image across the screen so that it was clear to the user how good the picture quality had come out, i.e. the phone screen displayed a sectioned image of 176 x 50 pixels. From this point the user has two options; to retake the picture, or start to decode the image.

```
public void commandAction(Command cmd, Displayable displayable) {
        if (cmd == retakeShotCommand) {
                midlet.retakeShot();
        }
        if (cmd == decodeCommand) {
                midlet.decodeImage(image);
        }
}
```

Processing the Image

All of the image processing is performed in the *ImageDecoder* class. Before attempting to read the barcode, the captured image must be converted into a readable format from which we can extract some relevant data. In order to achieve this, we need to read a single line across the centre of the barcode. Therefore the first step is to take a sample from the image we have captured by calling the *takeSample()* method.

This leaves us with a new image of size 1152 x 1 pixels. The MIDlet can now call the *startDecoding()* method. This method has three interesting features, coupled together with the purpose of transforming the PNG image into a monochrome image. The first section of code takes each individual red, green and blue (RGB) value for each pixel and stores it in an array of type integer.

```
int[] pixelsValues = new int[width * height];
img.getRGB(pixelsValues, 0, width, 0, 0, width, height);
```

After storing all of these RGB values, the next step is to convert this value into greyscale, as we have no use for colour. A simple for loop is enough to parse through the array, converting each pixel to its greyscale value as illustrated in the code sample below.

```
for (int x = 0; x < pixelsValues.length; x++) {
        int red = (pixelsValues[x] >> 16) & 0xff;
        int green = (pixelsValues[x] >> 8) & 0xff;
        int blue = pixelsValues[x] & 0xff;
```

Even though we now have a greyscale image, it cannot be converted to a monochrome image until we know what the halftone is for the image taken. This must be calculated every time, as each image has different colours, shades and lighting conditions. We perform this calculation by calling the *findHalftone()* method, passing the image as an argument.

```
private int findHalftone(int[] image) {
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;

        for (int x = 0; x < image.length; x++) {
                if (image[x] < min)
                        min = image[x];
                else if (image[x] > max)
                        max = image[x];
        }
        return (min + max) / 2;
}
```

What this method essentially does, is starts the local minimum with the highest possible integer value, and the maximum with the lowest possible value. We then parse through each greyscale pixel value. If the value is lower than the current minimum, it sets that value as the current minimum, and vice versa with the maximum values. Once we have parsed through the entire image, min and max will contain the minimum and maximum values which we sum up and half to get the exact halftone value, which is the value returned.

```
boolean[] monochrome = new boolean[pixelsValues.length];
for (int y = 0; y < pixelsValues.length; y++) {
        monochrome[y] = (pixelsValues[y] < halftone) ? true : false;
}
return monochrome;
```

The final step in the image processing, is to create a boolean array to store the new image data. We then examine each individual greyscale pixel, comparing the value of the pixel to the halftone value. Any value darker than the halftone, is assigned a positive value and subsequently, any value less being assigned a negative value. What we now have is a monochrome representation of the barcode sample, represented by a series of exactly 1152 zero's and one's.

Displaying the Monochrome Image

Displaying the monochrome image on the cell phone's display screen is handled by the *BarcodeCanvas* class. This class serves multiple purposes:

- Provides user with a visual representation of the decoded image
- User may compare monochrome barcode to the original barcode to clarify for correctness.
- Illustrates the ability of this program to integrate coupons and tickets feature

There are many methods in this class, however the one most relevant is the *paint()* method. J2ME has very limited graphical capabilities by design, therefore, a canvas is used to print either a black pixel or white pixel on screen, depending on the values in the monochrome array. This is achieved with the following code:

```
public void paint(Graphics graphics) {
        graphics.setColor(0xFFFFFFFF);
        graphics.fillRect(0, 0, getWidth(), getHeight());

        for (int y = 0; y < DISPLAY_HEIGHT; y++) {
                for (int x = 0; x < monochrome.length; x++) {

                        if (monochrome[x] == true) {
                                graphics.setColor(0x00000000);
                                graphics.drawLine(x, y, x, y);
                        } else {
                                graphics.setColor(0xFFFFFFFF);
                                graphics.drawLine(x, y, x, y);
                        }
                }
        }
}
```

For further improvements graphically, it would have been more ideal to use a tool such as J2ME Polish. This is an advance suite of tools used for creating powerful GUI's. However, time did not permit to integrate this feature.

Deciphering the Barcode

The most detailed and complicated of all the classes is the class for decoding the barcode, *BarcodeDecoder.java*. This class is responsible for decoding the raw monochrome representation of the barcode regardless of the amount of errors caused by a bad image capture of the barcode. Shadows or even lense refraction can give the illusion to make a single pixel, or even multiple pixels darker or lighter than the threshold level, thus giving a false read of the pixel in question. This is completely unavoidable without the use of a macro lense or a camera with a higher mega-pixel value.

Due to the low quality of the camera on cell phones, these and other factors can throw the reading of a barcode off quite substantially causing a black bar to appear thicker than in reality, or even a white space to appear thinner than its actual size. As seen earlier, the spacing in barcode and the widths of the bars is fundamental to their reading. Substantial blurring can even cause the complete loss of a bar altogether, rendering the image unreadable.

With the algorithm I developed for this application, most errors can be corrected. Unless a bar is not completely lost, i.e. it must occupy at least one pixel, then calculations can be performed and errors corrected to read the data correctly. This will be discussed in further detail later in this chapter.

When the MIDlet invokes the method to begin decoding the barcode, it passed the monochrome image to the method. This monochrome image is essentially an array of 1152 zero’s and one’s representing black and white space in the barcode, something quite similar to the following example:

“000000000000000011111100000000000111111000001111111111111111110000000000
111000011111110000000111111110000000011111111110000001111111111111100000000
000000000001111111111110000000000001110000011111100000011110000000111100000”

Should all of these pixel representations be of equal size, or even denominations of each other, then all we would need to do is divide by the lowest multiple. Blurring and other distortions make for very uneven distributions of pixels.

The first step in decoding the barcode from this data is to remove the unwanted white space from the data as this is useless information. The white space I refer to being the areas at the beginning and end of the barcode only. All other white space is also part of the coding. First, we remove the white space at the end of the barcode:

```
private boolean[] removeTrailingWhiteSpace(boolean[] monochrome) {
	boolean reachedBarcode = false;
	int endIndexPoint = monochrome.length;
	int i = monochrome.length - 1;

	do {
		if (monochrome[i] == true) {
			reachedBarcode = true;
			endIndexPoint = i;
		} else {
			i--;
		}
	} while (reachedBarcode == false);
		boolean[] newMonochrome = new boolean[endIndexPoint + 1];
		for (int j = 0; j < endIndexPoint + 1; j++) {
			newMonochrome[j] = monochrome[j];
		}
		return newMonochrome;
	}
```

Essentially what is done here, is we work backwards through the data until we reach an area of black pixels, i.e. the rear guard bars, and note this as the new end index point. We simply return a new array of such shorter length, hence removing trailing white space. The reason for working backwards first is simply to shorten the amount

of work to do, when going forwards through the data. Had we started at the beginning first, we would have to parse needlessly through the ending white space.

Following this, we then remove the white space at the beginning of the barcode.

```
private boolean[] removeInitialWhiteSpace(boolean[] monochrome) {
        boolean reachedBarcode = false;
        StringBuffer strBuffer = new StringBuffer();

        for (int i = 0; i < monochrome.length; i++) {
                if (monochrome[i] == true) {
                        reachedBarcode = true;
                }
                if (reachedBarcode == true) {
                        strBuffer.append((monochrome[i] == false) ? 0 : 1);
                }
        }

        boolean[] mono = strBufferToBoolArray(strBuffer);
        return mono;
}
```

Once complete, we now have only the barcode data remaining in our monochrome array. We must now try to work out what the size of each individual bar in the barcode should be in pixels. To do this we must consider the information we know about UPC barcodes:

- Each digit in a UPC barcode is composed of 7 bars of equal width
- There are exactly 12 numbers in the barcode
- Each barcode begins and ends with 3 guard bars on either end
- Every barcode has 5 bars to indicate the centre of the barcode
- All bars whether numbers of guard bars are of equal width

By performing a simple calculation, we know that there are:

$$(7 * 12) + (3 * 2) + 5 = 95 \text{ bars of equal length in a UPC barcode.}$$

We already know the barcode is 1152 pixels wide, however this also included the white space in the data. If we let:

Image size, i = 1152
Number of bars, k = 95
Let m = unknown variable white space

We need to find N, the precise pixel width per bar in the code;

$$N = \frac{i - m}{k}$$

Using a one mega pixel camera, the value of N generally varies from between 7 to 11 approximately, depending on the distance between the camera and barcode when the image is captured. The normal distance from the barcode should be 5 to 15cms when capturing the image.

Let us say for example, N = 8. Now let us assume we had a sample such as:

"11110000 00111100 00000000 00000111
11111000 00001111 00000000 00111110
00011111 11000000 00000011 11111100
00001111 11111100 00001111 11111111
11111111 11100000 00001111 00001111"

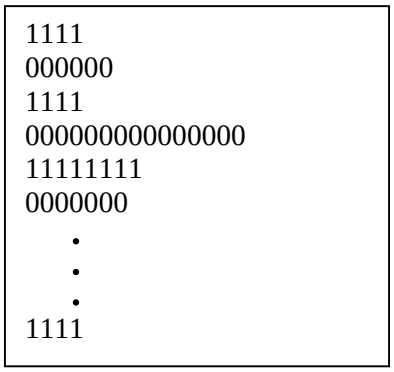
The problem with such a sample is that we cannot ascertain from the above data whether a group of "N" bars should be either zero or one. It must be a collection of one or the other, but can not be both. We also can not assume that a collection of bars is of one type just because there may be a majority. This could be a single bar being

blurred by two surrounding thicker bars. The solution to this problem is contained in the following code from the *correctErrors()* method from the *BarcodeDecoder* class.

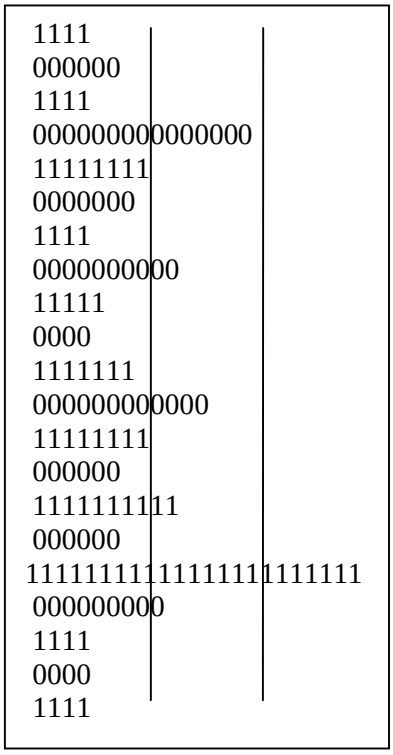
```
private boolean[] correctErrors(boolean[] monochrome) {
        .
        .
        .
        for (int i = 0; i < length; i++) {
                current = (monochrome[i] == false) ? 0 : 1;
                if (firstRead == true) {
                        previous = current;
                        currentCount = 1;
                        firstRead = false;
                } else {
                        if (current == previous) {
                                currentCount += 1;
                        } else {
                                correction = reduceError(currentCount);
                                for (int j = 0; j < correction; j++)
                                        strBuffer.append(previous);
                                previous = current;
                                currentCount = 1;
                        }
                }
        }
```

This method reads in a series of the same binary numbers and counts the number of occurrences of that number. If the number is the same as the previous, we add another to the count. Otherwise, we call the method to correct the error and start the next count.

What is effectively happening is all numbers are separating all of the numbers into groups:



After a change in the data bits, i.e. from a one to a zero or vice versa, we must pass the data segment and its count occurances to the *reduceError* method to calculate the true values. This method takes a row of data, and calculates by how many pixels, the bar is short of or greater than a single bar width as seen below. The horizontal lines representing one bar width in pixels (N = 8 as earlier). From this we can assume that any occurances of a pixel more than once and less than N, must represent one and one only bar length. Therefore we can pad this data to equal N.



Any number of occurrences greater than N we can be certain is at least one representation of a bar. We can not be certain if a reading greater than N is one bar representation with blurring, or two bars with blurring or being blurred over etc. Therefore we must set upper threshold level to indicate a boundary. This level being set to 80% of the calculated bar size in the application. This figure was arrived at by continuous testing at different levels. It was noted that the most accurate amount of correct readings were achieved at this level.

The *reduceError* method makes use of the mod function, due to the lack of support for floating points in CLDC 1.0.

```
private int reduceError(int count) {
        int remainder = 0;
        int correctedError = 0;
        int trueSize = 0;
        int thresholdLevel = ((barSize * THRESHOLD) / 10;
        correctedError = count / barSize;
        remainder = count % barSize;

        if (correctedError < MIN_VALUE) {
                correctedError = MIN_VALUE;
                remainder = 0;
        }

        if (correctedError >= MIN_VALUE && remainder > thresholdLevel) {
                correctedError += MIN_VALUE;
        }
        trueSize = correctedError * barSize;
        return trueSize;
}
```

Having corrected the errors by padding the data and removing noise, we now have a monochrome image of size 95 * N. In the above example, that would be 760 bytes. This is far more data than required, therefore we can compress the data into a single bit per bar representation, i.e. 95 bits.

```
private boolean[] compressBarcode(boolean[] monochrome) {
        boolean[] compressedBarcode = new boolean[NUM_BARS];
        int index = 0;
        for (int i = 0; i < monochrome.length; i += barSize) {
                compressedBarcode[index++] = monochrome[i];
        }
        return compressedBarcode;
}
```

Note that the original image was 1152 * 50 pixels. There are 16 bits in every pixel, giving us a total of 921,600 bits in the original image. We have now compressed the image data to a mere 95 bits.

The final phase in decoding the barcode is to now calculate the UPC from the modified data. Firstly we call the method:

"private StringBuffer calculateRawUpc(boolean[] barcode) {}"

This method will perform what I call calculating the raw UPC code. This refers to the translating of a binary digit such as zero, being read in as "0001101" or "1110010", and converting to a four digit number. Rather than being concerned about which side of the barcode we are reading, we simple calculate the changes in the bit, i.e. the previous example reads "3-2-1-1" regardless of barcode side.

Therefore the barcode with the UPC of "718037 111841" would be encoded in binary as "**101** 0111011 0011001 0110111 0001101 0111101 0111011 **01010**
1100110 1100110 1100110 1001000 1011100 1100110 **101"**.

We could quickly read this and translate to:

Number	Translation
0111011	1-3-1-2
0011001	2-2-2-1
0110111	1-2-1-3
0001101	3-2-1-1
0111101	1-4-1-1
0111011	1-3-1-2
1100110	2-2-2-1
1100110	2-2-2-1
1100110	2-2-2-1
1001000	1-2-1-3
1011100	1-1-3-2
1100110	2-2-2-1

Table 7 Binary Conversion

The next step in this process is then to call the *calculateUpc()* method passing the raw translation above in as a parameter. This method with then perform an analysis on these digits, and return the UPC code as follows:

Number	Translation
1-3-1-2	7
2-2-2-1	1
1-2-1-3	8
3-2-1-1	0
1-4-1-1	3
1-3-1-2	7
2-2-2-1	1
2-2-2-1	1
2-2-2-1	1
1-2-1-3	8
1-1-3-2	4
2-2-2-1	1

Table 8 Final Conversion

Finally, we then call the *checkUPCCode()* method. This is a procedure to verify that we have a correct UPC product code. It performs a calculation against the last digit in the number as discussed earlier, which verifies the validity of the code.

The decoding of the barcode is now complete. The MIDlet then calls on the *DisplayForm* class to show this information to the user. From here, the user can decide if they would like to connect to the cellular network, and retrieve the information from the product database.

Connecting to the Server

The MIDlet now starts a new thread in the class *HttpConnector*. This class sets up a HTTP connection to the web server, sending the UPC code as post requests. The receiving servlet polls the database, returning the valid records to the user. This is then displayed on the users screen.

```
public String sendPostRequest(String requeststring) {
        HttpConnection hc = null;
        DataInputStream dis = null;
        DataOutputStream dos = null;
        StringBuffer messagebuffer = new StringBuffer();

        try {
        hc = (HttpConnection) Connector.open(url, Connector.READ_WRITE);
        hc.setRequestMethod(HttpConnection.POST);
                .
                .
                .
```

The first thing we must do is open up the connection to the web server for both sending and receiving options. We then set the request method to “POST”.

Following this, we then need to send the UPC data byte by byte, then close the data output stream.

```
dos = hc.openDataOutputStream();
byte[] request_body = requeststring.getBytes();
for (int i = 0; i < request_body.length; i++) {
        dos.writeByte(request_body[i]);
}
dos.flush();
dos.close();
```

Now the response back from the servlet has to be read, therefore we open the data input stream and read the data into a buffer. Upon reaching the end of the data, we close the input stream and return the data back to the user for displaying.

```
dis = new DataInputStream(hc.openInputStream());
int ch;
long len = hc.getLength();

if (len != -1) {
        for (int i = 0; i < len; i++) {
                if ((ch = dis.read()) != -1) {
                        messagebuffer.append((char) ch);
                }
        }
} else {
        while ((ch = dis.read()) != -1) {
                messagebuffer.append((char) ch);
        }
}
dis.close();
```

Returning the Data

The final class running on the server side, is the *BarcodeServlet* class. This is a typical HTTP servlet which sits awaiting HTTP post requests from the MIDlet. It calls the method *getSearchResults()* which polls the database for any relevant data matching the submitted barcode UPC.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
response)  throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        ServletInputStream in = request.getInputStream();
        DataInputStream din = new DataInputStream(in);

        String upcRequest = din.readUTF();
        din.close();
        String upcResponse = getSearchResults(upcRequest);
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);

        int size = upcResponse.length();
        dout.writeInt(size);
        dout.writeUTF (upcResponse);
        byte[] data = bout.toByteArray();

        response.setContentType("application/octet-stream");
        response.setContentLength( data.length );
        response.setStatus( response.SC_OK );

        OutputStream out = response.getOutputStream();
        out.write(data);
        out.close();
    }
```

Project Evaluation

Looking at the project overall, I am satisfied with the outcome and final application. The barcode scanner reads UPC barcodes successfully through what can only be described as a sub-standard camera quality. There are a few improvements that I would like to have been able to address had time permitted.

Firstly, due to the inferior quality of the camera, a regular to small size barcode proved quite difficult to read. I believe that had some correction algorithm such as the Reed-Solomon or Hamming code been implemented, may have helped slightly in the error detection process, but overall would not have improved in the decoding of the barcode. For example, the Hamming code can detect and correct single bit errors, or detect double bit errors. However, the algorithm which I devised is capable of detecting these errors also. The problem lies with the quantity of errors due to the camera blurring, lense refraction and shadows cast on the barcode during image capturing.

The Reed-Solomon error correction code works by over-sampling a polynomial constructed from the data. This would have been the most suitable algorithm to implement. However, it was not feasible again due to the camera quality. Had the camera been able to take a larger image, of 1152 x 864 using MIDP, it would have been possible to take many samples through the image at different points along the y-axis. Unfortunately, I was limited to a mere 50 pixels, which actually distorted the data further that the original image capture.

I also believe that had this mobile phone made better use of the MMAPI, then it is quite reasonable to assume that a one mega-pixel camera is more than sufficient to read a barcode. I believe the limitations are more related to the device used in implementation, rather than the camera resolution as such. Further tests on other devices would be required to verify this.

Another improvement I would like to have included would have been a machine learning algorithm. Rather than manually adjusting threshold levels for the accuracy of the error correction by repeatedly testing, it would have been more suitable to have a dynamically changing value based on feeding a set of training data into the application. Again, time did not permit this.

The standard J2ME graphical capabilities are quite poor and do not make for an impressive display. A further improvement of this project would have been to integrate J2ME Polish or similar build tool. This tool is an advanced build tool and GUI for J2ME applications which provides a programmer with the tools for creating very impressive graphical displays along with other great features such as database integration.

Overall, I found working with J2ME quite different to any type of programming I had done before. It can be very limiting in what can be done, and at times can be frustrating to discover an API you would like to use, or have used before is not available in J2ME due to its limited functionality. However, this is also a good opportunity to develop ones own custom solution providing a more personalised application. I believe the project went quite well and I achieved my goal of successfully decoding barcode through a mobile phone.

Another improvement that I would have liked to implement fully into the project would have been to integrate a ticket and coupon system. The basic functionality is there at the moment in the project, when the device displays the monochrome barcode. In order for this to work more successfully, a better GUI would need to be implemented, however the core functionality exists.

Bibliography

[1] Mobile Information Device (JSR-37), JSR Specification, Java 2 Platform, Micro Edition http://java.sun.com/products/midp/

[2] Connected, Limited Device Configuration (JSR-30), JSR Specification, Java 2 Platform, Micro Edition, http://java.sun.com/products/cldc/

[3] Comparison of Data Matrix and QR Code, http://www.autoid.org/2001_documents/ANSI/ANSI_WG6/R9_Neg_IEC62090_5thCDV_0701.doc

[4] Scanbuy media, http://www.scanbuy.com

[5] Semacode Barcode Reader and SDK http://www.semacode.org

[6] SINTEF Barcode Reader, http://www.sintef.no/content/page1____1218.aspx

[7] J2ME - The Complete Reference (2003 McGraw Hill)

[8] Programming Java 2 Micro Edition (John Wiley & Sons)

[9] J2ME in a Nutshell (O'Reilly)

[10] Wireless Java with J2ME

Appendix

BarcodeScannerMidlet.java

```
package jd.midlet;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import jd.camera.CameraCanvas;
import jd.camera.DisplayCanvas;
import jd.debug.BarcodeCanvas;
import jd.decoder.BarcodeDecoder;
import jd.decoder.ImageDecoder;
import jd.display.DisplayForm;
import jd.network.HttpConnector;

public class BarcodeScannerMidlet extends MIDlet {

        private BarcodeDecoder barcodeDecoder;
        private CameraCanvas cameraCanvas;
        private DisplayCanvas displayCanvas;
        private DisplayForm displayForm;
        private BarcodeCanvas barcodeCanvas;
        private Displayable currentDisplay;
        private ImageDecoder imageDecoder;
        private HttpConnector httpConnector;
        private int barSize = 0;

        public BarcodeScannerMidlet(){
                cameraCanvas = null;
                displayCanvas = null;
                currentDisplay = null;
```

```
}

public void startApp() {
        currentDisplay = getDisplay().getCurrent();
        if (currentDisplay == null) {
                cameraCanvas = new CameraCanvas(this);
                displayCanvas = new DisplayCanvas(this);
                barcodeCanvas = new BarcodeCanvas(this);
                imageDecoder = new ImageDecoder();
                getDisplay().setCurrent(cameraCanvas);
                cameraCanvas.start();
        } else {
                if (currentDisplay == cameraCanvas) {
                        cameraCanvas.start();
                }
                getDisplay().setCurrent(currentDisplay);
        }
}

public void imageCaptured(byte[] pngImage) {

        Image image = Image.createImage(pngImage, 0, pngImage.length);
        cameraCanvas.stop();
        displayCanvas.setImage(image);
        getDisplay().setCurrent(displayCanvas);
}

public void decodeImage(Image image){
        Image imageSample = imageDecoder.takeSample(image);
        boolean[] monochrome = imageDecoder.startDecoding(imageSample);
        barcodeDecoder = new BarcodeDecoder();
        monochrome = barcodeDecoder.startDecoding(monochrome);
        barSize = barcodeDecoder.getBarSize();
        barcodeCanvas.setBarSize(barSize);
```

```
            barcodeCanvas.createMatrix(monochrome);
            getDisplay().setCurrent(barcodeCanvas);
    }

    public void showInformation(boolean[] monochrome){
            displayForm = new DisplayForm(this);
            displayForm.setBarSize(barSize);
            displayForm.createMatrix(monochrome);
            displayForm.setErrorDetection(barcodeDecoder.getErrorDetection());
            displayForm.setUpc(barcodeDecoder.getUpc());
            displayForm.start();
    }

    public void updateInformation(String upcDetail){
            displayForm.setUpcDetail(upcDetail);
    }

    public Display getDisplay(){
            return Display.getDisplay(this);
    }

    public void retakeShot() {
            getDisplay().setCurrent(cameraCanvas);
            cameraCanvas.start();
    }

    public void pauseApp() {
            if (getDisplay().getCurrent() == cameraCanvas) {
                    cameraCanvas.stop();
            }
    }

    public void destroyApp(boolean unconditional) {
            if (getDisplay().getCurrent() == cameraCanvas) {
```

```
                    cameraCanvas.stop();
                }
        }

        public void exitMidlet() {
                getDisplay().setCurrent(null);
                destroyApp(true);
                notifyDestroyed();
        }

        public void connectToServer(String upc) {
                httpConnector = new HttpConnector(this);
                httpConnector.setUpc(upc);
                httpConnector.start();
        }
}
```

CameraCanvas.java

```
package jd.camera;

import java.io.IOException;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Graphics;
import javax.microedition.media.Manager;
import javax.microedition.media.MediaException;
import javax.microedition.media.Player;
import javax.microedition.media.control.VideoControl;
import jd.midlet.BarcodeScannerMidlet;

/**
 *
 * @author John Delaney
 */
public class CameraCanvas extends Canvas implements CommandListener {

        private final BarcodeScannerMidlet midlet;
        private Player player;
        private VideoControl videoControl;
        private final Command exitCommand;
        private Command shootCommand;
        private boolean active;
        private int xAxis;
        private int yAxis;
        private int canvasWidth;
```

```
private int canvasHeight;
private int displayWidth;
private int displayHeight;
private String errorMsgType;
private String errorMsg;

public CameraCanvas(BarcodeScannerMidlet midlet) {
        this.midlet = midlet;
        player = null;
        videoControl = null;
        active = false;
        exitCommand = new Command("Exit", Command.EXIT, 1);
        addCommand(exitCommand);
        setCommandListener(this);
        initialiseCamera();
}

public void initialiseCamera() {
        try {
                player = Manager.createPlayer("capture://video");
                player.realize();
                videoControl =
                   (VideoControl) Player.getControl("VideoControl");

                if (videoControl == null) {
                        terminatePlayer();
                        errorMsgType = "Video Error: ";
                        errorMsg = "Video Control not found!";
                } else {
                        videoControl.initDisplayMode(
                             VideoControl.USE_DIRECT_VIDEO, this);

                        canvasWidth = getWidth();
                        canvasHeight = getHeight();
```

```
                displayWidth = videoControl.getDisplayWidth();
                displayHeight = videoControl.getDisplayHeight();
                xAxis = (canvasWidth - displayWidth) / 2;
                yAxis = (canvasHeight - displayHeight) / 2;
                videoControl.setDisplayLocation(xAxis, yAxis);
                shootCommand = new Command("Shoot",
                        Command.SCREEN, 1);
                addCommand(shootCommand);
            }
        } catch (IOException ioexception) {
            errorMsgType = "IOException: ";
            errorMsg = ioexception.getMessage();
            showAlert( errorMsgType + errorMsg);
            terminatePlayer();
        } catch (MediaException mediaException) {
            errorMsgType = "Media Exception: ";
            errorMsg = mediaException.getMessage();
            showAlert( errorMsgType + errorMsg);
            terminatePlayer();
        } catch (SecurityException securityException) {
            errorMsgType = "Security Exception: ";
            errorMsg = securityException.getMessage();
            showAlert( errorMsgType + errorMsg);
            terminatePlayer();
        }
    }

    private void takeSnapshot() {
        if (player != null) {
            try {
            byte[] pngImage = videoControl.getSnapshot(
                    "encoding=png&width=1152&height=50");
            midlet.imageCaptured(pngImage);
```

```
                } catch (MediaException mediaException) {
                        errorMsgType = "Media Exception:";
                        errorMsg = mediaException.getMessage();
                        showAlert( errorMsgType + errorMsg);
                }
        }
}

public void paint(Graphics graphics) {
        graphics.setColor(0x00FF0000);
        graphics.fillRect(0, 0, getWidth(), getHeight());

        if (errorMsgType != null) {
                graphics.setColor(0x00000000);
                graphics.drawString(errorMsgType, 1, 1,
                        Graphics.TOP | Graphics.LEFT);
                graphics.drawString(errorMsg, 1, 1 +
                        graphics.getFont().getHeight(), Graphics.TOP
                                | Graphics.LEFT);
        }
}

public void start() {
        if ((player != null) && !active) {
                try {
                        player.start();
                        videoControl.setVisible(true);
                } catch (MediaException mediaException) {
                        errorMsgType = "Media Exception: ";
                        errorMsg = mediaException.getMessage();
                        showAlert( errorMsgType + errorMsg);
                } catch (SecurityException securityException) {
                        errorMsgType = "Security Exception: ";
                        errorMsg = securityException.getMessage();
```

```
                showAlert( errorMsgType + errorMsg);
            }
            active = true;
        }
    }

    public void stop() {
        if ((player != null) && active) {
            try {
                videoControl.setVisible(false);
                player.stop();
            } catch (MediaException mediaException) {
                errorMsgType = "Media Exception: ";
                errorMsg = mediaException.getMessage();
                showAlert( errorMsgType + errorMsg);
            }
            active = false;
        }
    }

    private void terminatePlayer() {
        if (player != null) {
            player.close();
            player = null;
        }
        videoControl = null;
    }

    public void commandAction(Command cmd, Displayable displayable) {

        if (cmd == exitCommand) {
            midlet.exitMidlet();
        }
        if (cmd == shootCommand) {
```

```
                takeSnapshot();
            }
        }

        public void keyPressed(int keyCode) {
            if (getGameAction(keyCode) == FIRE) {
                takeSnapshot();
            }
        }

    private void showAlert(String err) {
        Alert alert = new Alert("");
        alert.setString(err);
        alert.setTimeout(Alert.FOREVER);
        midlet.getDisplay().setCurrent(alert);
    }
}
```

DisplayCanvas.java

```
package jd.camera;

import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;
import jd.midlet.BarcodeScannerMidlet;


/*
 * DisplayCanvas.java
*/

/**
 *
 * @author John Delaney
 */
public class DisplayCanvas extends Canvas implements CommandListener {

        private final BarcodeScannerMidlet midlet;
        private final Command retakeShotCommand;
        private final Command decodeCommand;
        private Image image = null;
        public DisplayCanvas(BarcodeScannerMidlet midlet) {
                this.midlet = midlet;
                decodeCommand = new Command(
                        "Decode", Command.SCREEN, 0);
                retakeShotCommand = new Command(
                        "Retake", Command.BACK, 1);
```

```
            addCommand(decodeCommand);
            addCommand(retakeShotCommand);
            setCommandListener(this);
        }

        public void paint(Graphics graphics) {
            graphics.setColor(0x00000000);
            graphics.fillRect(0, 0, getWidth(), getHeight());

                if (image != null) {
                    graphics.drawImage(
                        image, getWidth() / 2, getHeight() / 2,
                            Graphics.VCENTER |
                                Graphics.HCENTER);
                }
        }

        public void setPngImage(byte[] pngImage) {
            image = Image.createImage(pngImage, 0, pngImage.length);
        }

        public void setImage(Image img) {
            this.image = img;
        }

        public void commandAction(Command cmd, Displayable displayable) {
            if (cmd == retakeShotCommand) {
                midlet.retakeShot();
            }
            if (cmd == decodeCommand) {
                 midlet.decodeImage(image);
            }
        }
}
```

ImageDecoder.java

```
package jd.decoder;

import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

public class ImageDecoder {

	private static final int SAMPLE_HEIGHT = 1;
	public ImageDecoder() {
	}

	public boolean[] startDecoding(Image img) {
		int width = img.getWidth();
		int height = img.getHeight();
		int[] pixelsValues = new int[width * height];
		img.getRGB(pixelsValues, 0, width, 0, 0, width, height);
		for (int x = 0; x < pixelsValues.length; x++) {
			int red = (pixelsValues[x] >> 16) & 0xff;
			int green = (pixelsValues[x] >> 8) & 0xff;
			int blue = pixelsValues[x] & 0xff;
			pixelsValues[x] = (2 * red + 5 * green + blue) / 8;
		}

		int halftone = findHalftone(pixelsValues);
		boolean[] monochrome = new boolean[pixelsValues.length];
		for (int y = 0; y < pixelsValues.length; y++) {
			monochrome[y] = (pixelsValues[y] < halftone) ? true : false;
		}

		return monochrome;
	}
```

```
    private int findHalftone(int[] image) {
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;

        for (int x = 0; x < image.length; x++) {
            if (image[x] < min)
                min = image[x];
            else if (image[x] > max)
                max = image[x];
        }
        return (min + max) / 2;
    }

    public Image takeSample(Image image) {

        int imageWidth = image.getWidth();
        int imageHeight = image.getHeight();
        Image sampledImage =
            Image.createImage(imageWidth, SAMPLE_HEIGHT);
        Graphics graphics = sampledImage.getGraphics();
        for (int y = 0; y < SAMPLE_HEIGHT; y++) {
            for (int x = 0; x < imageWidth; x++) {
                graphics.setClip(x, y, 1, 1);
                int dx = (x * imageWidth) / imageWidth;
                int dy = imageHeight / 2;
                graphics.drawImage(
                    image, x - dx, y - dy, Graphics.LEFT
                        | Graphics.TOP);
            }
        }
        return Image.createImage(sampledImage);
    }
}
```

BarcodeDecoder.java

```
package jd.decoder;

public class BarcodeDecoder {
	private static final int ZERO = 3211;
	private static final int ONE = 2221;
	private static final int TWO = 2122;
	private static final int THREE = 1411;
	private static final int FOUR = 1132;
	private static final int FIVE = 1231;
	private static final int SIX = 1114;
	private static final int SEVEN = 1312;
	private static final int EIGHT = 1213;
	private static final int NINE = 3112;
	private static final int NUM_BARS = 95;
	private static final int THRESHOLD = 8;
	private static final int MIN_VALUE = 1;
	private static final int START_END_VALIDATION = 111;
	private static final int MID_VALIDATION = 11111;
	private static final int CODE_LENGTH = 4;
	private static final int START_STOP_LENGTH = 3;
	private static final int MID_LENGTH = 5;

	private int barSize = 0;
	private String upc = "";
	private StringBuffer strRawUpc;
	private boolean errorDetected = false;
	public BarcodeDecoder() {
	}
	public int getBarSize() {
		return barSize;
	}
```

```
public String getUpc() {
	return upc;
}

public boolean getErrorDetection(){
	return errorDetected;
}

public boolean[] startDecoding(boolean[] monochrome) {
	monochrome = removeTrailingWhiteSpace(monochrome);
	monochrome = removeInitialWhiteSpace(monochrome);

	barSize = calculateBarSize(monochrome);
	monochrome = correctErrors(monochrome);
	monochrome = compressBarcode(monochrome);
	this.strRawUpc = calculateRawUpc(monochrome);
	this.upc = calculateUpc(strRawUpc);
	boolean error = checkUPCCode(upc);
	if(error == true){
		errorDetected = true;
	}
	return monochrome;
}

private StringBuffer calculateRawUpc(boolean[] barcode) {
	StringBuffer strRawUpc = new StringBuffer();
	int current = 0;
	int currentCount = 0;
	int previous = 0;

	boolean firstRead = true;

	for (int i = 0; i < barcode.length; i++) {
```

```
                current = (barcode[i] == false) ? 0 : 1;
                if (firstRead == true) {
                        previous = current;
                        currentCount = 1;
                        firstRead = false;
                } else {
                        if (current == previous) {
                                currentCount += 1;
                        } else {
                                strRawUpc.append(currentCount);
                                previous = current;
                                currentCount = 1;
                        }
                }
        }
        strRawUpc.append(currentCount);
        return strRawUpc;
}

private String calculateUpc(StringBuffer strRawUpc) {
        StringBuffer upCode = new StringBuffer();
        String startCode = new String();
        String midCode = new String();
        String endCode = new String();
        int indexPoint = 0;

        startCode = readRawUpc(
                strRawUpc, START_STOP_LENGTH, indexPoint);
        indexPoint += START_STOP_LENGTH;

        for (int j = 0; j < 6; j++) {
                upCode.append(readRawUpc(
                        strRawUpc, CODE_LENGTH, indexPoint));
                indexPoint += CODE_LENGTH;
```

```
        }

        midCode = readRawUpc(strRawUpc, MID_LENGTH, indexPoint);
        indexPoint += MID_LENGTH;

        for (int j = 0; j < 6; j++) {
                upCode.append(readRawUpc(
                        strRawUpc, CODE_LENGTH, indexPoint));
                indexPoint += CODE_LENGTH;
        }

        endCode = readRawUpc(
                strRawUpc, START_STOP_LENGTH, indexPoint);
        String upcCode = strBufferToString(upCode);
        if(upcCode.length() != 12) {
                errorDetected = true;
        }
        return upcCode;
}

private String readRawUpc(StringBuffer strRawUpc, int length,
                int indexPoint) {
        String strTranlatedCode = new String();
        int number = 0;
        String strNum = new String();

        for (int i = indexPoint; i < (indexPoint + length); i++) {
                strTranlatedCode += strRawUpc.charAt(i);
        }

        number = stringToInt(strTranlatedCode);

        switch (number) {
                case ZERO:
```

```
            strNum = "0";
            break;

        case ONE:
            strNum = "1";
            break;

        case TWO:
            strNum = "2";
            break;

        case THREE:
            strNum = "3";
            break;

        case FOUR:
            strNum = "4";
            break;

        case FIVE:
            strNum = "5";
            break;

        case SIX:
            strNum = "6";
            break;

        case SEVEN:
            strNum = "7";
            break;

        case EIGHT:
            strNum = "8";
            break;
```

```
                case NINE:
                        strNum = "9";
                        break;

                case START_END_VALIDATION:
                        strNum = "X";
                        break;

                case MID_VALIDATION:
                        strNum = "X";
                        break;

                default:
                        strNum = " ERROR ";
                        break;
        }

        return strNum;
}

private boolean[] compressBarcode(boolean[] monochrome) {
        boolean[] compressedBarcode = new boolean[NUM_BARS];
        int index = 0;
        for (int i = 0; i < monochrome.length; i += barSize) {
                compressedBarcode[index++] = monochrome[i];
        }
        return compressedBarcode;
}

private int calculateBarSize(boolean[] monochrome) {
        return monochrome.length / NUM_BARS;
}
```

```
private boolean[] correctErrors(boolean[] monochrome) {
        int length = monochrome.length;
        int current = 0;
        int currentCount = 0;
        int previous = 0;
        int correction = 0;
        boolean firstRead = true;
        StringBuffer strBuffer = new StringBuffer();

        for (int i = 0; i < length; i++) {
                current = (monochrome[i] == false) ? 0 : 1;
                if (firstRead == true) {
                        previous = current;
                        currentCount = 1;
                        firstRead = false;
                } else {
                        if (current == previous) {
                                currentCount += 1;
                        } else {
                                correction = reduceError(currentCount);
                                for (int j = 0; j < correction; j++) {
                                        strBuffer.append(previous);
                                }
                                previous = current;
                                currentCount = 1;
                        }
                }
        }
        for (int j = 0; j < correction; j++) {
                strBuffer.append(current);
        }

        boolean[] mono = strBufferToBoolArray(strBuffer);
```

```
        return mono;
}

private int reduceError(int count) {
        int remainder = 0;
        int correctedError = 0;
        int trueSize = 0;
        int thresholdLevel = (barSize * THRESHOLD) / 10;
        correctedError = count / barSize;
        remainder = count % barSize;

        if (correctedError < MIN_VALUE) {
                correctedError = MIN_VALUE;
                remainder = 0;
        }

        if (correctedError >= MIN_VALUE && remainder > thresholdLevel)
        {
                correctedError += MIN_VALUE;
        }

        trueSize = correctedError * barSize;
        return trueSize;
}

private boolean[] removeInitialWhiteSpace(boolean[] monochrome) {
        boolean reachedBarcode = false;
        StringBuffer strBuffer = new StringBuffer();

        for (int i = 0; i < monochrome.length; i++) {
                if (monochrome[i] == true) {
                        reachedBarcode = true;
                }
                if (reachedBarcode == true) {
```

```
                    strBuffer.append((monochrome[i] == false) ? 0 : 1);
                }
            }

            boolean[] mono = strBufferToBoolArray(strBuffer);
            return mono;
        }

        private boolean[] removeTrailingWhiteSpace(boolean[] monochrome) {
            boolean reachedBarcode = false;
            int endIndexPoint = monochrome.length;
            int i = monochrome.length - 1;

            do {
                if (monochrome[i] == true) {
                    reachedBarcode = true;
                    endIndexPoint = i;
                } else {
                    i--;
                }
            } while (reachedBarcode == false);

            boolean[] newMonochrome = new boolean[endIndexPoint + 1];

            for (int j = 0; j < endIndexPoint + 1; j++) {
                newMonochrome[j] = monochrome[j];
            }

            return newMonochrome;
        }

        private String strBufferToString(StringBuffer strBuffer) {
            return String.valueOf(strBuffer);
        }
```

```
        private int stringToInt(String strInteger) {
                return Integer.valueOf(strInteger).intValue();
        }

        private boolean[] strBufferToBoolArray(StringBuffer strBuffer) {
                int length = strBuffer.length();
                boolean[] boolArray = new boolean[length];

                for (int i = 0; i < length; i++) {
                        int temp = (int) strBuffer.charAt(i) - (int) '0';
                        boolArray[i] = (temp == 0) ? false : true;
                }
                return boolArray;
        }

 public boolean checkUPCCode(String upcCode) {

    int[] upcArray = createUPCCodeArray(upcCode);

    return calculateCheckDigit(upcArray) == upcArray[upcArray.length - 1];
 }

private int calculateCheckDigit(int[] upcArray) {

     int q = calculateOddTotal(upcArray) + calculateEvenTotal(upcArray);
     return 10 - (q % 10);
}

private int calculateOddTotal(int[] upcArray) {
 int n = 0;
    for (int i = 0; i < upcArray.length; i += 2)
       n += upcArray[i];
    return n * 3;
```

```
    }

    private int calculateEvenTotal(int[] upcArray) {
        int m = 0;
        for (int i = 1; i < upcArray.length - 1; i += 2)
            m += upcArray[i];
        return m;
    }

    private int[] createUPCCodeArray(String upcCode) {
            int[] upcArray = new int[upcCode.length()];

        try {
            for (int i = 0; i < upcArray.length; i++)
                upcArray[i] = Integer.parseInt(upcCode.substring(i, i + 1));
        } catch (NumberFormatException e) {
            e.printStackTrace();
        }
        return upcArray;
    }
}
```

BarcodeCanvas.java

```
package jd.debug;

import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Graphics;
import jd.midlet.BarcodeScannerMidlet;

public class BarcodeCanvas extends Canvas implements CommandListener {
	private final BarcodeScannerMidlet midlet;
	private final Command exitCommand;
	private final Command translateBarcode;
	private static final int DISPLAY_HEIGHT = 50;
	private static final int NUM_BARS = 95;
	private boolean[] monochrome;
	private boolean[] displayMono;
	private int barSize;

	public BarcodeCanvas(BarcodeScannerMidlet midlet) {
		this.midlet = midlet;
		exitCommand = new Command("Exit", Command.EXIT, 0);
		translateBarcode = new Command(
			"Translate", Command.SCREEN, 1);

		addCommand(exitCommand);
		addCommand(translateBarcode);
		setCommandListener(this);
	}

	public void createMatrix(boolean[] mono) {
```

```
            monochrome = mono;
            // displayMono = createDisplayableBarcode(mono);
            repaint();
        }

        public void setBarSize(int barSize) {
            this.barSize = barSize;
        }

        public void paint(Graphics graphics) {
            graphics.setColor(0xFFFFFFFF);
            graphics.fillRect(0, 0, getWidth(), getHeight());
            for (int y = 0; y < DISPLAY_HEIGHT; y++) {
                for (int x = 0; x < monochrome.length; x++) {
                    if (monochrome[x] == true) {
                        graphics.setColor(0x00000000);
                        graphics.drawLine(x, y, x, y);
                    } else {
                        graphics.setColor(0xFFFFFFFF);
                        graphics.drawLine(x, y, x, y);
                    }
                }
            }
        }

        public void commandAction(Command command, Displayable displayable) {
            if (command == exitCommand) {
                midlet.exitMidlet();
            }
            if (command == translateBarcode) {
                midlet.showInformation(monochrome);
            }
        }
}
```

DisplayForm.java

```
package jd.display;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.StringItem;
import javax.microedition.io.*;
import java.io.*;

import jd.midlet.BarcodeScannerMidlet;

public class DisplayForm extends Form implements CommandListener, Runnable {
	private final BarcodeScannerMidlet midlet;
	private final Command exitCommand;
	private final Command retakeShotCommand;
	private final Command connectCommand;
	private StringItem status;
	private StringItem detail;
	private StringItem upc;
	private int barSize;
	private boolean errorDetected = false;
	boolean[] monoChrome;

	public DisplayForm(BarcodeScannerMidlet midlet) {
		super("Info");
		this.midlet = midlet;
		upc = new StringItem("UPC: ", "...");
		status = new StringItem("Status: ", "Decoding... ");
		detail = new StringItem("Detail: ", " ");
		append(status);
		append(upc);
```

```
        append(detail);
        exitCommand = new Command("Exit", Command.EXIT, 0);
        retakeShotCommand =
                new Command("Retake", Command.BACK, 1);
        connectCommand = new Command("Connect", Command.BACK, 1);

        addCommand(exitCommand);
        addCommand(retakeShotCommand);
        addCommand(connectCommand);
        setCommandListener(this);
}

public void setBarSize(int size){
        barSize = size;
}

public void setUpc(String code){
        this.upc.setText(code);
}

public void setUpcDetail(String details){
        this.detail.setText(details);
}

public void setErrorDetection(boolean error){
        errorDetected = error;
}

public void createMatrix(boolean[] mono) {
        monoChrome = mono;
}

public String booleanArrayToString(boolean[] array) {
        String strValues = "";
```

```
            strValues = "BAR SIZE: " + barSize;
            for (int i = 0; i < array.length; i++) {
                  if (i % barSize == 0)
                        strValues += "\n";
                  strValues += (array[i] == false) ? 0 : 1;
            }
            return strValues;
      }

      public void start() {
            //String values = booleanArrayToString(monoChrome);
            //upc.setText(values);
            midlet.getDisplay().setCurrent(this);
      }

      public void stop() {
      }

      public void commandAction(Command command, Displayable arg1) {
            // TODO Auto-generated method stub
            if (command == exitCommand) {
                  midlet.exitMidlet();
            }
            if (command == retakeShotCommand) {
                  midlet.retakeShot();
            }
            if (command == connectCommand) {
                  midlet.connectToServer(upc.getText());
            }
      }

      public void run() {
      }
}
```

HttpConnector.java

```
package jd.network;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;
import javax.microedition.lcdui.Alert;
import jd.midlet.BarcodeScannerMidlet;

public class HttpConnector implements Runnable {
	private BarcodeScannerMidlet midlet;
	private String url = "http://89.124.44.3:8080/BarcodeServlet";
	private String upc;
	public HttpConnector(BarcodeScannerMidlet midlet) {
		this.midlet = midlet;
	}

	public void setUpc(String upc) {
		this.upc = upc;
	}

	public void start() {
		Thread connectorThread = new Thread(this);
		connectorThread.start();
	}

	public void run() {
		String details = sendPostRequest(upc);
		midlet.updateInformation(details);
```

```
}

public String sendPostRequest(String requeststring) {
        HttpConnection hc = null;
        DataInputStream dis = null;
        DataOutputStream dos = null;
        StringBuffer messagebuffer = new StringBuffer();

        try {
                hc = (HttpConnection)
                        Connector.open(url, Connector.READ_WRITE);
                hc.setRequestMethod(HttpConnection.POST);
                dos = hc.openDataOutputStream();
                byte[] request_body = requeststring.getBytes();
                for (int i = 0; i < request_body.length; i++) {
                        dos.writeByte(request_body[i]);
                }
                dos.flush();
                dos.close();
                dis = new DataInputStream(hc.openInputStream());
                int ch;
                long len = hc.getLength();

                if (len != -1) {
                        for (int i = 0; i < len; i++) {
                                if ((ch = dis.read()) != -1) {
                                        messagebuffer.append((char) ch);
                                }
                        }
                } else {
                        while ((ch = dis.read()) != -1) {
                                messagebuffer.append((char) ch);
                        }
                }
```

```
                dis.close();

            } catch (IOException ex) {
                showAlert(ex.getMessage());
            } finally {
                try {
                    if (hc != null)
                        hc.close();
                } catch (IOException ex) {}
                try {
                    if (dis != null)
                        dis.close();
                } catch (IOException ex) {}
                try {
                    if (dos != null)
                        dos.close();
                } catch (IOException ex) {}
            }
            return messagebuffer.toString();
        }

        private void showAlert(String err) {
            Alert alert = new Alert("");
            alert.setString(err);
            alert.setTimeout(Alert.FOREVER);
            midlet.getDisplay().setCurrent(alert);
        }
}
```

BarcodeServlet.java

```
package jd.server;
import java.io.*;
import java.net.*;
import java.util.ArrayList;
import javax.servlet.*;
import javax.servlet.http.* ;
import java.sql.*;

public class BarcodeServlet extends HttpServlet {
   private Connection connection;
   private Statement statement;

   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DATABASE_URL = "jdbc:mysql://localhost:3306/barcode";
   static final String USERNAME = "delanej";
   static final String PASSWORD = "********";
   static final String FIELD_DELIM = "\t";

   String[] results;

   private void setResults(String[] results){
      this.results = results;
   }

   public void init(ServletConfig config) throws ServletException {
      super.init(config);
   }

   public void destroy() {
     }
```

```
    /** Processes requests for both HTTP <code>GET</code> and
<code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");

        ServletInputStream in = request.getInputStream();
        DataInputStream din = new DataInputStream(in);

        String upcRequest = din.readUTF();
        din.close();
       String upcResponse = getSearchResults(upcRequest);

       ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);

        int size = upcResponse.length();
        dout.writeInt(size);
        dout.writeUTF (upcResponse);

        byte[] data = bout.toByteArray();

        response.setContentType("application/octet-stream");
        response.setContentLength( data.length );
        response.setStatus( response.SC_OK );

        OutputStream out = response.getOutputStream();
        out.write(data);
        out.close();
    }
```

```
/** Handles the HTTP <code>GET</code> method.
 * @param request servlet request
 * @param response servlet response
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

/** Handles the HTTP <code>POST</code> method.
 * @param request servlet request
 * @param response servlet response
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

/** Returns a short description of the servlet.
 */
public String getServletInfo() {
    return "John Delaneys Barcode Data Processing Servlet";
}

public String getSearchResults(String searchTerm){
    String sqlQuery = "SELECT detail FROM information
                              WHERE upc='"+searchTerm+"'";
    return readFromDatabase(sqlQuery);
}
 private String readFromDatabase(String sqlQuery){

    ResultSet resultSet = null;
    //ArrayList tempArray = new ArrayList();
    String tuple = "";
```

```
try {

   // Load database driver class
   Class.forName( JDBC_DRIVER );

   // establish connection with database
   connection = DriverManager.getConnection(DATABASE_URL,
                                        USERNAME, PASSWORD);
   // Create the statement for querying the DB
   statement = connection.createStatement();

   resultSet = statement.executeQuery( sqlQuery );

   // Get the metadata
   ResultSetMetaData metaData = resultSet.getMetaData();

   // Count the columns
   int numCols = metaData.getColumnCount();

   while(resultSet.next()){
      // reset the tuple to prevent duplicates
      tuple = "";
      for(int i = 1; i <= numCols; i++){
         tuple += resultSet.getObject(i) + FIELD_DELIM;
      }
  } // end while

} catch( SQLException sqlException ){
   // detect any problems interacting with the database
   System.err.println(sqlException);
   sqlException.printStackTrace ();
}
```

```
        catch( ClassNotFoundException classNotFoundException ){
            // class not found exception
            System.err.println(classNotFoundException);
            classNotFoundException.printStackTrace ();
        }

        // ensure statement and connection are closed properly
        finally {
            try{
                statement.close();
                connection.close();
            } catch( SQLException sqlException ){
                // handle exceptions closing statement and connection
                sqlException.printStackTrace();
            }
        }
        //return this.results;
        return tuple;
    }
}
```