

Migrating an Application to Java2 Micro Edition: From Port to Portability

**Scott Palmer, Nat Panchee, Judy Sullivan,
Karen Thabet, and Sten Westgard**
Under the guidance of Professor Charles Tappert
Pace University

Abstract

A Nursing Information System (NIS) physical assessment application was ported to a Java 2 Micro Edition (J2ME) from a proprietary device running an obsolete operating system. The application was completely redesigned and rewritten, using J2ME to give the application cross-platform capabilities on multiple wireless mobile devices, and using object-oriented design to create portable classes for use in future applications. A small prototype was built, then scaled up to a full application, on Palm and Handspring handhelds running the PalmOS. A C++ conduit was written to allow the prototype to transfer data off a Palm handheld onto the PC. A complementary application was written for the PC, reusing many of the classes written for the handheld application. A wireless extension was written for the handheld, utilizing Java servlets on a web server to email the data as an attachment to a user-specified email address. Thus, the final application allows a user to record information from a physical assessment of a patient and send that data to a PC via synchronization through the conduit or send that data wirelessly. After the completion of the various applications, the balance of performance and portability design principles are analyzed for the J2ME handheld, as well as issues concerning code reuse between the handheld and PC applications.

Introduction

The Nurse Information System (NIS) is a system based on the Nightingale Tracker Device developed by FITNE, Inc., a member organization serving nursing schools and healthcare institutions throughout the United States and in several other countries. The Nightingale Tracker is currently used by Pace University's Leinhard School of Nursing, but its cost restricts its use to only a few students at a time. The Nightingale Tracker begins at \$2000.00 for one tracker and goes up to almost \$25,000.00 for a complete system consisting of 11 trackers and the FITNE supplied server. Pace University's search for a more cost-effective system led to the collaboration between FITNE and Pace to develop an application for widely available handhelds such as PalmOS and Windows CE based devices.

The NIS was developed to evaluate the same systems and symptoms as the Nightingale Tracker. However, in addition to its low cost, the NIS has several advantages over the Nightingale Tracker. The Nightingale Tracker software runs only on the FITNE Data Rover provided. This Data Rover is very bulky and awkward to carry and use. The Palm, Handspring, and other handheld devices are much lighter and can fit into any purse or pocket. The FITNE device also runs an operating system called Magic OS which is no longer supported by its developer. This limits the opportunity for upgrades and expansion of the Nightingale Tracker software. PalmOS and Windows CE are both actively used and supported at this time. The NIS is implemented using Java 2 Micro Edition (J2ME). Therefore, the NIS can be run on any device that is J2ME enabled and has at least 8MB memory.

The application walks the student nurse from beginning to end of a physical assessment. The nurse begins by entering patient identification information, including the date of the physical assessment, patient's ID number, nurse

information, doctor information, patient's medication history, patient's general medical history, and reason for patient's visit. After completing this information, the nurse proceeds to accessing various body systems. Access to the NIS anatomy systems can be done in two different ways. The first mode of access is provided by a graphical image of a body that, when pointed to, will take the nurse to the respective body system. The second manner in which the body systems can be accessed is by touching the MENU button next to the graphical image. A list of body systems will appear on the screen and touching the respective words accesses each system. At that point, the nurse will be presented with that particular body system's menu screen, which includes a list of body system assessment items. These items include access to textboxes for inputting subjective notes, assessment notes, and treatment plan. In addition, there are items that allow the nurse to collect patient information when reviewing body systems, performing a physical exam, and developing a treatment plan. Information collection is aided by checkboxes and drop-down-menus containing symptoms, physical assessments, and suggested treatment options. When the nurse is finished with this particular body system, he/she can return to the body image on the home screen to select another anatomy system. Once the physical assessment is complete, a report is generated on the Palm based on the data that was collected. This report can then be transferred to an instructor's PC.

Information on the Nightingale Tracker: FITNE provided no code, class lists, data structures, or documentation to the NIS project team. All that was supplied was the Nightingale Tracker device containing the finished application. Essentially, the team was given a black-box prototype. Team members could observe how the program functioned, but could not see any internal implementation details. Thus, the team built the program from the ground up, essentially using the working application on the Nightingale device as a source for requirements specification.

Analysis of Available Products for Implementation: Currently most of the healthcare software for mobile devices is platform-specific. A few products dominate the market: Patient Keeper (<http://www.patientkeeper.com>) has several thousand physicians using their software, which includes both personal and enterprise level applications for the Palm handheld, with functions for lab testing, hospital charges, clinical notes, etc. However, the core of their application is focused exclusively on the Palm. (They have do a separate version for Windows CE, but this introduces serious overhead into the development process as changes must be propagated across two platform versions).

Simple database applications for the healthcare market have also been created using programs like PendragonForms and HandDBase. For instance, the Visiting Nurses Association Home Health System (VNAHHS) used PendragonForms (a quick visual IDE & database tool for the Palm) to create an application for their nurses in the field (source: <http://palmpoweredenterprise.com/issues/issue200110/nurses001.html>). These programs provide only a limited set of functions (for instance, there is little control over screen presentation, and in general, no allowance for the inclusion of complex graphics in the resulting applications), and are, like PatientKeeper, focused only on the PalmOS.

Other companies like Air2Web (<http://www.air2web.com>) have a focus on wireless technology with a healthcare division. Again these companies are platform specific. One of the chief goals of the NIS project was to develop a cross-platform application, one that would work on a Palm, PocketPC, cell phone, wireless pager, or any other J2ME-capable device.

A very few companies, such as Catapult Technologies (<http://www.catapult-technologies.com>) have actually entered the healthcare market with J2ME applications. However, at the time of the NIS project inception, and even as late as March 2002, Catapult Technologies had not given any public details about its J2ME application. This project was unveiled at the JavaOne Conference in late March.

Methodology

The primary objective of the Nurse Information System (NIS) was to develop an application that mimics the FITNE Data Rover, but that can be implemented in a more portable and cost-effective device such as a Palm Pilot. Since there were no products on the market for Personal Digital Assistants (PDA) that had the exact look and feel of the FITNE Data Rover, the NIS team had no choice but to build the application from scratch. J2ME was the language of choice when it came to deciding which language to use to build the NIS application. This decision was based on the particular benefits J2ME provided as well as the abilities of the NIS team. The advantages of using J2ME are that it offers cross-platform capability and that it works on any device that can run a Java Virtual Machine (JVM) (e.g. Palm OS, Windows CE, and mobile phones). Additionally, it is comprised of a relatively small set of new classes to learn and is a readily available inexpensive development tool.

The challenge of the Nurse Information System (NIS) is to fit a large application, one used to aid nursing students in performing a patient physical assessment, onto a small footprint device. The NIS must implement an architecture that can dynamically bring into memory only the information that the user needs to see just at that moment, while keeping all of the information on file.

The NIS architecture is divided into four levels:

1. **General User Interface (GUI) Layer:** This layer encompasses all the *presentation logic*, all the essential screen elements of the application. A key feature is that every essential screen element must have the ability to dynamically rewrite itself to present the current information the user wants to see. Based on user-triggered events, the GUI layer obtains information about what to present next from the BOOP layer (see below). The GUI layer also signals to the BOOP layer when it should dynamically create a different AnatomySystem or Patient Information for viewing.
2. **Body Object-Oriented Programming (BOOP) Layer:** This layer encompasses all the *business logic*, so to speak (perhaps “body logic” is a more appropriate term). For instance, the BOOP layer understands that the Body has a number of Anatomy Systems, that each Anatomy System has a set number of Anatomy Subsystems, and that each Anatomy Subsystem may consist of a number of Conditions, Text fields, and/or Text boxes. The special relationships that exist between the data entities such as the Body, AnatomySystem, AnatomySubsystem, and Condition are also encapsulated in this layer. Like the GUI layer, the BOOP layer can be set dynamically. In an environment with more resources, the entire BOOP layer would be kept in memory. In the small footprint environment, it will only create and keep in memory the current Anatomy System (or Patient Information) in memory, writing all other components to file when not in view by the user. The BOOP layer also signals the DataStorage layer when it should write information in memory to file, and when it should read information from file into memory.
3. **DataStorage Layer:** This layer encompasses all the *persistence logic* for storage, reading and writing to file. A traditional database is not available for use, due to the development choice of Java 2 Micro Edition (J2ME). Instead, all data must be stored in byte arrays. The DataStorage layer encapsulates all of the structure and values of the various BOOP categories into specific read and write commands on byte arrays. In addition, the DataStorage layer has an extremely important feature: it builds the initial structure of the body in the BOOP layer.
4. **Conduit Layer:** This layer is a critical Palm-specific function. It allows the files created by the user to be transferred from the Palm to a PC. Without the conduit, the file cannot interact with the “greater world” of the PC, Internet, etc., and the application becomes trapped on the Palm.

This architecture roughly fits into a 3-tier approach, if you consider that the Conduit layer is a special function of the DataStorage layer. The 3-tier approach typically separates the presentation logic, the business logic, and the persistence logic.

One key feature of NIS is that it is never a fixed set of anatomy systems, subsystems, conditions, text boxes, and/or text fields. The Data Storage layer converts information stored in a file into a BOOP representation, which is then rendered by the GUI layer into a screen element. Thus, NIS allows an ever-changing combination of elements, questions, and structures to be represented. This dynamic ability is built into the architecture: the GUI presents only the information that the user requests to see; the BOOP layer only keeps in memory the relationship and structure logic of the system currently in view; and the DataStorage layer preserves all other information on file, writing systems that are coming “out of view” to file, and reading and building the new systems as they are requested to come “into view.” The *dynamic presentation ability* allows the NIS to exist on a small footprint device. The *dynamic business* and *datastorage ability* allows the program to create a physical assessment of unlimited size and variation – so the program can evolve and adapt to the user’s needs.

The Body Object-Oriented Program (BOOP) Layer

This layer encompasses all the *business logic*. For instance, the BOOP layer understands that the Body has a number of Anatomy Systems, that each Anatomy System has a set number of Anatomy Subsystems, and that each Anatomy Subsystem may consist of a number of Conditions, Text fields, and/or Text boxes. The special relationships that exist between the data entities such as the Body, AnatomySystem, AnatomySubsystem, Condition, are also encapsulated in this layer. Like the GUI layer, the BOOP layer can be set dynamically. In an environment with more resources, the entire BOOP layer would be kept in memory. In the small footprint environment, it will only create and keep in memory the current Anatomy System (or Patient Information) in memory, writing all other

components to file when not in view by the user. The BOOP layer also signals the DataStorage layer when it should write information in memory to file, and when it should read information from file into memory.

The BOOP layer interacts with the GUI layer in the following way: A GUI class (Glist, for example) will be passed a BOOP class reference. The GUI class will then use getX methods to obtain information about that particular BOOP instantiation, and render the appropriate number of screen elements. Because the GUI layer is built to be dynamic and reconstructs itself continually, it has no knowledge of what is contained in any particular AnatomySystem or AnatomySubsystem; the BOOP layer is the temporary repository of the knowledge of that structure.

The BOOP layer temporarily “saves state”: If a user makes a change to a GUI screen, that change is made in the appropriate BOOP class, and temporarily stored there. This information IS NOT written to file until a user makes a larger change. When a user changes from one AnatomySystem to another (for instance, from Head to Neck), at that time all the information from the current BOOP instantiation is passed to the DataStorage. The reason that a user change is not immediately written to file is due to the constraints of J2ME – when one change is made to a record, the entire record must be rewritten. This consumes precious time and resources not available on a small device. Waiting until there is a change of AnatomySystem results in fewer reads and writes to file, and should improve performance of the application.

The BOOP layer is dynamic: Body object can hold any number of AnatomySystems. An AnatomySystem can hold any number of Conditions, TextFields, and/or Textboxes. A TreatmentPlan Subsystem can hold any number of Ncheckboxes. This design allows flexibility in the construction of physical assessment. For instance, the alpha prototype was built with only two AnatomySystems (Head and Neck), while the full application has fifteen AnatomySystems. However, the code for both cases is the same.

The NIS Data Storage Layer

Overview

The classes that compose the NIS Data Storage Layer have two main responsibilities: to save data from the Java RecordStore and to retrieve data from the Java RecordStore. However, because the Java RecordStore can only hold data in the form of byte arrays, the data storage classes have the additional responsibility of converting data from the NIS Application into byte arrays.

The Byte Array

The most important aspect of the data storage layer is the byte array. The order and placement of the bytes in the array define the meaning of the information found in the array.

The NIS data storage classes view each Anatomy System (and the PatientInfo class) as if it were composed of three layers:

First Layer: Anatomy System
Second Layer: Review of Systems, Physical Exam, Assessment, or Treatment Subsystem
Third Layer: Conditions (ChoiceGroups), NTextFields, NTextBoxes, or CheckBoxes

It is important to note that each layer is composed of the layer that follows it. For example, the first layer is composed of the second layer; similarly, the components of the third layer create the second layer.

Each of the above three layers are represented by IDs and numbers in the actual byte array as follows:

First Layer: |- SYSTEM START-|-SYSTEM ID-|SYSTEM SIZE-|
Second Layer: |-LAYER TWO START-|-LAYER TWO ID-|-LAYER SIZE-|
Third Layer: |-LAYER THREE START-|-LAYER THREE ID-|-LAYER SIZE-|-TYPE-|-VALUE-|-SIZE OF TEXT-|-TEXT-|

System Start, Layer Two Start, Layer Three Start – indicate the start of their respective layers

System ID, Layer Two ID, Layer Three ID – are specific IDs found in the NISIDS interface

Value – indicates user’s selection

Text – the actual text included in text-boxes and text fields

The best way to understand the creation of a byte array is to walk through a simple example. The following discussion will be based on a simple anatomy system called HEAD, which has a PHYSICAL EXAM component that is composed of one TEXTBOX component.

The following are ID numbers created for our example (which would be defined in NISID):

ANATOMY SYSTEM START = 101
SECOND LAYER START = 102
THIRD LAYER START = 103

HEAD = 1
PHYSICAL EXAM = 7
SUBJECTIVE = 5
TEXTBOX = 4

The text in the TEXTBOX is as follows “This is the text”.

Given the above information, we can construct the following:

First Layer: |-101-|-1-|-22-|
Second Layer: |-102-|-7-|-20-|
Third Layer: |-103-|-5-|-18-|-4-|-0-|-16-|-This is the text-|

The actual byte array is as follows:

|-101-|-1-|-22-||-102-|-7-|-20-||-103-|-5-|-18-|-4-|-0-|-16-|-This is the text-|

The Classes

The classes that create the byte array can be divided into four parts, each with their own responsibilities. A brief overview of each division is as follows:

Manager Classes - These classes call upon other classes (either from the NIS Data Store, or the Java 2 ME library) to perform the tasks of Data Store Level.

NISRecordManager

The NISRecordManager is a wrapper class that encapsulates the functionalities provided by Java 2ME to create, insert, delete, and modify records in the Java RecordStore. Calls to methods in this class first perform a check of the data (for proper NIS formatting) before entry into the database.

NISDataManager

The NISDataManager is the class with the save and set method. A reference to the current Body is passed to this class when the constructor is called. The save method of this class takes as arguments the patient id and the name of the system to be saved. The class then makes calls to the NISRecordManager to determine if previous information has been saved for the current patient id. If previous information is found, the NISRecordManager returns a byte array of the information. The previous information is sent back to the NISDataManager and is used to obtain a summary (such as a determination if the information already exists). Finally, as the NISDataManager traverses through the Body and Anatomy System, it makes calls to the NISRecordWriter. When this process is completed, the NISRecordWriter returns a byte array, which is sent to the NISRecordManager to be saved in the Java RecordStore.

Reader Classes - These classes are called during the “loading” process and set the BOOP layer.

NISRecordReader and NISPatientReader

The NISRecordReader is a class whose primary responsibility is to read records from a byte array. (It does NOT read records from the Recordstore—that’s handled by the NISRecordManager) The NISRecordReader class converts byte arrays into NIS data using a template that is also used by the NISRecordWriter. It should be noted that this is NOT the class that loads data into an Anatomy System. This class is used in conjunction with the NISDataManager to set the BOOP layer.

Writer Class - These classes are called during the “saving” process and records BOOP layer information.

NISRecordWriter

The NISRecordWriter is a class whose primary responsibility is to write NIS information into a byte array. Methods in this class write bytes into an output stream. The resulting stream returns a byte array that is eventually saved into the Java RecordStore.

Interface - – The interface defines IDs used by the datastore level.

NISIDs

This is an interface that is implemented by all classes that either write or read from a byte array obtained from the recordstore. It is composed of static final byte declarations to assign byte values to AnatomySystem and AnatomySubsystems.

The GUI Classes

This layer contains all graphical user interface components through which the user interacts with the NIS.

The “NISApplication” Midlet

This is the class that makes all things possible on the PalmOS. It is the bridge between the PalmOS and all the other java classes of the NIS. Its functions include instantiating an object of the “BodyTouch” class, starting the application, pausing the application (which will not be necessary for our purposes) and breaking down (or destroying) the entire application when the user is finished with it.

The “BodyTouch” class

By having this class extend the built in “Canvas” class, it becomes the first class that the user actually sees and interacts with. It displays a modified graphic of Leonardo daVinci’s famous anatomical drawing and a graphical representation of text, all of which are touch sensitive. When the user touches the desired text or body part with their stylus, it calls for the corresponding “Glist” to appear.

This class also provides the user with several buttons at the bottom which provide a link to the patient info screen, a link to the current report screen, or a link which allows the user to exit the program entirely.

The “GPatientInfo” class

This class extends the built in “List” class, which displays a clickable list of patient information choices for the user. Its functions include appending the date/patient ID and linking to the dynamic GTextBox class. This allows the user to enter such things as Nurse/Dr info, Medication/Medical history and the reason for this current visit.

The “GTextBox” class

This class extends the built in “TextBox” class. It displays a blank text box with a title bar of the current option chosen. The user can then type in the related information.

The “GList” class

Much like the GPatientInfo class, this class also extends the “List” class and dynamically displays a list based on the parameters sent to it from the “BodyTouch” class (i.e., when the head is touched, BodyTouch tells GList which list to display). The Lists themselves vary but usually connect to a GTextBox, a GChoiceForm, or a GTreatment.

The “GChoiceForm” class

This class extends the built in “Form” class which allows us to provide the user with choicegroups (dropdown menus). These choicegroups are dynamically generated depending on a variety of factors. One choicegroup contains all the conditions involved with the particular anatomy system and the second group has the three possible assessment values (usually yes/no/not assessed) associated with the chosen condition. When a condition is chosen the class calls for its current condition value and displays it. The user can then change that value if needed. Once a condition and its new assessment value are chosen, they become linked together again.

The “GTreatment” class

When this class is called from the “GList” (by the user selecting “Treatment Plan”), it displays the corresponding checkboxes (if any) for the current anatomy system. It then listens for and acts upon any items that are checked by the user.

The “GReport” class

This class also extends “Form”. When it is called from the GList class, it traverses the BOOP (PatientInfo, AnatomySystem, NTextBox, etc.) and extracts/displays all the current information entered for the patient. The user can then review the information and exit the program or (if, upon review, some information needs to be changed) return to the body to correct it.

The Conduit

An Overview of Conduits

A conduit is a desktop plug-in (DLL or ActiveX COM component) made in a Personal Computer (PC or desktop) development environment. This is not code that runs on the Palm handheld, but rather an executable library that is loaded by the HotSync Manager during the synchronization process.

What Does a Conduit Do?

A conduit is responsible for the application’s data during synchronization between the handheld and a PC. The conduit needs to:

- Open and close databases on the Palm handheld.
- Determine the extent of synchronization necessary--upload only, download only, or a combination of both.
- Appropriately adds, deletes, and modifies records on the handheld and on the desktop.
- Working within a multi-user environment where more than one Palm handheld may be syncing to the same network or desktop computer.
- Converting the data in the application’s database records to appropriate data structures on the PC.
- Optionally, comparing records so that only modified records are synched.

The conduit is responsible for saving the data on the PC. If the conduit uploads to a file for a PC application, it needs to read and write data in that application’s format. The conduit may read and write records from a database on the desktop or some database on the network. As a result, each conduit handles storing and retrieving desktop data differently.

Conduit Development on Windows

Palm provides a Conduit Development Kit to facilitate development of conduits. It is possible to develop conduits using several programming languages including: C/C++, MFC, Visual Basic, or Java.

The Conduit

To develop the Palm Conduit, the team decided to use the Visual C++ and the Palm Conduit Development Kit. The conduit, as well as the installation and deinstallation programs, were written in C++. Although the CDK was available in Java, C++ was chosen to avoid complications for the end user.

The conduit has three required entry points, including:

- GetConduitName()
- GetConduitVersion()
- OpenConduit()

The code found in these methods has the task of opening the NIS databases and retrieving the data arrays in byte format. The conduit then creates a data file on the PC for the NIS patient information and the NIS primary data called patient.nis and primary.nis, respectively. These two files are used by the NISPC application.

The NISPC

The NISPC is an application that runs on the user’s PC allowing them to open and read files sent to them by the NIS Palm application. After allowing the user to find these files, the NISPC then reads the files and decodes them. Based on what it finds in these files, GUI panels are dynamically created and displayed. The user also has the option of exporting the information directly into a text file. The NISPC uses Java Swing components and currently consists of 21 classes, 13 of which are reused classes taken directly from the Palm side of the NIS project.

The Wireless Extension of NIS

Given the proliferation of devices in the Mobile Information Device Profile (MIDP), it would be highly impractical to create data conduits and other custom connection methods for each and every device on which the NIS might be installed. Indeed, it would defeat the core concept of the J2ME MIDP, which is to enable application development without a specific device in mind. One of the main strengths of J2ME is its ability to bridge platforms. Applications written in J2ME MIDP can be deployed to cell phones, pagers, and PDAs with similar expectations of performance. But if the only data transfer performed by the NIS application is through a PalmOS C++ conduit, then the application is *de facto* restricted to installations on the Palm platform. Thus, in order to insure that the NIS has a cross-platform future - that it can migrate from one device to another - it must take advantage of the J2ME networking features.

The NIS is portable, a nursing student can take it to a patient's home and perform the physical assessment. However, in order to get that data to an instructor, the student would have to return to his/her computer, synchronize the Palm with the computer, and only then could the student transfer the report to the instructor. An obvious improvement of the NIS would be to give the student the ability to transmit the report directly to the instructor directly from the patient's home. Since the NIS is portable, the data transmission should be portable, too.

An Overview of Java 2 Micro Edition Networking Technologies

The Java 2 Micro Edition has a built-in set of networking features. The Connected Limited Device Configuration (CLDC) provides a Generic Connection Framework and the MIDP provides a `HttpConnection` interface, which is part of the `javax.microedition.io` package, that defines the necessary methods and constants for an HTTP connection. Note that the HTTP connection is the only protocol a MIDP implementation *must support*, all other protocols are optional. In effect, this means that all applications wishing to insure their cross-platform capability must rely only on the HTTP connection for networking -- nothing else is guaranteed to be available.

This highlights a weakness of J2ME which is a by-product of its cross-platform strength: the features available to a J2ME application are severely limited to those capabilities that all devices can support. Thus, J2ME applications can suffer from a "lowest common denominator" approach to applications. A Palm or PocketPC may offer many sophisticated native features, but J2ME applications running on those platforms can only take advantage of the features that are also found on cell phones and wireless pagers.

Use Case: How the Wireless Extension Works

The NIS Wireless Extension works as follows: a user of the NIS program completes a physical assessment of a patient. Next, they view the Report, which is generated and displayed by the `GReport` class. The Report screen displays a "Send" button, which, when pressed, takes the user to a new screen. This Send screen allows the user to enter a target email address. Once entered, the user can then click a "Send Report" button. At this point, the user sees a `WaitScreen`, which displays an animation as well as information on the progress of the data transmission. Because of the slow speed of data transmission and the intermittent connectivity, this data transmission does take some time, although less than a minute. At the end of the transmission, an alert screen displays information to the user about the success or failure of the data transmission.

Once the data is transmitted, it is received by a Java servlet on a web server. The servlet checks the data to make sure it has received it correctly, converts the data into two specific files (`patient.nis`, which contains the administrative information about the patient, and `primary.nis`, which contains the actual data about the physical assessment of the patient), and attaches those files to an email message which is then sent to the email address submitted by the user. At the end of this, it sends a message back to the mobile device indicating the success or failure of this series of tasks. It is important to note that the tasks carried out by the servlet are transparent to the user; the user never interacts directly with the server and needs no knowledge of the existence of the server.

At some time later, another user (the same user who performed the physical assessment or someone else, for instance, the instructor) will receive an email message with the two files attached. These two files can then be loaded into the NISPC application, which allows the user to view the complete Report on the PC and generates a simple text file of the report.

Prototype Environment

The NIS Wireless Extension was developed and tested on a Palm VII wireless PDA with Palm.Net wireless connectivity (which is carried over the Cingular network). The web server in use was

<http://www.westgardpeer.com>, hosted on <http://www.webappcabaret.com>, a website hosting company that offers support for Java Servlets, JavaMail, and other Java 2 Enterprise Edition (J2EE) technologies at a relatively low cost.

The use of the Palm VII as the prototype device created some additional complications. The Palm VII makes use of a device-specific inethhttp protocol for wireless transmission. All HTTP connections made through the device must use the inethhttp protocol. To prevent this requirement from making the application a Palm VII-specific device, a simple if clause tests for the existence of the inethhttp protocol, and if it does exist, all URLs are converted from <http://somewebsite...> to “inethhttp://some website”.

Another limitation of the Palm VII inethhttp protocol is that the headers and fields in the HTTP connection are severely limited and all data being transmitted must be URL encoded. However, since these are limiting factors, it does mean that an application that works on the Palm VII will work on other devices as well; the application simply will not take advantage of some of the HTTP capabilities.

Balancing Performance and Portability

Two principles guided the development of the NIS application. The first principle was to optimize performance so that the NIS physical assessment application would run as quickly and efficiently as possible on a small device and provide a good user experience. The second principle was to build a system that looked beyond the narrow confines of the physical assessment application toward a flexible set of classes that could accommodate entirely new applications with minimal change. Just as the core goal for NIS was to make the application portable from device to device, the principle in design was to make the classes portable from application to application. At times, performance and portability were in conflict with each other, and a balance had to be struck.

There are several standard methods for optimizing J2ME applications for performance. The NIS application made use of the following techniques:

1. Using StringBuffers instead of Strings to avoid the overhead of String concatenation. For example, the NTextBox and NTextField classes use StringBuffers to store text. StringBuffers are also used during the data conversion performed by the Wireless Extension.
2. Reusing and recycling objects to avoid the memory expense of new object creation. For example, instead of creating 15 List screens, one for each anatomy system (head, neck, etc.), only one instance of the GList class is created and it is redrawn and rewritten as needed.
3. Releasing resources once they are used. This includes setting objects to null and calling the system garbage collector (System.gc()) when necessary, allowing memory to be freed up for new operations. For example, the data storage component, which relies heavily on byte arrays, takes great pains to release memory once saving and retrieving methods have completed. The Wireless Extension also makes sure to release its network connections as soon as the response to its data transmission is complete.

Several other techniques for optimization had to be intentionally disregarded as a consequence of the development principle of portability. These included:

1. Use built-in classes where possible. For example, a truly optimized application would have made use of the standard List, TextBox, TextField, etc., classes, instead of the special GList, GTextBox, etc. Specialized classes “cost” more in memory. However, for the NIS, it was a better choice to encapsulate all the methods and members. Using the built-in classes would have meant placing all the methods in the main midlet class, resulting in one oversized class that contained code for many separate conceptual classes. Adhering to this technique would have brought better performance at the expense of unmanageable class files.
2. Code for specifics rather than in abstractions. This is the concept of “hard-coding” in all the settings of the application. For instance, in J2ME it would be faster to have a condition (i.e., dizziness) represented by a simple StringItem than a specialized class called Condition, which had a String as a member. The Condition class has more overhead than a simple StringItem. However, hard-coding means abandoning all flexibility; any changes in the number, type, and behavior of conditions would have to be changed in each and every instance throughout the program. Maintenance of such a program is extremely difficult. Changing the behavior of the Condition class allows one change to propagate out to the hundreds of conditions that exist.

In both cases, tradeoffs in performance were made for to gain efficiencies in design, development, and maintenance. These efficiencies also have the benefit of preserving, as much as possible, the ability to move components to new applications. Striking the balance between performance and portability is difficult. The NIS application optimizes performance where it can, but not at the expense of eliminating the future reuse of the application components.

Code Reuse in NIS and NISPC

The motto of Java is “write once, run everywhere.” However, this was made more difficult when Java split into three versions (Micro Edition(J2ME), Standard Edition(J2SE), and Enterprise Edition (J2EE)). Most J2EE technologies and many J2SE can not run on J2ME. However, much of what works on J2ME will scale up and work on other versions of Java, with the exception of the specific classes -- most notably the `microedition.lcdui` and `microedition.rms` packages.

Component	NIS Palm for Palm OS	NISPC
GUI	8 classes, 1.7 KLOC	No reuse possible. 4 classes, .66 KLOC
BOOP	10 classes, 2.7 KLOC	100% reuse. 10 classes, 2.7 KLOC
DataStorage	7 classes, 2.1 KLOC	89.1% reuse. 4 classes, 1.0 KLOC
TOTAL	25 classes, 6.5 KLOC	85.6% reuse. 18 classes, 4.3 KLOC

As the chart shows (figures for prototypes as of 3/18/02), component reuse was high for the NISPC application. We were able to completely reuse the business logic on the PC-side, which is logical, since this component involved neither presentation nor persistence, both of which have J2ME specific packages and therefore are difficult if not impossible to reuse. Reuse of the datastorage component was achieved because many of the datastorage classes were designed to interact with a wrapper class that went around the version-specific recordstore package. By encapsulating the version-specific classes, it was possible to then refactor the class so that it would use the normal `java.io.*` classes and read from a text file. However, even when GUI classes could not be copied over from the Palm, the significant concepts were reused on the PC side. The Report displayed on the PC application mimics the presentation of the Report on the Palm, and the PC classes duplicate the methods found in the GReport class.

Clearly, there is a benefit to using Java on the Palm and Java on the PC. Had Palm-specific code been used, for instance, it is highly unlikely that that code would have been of any use on the PC side. Instead, a high level of reuse was achieved and conceptually the two versions work the same way. Given this success, it is also highly probable that many of the classes could be ported to another environment, such as a webserver, for enterprise scale applications.

Conclusion and the Future of NIS

Porting the NIS application to Java 2 Micro Edition was not only possible, it was beneficial. The NIS application is now liberated from a single proprietary device and can be installed on any number of J2ME-enabled devices. Furthermore, the use of Java, plus a balancing of performance and portability, makes many of the components of the NIS portable to future applications. New types and versions of assessments for nurses and even other healthcare professionals can be readily accommodated by the current classes of the NIS. The datastorage component has the greatest potential reuse, since it maintains the greatest degree of independence from the specific nature of the application.

Performance gains could be achieved by obfuscating the code - a process which reduces the *.jar files to the absolute minimum necessary for operation. Smaller *.jar files would make more memory available for program operation. Another performance improvement could be achieved by deploying the application to other third-party Java Virtual Machines (JVMs). Zucutto (Whiteboard), Esmertec (Jbed), IBM (J9), Kada Systems, aJile Technologies, and many other companies offer JVMs that work on a broader range of devices with better performance and, in some cases, more features. This is an example of another of Java's strengths -- because multiple vendors can produce JVMs, competition forces innovation and improvement in JVM performance.

Finally, NIS users would benefit from installation utilities, which take care of the installation of the JVM/JRE on the PC side, as well as automated installation and registration of the conduit. These utilities are readily available but are beyond the financial means of the team.

References

Feng, Yu and Jun Zhu. *Wireless Java Programming with Java 2 Micro Edition*, Indianapolis, Indiana: Sams Publishing, 2001.

Muchow, John W. *Core J2ME Technology & MIDP*, Upper Saddle River, NJ: Prentice Hall, 2002.

Riggs, Roger, Antero Taivalsaari, Mark VandenBrink. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*, Upper Saddle River, NJ: Addison-Wesley, 2001.

Walrath, Kathy and Mary Campione. *The JFC Swing Tutorial, A Guide to Constructing GUIs*, Reading, Massachusetts: Addison-Wesley, 1999.

Web sites:

<http://fitne.net>

<http://java.sun.com>