

THE CREATION OF CARGO SCANNER SOFTWARE TO IMPROVE THE
CONTAINER PACKING PROCESS

By

Jonathan Berkey Adams

B.S., Liberty University, Lynchburg, Virginia, 2005

Professional Paper

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

Spring 2007

Approved by:

Dr. David A. Strobel, Dean
Graduate School

Dr. Joel E. Henry, Chair
Department of Computer Science

Dr. Yolanda Jacobs Reimer
Department of Computer Science

Dr. George McRae
Department of Mathematical Sciences

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS	iv
CHAPTER 1 THE NEED FOR A SOFTWARE SOLUTION	1
Introduction.....	1
The Container Packing Problem.....	1
Background.....	2
The Problem.....	3
The Solution.....	4
CHAPTER 2 SOFTWARE REQUIREMENTS	7
Container Selection.....	7
Scanning Cargo Items.....	10
The Packing Process.....	15
CHAPTER 3 SOFTWARE DESIGN	26
Overview.....	26
The frmContainerType Form.....	28
The frmMain Form.....	30
The frmCargoManifest Form.....	34
The XMLReader Class.....	37
The XMLWriter Class.....	39
CHAPTER 4 IMPLEMENTATION	41
CHAPTER 5 SOFTWARE TESTING	43
CHAPTER 6 SOFTWARE DEPLOYMENT	46

CHAPTER 7 PROJECT ASSESSMENT	47
Assessment.....	47
Future Direction.....	48
REFERENCES	52
APPENDIX A	53
APPENDIX B	54

LIST OF ILLUSTRATIONS

Figure 1.1 Solution Process.....	6
Figure 2.1 Container Selection Window.....	7
Figure 2.2 Container Selection Error.....	8
Figure 2.3 Selecting Containers.....	9
Figure 2.4 Completed Container Selection.....	9
Figure 2.5 Main Window.....	11
Figure 2.6 Scanned Cargo.....	11
Figure 2.7 Cargo Item Not Found.....	12
Figure 2.8 Removing Cargo.....	13
Figure 2.9 Clearing the List.....	14
Figure 2.10 Clear List Warning.....	14
Figure 2.11 Server Connection Error.....	16
Figure 2.12 Manifest Window.....	19
Figure 2.13 Side Scrolling the Manifest Window.....	20
Figure 2.14 Sorting the Manifest List.....	21
Figure 2.15 Removing Item from Manifest List.....	22
Figure 2.16 Exit Cargo Scanner.....	23
Figure 2.17 Packing Process (Part 1).....	24
Figure 2.18 Packing Process (Part 2).....	25
Figure 3.1 Class Interaction.....	27
Figure 3.2 The frmContainerType Form.....	28
Figure 3.3 The frmMain Form.....	30
Figure 3.4 The frmCargoManifest Form.....	34
Figure 3.5 The XMLReader Class.....	37
Figure 3.6 The XMLWriter Class.....	39

CHAPTER 1

THE NEED FOR A SOFTWARE SOLUTION

Introduction

This paper will cover the work performed to create computer software that will help simplify the current packing process employed when using the Container Packer software created by Dr. Joel Henry of the University of Montana. This paper is organized as follows: Introduction to the container packing problem, Overview and background of the Container Packer software, and the problem solved. Since a majority of the work consisted of writing computer software, the software will be covered in great detail. First, the requirements for the software will be discussed, followed by the design, implementation, testing, and deployment of the software. The paper will conclude with an assessment of the work and future directions or enhancements that may be made to the software.

The Container Packing Problem

A common problem in computer science that many people try to solve is the Container Packing Problem. In this problem, N cargo items must be packed into a container type. A container has a weight limit and this weight limit cannot be exceeded. Each cargo item has a length, width, height, and weight. No item within a container may overlap the space occupied by another.

The optimal solution to this problem would be a packing order of the cargo items that results in the minimal number of containers of a single type being used. To find this optimal packing order, all possible combinations of cargo items must be attempted in a

container. This means that $N!$ packing combinations will be tested. This is known to be an NP-Hard problem. The problem can not be solved in polynomial time and as the problem size (N) increases, the time to solve the problem will increase by some exponential function of the size.

Background

Several years ago, Dr. Joel Henry was tasked under a research and development contract with Development, Planning, Research, and Analysis (DPRA) to create a computer program that, when given a list of cargo items and a list of containers, would approximate the number of containers required to pack the items and determine a packing order for the cargo items.

The Container Packer program accepts two files as input. One is a list of cargo items to be packed. Each cargo item has a Unit ID (a way of grouping the cargo items that should be packed together), Cargo ID (allows each cargo item to be uniquely identified), length, width, height, and weight. The other file is a list of all the different types of containers into which the cargo items may be packed. Each container in the list has a name, type, a Boolean value indicating if there is a maximum packing height, a Boolean value indicating if there is a maximum packing length, dimensions of the container in both metric and imperial measurement units, and a maximum weight that it can hold.

The program then runs a packing algorithm using the input files to determine a packing order. Since this problem is considered to be NP-Hard, the program uses heuristics to approximate the best solution. First, all the cargo data is stored in data

objects. The cargo information is only stored one time. The cargo objects are then separated into groups based on the Unit ID and organized by length, width, and height.

The cargo items are then packed into containers using a best-fit algorithm. Packing always begins at the back, left, bottom corner of an empty container. Initially, there is only one space to pack, the entire container. As items are placed in a container, new spaces are created beside, in front of, and on top of cargo items. The largest empty space within a container is always packed first. The cargo item consuming the largest volume within that space is then packed. No backtracking is used in this algorithm. This means cargo items are only packed once. They are never removed to find a more efficient packing scheme. Not all cargo items are considered for each space to be packed. Only items that will fit in the space are taken into account. This increases the speed of the algorithm execution. Because of the complex data structures that are used to store cargo and space data, the algorithm maintains a $O(1)$ retrieval of any cargo or free space data, regardless of the number of items being packed. The algorithm also packs containers to their maximum weight limit 95% of the time.

Once a cargo item has been successfully packed into a container, the container number and the position that the cargo item is to be placed within the container is stored. After all cargo items have been packed, each cargo item along with its container number and position data is written to an output file. Then a list of the containers to be packed along with the container size and weight after packing is written to a file. These two files are used to perform the physical packing of the cargo.

The Problem

When used commercially, the current packing process requires several steps.

First, a person creates an XML file of each cargo item to be packed. This file contains each cargo item's Unit ID, Cargo ID, length, width, height, and weight. This list is created through keyboard entry. Then, a second XML file is created that contains a list of the containers, which may be used to pack each item. This list is also created through keyboard entry. The person that created the lists then runs the Container Packer software. The Container Packer creates a final XML packing list that contains the cargo items and the containers that each item will be packed into. The computer that runs the Cargo Packer software resides in a different location than where the physical packing takes place, so the list must be printed out and given to the person who actually places each item in the containers.

Problems are encountered when this program is used in the real world: the data entry process is tedious and time consuming and two people who work in different locations are required to complete the process.

The Solution

The purpose of this project is to streamline the current packing process. This can be done by allowing a single person to create the packing list, run the program, view the final packing list, and physically pack the cargo items all from a single location. This can be done by providing the physical packer with a handheld computing device, such as a PocketPC, that has both barcode scanning and wireless LAN technology.

For this project, software was created to run on a PocketPC with barcode scanning and wireless LAN technology. This software allows a single person to scan cargo items with the PocketPC. When the items are scanned, an XML list of the cargo items is created. The software will also allow this list to be wirelessly transmitted to a server,

which runs the software that performs the packing algorithm. The PocketPC software will wait for the final packing list to be sent back from the server. Upon receiving the final packing list, the PocketPC will then display the final packing list so that physical packing may begin.

The new packing process will require fewer steps. First, each cargo item will have a barcode placed on it. The physical packer will use the PocketPC to scan each cargo item in order to create the packing list. Then, the packer will also select from a list the container type in which to pack the items into. Upon completion of these tasks, the PocketPC will then wirelessly transmit the packing list to a central server which runs the software that performs the packing algorithm. The server will then transmit the final packing list back to the PocketPC so that the physical packer may view the list and begin packing each cargo item in the containers. **Figure 1.1** displays this process.

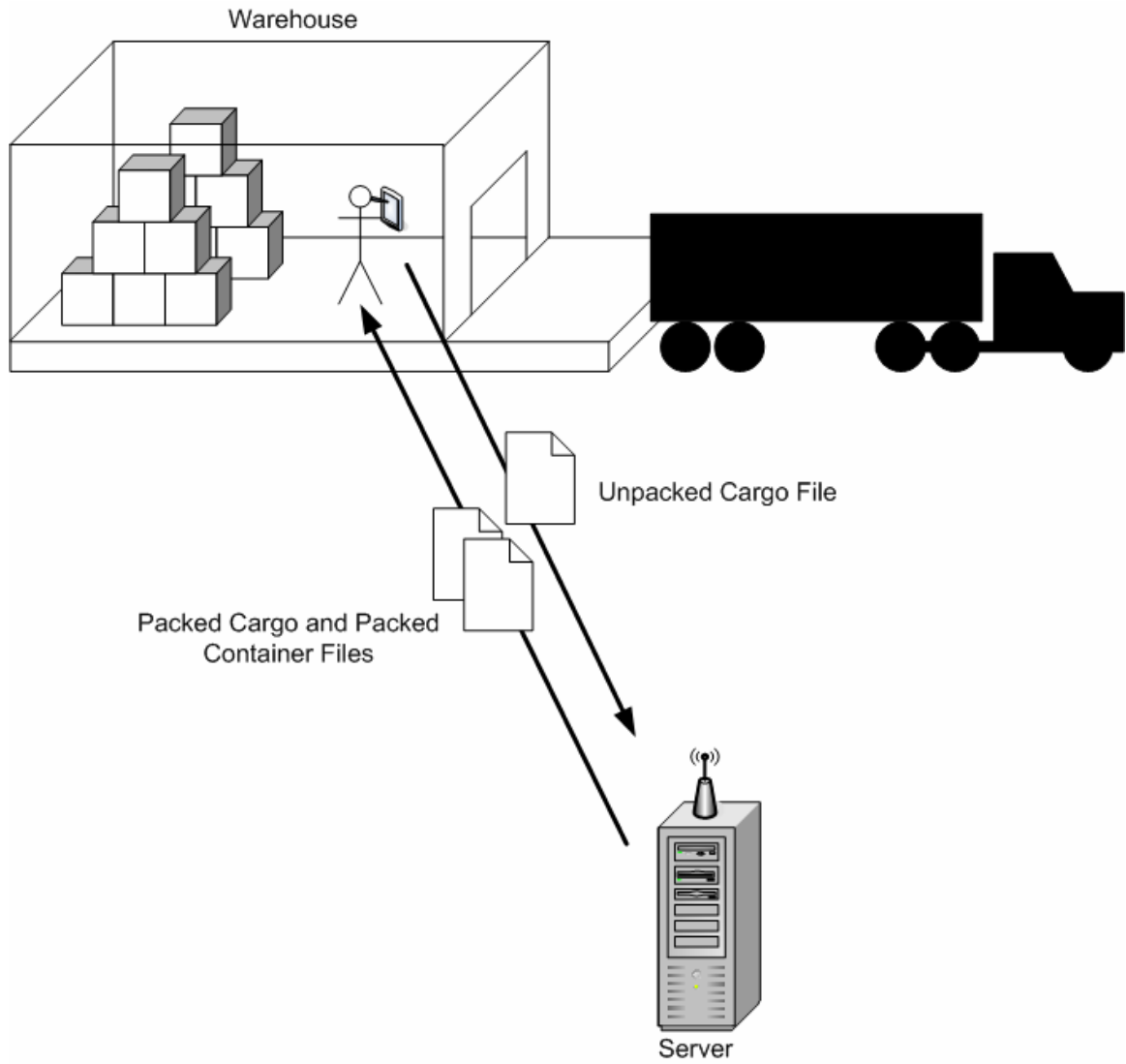


Figure 1.1 - Solution Process

CHAPTER 2

SOFTWARE REQUIREMENTS

Container Selection

When the user launches the Cargo Scanner program, they are presented with the Container Selection window (**Figure 2.1**). This window allows the user to select the type of container in which to pack the cargo items into and the units of measurement that will be used. The window will not allow the user to advance unless the container type and measurement units have been selected (**Figure 2.2**).

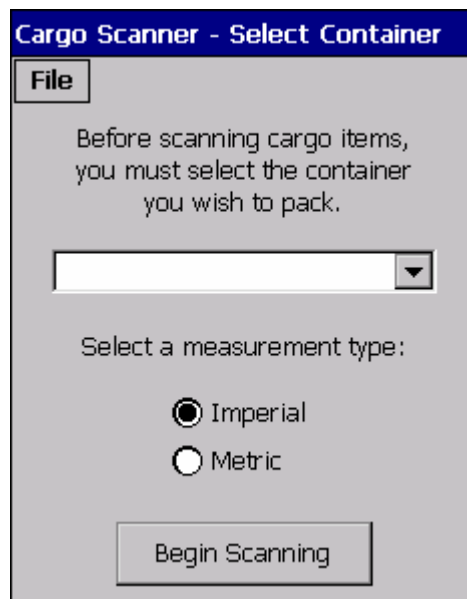


Figure 2.1 - Container Selection Window

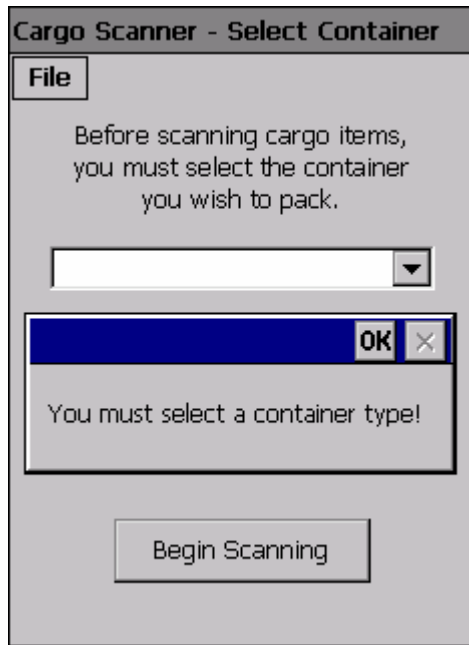


Figure 2.2 - Container Selection Error

The drop-down list contains all the different types of containers that may be selected for packing. Only one container type may be selected at a time. This list is a static list. If a new container type is to be added, the code must be edited appropriately, recompiled, and reinstalled on the PocketPC.

The measurement type is selected by pressing on the appropriate radio button. When the window is first loaded, Imperial is selected by default. Only one measurement type can be selected at a time. If Imperial is currently the selected measurement type and the user presses on the radio button for Metric, Imperial is deselected and Metric becomes the newly selected measurement type.

When the arrow of the container drop-down list is pressed, the user is presented with the entire list of available containers (**Figure 2.3**). Then, the user may select one of the containers from the list for packing (**Figure 2.4**). Once a container type has been

selected, the user may change the measurement type or press the “Begin Scanning” button.

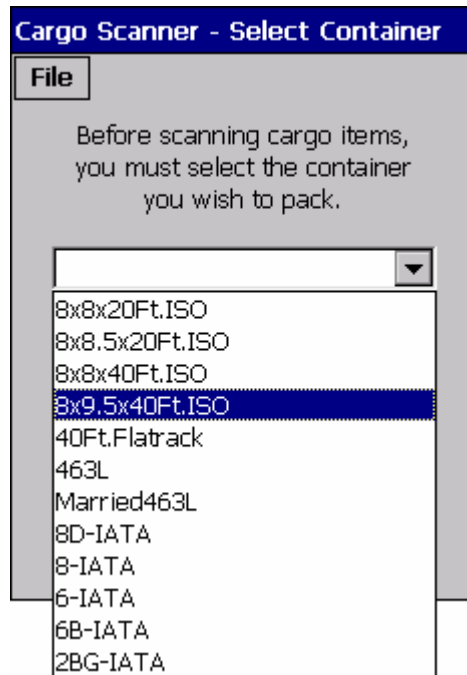


Figure 2.3 - Selecting Containers

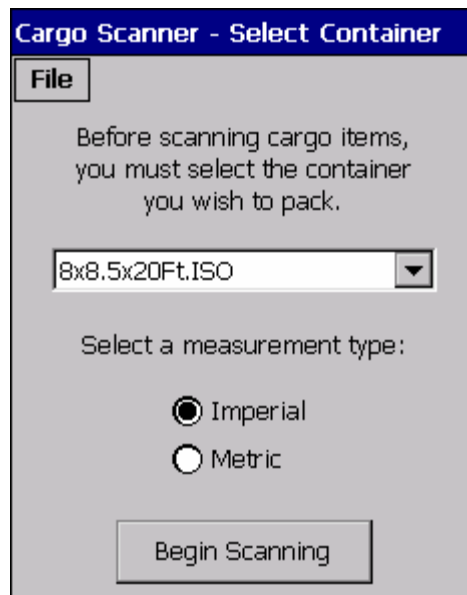


Figure 2.4 - Completed Container Selection

Scanning Cargo Items

After the user presses the “Begin Scanning” button on the Container Selection window, the container type and measurement units are saved, the Main window for the Cargo Scanner is opened (**Figure 2.5**) and the barcode scanner is enabled. When the barcode scanner is enabled, the barcode scanner laser can be used to scan cargo items. The barcode scanner laser is turned on by holding down either of the yellow, rubber buttons on the sides of the PocketPC.

The barcode scanner laser will shut off under one of three conditions: an item was scanned, the button was held down for more than three seconds, or the user released the button.

In order to begin creating a list of cargo items for packing, all cargo items must have barcodes that contain the unique Cargo ID of that particular cargo item. There must also be an XML file on the PocketPC that contains a list of all the cargo items and the physical properties of the cargo items. This is the Cargo Lookup file. See **Appendix A** for a sample of this file.

When a cargo item is scanned, the Cargo ID of that cargo item is stored. The Cargo Lookup file is opened and a sequential search for the scanned item is performed using the Cargo ID. When the cargo item is found in the list, its length, width, height, and maximum weight are extracted from the file and loaded into a data structure that is displayed in the Main window (**Figure 2.6**).



Figure 2.5 - Main Window



Figure 2.6 - Scanned Cargo

The status message at the bottom of the list is updated if a Cargo ID is not found in the Cargo Lookup file. The updated message informs the user that the cargo item does not exist in the Cargo Lookup file (**Figure 2.7**).

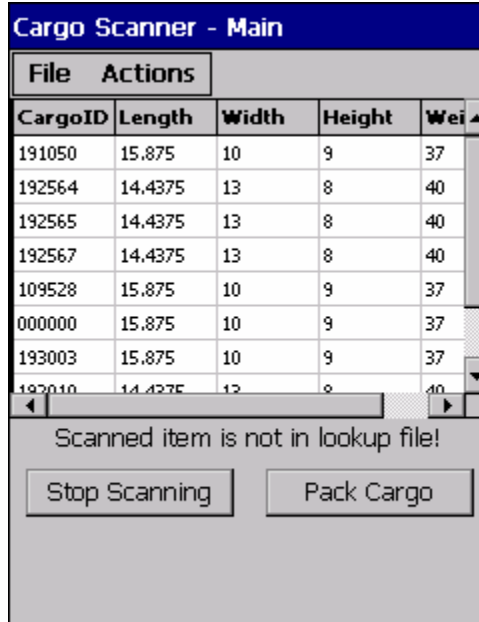


Figure 2.7 - Cargo Item Not Found

Cargo items may be deleted from the list at any time. In order to delete a particular cargo item from the list, the user may press on any cell within the row of the desired item to delete. Upon pressing, a context menu appears with the menu item **Delete Cargo Item (Figure 2.8)**. When this menu item is selected, the selected cargo item will be deleted from the list.

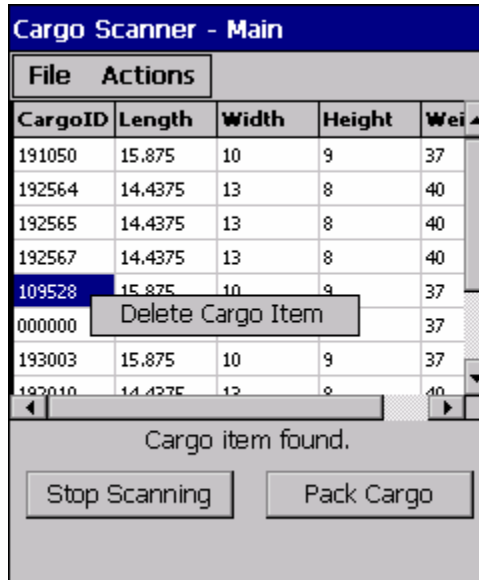


Figure 2.8 - Removing Cargo

The entire list of cargo items may be deleted at any time by selecting **Actions** from the main menu, and then selecting **Clear List** (Figure 2.9). If **Clear List** is selected from the **Actions** menu, a dialog box is displayed with a warning message giving the user a chance to cancel this action (Figure 2.10). If the user selects **No**, the action is cancelled and the user can continue scanning cargo. If **Yes** is selected, all cargo items in the list are deleted, the Main window is refreshed with a blank list, and the barcode scanner is enabled.

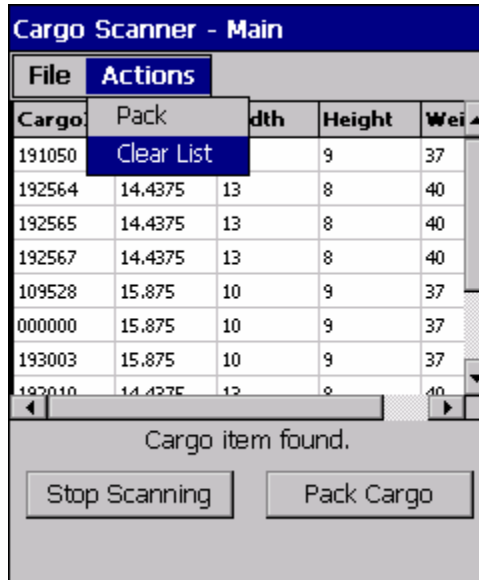


Figure 2.9 - Clearing the List

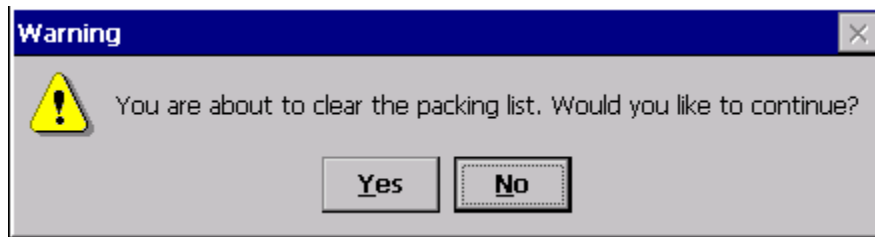


Figure 2.10 - Clear List Warning

The barcode scanner may be disabled at any time during the scanning process by pressing the **Stop Scanning** button. When the user presses this button, the barcode scanner is disabled and any memory or data structures associated with it are released. Also, the text on the button changes to “Start Scanning.” Disabling the barcode scanner turns off the ability to activate the barcode scanner laser. When disabled, pressing the yellow, rubber buttons on either side of the PocketPC does nothing.

The barcode scanner can be enabled again by pressing what is now the **Start Scanning** button. This activates the barcode scanner and allocates any resources that are required by the barcode scanner hardware.

Once all the desired cargo items have been scanned and added to the list, packing can begin by either pressing the **Pack Cargo** button or selecting **Pack** from the **Actions** menu (**Figure 2.9**).

The Packing Process

There are several steps to the packing process. Since the Container Packer software requires significant memory and processing power, it cannot run efficiently on a PocketPC. The Container Packer software runs on a server connected to a wireless router. So, the first step of the packing process for the PocketPC is to establish a connection with server. The connection is established using a Transmission Control Protocol (TCP) Socket. A connection cannot be made unless the server is ready.

Geddy Tarbell, another student working on an independent study, created a Windows Service which runs on the server that acts as an intermediary between the PocketPC and the Container Packer software. A Windows Service is a program that runs in the background of the computer as long as it is turned on. The Container Packer Service or CPS, when first started, creates a socket that will accept connections from any IP address on port 48888. The CPS performs a blocking wait which means program execution stops until a connection attempt is made on port 48888.

When the user presses the **Pack Cargo** button, a socket is created and attempts to make a connection to the server. If a connection was successfully established, the Cargo Scanner moves on to the next step of the packing process. If a connection with the server

could not be made, then an error message is displayed and control of the program execution is returned to the Main window (**Figure 2.11**).

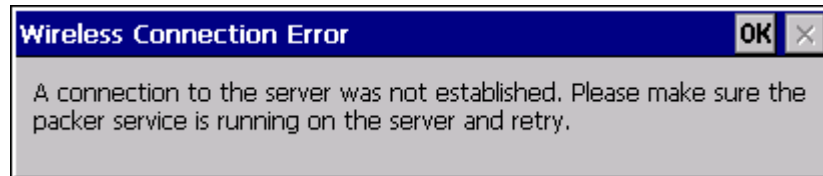


Figure 2.11 - Server Connection Error

The second step of the packing process is to create a cargo item XML file containing the container type, measurement units, and all the scanned cargo items with their physical properties. This XML file follows the specifications defined by Henry (Henry, 4). First, the XML header is written to the file. Then, the container type and measurement units are added. Next, each scanned cargo item is taken from the list and each item's Cargo ID, length, width, height, and weight are added to the file. Once all of the cargo items have been written to the XML file, the closing XML tags are inserted and the file is closed. A sample cargo item file can be viewed in **Appendix B**.

The third step is to transfer the cargo item file from the PocketPC to the CPS on the server. To perform this, the Cargo Scanner reads through the cargo item file line by line and places each line into an array of strings. This array grows as needed. Initially, the array can hold 20 string objects. When the array becomes full, a function is called that increases the capacity of the array by another 20 elements.

After the file has been completely stored in the array, the PocketPC then breaks the array into bytes and begins transmitting the bytes through the TCP socket that was created in Step 1. Since TCP is the underlying protocol for network communication, if

any data gets lost or corrupted during transmission, the data is retransmitted as needed without any extra programming. Once the entire array has been broken into bytes and successfully transmitted across the network to the server, the PocketPC completes the transmission by sending an “end of file message”. This message lets the CPS on the server know that it has received the entire file and may begin using that file with the Container Packer software. The Cargo Scanner then begins waiting for the CPS to perform the packing and send back two files.

Step four consists of the CPS running the Container Packer software using the aforementioned cargo item file. This results in the creation of two files on the server, a Cargo Manifest file and a Container Manifest file. The Cargo Manifest contains each cargo item, the container number that that item will be packed in, the X, Y, and Z coordinates within the container that the back, left corner of the item will be placed, and packing index of that item. The packing index is the order in which items should be placed into a container.

The Container Manifest contains information for all of the containers required to pack all of the items from the Cargo Manifest file. This file contains the following information for each container: container name, container number, packed weight of the container, height of the container if it has an open top, length of the container if it has open sides, and the width of the container.

In step 5, the CPS sends both the Cargo Manifest and the Container Manifest to the Cargo Scanner. Since the Cargo Scanner still has an open network connection with the CPS and is ready to accept files, the CPS can immediately begin file transmission. The Cargo Manifest file is broken into an array of strings, each element in the array is

broken into bytes, and the bytes are placed in data packets and transmitted across the network. After it has completed sending the file, it sends an end of file message to the Cargo Scanner and begins sending the Container Manifest in the same fashion it sent the Cargo Manifest. When it finishes sending the Container Manifest, it sends another end of file message.

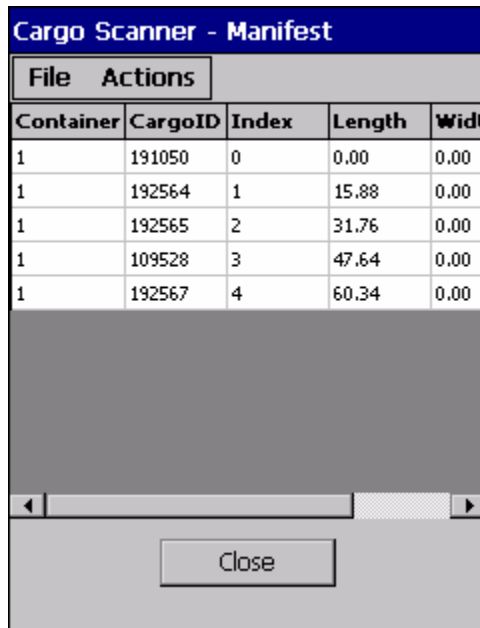
Once the Cargo Scanner begins receiving the packets from the CPS, it begins reassembling the Cargo Manifest file. This is a delicate task. Each packet may contain multiple lines or only pieces of a line of the file. This is extremely difficult to detect. In order to maintain the original format of the file, the data from the incoming packets is stored as one large string. As an incoming packet is received, the data from that packet is added to the string. This can result in string objects consisting of thousands of characters.

The Cargo Scanner checks for the end of file message in every packet. When it finds the end of file message, it adds any preceding file data to the current string, and writes the entire string to the Cargo Manifest file. The end of file message may also contain some data for the Container Manifest file. If this is the case, that data is stored in a new string that is specifically for the Container Manifest. Since the end of file message was received, all of the incoming data will now be strictly for the Container Manifest. The Container Manifest file is created using the same methodology as the Cargo Manifest. Every incoming packet is checked for the end of file message. If the message is encountered, the string is written to the Container Manifest file. After the end of file message has been received by the Cargo Scanner, no other communication between the PocketPC and the server is necessary so the socket is closed. The Cargo Scanner

proceeds to step 6 and the CPS goes back to a state where it waits for an incoming connection.

Even though the Cargo Scanner receives the Container Manifest file from the CPS, it never uses the file. This file is sent to the Cargo Scanner so it is there for use if the Cargo Scanner software is expanded.

In step 6, the Cargo Scanner opens the Manifest window and begins reading the Cargo Manifest file. Each cargo item is stored in a data object. Each object contains all the cargo items properties that were specified in Step 4. After all the cargo items have been stored in objects, a list in the Manifest window is populated with each object (Figure 2.12). A diagram of the entire packing process is provided in Figure 2.17 and Figure 2.18.



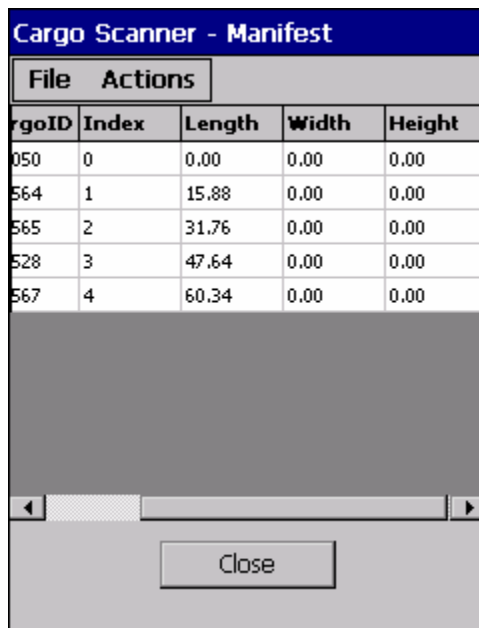
The screenshot shows a window titled "Cargo Scanner - Manifest". It contains a table with the following data:

Container	CargoID	Index	Length	Width
1	191050	0	0.00	0.00
1	192564	1	15.88	0.00
1	192565	2	31.76	0.00
1	109528	3	47.64	0.00
1	192567	4	60.34	0.00

Below the table is a horizontal scrollbar and a "Close" button.

Figure 2.12 - Manifest Window

Due to the width of the columns, the list is wide and the horizontal scrolling arrows must be used to view the entire contents of the list (**Figure 2.13**). The size of the columns may be changed during program execution. To change the size of any column, the user may press and hold the column header border and drag it to the desired location. If more items exist than can be viewed on the screen, vertical scrolling arrows are added so that the other items may be seen.



Cargo Scanner - Manifest				
File		Actions		
CargoID	Index	Length	Width	Height
050	0	0.00	0.00	0.00
564	1	15.88	0.00	0.00
565	2	31.76	0.00	0.00
528	3	47.64	0.00	0.00
567	4	60.34	0.00	0.00

Figure 2.13 - Side Scrolling the Manifest Window

By default the items in the list are ordered by packing index. The list may be sorted based on any property of the cargo items. To sort the manifest list, the user may press on the **Actions** menu item and then choose **Sort by...**. A list of all the available sorting options will be displayed. The user can then press the desired sorting property and the list will then update with the appropriate sorting (**Figure 2.14**).

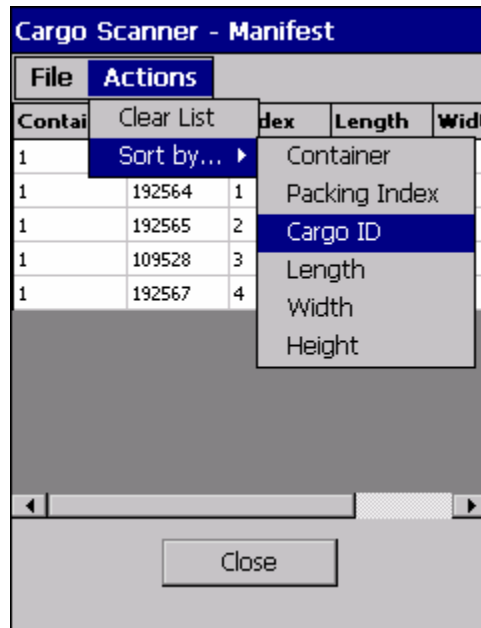


Figure 2.14 - Sorting the Manifest List

As items are packed into their appropriate containers, they can be quickly removed from the manifest list. To remove a cargo item from the manifest list, the user may just press any cell within the row of the desired item to remove and a context menu will appear. This menu provides the option to remove the item. Once the user presses on **Delete Cargo Item (Figure 2.15)**, that cargo item is removed from the list and the user may continue packing the next item.

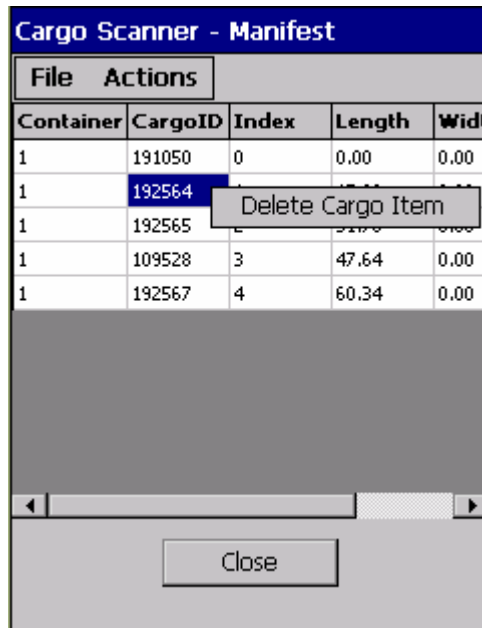


Figure 2.15 - Removing Item from Manifest List

When all items have been packed into the appropriate containers, the user may either exit the Cargo Scanner software or begin packing new containers. To pack a new container, the user may click on the **Close** button on the Manifest window. This will close the current manifest window and any cargo items remaining in the list will be lost. A new Container Selection window will then be opened and the user may proceed with the container and measurement unit selection.

To exit the Cargo Scanner software, the user may press the **File** menu option and then select **Exit**. The program can be shut down in this manner from any window within the Cargo Scanner software (**Figure 2.16**).

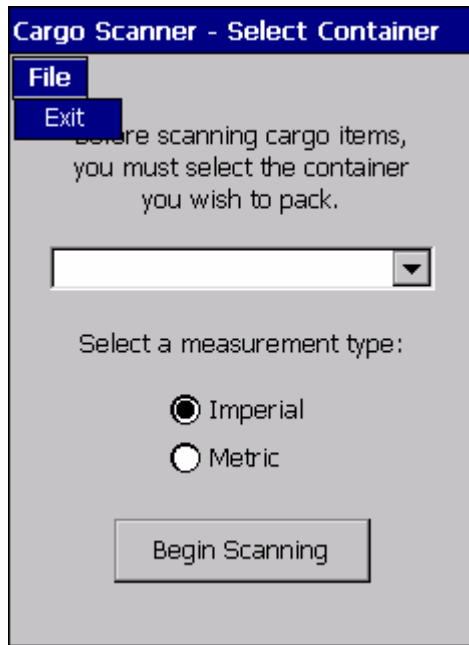


Figure 2.16 - Exit Cargo Scanner

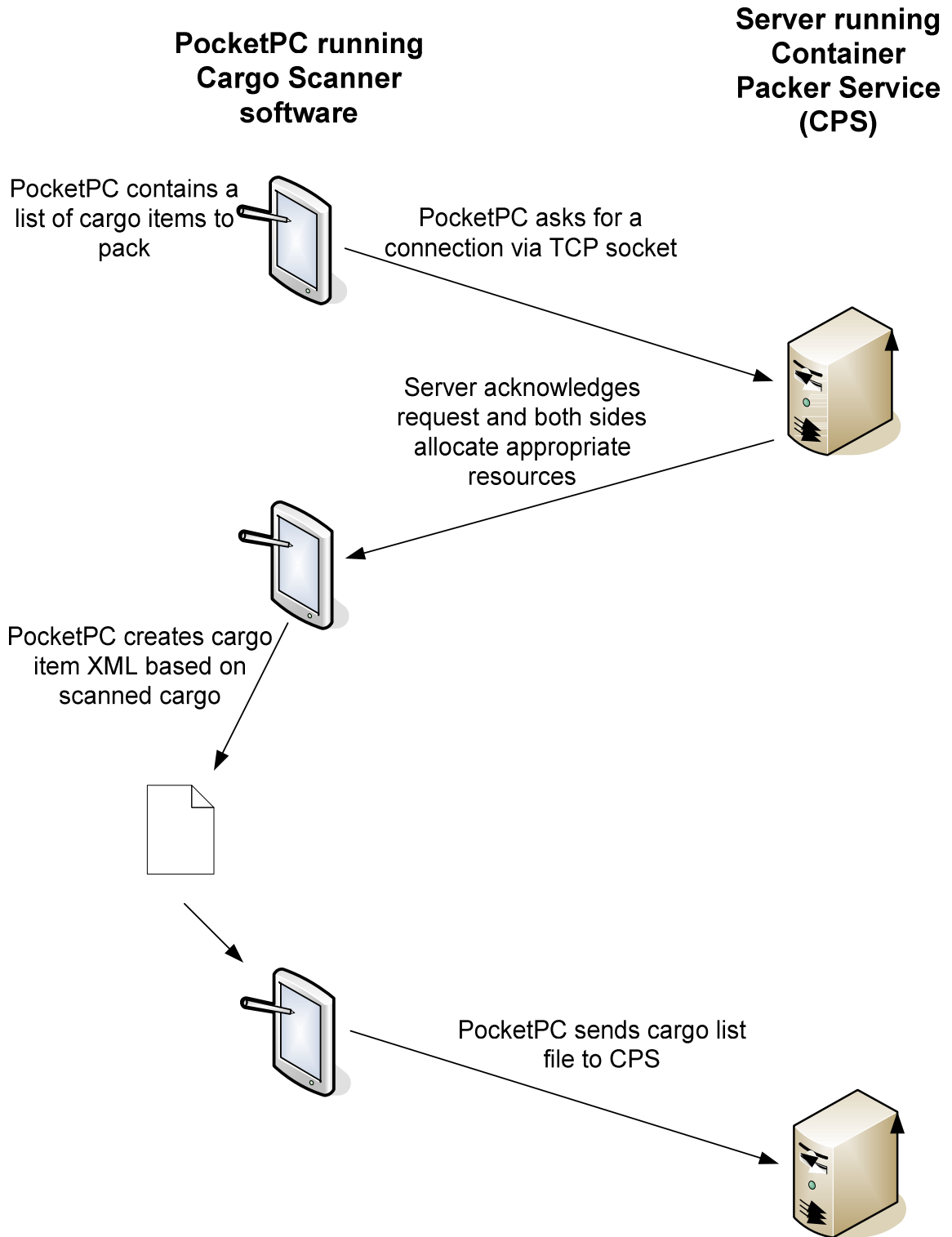


Figure 2.17 - Packing Process (Part 1)

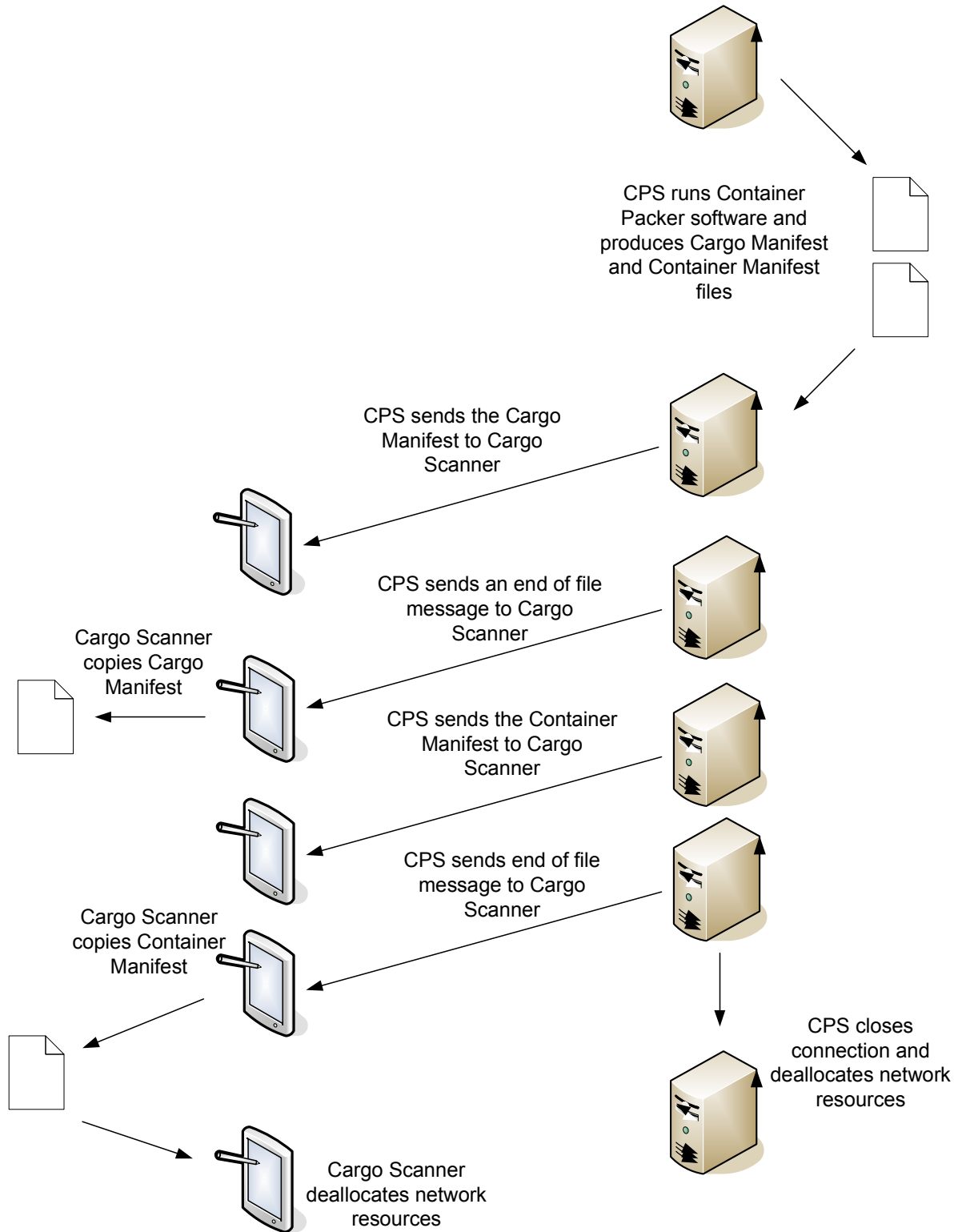


Figure 2.18 - Packing Process (Part 2)

CHAPTER 3

SOFTWARE DESIGN

Overview

The design of the Cargo Scanner is very simple. A total of three forms and two classes were created to carry out the cargo scanning process. **Figure 3.1** displays the interactions between the forms and classes. When the user starts the Cargo Scanner software, the frmContainerType form is displayed. See **Figure 3.2** for a list of properties and methods for this form. After the user finishes entering the container type and measurement units, the frmMain form (**Figure 3.3**) is instantiated. This form has two properties: strContainerType and strMeasurementType. These two properties are set by the frmContainerType when the frmMain is instantiated. This is done because the frmMain form must know the container type and measurement units so that it may be written to an XML file. From the frmMain form, the user scans the appropriate cargo items. Every time an item is scanned, frmMain instantiates the XMLReader class (**Figure 3.5**) in order to find the cargo item data in the Cargo Lookup XML file.

Once the user is ready to pack the cargo items, the XMLWriter class (**Figure 3.6**) is instantiated. This class is used to write the container type, measurement units, and all cargo items to an XML file. After the XML file has been created, frmMain makes a connection with the CPS running on the server and sends the XML file to it. It then waits for the manifest files to be sent back.

Once frmMain receives the Cargo Manifest and Container Manifest files from the CPS, the frmCargoManifest form (**Figure 3.4**) is instantiated and loaded. This form handles the display of the manifest files. When the form loads, it opens the Cargo

Manifest file and begins reading each cargo item into a data object. After the cargo items are loaded into a data object, they are then displayed on the frmCargoManifest form.

The software design is based on knowledge of the desired logical packing process. Only two classes were used because there are only two points within the software execution that the data is operated upon: when it is being read from a file and when it is being written to a file. The XMLReader and XMLWriter classes handle these two operations.

The three forms are designed to handle the three different sets of data. The forms, frmContainerType and frmMain handle the input of data. The frmContainerType form handles the input of the container type and measurement units while the frmMain form handles the input of cargo items. The frmCargoManifest form is designed to handle the display of the resulting Cargo Manifest file.

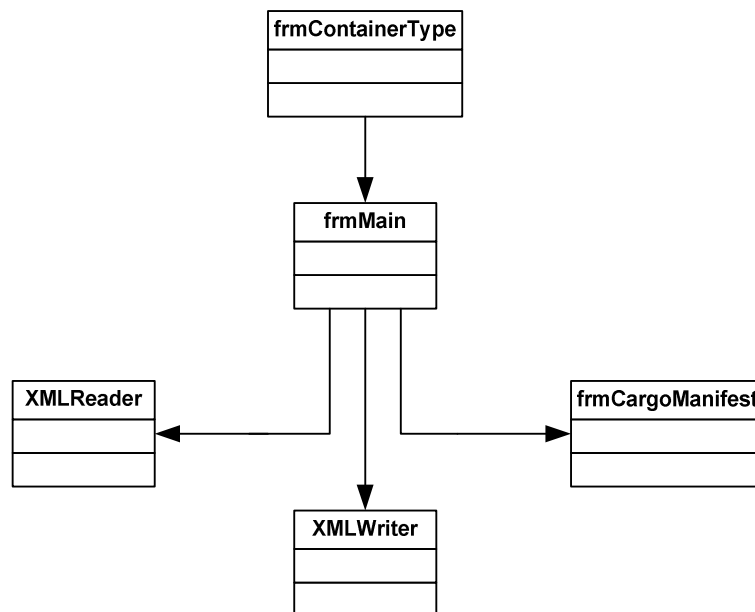


Figure 3.1 - Class Interaction

The frmContainerType Form

frmContainerType
-cmbContainerType : ComboBox -rbImperial : RadioButton -rbMetric : RadioButton -btnReturnToMain : Button -mmuMain : MainMenu -mmuFile : MenuItem -mmuExit : MenuItem -lblInstructions1 : Label -lblInstructions2 : Label -blnImperialChecked : bool = true
-Main() -frmContainerType_Load(in sender : object, in e : EventArgs) -rbMetric_CheckedChanged(in sender : object, in e : EventArgs) -rbImperial_CheckedChanged(in sender : object, in e : EventArgs) -mmuExit_Click(in sender : object, in e : EventArgs) -btnReturnToMain_Click(in sender : object, in e : EventArgs)

Figure 3.2 – The frmContainerType Form

(See Figure 2.1 for a screenshot of this form)

The frmContainerType form has six methods that are used during program execution. The Main method is the main entry point of the Cargo Scanner software and it makes certain that the frmContainerType form is displayed.

The frmContainerType_Load method is automatically called when the frmContainerType form is displayed. Currently, it does nothing. This is where code would be inserted if the cmbContainerType ComboBox were to be a dynamic list.

The rbMetric_CheckedChanged and rbImperial_CheckedChanged methods are called anytime one of the radio buttons on the frmContainerType form is selected. These methods make certain only one measurement unit (radio button) is selected at a time.

The mmuExit_Click method handles the event of a user selecting the **Exit** option from the **File** menu. This closes any open forms and ensures the scanner is turned off.

The btnReturnToMain_Click method is called when the user clicks the **Begin Scanning** button on this form. It creates an instance of the frmMain form and sets the strContainerType and strMeasurementType properties of frmMain. Once these properties have been set, the frmMain form is displayed.

The frmMain Form

frmMain
<pre> -btnPack : Button -btnScan : Button +lblMessage : Label -mmuMain : MainMenu -mmuFile : MenuItem -mmuExit : MenuItem -mmuActions : MenuItem -mmuPack : MenuItem -mmuClearList : MenuItem -cmDataGridClick : ContextMenu -miDeleteCargoItem : MenuItem -ofdOpen : OpenFileDialog -sfdSave : SaveFileDialog -dgScannedCargo : DataGrid -bcrBarcodeReader : BarcodeReader -END_CARGO_FILE : string = "END_CARGO_FILE" -END_CONTAINER_FILE : string = "END_CONTAINER_FILE" -CARGO_FILE_FOR_SERVER : string = "CargoToPack.xml" -PACKED_CARGO_FILE : string = "PackedCargo" -PACKED_CONTAINER_FILE : string = "PackedContainer" -FILE_PATH : string = Convert.ToString(@"\Program Files\WirelessCargoScanner_CS\") -strMeasurementType : string -strContainerType : string -blScanning : bool -tcpToServer : TcpClient = new TcpClient() -dtScannedCargo : DataTable = new DataTable("ScannedCargo") -dsScannedCargo : DataSet </pre>
<pre> -frmMain_Load(in sender : object, in e : EventArgs) -btnPack_Click(in sender : object, in e : EventArgs) -btnScan_Click(in sender : object, in e : EventArgs) -bcrBarcodeReader_ListChanged(in sender : object, in e : ListChangedEventArgs) -mmuExit_Click(in sender : object, in e : EventArgs) -mmuPack_Click(in sender : object, in e : EventArgs) -mmuClearList_Click(in sender : object, in e : EventArgs) -dgScannedCargo_Click(in sender : object, in e : EventArgs) -miDeleteCargoItem_Click(in sender : object, in e : EventArgs) -BeginPacking() +CreateCargoXMLFile(in strFileName : string) +GetCargoData(in strCargoIDFromScanner : string) +ConnectToServer() : bool -SendFileToServer() +WaitForFilesFromServer() : string[] +ReadFile(in strFileName : string) : string[] +EnlargeArray(in strarrCurrent : string[]) : string[] -WriteToFile(in strFilename : string, in strData : string) +SetMeasurementType(in strMeasurement : string) +GetMeasurementType() : string +SetContainerType(in strContainer : string) +GetContainerType() : string </pre>

Figure 3.3 – The frmMain Form

(See Figure 2.6 for a screenshot of this form)

The frmMain form is the workhorse of the Cargo Scanner software and therefore has many methods. The frmMain_Load method is called when the frmMain form is first displayed. It ensures the barcode scanner is turned on and it sets up the datagrid to begin displaying the scanned cargo items.

The btnPack_Click method is called when the user presses the **Pack Cargo** button. This method calls the BeginPacking sub procedure as long as at least one cargo item has been scanned.

The btnScan_Click method is called when the user presses the **Stop Scanning** button. This method turns off the barcode scanner and changes the button's label to "Begin Scanning." If the button is pressed again, the barcode scanner is turned on and the button's label is changed back to "Stop Scanning."

The bcrBarcodeReader_ListChanged method is called anytime the barcode scanner scans a barcode. This method calls the GetCargoData sub procedure using the most recently scanned barcode as the lookup value.

The mmuExit_Click method handles the event of a user selecting the **Exit** option from the **File** menu. This closes any open forms and ensures the scanner is turned off.

The mmuPack_Click method handles the event of a user selecting the **Pack** option from the **Actions** menu. This method calls the BeginPacking sub procedure.

The mmuClearList_Click method is executed when a user selects **Clear List** from the **Actions** menu. The method deletes all of the scanned cargo items from the datagrid.

The dgScannedCargo_Click method handles the event of the user selecting one of the scanned cargo items in the datagrid. This method displays the context menu so that the user may delete the currently selected cargo item.

The miDeleteCargoItem_Click method is called when a user selects **Delete Cargo Item** from the context menu that is displayed by dgScannedCargo_Click. This method deletes the currently selected cargo item from the scanned cargo list.

The BeginPacking sub procedure starts the packing process that is displayed in **Figure 17**. This sub procedure calls the CreateCargoXMLFile and ConnectToServer sub procedures. If a connection to the server is successfully created, then it also calls the SendFileToServer and WaitForFilesFromServer sub procedures. Once the WaitForFilesFromServer sub procedure finishes executing, BeginPacking sets the strPackedCargoFile and strPackedContainerFile properties on the frmCargoManifest form and displays it.

The CreateCargoXMLFile sub procedure instantiates the XMLWriter class and uses the XMLWriter methods that are necessary to create the XML file needed by the server.

The GetCargoData sub procedure takes a cargo ID provided to it by the barcode scanner and uses the XMLReader class to look up that cargo ID in an XML file, which contains a list of all the cargo items that may be scanned and their respective attributes. Once it finds the cargo item in the lookup file, it places that information in the datagrid.

The ConnectToServer function attempts to establish a TCP Socket connection with a server running the CPS. If the connection is successful, the function returns a Boolean true, otherwise it returns a Boolean false.

The `SendFileToServer` sub procedure uses the TCP Socket to transmit the scanned cargo file byte by byte. It calls the `ReadFile` sub procedure to place the file into an easy to use format for transmission.

The `WaitForFilesFromServer` function handles the reception of the manifest files from the server. It stores the incoming data from the server in a string. Once the entire file has been received, it calls the `WriteToFile` sub procedure. After successfully reconstructing the both manifest files, it returns the names of both of the files.

The `ReadFile` function takes a file and reads it into an array of strings. If the array is not large enough to hold all the lines of the file, the `EnlargeArray` function is called. Once the entire file has been written to the array, `ReadFile` returns the array of strings.

The `EnlargeArray` function takes an existing array and adds 20 elements to the end of it. After the new elements have been added, it returns the new array and it still contains any data that was previously stored in it.

The `WriteToFile` sub procedure takes a string of characters and writes them to the specified file.

The `SetMeasurementType` sub procedure allows the private `strMeasurementType` property to be set.

The `GetMeasurementType` function returns the value of the private `strMeasurementType` property.

The `SetContainerType` sub procedure allows the private `strContainerType` property to be set.

The `GetContainerType` function returns the value of the private `strContainerType` property.

The frmCargoManifest Form

frmCargoManifest
<pre>-dgPackedCargo : DataGrid -btnClose : Button -mmuMain : MainMenu -mmuFile : MenuItem -mmuExit : MenuItem -mmuActions : MenuItem -mmuClearList : MenuItem -mmuSortBy : MenuItem -mmuCargoID : MenuItem -mmuLength : MenuItem -mmuWidth : MenuItem -mmuHeight : MenuItem -mmuContainer : MenuItem -mmuIndex : MenuItem -cmPackedCargo : ContextMenu -miDeleteCargoItem : MenuItem -dtPackedCargo : DataTable = new DataTable("PackedCargo") -dvCurrentView : DataView -intItemCount : int -strPackedCargoFile : string -strPackedContainerFile : string -FILE_PATH : string = Convert.ToString(@"Program Files\WirelessCargoScanner_CS\")</pre>
<pre>-frmCargoManifest_Load(in sender : object, in e : EventArgs) -btnClose_Click(in sender : object, in e : EventArgs) -mmuExit_Click(in sender : object, in e : EventArgs) -mmuContainer_Click(in sender : object, in e : EventArgs) -mmuIndex_Click(in sender : object, in e : EventArgs) -mmuCargoID_Click(in sender : object, in e : EventArgs) -mmuLength_Click(in sender : object, in e : EventArgs) -mmuWidth_Click(in sender : object, in e : EventArgs) -mmuHeight_Click(in sender : object, in e : EventArgs) -mmuClearList_Click(in sender : object, in e : EventArgs) -dgPackedCargo_Click(in sender : object, in e : EventArgs) -miDeleteCargoItem_Click(in sender : object, in e : EventArgs) -PopulateDataGrid() +GetItemCount() : int +GetCargoManifestFileName() : string +SetCargoManifestFileName() +GetContainerManifestFileName() : string +SetContainerManifestFileName()</pre>

Figure 3.4 – The frmCargoManifest Form

(See Figure 2.12 for a screenshot of this form)

The frmCargoManifest form handles the display of the Cargo Manifest file. The frmCargoManifest_Load method is called when the form is first displayed. It prepares

the datagrid to display the packed cargo items. Once the datagrid is ready, the PopulateDataGrid sub procedure is called.

The btnClose_Click method is called when the user presses the **Close** button on the form. This method closes the frmCargoManifest form and loads the frmContainerType form so that a new cargo list may be created for packing.

The mmuExit_Click method handles the event of a user selecting the **Exit** option from the **File** menu. This closes any open forms and makes certain that the scanner is turned off.

The mmuContainer_Click method handles the event of a user selecting **Container** from the **Sort By...** menu. This sorts the cargo items in the datagrid by their container in descending order.

The mmuIndex_Click method handles the event of a user selecting **Index** from the **Sort By...** menu. This sorts the cargo items in the datagrid by their packing index in descending order.

The mmuCargoID_Click method handles the event of a user selecting **Cargo ID** from the **Sort By...** menu. This sorts the cargo items in the datagrid by their Cargo ID in descending order.

The mmuLength_Click method handles the event of a user selecting **Length** from the **Sort By...** menu. This sorts the cargo items in the datagrid by their position length-wise in the container in descending order.

The mmuWidth_Click method handles the event of a user selecting **Width** from the **Sort By...** menu. This sorts the cargo items in the datagrid by their position width-wise in the container in descending order.

The mmuHeight_Click method handles the event of a user selecting **Height** from the **Sort By...** menu. This sorts the cargo items in the datagrid by their position height-wise in the container in descending order.

The mmuClearList_Click method is executed when a user selects **Clear List** from the **Actions** menu. The method deletes all of the packed cargo items from the datagrid.

The dgPackedCargo_Click method handles the event of the user selecting one of the packed cargo items in the datagrid. This method displays the context menu so that the user may delete the currently selected cargo item.

The miDeleteCargoItem_Click method is called when a user selects **Delete Cargo Item** from the context menu that is displayed by dgPackedCargo_Click. This method deletes the currently selected cargo item from the packed cargo list.

The PopulateDataGrid sub procedure reads the Cargo Manifest file and stores each cargo item in an XML data object. The datagrid is then populated with all the cargo items using the XML data objects.

The GetItemCount function returns the value of the private intItemCount property. This property contains the number of cargo items that were read into XML data objects during the PopulateDataGrid sub procedure. Currently, this function is not used. It is there for future expansion.

The SetCargoManifestFileName sub procedure allows for the private strPackedCargoFile property to be set. The property holds the file name of the Cargo Manifest file.

The GetCargoManifestFileName function returns the value of the private strPackedCargoFile property.

The SetContainerManifestFileName sub procedure allows for the private strPackedContainerFile property to be set. The property holds the file name of the Container Manifest file.

The GetContainerManifestFileName function returns the value of the private strPackedContainerFile property.

The XMLReader Class

XMLReader
<pre> -trCargoLookup : TextReader -intUnitID : int -strCargoID : string -dblLength : double -dblHeight : double -dblWidth : double -dblWeight : double +XMLReader() ~XMLReader() +FindCargo(in strCargoIDToFind : string) : bool -AddCargoTag(in strCargoIDToFind : string) : string -StripTags(in strItemToStrip : string, in strLeftTag : string, in strRightTag : string) : string -FormatForLength(in strLengthWithTags : string) -FormatForWidth(in strWidthWithTags : string) -FormatForHeight(in strHeightWithTags : string) -FormatForWeight(in strWeightWithTags : string) +GetUnitID() : int +GetCargoID() : string +GetCargoLength() : double -SetCargoLength(in dblNewLength : double) +GetCargoHeight() : double -SetCargoHeight(in dblNewHeight : double) +GetCargoWidth() : double -SetCargoWidth(in dblNewWidth : double) +GetCargoWeight() : double -SetCargoWeight(in dblNewWeight : double) +LookupType(in strType : string) </pre>

Figure 3.5 – The XMLReader Class

The XMLReader class is used to find cargo items within the Cargo Lookup file. XMLReader() and ~XMLReader() are a constructor and destructor, respectively. The constructor is called by default when the class is instantiated and it currently performs no tasks. The destructor closes any files it may have opened during execution.

The FindCargo method reads through an open XML file line by line, searching for a given Cargo ID. If the Cargo ID is found in the Cargo Lookup file, it pulls out all the lines of data for that cargo item. The data is formatted using the Format sub procedures within the XMLReader class. Once the data is formatted, the class properties are set. They are intUnitID (if UnitID was turned on by the CPS), strCargoID, dblLength, dblHeight, dblWidth, and dblWeight.

The AddCargoTag function adds the proper XML tags to the Cargo ID from the barcode scanner and returns it as a string. This allows for an easy comparison of the current Cargo ID against any Cargo ID within the Cargo Lookup file.

The StripTags function strips the desired tags from a line of an XML file. The function returns the data between the tags as a string.

The FormatForLength sub procedure calls the StripTags function and sets the class property dblLength.

The FormatForWidth sub procedure calls the StripTags function and sets the class property dblWidth.

The FormatForHeight sub procedure calls the StripTags function and sets the class property dblHeight.

The FormatForWeight sub procedure calls the StripTags function and sets the class property dblWeight.

The GetUnitID method returns the value of the private intUnitID property.

The GetCargoID method returns the value of the private strCargoID property.

The GetCargoLength method returns the value of the private dblLength property.

The GetCargoHeight method returns the value of the private dblHeight property.

The GetCargoWidth method returns the value of the private dblWidth property.

The GetCargoWeight method returns the value of the private dblWeight property.

The LookupType method is used to set the type of look up to be performed, either cargo or container. It then opens the appropriate XML file based on this choice. Currently, only the cargo look up is performed. The container look up option is there for the frmContainerType form in the event the cmbContainerType becomes a dynamic list and the values are read from an XML file.

The XMLWriter Class

XMLWriter
-fiCargoXML : FileInfo -strwCargoXML : StreamWriter
+XMLWriter() ~XMLWriter() +CreateFile(in strFileName : string) : bool +WriteDeclaration() +WriteBeginRoot() +WriteEndRoot() +WriteContainer(in strContainerType : string, in strMeasurementType : string) +WriteCargo(in intUnitID : int, in strCargoID : string, in dblLength : double, in dblWidth : double, in dblHeight : double, in dblWeight : double) +Close()

Figure 3.6 - The XMLWriter Class

The XMLWriter class handles the creation of the XML file the CPS requires for packing. XMLWriter () and ~XMLWriter () are a constructor and destructor, respectively. The constructor is called by default when the class is instantiated and it currently performs no tasks. The destructor closes any files it may have opened during execution.

The CreateFile method is used to create the file using the strFileName argument as the name of the file. If the file was successfully created, it returns a Boolean true. Otherwise, it returns a Boolean false.

The WriteDeclaration method writes the appropriate XML header to the file.

The WriteBeginRoot method writes the beginning outer most tag of the XML file.

The WriteEndRoot method writes the ending outer most tag of the XML file.

The WriteContainer method takes as input the name of the container and the measurement units being used. These values are then written with the appropriate tags to the XML file. This method should only be called once and immediately after the WriteBeginRoot method was called.

The WriteCargo method is used to write each cargo item to the XML file. It will be called once for every cargo item in the scanned cargo list. It takes all of the cargo item properties as input and writes them to the XML file with the appropriate tags.

The Close method is called after the WriteEndRoot method is called. It is used to close any open streams to the XML file.

CHAPTER 4

IMPLEMENTATION

The development process that was primarily used consisted of coding a small portion of the software at a time and testing that portion before moving on. In all of the classes, a method or sub procedure was implemented and tested to ensure that it would produce the correct results if given the correct input.

Coding began with the frmMain form. Since this was going to constitute a large (and most difficult to code) portion of the software, it was appropriate to start with this form. The frmMain form would handle the cargo item scanning and sending the files to the CPS, and since the CPS had not been coded yet, writing the code to handle the barcode scanning was done first. Once the necessary information was retrieved from the barcodes, the XMLReader class was started.

The first portion of code to be written for the XMLReader class was the FindCargo method. While coding the FindCargo method, the AddCargoTags, StripTags, and the formatting methods were created. These are all methods and sub procedures that FindCargo uses to set the private class data members. After this portion was coded and tested, the setters and getters were developed so that external classes could access the private data members.

Then work resumed on the frmMain form. Now that the software could access all of the cargo items physical properties based on a Cargo ID, code to display this information was created. After the display code was completed, testing of the interaction of the frmMain form with the XMLReader class and the XMLReader class as a whole was performed.

Then, work began on the XMLWriter class. The author felt very proficient at writing code that sent information to a file, therefore, most of the class was coded before it was actually tested. The only method not coded was the WriteContainer method. The frmMain form was then used to test the XMLWriter class.

At this point, there was no way to get the container and measurement data from the user. There was no room on the frmMain form to allow the user to make these selections so the frmContainerType form was created. Setters and getters were added to the frmMain form to allow frmContainerType to send the container type and measurement data to frmMain. Since frmMain now knew this information, the WriteContainer method for the XMLWriter class was written.

Next, the code was created in frmMain that would establish the connection to the server and send the XML file of packed cargo. After this was adequately tested, the code for receiving files was written and tested.

The one remaining step to code was the display of the Cargo Manifest file. The frmCargoManifest form was created to perform this action. First, the sub procedure, which reads in the data from the file and displays it, was written and tested. Then, the code to sort the cargo items was written and tested.

CHAPTER 5

SOFTWARE TESTING

Testing this software proved to be a difficult task. This was mainly due to the software's dependence upon the PocketPC hardware, namely the barcode scanner and the wireless network card. Microsoft Visual Studio offers an emulator that mimics a PocketPC platform and allows for PocketPC software to be tested on a desktop. This emulator could not be used for testing the Cargo Scanner software. Whenever an attempt was made to run the Cargo Scanner software using the PocketPC emulator, the emulator would crash because it could not properly load the DLL files required by the barcode scanner. To overcome this setback, most of the code was first written and tested on the PC ignoring the barcode scanner hardware. Once the code worked properly on the PC, it was then inserted in the PocketPC code, executed, and tested.

As previously stated in the implementation portion of this paper, unit testing was performed as each unit of code was finished and integration testing was performed as completed classes were incorporated into the project. A majority of the testing focused on three main areas of the software: scanning the cargo items, network transmission of the files, and displaying the manifest cargo file.

The scanning of cargo items underwent a great deal of unit and integration testing. During unit testing, cargo items were scanned and the data object returned by the barcode scanner was stored. The Cargo ID was then extracted from this data object and displayed. When the Cargo ID was displayed, it was manually checked against the Cargo ID that was displayed on the barcode label. Once the Cargo ID could be extracted reliably, the XMLReader class was tested to make certain that it would return the

appropriate data when supplied a Cargo ID. When unit testing of the XMLReader class was completed, it was then integrated with the barcode scanner and retested. After every item was scanned, the data displayed on the screen by the XMLReader class was manually checked against the Cargo Lookup file that the software was using to retrieve the data.

Testing the transmission of files over the network was the most difficult phase of the entire project. Since the CPS was not completed at the time of the creation of the Cargo Scanner software, it was not possible to test the actual transmission of files over the network. However, an alternative means of testing was found. First, I created a program to run on the PC that contains the PocketPC emulator. This program would perform the sending and receiving of files that the CPS would eventually carry out. This was accomplished by having the program create a TCP Socket and listen for a connection request using the loopback IP address of the PC. This address is 127.0.0.1. When it receives a request, it establishes the connection and begins receiving the XML file from the Cargo Scanner software. Once the entire file is received, it then sends the Cargo and Container manifest files to the Cargo Scanner and closes the connection.

In order for the Cargo Scanner to communicate with the fake CPS program running on the PC, it could not be executed from the PocketPC. The Cargo Scanner code had to be written so it could run within the PocketPC emulator. This would allow the Cargo Scanner software to access the PC's loopback address. Once the Cargo Scanner could connect to the fake CPS, the ability to send and receive files was extensively tested. This was done by examining each line of the file as it was sent and received by both the Cargo Scanner and fake CPS. This was done to ensure that the data being sent was being

received per specifications. The resulting XML files were then compared to the originals to make certain that no data was lost.

The next set of testing was done on the ability to display the Cargo Manifest file. This was done by supplying the frmCargoManifest form with a valid Cargo Manifest file. Each cargo item and all its data from the Manifest File was displayed within the form. All the data displayed on the form was manually checked against the XML file to make sure the data was correct.

The final step of testing was system testing. This was performed once the CPS software was completed. To perform this task, the Cargo Scanner software was used as it would be used in the real world. Several cargo items were scanned using the PocketPC and the list of items was created and written to an XML file. The XML file was received by the CPS and the CPS performed its operations. The CPS would then send the resulting Cargo and Container manifest files to the Cargo Scanner. The Cargo Scanner would then display the each cargo item from the manifest file on the screen. To test the results, the Cargo and Container manifest files on the PocketPC were compared to the same files on the server running the CPS.

It should be noted, the assumption was made that the files being sent by the CPS were valid manifest files. The Container Packer software has been extensively tested in the past so this was a safe assumption.

CHAPTER 6

SOFTWARE DEPLOYMENT

Deploying the Cargo Scanner software is an easy and straightforward process. After the code has been compiled, it must be built into a CAB file. This is done by selecting **Build**, then **Build Cab File** from the main menu with Microsoft Visual Studio 2003. The CAB file will be on the hard drive of the PC so it must be transferred to the PocketPC for installation. This is done by simply connecting the PocketPC to the PC via USB or serial cable. Once connected, the PocketPC should act as an external drive of the PC. It can be accessed through Microsoft ActiveSync Version 4.2 or through the My Computer icon. Then, the CAB file need only be dragged from the 'cab' folder within the Cargo Scanner project folder to any folder on the PocketPC. The CAB file will then be copied to the PocketPC.

Next, use the PocketPC to navigate to the location of the CAB file. To install the Cargo Scanner software, double tap on the CAB file. The Cargo Scanner software will be installed to the directory: “\Program Files\WirelessCargoScanner.”

To begin using the Cargo Scanner software, select **Programs** from the **Start** menu, then select **Wireless Cargo Scanner**. The Cargo Scanner software will start and the Container Selection window will be displayed.

CHAPTER 7

PROJECT ASSESSMENT

Assessment

This project was difficult in many respects. The most difficult part was managing the project from start to finish. First, the required hardware and software had to be identified and acquired. After two weeks of research, the list of what was needed and an estimated cost to acquire the items was agreed upon by the author and Dr. Henry. Then, the project had to be proposed to The Office of Research and Sponsored Projects in order to acquire the necessary funding to buy the needed supplies. After the project was approved, all the equipment was purchased from multiple websites. When the equipment arrived, one of the items was not as it was advertised on the website. The item had to be returned and a replacement shipped. This entire process spanned four months.

Once the necessary equipment was obtained, development began. This stage of the project had its share of challenges as well. Due to the incompatibilities between the development environment and the target platform, a new development environment had to be installed and many of the environment features, which would have made programming much easier, were lost. This added three weeks to my development time. The total time for development was two and a half months, bringing the total time to complete the project to six and a half months.

Many lessons were learned during this project. The first was learning how to deal with other people and companies. People who are not directly involved in a project do not seem to offer much help or consideration for their small involvement. Much time was spent waiting for people to make decisions and allocate resources in order to get

started. More time was lost arguing with companies and waiting for them to fix their mistakes.

Even though these situations were very frustrating and time consuming, they provided opportunities to learn about time management and performing tasks in parallel. The first part of the development process is planning. Much more development was possible in the first four months. This would include developing initial designs of the user interface, determining what classes were needed and how they would interact with each other, and determining if there is any pre-existing code that could have been used.

More time should have been invested researching the equipment that would be used. The operating system on the PocketPC was incompatible with the latest Visual Studio development environment. Although all problems cannot be avoided, this one could have been through a more diligent investigation into the specifications of the PocketPC.

C# was a language largely unfamiliar to the author and therefore was a concern as the programming language for this project. It turned out to be a fantastic choice. C# was very similar to C++ and VB.Net, which the author knows well. Because of this, coding provided minimal difficulties.

Future Direction

After three months of development, there is much that can still be done to enhance and extend the Cargo Scanner software. One such enhancement would be the ability of the software to dynamically find a server that is running the Container Packer Service. Currently, the IP address and the port number of the server are hard coded into the program.

Another enhancement to the software would be the ability to save and open both the scanned cargo lists and the cargo manifest lists. Saving the files would not require much coding effort. Currently, the scanned cargo list is written over every time the pack button is pressed. This is the only time the file is written. **Save** and **Open** options should be added to the **File** menu on the Main window. This way, a user would be able to save the scanned cargo list without pressing the **Pack Cargo** button. The user could also open old scanned cargo lists at any time. The **Save** and **Open** options should also be added to the Manifest window.

Currently, the list of containers in the drop-down list on the Container Selection window is a static list. The container names are hard coded into the program. It would be a better idea for this to be a dynamic list. This could be done by having the CPS software send the Cargo Scanner the file listing all the different containers. Once the Cargo Scanner receives this file, it populates the drop-down list with the latest container names. This allows for the list to be easily updated and only one file needs to be edited.

Many features could be added to the Manifest window. It could be changed to allow for more information to be displayed. Even though the Cargo Scanner receives the Container Manifest file, it is currently not being utilized. It is possible that the Manifest window could be arranged as a tabbed window. One tab could have the Cargo Manifest list and the other tab could have all the information regarding the containers. A drop-down list could contain the different Container IDs and when the user selects one, that containers information would be displayed. This information would include the packed weight, height, length, width, and the number of items within the container.

Also, the number of cargo items in the Cargo Manifest list could be displayed somewhere in this window. There is a property of the frmManifest class that keeps track of this. Due to lack of space on the screen, it is not currently displayed anywhere.

Another useful enhancement would be to display only between ten and twenty items on the Cargo Manifest list at a time. The Cargo Manifest list can easily grow to thousands of items. It is very difficult to use a vertical scrollbar in order to view that many cargo items. It might not be a bad idea to have a button/buttons that allow the user to view the list, several items at a time.

It might also be beneficial to restructure the way items are removed from the Cargo Manifest list. Rather than selecting an item from the list and selecting **Delete** from the context menu, it may be faster for the user to just rescan the cargo item. This would require that the barcode scanner be turned on once the Cargo Manifest list is loaded. When a user scans an item, the list of cargo objects is searched for that item's Cargo ID. When the Cargo ID is found, the cargo object is removed from the list and the Manifest window would be updated.

Finally, it would be extremely beneficial to provide visualizations for packing the cargo. Currently, the user must determine where each item goes based on the X, Y, Z coordinates of the back left corner of the item. This would be a very difficult task. The visualization for the Container Packer that runs on desktops uses more memory and processing power than the PocketPC offers. A cargo visualization for the PocketPC would have to be a stripped down version of the current one. Instead of displaying all cargo items within the container, it might only show a few items at a time. Another

possibility is that the user selects a cargo item from the manifest list and a new window appears with a display of where the item goes within the container.

References

Henry, Joel. Overview of an ActiveX Control that Performs Container Stuffing .NET 2005 Version. February, 2006.

Appendix A

Example Cargo Lookup File

```
<?xml version="1.0" encoding="utf-8" ?>
<Lookup>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>J2B|3UM77|44|75</CargoID>
    <Length>3.875</Length>
    <Width>2.625</Width>
    <Height>1.375</Height>
    <Weight>37</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>J2B|3UM77|43|74</CargoID>
    <Length>15.875</Length>
    <Width>4.625</Width>
    <Height>4.375</Height>
    <Weight>37</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>J2B|3UM77|42|73</CargoID>
    <Length>45.214</Length>
    <Width>9.625</Width>
    <Height>11.859</Height>
    <Weight>37</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>191050</CargoID>
    <Length>15.875</Length>
    <Width>9.625</Width>
    <Height>9.375</Height>
    <Weight>1.23</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>192564</CargoID>
    <Length>15.875</Length>
    <Width>9.625</Width>
    <Height>9.375</Height>
    <Weight>37</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>192565</CargoID>
    <Length>15.875</Length>
    <Width>9.625</Width>
    <Height>9.375</Height>
    <Weight>37</Weight>
  </Cargo>
</Lookup>
```

Appendix B

Example Cargo Item File

```
<?xml version="1.0" encoding="utf-8" ?>
<movementplan>
  <Containers>
    <ContainerName>8x9.5x40Ft.ISO</ContainerName>
    <MeasurementUnits>imperial</MeasurementUnits>
  </Containers>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>192564</CargoID>
    <Length>15.875</Length>
    <Width>10</Width>
    <Height>9</Height>
    <Weight>37</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>192565</CargoID>
    <Length>15.875</Length>
    <Width>10</Width>
    <Height>9</Height>
    <Weight>37</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>192567</CargoID>
    <Length>13.65</Length>
    <Width>10</Width>
    <Height>4</Height>
    <Weight>25</Weight>
  </Cargo>
  <Cargo>
    <UnitID>123456</UnitID>
    <CargoID>109528</CargoID>
    <Length>12.7</Length>
    <Width>4</Width>
    <Height>11</Height>
    <Weight>65</Weight>
  </Cargo>
</movementplan>
```