Developing a Generic Academia Mobile Short Messaging System Using the Notion of Design Patterns

MSc. Thesis

By

Jesse Canuel

Submitted in Partial Fulfilment for the Degree in MSc in Computer Science, Department of Computer Science, Lakehead University,

Under the Supervision of Dr. Sabah M. A. MOHAMMED

Thunder Bay, Ontario, CANADA

April 2005

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-10650-6
Our file Notre référence
ISBN: 0-494-10650-6

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

I would like to thank my supervisor Dr. Sabah M.A. MOHAMMED, for his clear advice and encouragement during this thesis.

I would also like to thank my internal examiner Dr. Jinan A.W. FIAIDHI for her prompt and precise evaluation of this thesis.

I would also like to Dr. Francis Allaire for dedicating his personal time to thoroughly correcting this paper.

I would also like to thank all my classmates and Lakehead University for helping me in furthering my education.

Last and by no means least, thanks to my family, my co-workers and friends who have given me their support and encouragement throughout my studies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Designing reusable applications is vital area of research. Design patterns are an innovative notion that promotes reusability. This thesis attempts to develop generic and reusable Short Messaging Systems that can be used by the academia environment using the notion of design patterns.

In this thesis the design and implementation of the two essential parts of any Short Messaging System is carefully investigated and compared to the relevant traditional approaches. Both the Short Messaging Service Center (SMSC) and the Mobile Station (MS) have been fully implemented as generic units based on selected design patterns.

In chapter 2, we investigated possible developing platforms to house the SMSC. Different servers were also discussed. Ideally, the developing platform should not limit the choice of server. JSP was used to develop the SMSC application because it was found to operate properly on any server. The second part of chapter 2 discussed three possible implementations of the SMSC application. The first uses a simple architecture of only one JSP page. The second realizes that true generality is only achieved through the use design patterns and it implements the SMSC application with the aide of the Model, View and Controller (MVC) design pattern. The third implementation tries to enforce this crucial design pattern through Struts, which is based around the MVC design pattern.

Chapter 3 includes three stages that were used to develop a SMSC application for sending and receiving SMS messages. The first stage included a crude SMSC application that lacked all structure. It placed all of the business logic with the presentation logic, thus, making the page hard to read. The second stage proposed separating the business logic and presentation logic with a design pattern. The MVC design pattern was used and a lot of structure was gained. Now that we have this great design pattern aiding the architecture of the SMSC application we needed a way to enforce it. In the third implementation we used Struts, which automatically applies and enforces the MVC design pattern.

In Chapter 4, we explained the architecture of a MS. We also outline the necessary steps for a mobile device to send and receive SMS messages. J2ME is introduced as the preferred developing platform for a MS application. This chapter also introduces two MS applications for sending and receiving SMS messages. The first SMS application was developed by Sun Microsystems and is easily deployable. However, it didn't have a design pattern so the second SMS application proposed two design patterns that will serve the architecture some structure. The two design patterns were the MVC and the Wizard Dialog.

Finally, in Chapter 5, conclusion and future research trends are discussed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Acknowledgements ii

Abstract iii

1. Reviewing Short Messaging Services Technologies 1

1.1 Overview 1
1.2 SMS Architectures 5
1.3 Current SMS Technologies 7
1.3.1 Current SMS Technology for SMSC 7
1.3.2 SMS on the Mobile Station 8
1.3.3 WAP Short Messaging Systems 9
1.3.4 .Net CF 9
1.3.5 J2ME SMS 10
1.4 Design Patterns 13
1.4.1 Strategy Design Pattern 15
1.4.2 Command Design Pattern 16
1.4.3 Model View Controller Design Pattern 17
1.4.4 Using Design Patterns in MS Design 19
1.5 Useful APIs for SMS Systems 20
1.5.1 Java Server Pages 21
1.5.2 JavaPhone API 23
1.5.3 .NET Mobile Web SDK and ASP.NET 23
1.6 XHTML 24
1.7 Summary 27

2. Designing a Generic Faculty Short Message System Center 28

2.1 Why Generic SMSC 28
2.2 Deploying a Web-Based SMSC using SimpleWire 29
2.3 Deploying a Web-Based SMSC System 30
2.3.1 Deploying a Web-Based SMSC System using J2EE JSP 30
2.3.2 Deploying a Web-Based SMSC System using Tomcat Apache32
2.4 Composing a Java Server Page 34

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4.1 An Enhanced SMSC Model ... 35
2.4.2 An Enhanced SMSC Model Based on MVC Design Pattern ... 36
2.4.3 An Enhanced SMSC Model Based on MVC Design Pattern using Struts ... 37
2.5 Summary ... 38

3. Implementing a Generic Faculty Short Message System Center Using Various MVC Design Pattern Implementations ... 40

3.1 SMSC using JSP ... 40
3.2 MVC SMSC using JSP ... 41
3.3 MVC SMSC Implementation Using Struts ... 44
3.4 Summary ... 47

4. Developing a Generic Mobile Station ... 49

4.1 Introduction ... 49
4.2 MS General Architecture ... 49
4.3 Choosing the Right Test-beds for MS Stations ... 52
4.4 Traditional MS SMS Standards ... 53
4.5 Developing the MIDP of the MS Station ... 54
4.6 Reviewing some Essential J2ME API ... 55
4.7 Developing a Generic Mobile Station SMS application using J2ME ... 58
4.7.1 Sun Microsystem's MS SMS Application ... 59
4.7.2 A Generic MS Application using the MVC and Wizard Design Patterns ... 61
4.8 Summary ... 63

5. Conclusions and Future Research ... 67

5.1 Thesis Summary and Findings ... 67
5.2 Analytical Comparison of the Generic SMS Applications ... 68
5.3 Chapter Summary ... 69
5.3.1 Chapter 1 ... 69
5.3.2 Chapter 2 ... 69
5.3.3 Chapter 3 ... 70
5.3.4 Chapter 4 ... 70

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3.5 Chapter 5 71
5.4 Future Research Directions 71

References 75

Appendices 85

Appendix A SMSC source code using JSP 85
Appendix B MVC SMSC source code using JSP 91
Appendix C MVC SMSC using Struts Source Code 99
Appendix D An Generic MS J2ME Example 109
Appendix E A Generic MS Application for Sending and Receiving SMS messages 120

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Reviewing Short Messaging Services Technologies

1.1 Overview

With the ever-growing population of wireless networks and mobile devices, such as, cell-phones, pagers, personal digital assistants (PDAs), etc., there must be a way to send and receive information to and from these devices. There are numerous ways of sending and receiving text-messages to these devices and one of the more popular methods is using Short Message Service (SMS) or also known as Cell Broadcast System (CBS). Some of the other popular methods include Wireless Application Protocol (WAP), i-mode, or General Packet Radio Service (GPRS). SMS is known for having a simple user interface, unlike WAP's rich Web-like interface [Xu, 2003]. SMS is widely supported in wireless telephones in most European and Asia-Pacific countries [Laird, 2001] and it will continue to grow in the other countries, such as, in the USA.

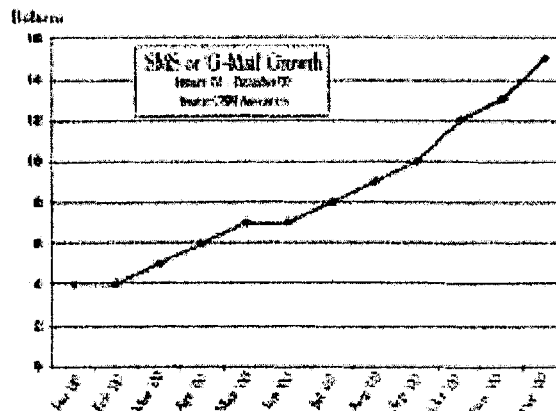

Figure 1.1: Worldwide growth of SMS use

SMS originated in Europe around 1991. It was part of the Global System for Mobile Communications (GSM) Phase 1 standard. The first SMS message was sent in December 1992 from a PC to a mobile phone on the Vodafone GSM network in the UK [Harron, 2002]. From there SMS just took off and it continued to grow rapidly. The use of the text messaging is growing every month. Between January & December 2000, SMS use grew from 4 billion to 15 billion messages per month [Butts, 2001]. Please refer to Figure 1.1 where it clearly illustrates an increase of 375% in one year.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

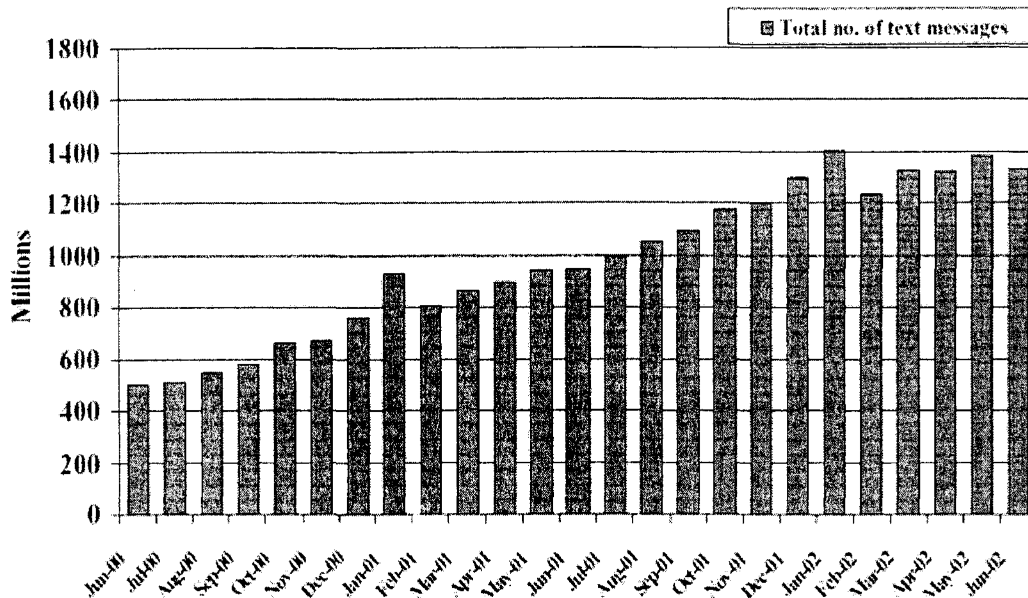

Figure 1.2: Text Messaging Growth (SMS): UK GSM Network Operator Totals June 2000

What a phenomenal amount. SMS slowly picked up popularity in North America but it is among one of the favourites now though. Figures released by the Mobile Data Association (MDA) revealed that Person-to-Person SMS text messaging for June 2002 stand at 1.3 billion and are up by 380 million on the previous year. Please refer to Figure 1.2 [Harron, 2002].

The architecture of SMS is relatively simple. Typically, a mobile device would send another mobile device a text message. Either the receiving end or the sending end doesn't have to be mobile. It could be a PC, Fax Machine, Email, or any IP address. When a text-message is sent, it is sent to the Short Messaging Service Centre (SMSC). The SMSC will manage sending the message to the mobile device, even if the mobile device is off or out of range. The SMSC will send a request to the mobile device's Home Location Register (HLR) to find the roaming customer. The HLR will then inform the SMSC if the customer is available. If the customer's mobile device is "inactive" then the SMSC will store the message for a period of time. When the mobile device becomes active once again the HLR will notify the SMSC and inturn the SMSC will SMS the message to the mobile device. The actual architecture of SMS will be discussed more thoroughly later.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SMS has the following benefits:

- It is inexpensive
- Convenience of "anytime and anywhere"
- Popularity

Since SMS is so cheap the cost of sending SMS messages is less than other data oriented mobile services such as WAP [Xu, 2003]. The cost of sending a message has two different types of costs, the cost of the phone and the cost of the message. Practically most phones are SMS enabled, whereas, WAP enabled phones are expensive. The cost of sending a message is 0.02 Euro in Philippines, 0.04 Euro in Japan and 0.11 Euro in Western Europe [Marcussen, 2002].

The benefit of "anytime and anywhere" is a great advantage to SMS. As long as the mobile device is equipped with SMS and its switched on then that user can send a SMS message "anytime". The user doesn't have to worry about if the receiving mobile device is on or off because the SMSC will handle delivery of the message. The "anywhere" is more or less a feature of the mobile device.

Popularity becomes a benefit to SMS because it helps to bring the prices of SMS enabled phones lower. Also if SMS is a service that everyone wants then it will be a cheap service. The more popularity that SMS is exposed to the better off it is.

There are two different types of SMS applications:

- Point-to-Point: An acknowledgement of receipt is provided to the sender.
- Cell Broadcast: This allows for a number of unacknowledged (general) messages to be broadcasted.

The two types of user applications are categorized under Consumer-based and Commercial/Enterprise applications. It is not uncommon to find 90% of a network operator's total SMS traffic being accounted for by the applications to come in this section [Buckingham, 2000]. Examples of SMS consumer applications generally include:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Peer-to-Peer: This is usually the exchange of a simple text-message from one mobile-device to another. The message is typically inputted from the mobile device's keypad. A message might consist of, "Can you meet me for supper?" This is the most common type of use of SMS [Malhotra, 2001].

- Information Services: This is when SMS is used for retrieving stock quotes, weather reports, lottery results, etc. Basically, any information that can fit into a short message can be delivered by SMS.

  An information service is usually received after a mobile-device makes a request. For instance, to receive the winning lottery numbers one might send "WLN", to a predefined number. Then a few moments after sending the message the mobile-device would receive the latest lottery numbers.

  Information services should be simple to use, timely, personalized and localized [Buckingham, 2000].

- Advertising: SMS can be used as a form of low-cost advertising for businesses. For instance, a cell-phone provider could send great cell-phone packages to their subscribers. Another form of advertising would be if the mobile-user has provided their cell-phone number to a business, then that business can send them special alerts. These alerts could be anything from informing them of great sales, or that their order is ready for a pickup, or maybe that there are some clothes in the store that is their size. SMS can be a very cost effective advertising tool.

- Voice and Fax mail notifications: This simply sends the mobile-device a text-message that they have new mail waiting. The message would typically state from who the mail came from and possible the subject (if applicable).

Listed below are some examples of SMS Commercial/Enterprise applications:

- Customer Service: Quite often SMS can assist in avoiding expensive customer service centres person-to-person voice calls. A lot of businesses (or corporations) have multiple sites spread out over entire continents and even over different countries. They can cost efficiently send SMS messages to anywhere in the world, whereas, making telephone calls suffer from varying long-distance charges. Most long-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

distance calls vary from country to country but a SMS text-message has a flat rate.

- Job Dispatch: SMS can be used to deliver jobs to a worker out in the field. For instance, a worker could leave the office in the morning with only 2 work-orders but providing that he has a mobile-device then he could receive more work requests. This helps a business operate more efficiently because the more recent work requests could have more urgency.

1.2 SMS Architectures

The SMS architecture is concerned with delivery of the message. It also deals with what happens when the customer's mobile device is turned off or is out of range. The architecture also needs to concern itself with how the Mobile Station should receive information. A Mobile Station is the mobile receiving device. The architecture defines how SMS operates from all aspects. The following are descriptions of the SMS architecture elements [Malhotra, 2001]:

- MS Mobile Station, a wireless terminal that is capable of receiving and sending alphanumeric messages.
- SIM Subscriber Identity Module, otherwise known as a SIM card. This card is for identify the subscriber. This card will also be used for storing old undeleted messages on the mobile device.
- BS The Base Station is for communicating between the Mobile Station and the Mobile Switching Centre. The BS consists of controllers, Base Station Controllers (BSC), and Base Transceiver Stations (TBS), also known as "cells".
- SME Short Message Entity, which can be a device like a mobile phone, which is capable of receiving and sending alphanumeric messages.
- STP Single Transfer Point is used for operating on possible foreign networks such as X.25 or TCP/IP.
- HLR Home Location Register is for storing information in the database about the Mobile Station's subscriber record and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

possibly configuration record.

- VLR Visitor Location Register is the temporary data store in each Mobile Switch Centre where information about roaming subscribers is stored. This is what gives a mobile device the ability to go roaming outside of the subscription.

- SMSC The responsibility of the Short Message Service Centre is for storing and forwarding messages to and from the Mobile Station. This is achieved through a combination of hardware and software.

An SMS system can be viewed as a three-tier architecture. It consists of the Interface Layer, Implementation Layer and Transport Layer. The three layers are used for the following functions:

- Interface Layer: This layer is generic and doesn't depend on any messaging protocols. It typically will contain messaging interfaces that are used in providing the basic definition of a message [Ghosh, 2003]. The basic definition is used for sending and receiving SMS messages.

- Implementation Layer: The implementation layer is used for ensuring that the SMS message is the proper length and if it's not then this layer will perform segmentation and concatenation of the message for the underlying protocol [Ghosh, 2003]. It also contains classes that can implement the Interface Layer to access wireless messaging functionalities on a mobile device.

- Transport Layer: This layer contains the actual implementation protocols that carry messages to the mobile device. This layer can also contain additional security protocols [Vogler, 2000].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3 Current SMS Technologies

SMS can be developed in multiple different languages (or environments). For instance, SMS supports C, Java, Perl, Visual Basic, ActiveX, etc, which environment is chosen largely depends on the developer's preference. The chosen environment can also depend on how much exposure the developer wishes their SMS application to have. If an SMS system is developed on a programming language that isn't largely supported then the SMS System isn't very useful. What different SMS Systems could be developed? Well, there are two different flavours of SMS technology. We could have SMS for the mobile device (sending or receiving) or SMS technology for PCs or servers.

1.3.1 Current SMS Technology for SMSC

As mentioned earlier, SMSC is the SMS Centre and is responsible for storing and forwarding messages to and from the mobile device. That's the short of it, but it's actually responsible for a little bit more. The SMSC is responsible for storing and forwarding but it's also responsible for queuing messages, billing the sender, and/or returning receipts if necessary. In order for a SMS message to reach it's destination it must pass through a SMSC. What is a SMSC? It's a combination of hardware and software. The software has the feature of sending a text message via a website. This can be a very useful feature if someone wishes to send a message from his or her PC. Then the recipient will be guaranteed to receive the SMS message because of the SMSC.

There are quite a few services that offer SMSC and quite often these services are free, i.e., one could send a SMS text message to a mobile phone for free. However, this wasn't always the case because the pioneers to SMSC, which are BellSouth Mobility, PrimeCo, and Nextel, among others [CSL, 2003], only offered the service on the action mobile device. SMS had to slowly evolve and develop from there. As Figure 1.3 illustrates it can handle any form of email, voice mail, websites, or mobile devices. Since the SMSC can handle all of these different forms it makes SMS a better system. The more various different forms SMSC can handle the more generic the SMS system is and the more generic then the more useful it will be.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

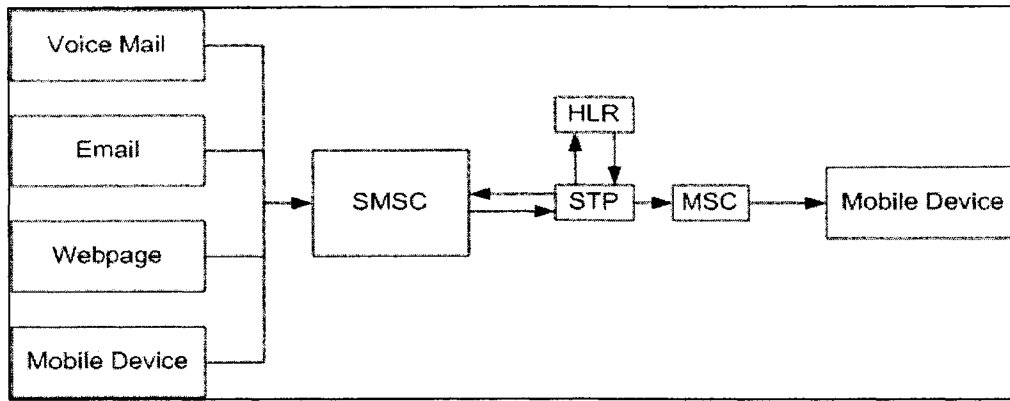

Figure 1.3: SMSC basic network

1.3.2 SMS on the Mobile Station

For the mobile device there are a limited number of SMS applications. One could use J2ME, Wireless Application Protocol, or a native platform. The server-side of the SMS application has a little bit more say as to what has to handle the incoming (or outgoing) SMS message. If the SMS application was built around J2ME then the server should know how to read and handle this J2ME application message.

To give a brief comparison of J2ME, WAP, and other native platforms, they are highly unalike. J2ME applications are known for having more features and security than WAP applications [Yuan, 2002]. WAP pages are known for having less risk of software crashes and/or virus attacks. This is because WAP is a think-client development protocol; J2ME is a development platform specifically for smart applications.

J2ME is slightly ahead of the native platforms because it allows one to write platform independent applications and thus the applications are highly portable. The Java platform's portability stems from its execution model [Yuan, 2002]. In general, the portability stems from the Java Virtual Machine (JVM) and how the JVM processes Java bytecode into machine code. All of this is executed at runtime. Native applications typically don't verify the code before executing it but the JVM does a two-byte bytecode verification. J2ME suffers from the fact that the standard J2ME API does not provide a way to access an underlying

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

device's SMS features [Yuan, 2002]. This implies that not all J2ME devices are SMS compatible.

1.3.3 WAP Short Messaging Systems

Wireless Application Protocol (WAP) was developed by the WAP Forum as a standard specification. It was designed with mobile devices in mind. It bridges the gap between the mobile world and the Internet [Brady, 2000]. The WAP Forum's goal was to offer a weblike experience on a mobile device. In order for a phone to be WAP enabled a relatively simple microbrowser has to be installed. The microbrowser requires limited resources, because the microbased-services and applications reside temporarily on the server.

The WAP Forum developed a new wireless webpage language, which they called Wireless Markup Language (WML). Any WAP application is written in WML. They are designed for low power and are a subset of XML. WML requires a DTD, which lists the local tags that can be used [Brady, 2000]. Since the WAP Forum developed WML it only makes sense that they specify the DTD. The problem with WML is that not all phones can intrepret it. Some phones can display WML and others can use stripped-down versions of HTML [Biggs, 2002].

WML organizes webpages in a card/deck metaphor. The deck is referred to as a complete webpage and the card is a small portion of the webpage. Typically cards are designed for the size of the mobile display device, i.e., a card wouldn't be larger than a cell-phone's screen.

WAP requires a WAP gateway. The reason for this is because the mobile device and a server communicate in two different languages. There must be an extra server between them to handle the translation [Brady, 2000]. The WAP gateway is often referred to as WAP Proxy.

1.3.4 .Net CF

.Net CF is one of the leading standards for developing applications on a mobile device. .Net CF stands for .Net Compact Framework. It is part of the Microsoft .Net environment, which implies that .Net CF will inherit a lot of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

.Net framework. However, it is compact so this is considered a lightweight version. Microsoft .Net CF is currently the overwhelmingly preferred development and run platform for applications on mobile hardware which use the latest Microsoft Windows CE compact operating system [Yuan, 2003]. How suitable would .Net CF be for developing a generic Mobile Station application for sending and receiving SMS messages?

First, let's look at how the .Net CF operates and functions. All code written on the .Net Framework platform is called managed code [CM20143]. When code is "managed" it comes with a few assurances:

- There are no bad pointers
- It's impossible to create memory leaks
- Supports strong type-safety

The Common Language Runtime (CLR) for the .Net CF runs regular .Net byte code applications. The .Net CF API is just a subset of the standard .Net API library. The .Net CF API's primary concern is for mobile application development [Yuan, 2003].

.Net CF is a lightweight platform that is great for developing a mobile application. However, it's lacking generality. As with most Microsoft developed products, the .Net CF only operates on the Windows CE operating system. Although, the Windows CE is highly deployed and used worldwide it still only consists of a small part of today's mobile device population [Yuan, 2003] [CM20143]. The .Net CF is great to develop in but it's not highly supported. Thus, .Net CF should not be considered when contemplating a generic MS application for sending and receiving SMS messages.

1.3.5 J2ME SMS

J2ME is Sun's answer to mobile devices. It stands for Java 2 Platform, Micro Edition. One of the huge benefits for using Java is that its platform independent and can therefore run on multiple platforms. The hugest constraint with programming for a mobile device is the fact that the device has limited memory, battery life, display size, processing power, and network bandwidth. As Yuan and Long state it, "It would be impossible to port the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

complete functionalities of an application running on a sophisticated set-top box to a cell-phone." [Yuan, 2002]

J2ME was developed not to handle every single device, because that wouldn't be feasible but to handle most devices. The design of J2ME had the J2SE (Java 2, Standard Edition) in mind. J2ME includes Java virtual machines and a set of standard Java APIs defined through the Java Community Process, by expert groups whose members include leading device manufacturers, software vendors, and service providers [Kluyt,2002]. J2ME basically took the classes of J2SE and kept what was suitable and micro-sized the rest. By "micro-sizing", they either deleted some of the not so necessary functionality or they decreased large objects to more micro objects. They tried to do all of this without taking away from necessary Java components [Muchow, 2002].

There are two different types of J2ME configurations. There is the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). The CLDC is typically for small resource constrained devices, for instance, cell-phones. The CDC was built with high-end PDA's in mind that are equipped to handle powerful processes. Configurations comprise a virtual machine and a minimal set of class libraries. They are used for providing the basic functions that a mobile device might need. These classes are nothing more than a suggestion and can be elaborated onto if one so chooses. A device that implements the CDC has the following characteristics [Muchow, 2002]:

- 512 kilobytes (minimum) memory for running Java programs
- 256 kilobytes (minimum) for run-time memory allocation
- Network connectivity, possibly persistent and high-bandwidth

And the typical characteristics of a device that implements the CLDC:

- 128 kilobytes of memory for running Java programs
- 32 kilobytes of memory for run time memory allocation
- A limited user interface
- Runs on battery power
- Wireless network connection, low bandwidth

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One typical drawback to the CLDC is its limited power to do mathematical processing. This actually becomes a security hazard that has to be addressed. Implementing secure applications is much harder, due to the CLDC configurations limited mathematical functionalities and the scant processing power of many of the underlying devices [Yuan, 2002]. Surprisingly, CLDC mobile devices are the most widely used of the mobile devices so enabling security on these devices is very important [Muchow, 2002].

In this section, J2ME was introduced. J2ME can be thought of as a subset of the Java platform designed specifically for the development of mobile device applications. In Table 1.1, the comparison between .Net CF and J2ME is outlined. The J2ME has some obvious advantages over the .Net CF.

	.Net CF	**J2ME Connected Device Configuration**	**J2ME Connected Limited Device Configuration**
Device Requirement	Powerful, expensive	Powerful, expensive	Cheap, pervasive
Cost	High	High	Medium
Language Supported	C#, VB.Net	Java	Java
Platforms	Pocket PC, Windows CE	Major mobile platforms except Palm OS	All mobile platforms
Market Focus	Enterprise	Enterprise	Consumer and Enterprise

Table 1.1: A comparison between .Net CF and J2ME

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.4 Design Patterns

What are design patterns? Design patterns are solutions to identified reoccurring problems at the development stage of an application. Once a problem has been identified then the best practices of experienced object-oriented software developers is applied [Geary, 2001], also known as, applying a design pattern.

These patterns are important for a number of various reasons. They allow the developer to verbally explain their code (or logic) in a unified method. A C developer might not understand Java but they will understand the concept behind a design pattern so a design pattern can be used to break the language barrier. Design patterns are also important because the developer can learn from other developers quickly and efficiently. If they notice the pattern deriving in their application logic then the design pattern can be applied to help them with this problem. Design patterns let you leverage the developer community's collective experience by sharing problems and solutions that benefit everyone [Hurst, 2002].

The Gang of Four (GOF) are commonly referenced when referring to design patterns. The GOF are among the forefathers of most design patterns. Their particular design patterns are grouped into three categories Creational, Structural, and Behavioural. These three categories have certain characteristics that will aide a developer if they start experiencing a recurring problem. If a developer is experiencing problems developing a certain piece of software then they might turn to a Creational design pattern; however, if they're experiencing a performance problem with an application that has already been developed then they might apply a Behavioural design pattern to correct this problem.

Table 1.2 illustrates the possible design patterns for the three classified categories that the GOF developed. As one can visibly see there are many solutions for the three categories. This is extremely useful since there are far too many behavioral problems for a single design pattern approach to solve all of them.

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Creational	Structural	Behavioral
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Table 1.2: Design pattern space [Gamma et al, 1994]

Design patterns are crucial to the architecture of any developing software. They offer an abstract way of developing code. When design patterns are applied properly they assist in making the application in question very generic. This is achieved by providing structure to the application. Also, everything in the application has a proper place and a purpose. For instance, if the developer decides that the presentation logic could use some more work, well, in a classical sense then the entire application would require an overhaul. If a design pattern is applied to the design of the application at the development stage then replacing the presentation logic is really quite simple. The reason for this is that the presentation logic is separate from the business logic. Thus, because a design pattern was used everything has a proper place.

As Table 1.2 illustrates, there are multiple design patterns already developed, in fact, books upon books have been written about them. In other words, there are far too many to discuss here. However, a few selective design patterns will be analysed to see if any can be applied to the developing SMSC System. The next three design patterns were selected on purpose in an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

attempt to find an appropriate design pattern to aide the architecture of the SMSC application. If a design pattern is applied properly then it can easily become the most beneficial part of the application. This is the reason why so much focus has been placed on design patterns.

1.4.1 Strategy Design Pattern

The Strategy design pattern is one of the classical design patterns that has been classified by the GOF as a behavioral design pattern. This particular design pattern is used for encapsulating a family of algorithms and separating them into their own algorithm. This makes them highly interchangeable at runtime. More simply put, an object and its behaviour are separated and put into two different classes [Garcia, 2000].

When should this design pattern be applied? Use the strategy pattern whenever [Tarr, 2000]:

- Many related classes differ only in their behaviour
- You need different variants of an algorithm

There are several advantages to using this design pattern. For one, it makes maintenance on a single object easier by separating that class into separate subclasses based on behaviours. The subclasses are known as a Strategy.

An example of an applied Strategy pattern is when a class wants to decide at run-time which algorithm it should use to sort an array. If there are lots of sort algorithms available then this task can become very sticky. However, if we encapsulate the different sort algorithms using the Strategy pattern then this task becomes relatively easy. A class diagram of this example has been provided in Figure 1.4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

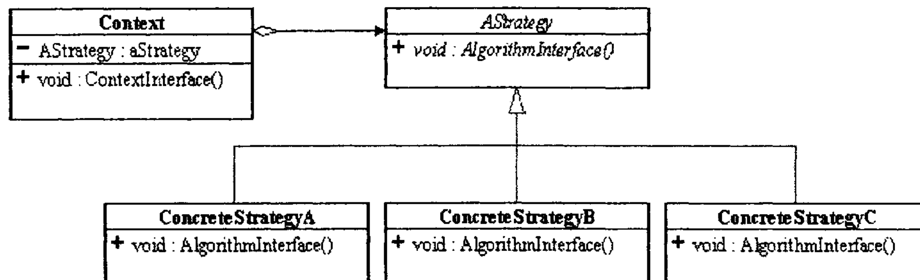

Figure 1.4: Strategy Design Pattern Class Diagram

1.4.2 Command Design Pattern

The Command design pattern is also a classical design pattern. The Command design pattern is for sending an abstract command to a class. In this case, the object that orders the command doesn't care who will handle the command. The command design pattern encapsulates the concept of the command into an object [Garcia, 2000].

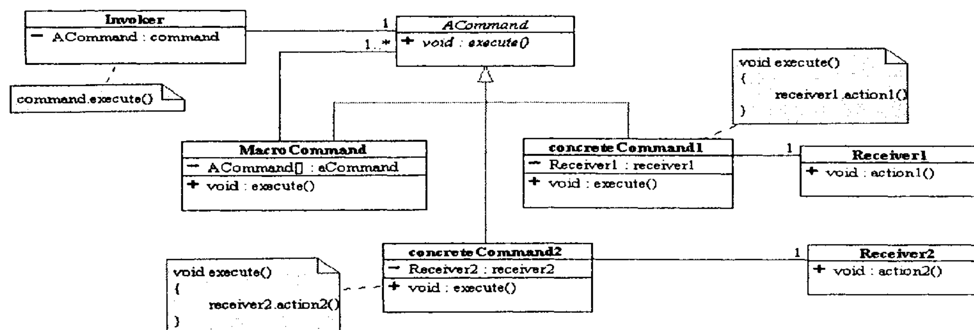

Figure 1.5: Command Design Pattern Class Diagram

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the object that orders the command doesn't care who the recipient is then who does? Well, the actual Command object will receive the command and distribute it to the proper object. One of the advantages of this design pattern is that all of the objects will be using a central Command object. Thus, if the recipient of a command changes then only one object needs to be updated. Figure 1.5 shows a Command pattern class diagram [Geary, 2002].

If we were developing a Graphical User Interface (GUI) that has more than one button and each button does a different action. For each button there will be a menu item that performs the same action. What is the most efficient way to develop an application with these specifications? Probably one of the most efficient ways to solve this problem is to create an action listener for all buttons and menu items [Geary, 2002] [Tarr, 2000]. This is actually the Java Swing solution and almost all object-oriented framework implement the Command pattern [Geary, 2002].

1.4.3 Model View Controller Design Pattern

Model-View Controller (MVC) is a design pattern that is widely used because of its architectural pattern. MVC is not a classical design pattern and it cannot be classified under the three categories presented by GOF. It is classified as a Architectural Design Pattern [Gamma et al, 1994]. MVC is good at distinguishing the separation between the user interface and application control.

The Model contains the core functionality of application components [Ping et. al, 2003]. It is used for representing low-level behavioural states. The Model should do all of the transformations on that state and effectively it manages the data of the state. The Model has no knowledge of either the View or the Controller.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

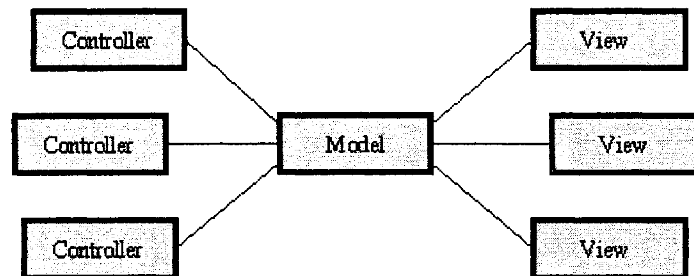

Figure 1.6: The Model/View/Controller architecture

The View is for visual display. It actually should not have any logic to process. A Model can have more than one View [Sundsted, 1998]. The View's purpose is to retrieve data from the Model via objects that were previously created from the Controller. It needs to be notified when the state changes [Sundsted, 1996]. The View has no knowledge of the Controller.

The Controller is in charge of the Model object. It will create any objects that the View will use and it also manages any requests. The Controller is the object that provides the means for user interaction with the data represented by the Model [Sundsted, 1996]. It will provide the means for information in the Model to change and it will also inform the View of the changed state. It too interacts with the Model via a reference to the Model object.

One of the most general forms of the MVC design pattern is displayed in Figure 1.6. In this example, there are multiple controllers, one Model, and multiple views. The benefit of MVC is that there is a clear separation between each of the components of a program. Also, the binding between the Model and the View is dynamic, which implies that it occurs at run-time, rather than at compile time [Sundsted, 1996].

One of the benefits of the MVC design pattern is that a lot of programming languages are built around this design pattern. Thus, if the programming language is built around MVC then MVC can easily be applied to a developing program. Java is among one of the many languages that use the MVC design pattern and it was carried over into the design of JSP, which is what will be discussed next.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The MVC design pattern could be used in a generic SMSC application by applying its superior architectural design to aide in the construction of the application. If MVC is applied then the user has the satisfaction of knowing that everything has a proper place in the architecture of the application. Thus, the application will have a solid generic background in the development stage and an easily maintainable architecture for the future.

1.4.4 Using Design Patterns in MS Design

Since the MS operates entirely different from regular PC's some various different design patterns exist. The Cascading Menu pattern, the Wizard Dialog pattern, or the Slide Show pattern is just some of the MS design patterns that can be applied. One will notice quite quickly that these design patterns are just as important as the classical design patterns.

The Cascading Menu pattern is based off of the MVC design pattern but it's a scaled down version. There is no Controller in this pattern. The View and the Model will communicate directly with each other. The View will render itself based on the current state of the Model [Hui, 2002]. Thus, the View is changing dynamically as the Model updates.

The Wizard Dialog pattern reflects current installation wizards that are driven by two buttons, "Next" and "Back". This design pattern is often useful because it will collect all of the user input via asking a series of questions before executing a command [Hui, 2002]. This design pattern will effectively replace a web input form that couldn't possibly fit on a mobile device's display screen. Instead of asking for the user's name and address all at once, the Wizard Dialog design pattern will ask for the user's name and wait for him to press the "Next" button before prompting him for his address.

The Pagination pattern is a design pattern based on breaking pages content up into small viewable pages that can be seen on the mobile device. Quite often pages are too large for the mobile device's display screen. The smaller pages will contain a subset of the complete content [Hui, 2002]. The user will go from page to page by pressing a key on the mobile device.

The final design pattern that will be discussed that is particular to the MS is the Slide Show pattern. This design pattern also mimics the desktop version because it's a series of screens that will automatically go from screen to screen without any user interaction. The sequence appears as it is programmed; users

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cannot control the sequence's pace or order [Hui, 2002]. Typically, there is a long enough pause between slides to let the user view the content on the screen before going to the next screen.

Which design pattern would suit the needs of a generic MS application for sending SMS messages? Two design patterns that do not apply are the Slide Show and Pagination pattern. The Slide Show is for displaying a series of screens on the mobile device that are automated. The Pagination pattern is used for displaying a lot of text on the screen. A generic MS application for sending SMS messages doesn't require these design features. It doesn't have a whole bunch of screens that need to be automated and it has no large amounts of text that need to be displayed. It might have some text to be displayed when a message is received; however, SMS text-messages are not allowed to be large enough to require the assistance of the Pagination design pattern. A generic MS application could use the Wizard Dialog pattern or the Cascading Menu pattern. Since the Cascading Menu pattern is for inputting information before performing a task then this would appear to be more appropriate. However, the Wizard Dialog pattern also effectively collects data and it has "Next" and "Back" buttons.

The generic MS application will apply the Wizard Dialog pattern and also the MVC design pattern. The Wizard Dialog pattern will provide structure for collecting the destination address and the text-message prior to sending or replying to a SMS message. This will be a useful design pattern because the user will have the ability to navigate through the use of the buttons. The MVC design pattern will handle the presentation logic and the business logic. It will actually encapsulate the Wizard Dialog pattern and govern its very existence.

1.5 Useful APIs for SMS Systems

To develop a generic SMS System one needs to use building blocks that are easily transferable to just about anywhere. This implies that the developing technology used is a crucial step in the development of the integral SMS System. It should be widely supported and also preferably as generic as possible, meaning that it shouldn't make any assumptions. Another way to look at the system being generic is if some piece of code was written for a mobile device but the exact same code worked on the PC without any alterations then that would be very generic.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Another component that will be investigated is Java Server Pages (JSP). These server pages are highly generic and transferable to any system. JSP is used for enhancing servlets.

1.5.1 Java Server Pages

Java Server Pages (JSP) was introduced by Sun Microsystems and is a fundamental part of J2EE. It was originally developed as an alternative to Microsoft's Active Server Pages (ASP). JSP technology is an open freely available specification developed by Java Community Process (JCP) [Mahmoud, 2003]. JSP is for dynamic webpages that easily elevate servlets to the next level. It also makes it easy to separate the static and dynamic parts of a webpage. Thus, the confidential systematic part of the webpage can remain hidden.

Before JSP and before dynamic webpages there were CGI scripts. CGI is often referred to as a first generation solution. The problem with CGI is that for every request a new script was required. Therefore, CGI was dynamic but it wasn't easily serviceable nor upgradeable. It also suffered from the vicious write, compile and deploy lifecycle.

Second generation mingled the static and dynamic parts of the web together. These solutions included web server vendors providing plug-ins and APIs for their servers [Mahmoud, 2001]. Problems with the second generation occurred when the solutions became platform dependent. An example of this would be Microsoft's ASP. In order to operate ASP the server must be operating a Microsoft server. This very well could be a drawback because the server shouldn't have to be Microsoft just because the software is Microsoft. There are some third-party plug-ins that will do the transformation from ASP back to an alternative server, such as the Tomcat server. The third-party software is called a "porting product".

Another second generation example is servlets. These use Java technology and can easily be used to write server-side scripts. The problem with servlets is that they still suffer from the same life-cycle that a CGI script suffers from. They suffer from the write, compile and deploy lifecycle [Mahmoud, 2001].

JSP is a third generation solution. It extends ASP but is compatible with any platform. This makes perfect sense because it was developed with Java technology and Java's motto is, "write once, run anywhere."

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In order to get JSP up and running on a web server one needs to conform to the JSP and servlets standards. One-way to do this is to download J2EE off of Sun's website (free of charge) and install it. A JSP page is nothing more than regular XHTML code (or XHTML-tags) with strategically placed bits of Java code throughout the page. How does this work? Well, when the browser is interpreting the JSP page, the web server will compile the code into a Java servlet. This servlet is nothing more than a second-generation solution. The servlet engine then loads the servlet class, which executes it to create dynamic XHTML to be sent to the browser [Mahmoud, 2001].

The Table 1.3 below will do a quick comparison of JSP to ASP and other solutions, such as, CGI scripts.

	JSP	**ASP**	**Other (i.e. CGI Scripts)**
Platforms	Most popular platforms	Microsoft Windows (other platforms requires the third-party software that was discussed earlier)	Most popular platforms
Web Server	Any web server	Personal Web Server or Microsoft IIS	Any Web Server
Scripting Language	Java	VBScript, Jscript	Perl
Customizable Tags	Yes	No	No
Reusable Cross-Platform Components	Yes	No	No
Database Integration	ODBC or JDBC	ODBC	-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	JSP	ASP	Other (i.e. CGI Scripts)
Dynamic HTML	Yes	Yes	No

Table 1.3: A comparison of JSP to ASP and other solutions

1.5.2 JavaPhone API

The JavaPhone API was developed by Sun MicroSystems and some of the key leaders in the telecommunication industry. It was developed to assist the market's demands in creating a unified telephone. This miracle telephone would be able to surf the Internet, manage your day, store telephone numbers, run Java Applets, etc [Green,2004]. Basically, it would do everything that a PC can do plus have the mobility of a cell-phone. Well, what better programming language to develop a telephone like this other than Java? It's platform independent and the unified telephone would fall under the, "Write once, Run anywhere," motto that accompanies Java.

The JavaPhone API has lots of diversity and many dynamic features. A developer could do almost anything in developing a JavaPhone application. They could even develop a telephone that supports a SMSC System. However, the JavaPhone API cannot be used directly to develop the SMSC Center. A JavaPhone Application is the entire phone's interface, not just one part of it. When it's used directly to develop a single feature, such as the SMSC Center, then we would effectively be using plain Java to develop this feature. JavaPhone API is the platform for developing applications and deploying dynamic information services on Internet screenphones or wireless smartphones [Knudsen, 2003].

1.5.3 .NET Mobile Web SDK and ASP.NET

Microsoft introduced the .NET-programming environment in the year 2000 at their Professional Developers Conference. The actual environment is known for being an entire framework. The .NET framework includes ASP.NET, which is in fact the next level of ASP. ASP.NET pages can be developed in any .NET language such as, VB.NET, C++, etc. [Sivakumar, 2001]. Figure 1.7 shows the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

.NET framework architecture. As the figure explains, the .NET framework is convenient because the developer only needs to write code once and it will work on any web browser or any mobile device.

The .NET framework eliminates browser checks and it decides whether to deliver HTML versus WML content based on the target it. There is no need to learn multiple languages because all that is required is ASP.NET, which implicitly implies that there is no need to learn WML. The .NET framework has a very nice drag and drop application development. Why wouldn't everyone want to develop pages in the .NET framework? Well, one is limited to Microsoft products and operating systems or servers. Also, when new versions of WML or HTML are released, one will have to wait until Microsoft announces support for the new version [Sivakumar, 2001]. Since one of the main concerns is generality then the .NET environment is not ideal for this SMSC.

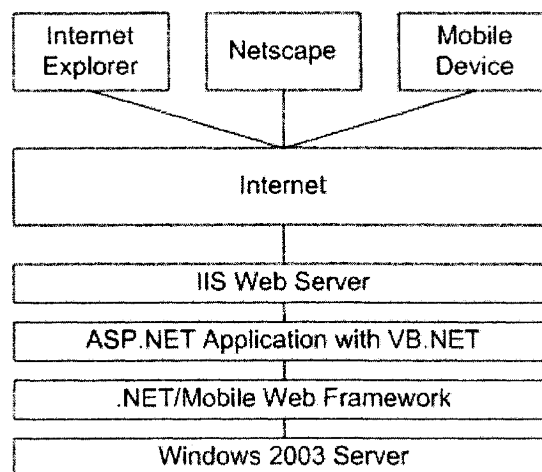

Figure 1.7: .NET Architecture

1.6 XHTML

Is Hyper Text Markup Language (HTML) generic enough for this SMSC? It is a fairly known fact that the HTML Document Type Definition (DTD) is web-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

browser dependent [Balani,2001]. This implies that the Netscape web-browser version of HTML might vary slightly to Microsoft's Internet Explorer web-browser. True, that the variances would probably be small enough that the developing SMSC could ignore it but its best not to. Thus, we need a better a solution.

XHTML is shorthand for Extensible Hyper Text Markup Language. It is basically one step above HTML. XHTML is combination of XML with HTML and is often written arithmetically like [LCTTP, 2001]:

XML + HTML = XHTML

What makes XHTML a better choice than HTML? Well, because it is part XML it then must conform to XML rules. This implies that all tags must be closed. For instance, if the <br> tag is used then it should be closed by doing either of the following:

</br>
or </ br> (recommended)

One of the other XML rules is that the tags should be in lower-case [Pemberton et. al., 2000]. In comparison to HTML where either of the following is valid:

or

XML is case-sensitive and thus one should always be extremely careful about capitals. XML also doesn't allow elements to be improperly nested. It's based on a First In Last Out (FILO) methodology. For example:

<i> this is valid </i>
<i> this is valid only in HTML </i>

XML attributes should always be in lower-case with the values in quotes. HTML doesn't care either way about this. One might place the attribute value in quotes but it is not mandatory. It's entirely up to the individual. The nice thing about XHTML is that all of this ambiguity is resolved because XML is a stricter language.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Why doesn't XHTML vary from web-browser to web-browser? The answer to that question is that because XHTML allows the programmer to define the Document Type Definition (DTD), whereas, HTML makes use of a predefined DTD. Thus, depending on the web-browser the HTML-tags will vary because the DTD is built into the web-browser.

XHTML declares that the DTD should always be the first line in an XHTML document [LCTTP, 2001]. The web-programmer could do one of two things. They could create a personal DTD and define all of the tags or they could use a predefined DTD. The drawbacks of the first method are that they would have to write all of the tags that their website might possibly use. For example, if a web-browser was attempting to interpret an XHTML web page but there was an undefined tag then it would display an error message. Creating personal DTD's can be tedious and time consuming. Luckily there is still the alternative method of using a public DTD. One simply specifies the URL in the first line of the page and the browser knows to reference that DTD whenever it discovers a new tag. The most popular spot for referencing a public DTD is at W3C's website. W3C is the maker of XHTML and thus their DTD is always current and always expanding. Therefore, if there is a new XHTML tag then as soon as it's been publicly approved it will be added to their DTD.

W3C has three different flavours of DTD's and they vary depending on one's needs. The first is called Strict and should be used for a clean markup that will have a presentation that is clutter-free. It should also be used with Cascading Style Sheets (CSS) [LCTTP, 2001]. The next DTD is called Transitional and should also be used for a clean markup that will have a presentation that is clutter-less. This one could be used if the browsers don't support Cascading Style Sheets. The third and final DTD is called Frameset. This is used when the XHTML page wishes to utilize HTML's Frame tag. The frame tag is used for partitioning the window.

The "Generic Web-Based SMSC" does use XHTML because it is obviously more generic than HTML. The SMSC also utilizes W3C's Transition DTD. Although, the Cascading Style Sheet is a useful feature it is not highly supported in all web-browsers and its interpretation varies from one web-browser to the next. The idea is to create as general of a SMSC as possible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.7 Summary

In this chapter, SMS was introduced as a text-messaging system. Although there are numerous other SMS Systems already developed most of them suffer from various different drawbacks. For instance, in Barry Harron's [Harron, 2002] thesis, he used Java technology, which is good because Java is a generic programming language but his application was only concerned with the PC sending a message to the mobile device. Therefore, he didn't develop a generic SMS System but a generic SMSC. Another drawback occurs in Vivek Malhotra [Malhotra, 2001] paper because the source code will only work on a Microsoft PC or Server. The reason being is that it is written using Active Server Pages and VBScript, both of which are not supported on every Operating System. The need for an entirely generic SMS System is evident and not just the SMSC but the whole SMS System. After all this is one of the key attributes to a successful SMS System. The other keys are for it to be cost efficient, easily deployable, and for it to work anywhere at anytime.

Half of the battle in developing generic SMS applications is choosing the proper developing platform and the other is choosing the proper architecture. The first half can be easily achieved through the use of Java technology. Java has been proven to be a generic language to develop applications for, either a SMSC or a mobile device. Since Java can be developed to run on any platform or almost any mobile device this makes it very cost efficient, easy to deploy and it can work anywhere at anytime. The other half has to pay close attention to design patterns. Applying a design pattern, such as MVC, will help immensely to make the application generic.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Designing a Generic Faculty Short Message System Center

2.1 Why Generic SMSC

Having a generic Short Message System Center (SMSC) is very crucial. How can a generic SMSC be achieved? Well, generality is most often achieved through a solid developing platform and having the proper design architecture. The developing platform should be deployable on any server. The design architecture is very important too because it is the backbone of the application. Essentially the developer needs to understand how to properly develop the application currently and in the future. Therefore, a design pattern, i.e., MVC, will be applied to the architecture of the SMSC application. The MVC design pattern will provide a good solid backbone for the SMSC application.

In this Short Message System Center (SMSC) a couple of things will appear to be different than other SMSC's but they will be rightfully justified. This is a web-based SMSC opposed to an application based SMSC. This makes the SMSC accessible from any PC at anytime. One might argue that this SMSC will now require an Internet connection, which is true but an Internet connection is already required for the sending process of an application-based SMSC. An application-based SMSC suffers from the drawback that it must be installed on every PC that the user wishes to send an SMS text message from. Thus, if one had a PC at home and another at work then they would have to install it multiple times. Another drawback occurs when the application needs to be upgraded or patched. Well, the n-user must do this on their own or it has to be built into the application to check for updates. If it is built into the application then it will be stealing the user's bandwidth while they're trying to utilize it for something more productive.

However, a web-based SMSC doesn't suffer from this at all. All of the SMSC content lies on the web-server and all of the updating is completed on one spot, i.e., the server. Then, the next time that they login to send a text-message the update will already be present. All of this updating and installation will be completed without any interactions from the user. A web-based SMSC naturally does not have to be installed on every PC. The user simply has to remember the URL, which could be made easy with the proper Domain Name Server (DNS). For instance, it's easy to remember where to go to check one's mail if it's simply by replacing "www" with "mail", i.e., mail.yahoo.com.

Since all of the installation will be completed on the server, one must be very careful choosing the server. The server that is chosen will largely depend

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on which environment the SMSC was written in. If the server is a Microsoft based server then one could use Active Server Pages (ASP) or Java Server Pages (JSP). If the server is anything other than Microsoft then ASP will not work without some help from third party software. However, JSP still works regardless of the server. Therefore, a "Generic Web-Based SMSC" should be written entirely in JSP. This chapter provides details on designing such a generic SMSC system.

2.2 Deploying a Web-Based SMSC using SimpleWire

The actual sending of the message could be handled any number of ways but for the purpose of this SMSC, SimpleWire can be chosen for the purpose of deploying it on the web. It is a SMS web-based company that has a web-server dedicated specially for sending and receiving SMS text-messages to and from mobile devices. It can also receive SMS text-messages from a PC or another server, which is precisely what the "Generic Web-Based SMSC" needs.

Initial registration is free with SimpleWire. When one registers with SimpleWire she is provided with a SimpleWire demo account. The account comes with a virtual phone number, subscriber's ID, and a subscriber's password. The virtual phone number is good for sending messages to the SimpleWire account. When a message is in the SimpleWire inbox it looks like Figure 2.1.

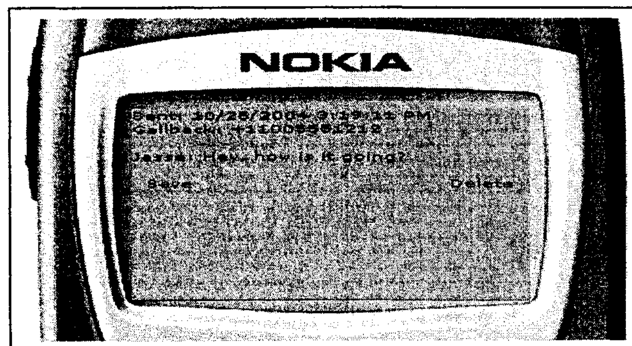

Figure 2.1: SimpleWire Inbox

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Upon registering for the SimpleWire account she is provided with a wide variety of SimpleWire SDK's for developing languages such as, C, C++, Java, ActiveX, etc. The Java SDK was chosen here because of its ability to operate on many different operating systems. Listing 2.1 illustrates how a plain-text message can be deployed using SimpleWire API. In order to make the text-message demo operate properly all one must do is replace the Subscriber's ID, password and virtual phone number with the ones that SimpleWire provides upon registering the account. Also, before running the application one must ensure that they're connected to the Internet. If there is no Internet connection present or if there is an error with the subscriber's ID, password, or virtual phone number then the output will simply say, "Message was not sent!" The demo is not sophisticated enough to distinguish between the different errors. However, SimpleWire does return an error code and one could easily interpret the error.

2.3 Deploying a Web-Based SMSC System

The SMSC System should be developed to deploy on any server. Java is a platform independent language and the same can be said for JSP. Therefore, a server requirement should be that it has to be able to run Java based applications. The other requirement of the server is almost redundant but it must be specified. The server should also be able to run on any platform.

If the developing language isn't constraining then the SMSC System should in theory work on any server. For this particular SMSC two servers were chosen. The first was Java 2 Enterprise Edition (J2EE) and is Sun's Java server. The second was Tomcat, which is part of the Apache project. Both are Java servers that naturally support the use of JSP and JSP's precedent, servlet's.

2.3.1 Deploying a Web-Based SMSC System using J2EE JSP

J2EE stands for Java 2 Enterprise Edition and is Sun Microsystems' version of creating a Java server. Although JSP will function on any server J2EE was chosen because it's cost effective and it's a good generic environment to develop the "Generic Web-Based SMSC" in. Plus, J2EE is 100% employable on any operating system. J2EE has nothing to do with the final output of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SMSC. The JSP written makes no J2EE assumptions and it will work on any server that supports JSP. One can acquire J2EE by downloading it for free from Sun's website. J2EE comes packaged in two different formats, one that can be unpackaged on a Microsoft based machine and the additional one that functions on other operating systems.

After installing J2EE one must create a new web application. The web application will be used for setting all of the customizable parameters for the "Generic Web-Based SMSC". It will also be used for compiling the SMSC. To create a new web application the following steps should be followed:

1. Start the "Deploytool"
2. Click File->New->Web Component
3. Click "Create new stand-alone WAR Module"
4. Choose an appropriate location for the WAR module
5. Under context Root enter something like:
   /faculty
   - There's no need to place an extra forward slash '/'
   - Whatever is entered here will be where the user has to go in order to send an SMS text-message, i.e.,
     http://localhost:8080/faculty
6. Click "edit contents" and add a JSP page
   - You must add a JSP page, even it it's blank
7. Click "Next"
8. Select "JSP" and click "Finish"

Now the JSP page can be viewed by following the above link.

Whenever a change occurs to the JSP page one must recompile the entire page. This will instruct J2EE to re-cache the servlet. Thus, when an individual enters the site they will be accessing the cached servlet. This will ensure a quick load of the JSP page because it will only be executed as opposed to being recompiled and then executed every time someone accesses

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it.

```
/*****************************************************************
 * Copyright (c) 1999-2001 Simplewire, Inc. All Rights Reserved.
 * Shows how to send a wireless message containing text in Java.
 *
 * Please visit www.simplewire.com for sales and support.
 *
 * @author Simplewire, Inc.
 * @version 2.6.1
 * @since  jdk1.2
 *****************************************************************/

import com.simplewire.sms.*;

public class send_text {
   public static void main(String[] args) throws Exception {
      SMS sms = new SMS();

      // Subscriber Settings
      sms.setSubscriberID("123-456-789-12345");
      sms.setSubscriberPassword("Password Goes Here");

      // Message Settings
      sms.setMsgPin("+11005101234");
      sms.setMsgFrom("Demo");
      sms.setMsgCallback("+11005551212");
      sms.setMsgText("Hello World From Simplewire!");

      System.out.println("Sending message to Simplewire...");

      // Send Message
      sms.msgSend();

      // Check For Errors
      if(sms.isSuccess())
         System.out.println("Message was sent!");
      else {
         System.out.println("Message was not sent!");
         System.out.println("Error Code: " + sms.getErrorCode());
         System.out.println("Error Description: " + sms.getErrorDesc());
         System.out.println("Error Resolution: " + sms.getErrorResolution() + "\n");
      }
   }
}
```

Listing 2.1: Simple Send Text Example using SimpleWire API

2.3.2 Deploying a Web-Based SMSC System using Tomcat Apache

Tomcat can be downloaded from the Apache website. Tomcat was developed for open source purposes to be used with servlets and JSP pages. Tomcat itself is the servlet container that is used in deploying servlets and JSP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pages on the web. The version of Tomcat that was downloaded for the SMSC System implements the Servlet 2.3 and JSP 1.2 specifications from Java Software [Apache, 2005]. Since Tomcat is linked to Java, the Java SDK is required in order for Tomcat to operate properly.

To get the Tomcat server operating the following steps should be carried out. Download the Windows zip file for a Windows platform, or download the tar for almost any other platform. Once the file has been uncompressed verify that a compatible version of Java SDK is present. Compatible versions are specified in the "RUNNING.txt" file that is present with the Tomcat download. If the current version of Java is non-compatible then download a new version from Sun's website. The final step requires that a new environment variable be created called, "CATALINA_HOME." This variable will store the path of the directory into which Tomcat has been installed. Now the Tomcat server is ready for deployment. To start the server, execute the startup script located in Tomcat's bin directory.

After the server has started its time to create a web application, e.g., the SMSC System. To create a new web application these basic steps should be followed:

1. Create a new directory inside the "webapps" directory, e.g., "SMSC_System"
    - This will be used for referencing the new web application on the web
2. Place a valid JSP page inside the new directory, e.g., "smsc.jsp"
3. Restart the Tomcat server
4. In a web-browser, reference the new web application by entering something similar to following:
    - http://localhost:8080/SMSC_System/smsc.jsp

If changes are made to the JSP page while the Tomcat server then simply refresh the webpage and Tomcat will automatically detect the change, recompile the servlet and redeploy it. All without having to bring down the server.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4 Composing a Java Server Page

We already know how a JSP page is compiled and we know partly how it is deployed. Well, then there is only one more question that needs to be answered. How can one develop a generic JSP page?

A JSP page is a combination of HTML and server side script. In this case, the server side script will be Java. The actual page can be composed in several different manners. For instance, one could separate the HTML and Java by placing all of the Java in a separate file. Or, another example would be to, place all of the HTML and Java in the same file.

By placing all of the HTML and Java in separate files then the personal methodology is completely hidden. The JSP page will handle most of the methodology and JavaScript will handle sometimes a little bit of it. Quite often individuals will actually go as far as to place his or her JSP page in a completely separate place on the server and most often it will be password protected. This ensures that the personal algorithms will not be comprised. Another advantage of this method is that the same algorithm can be applied to multiple other HTML files. Thus, the server side script is updated once and it affects multiple JSP pages. Effectively, the static Java code is becoming more dynamic. Please refer to Listing 2.2 and Listing 2.3 where this example has been illustrated with basic HTML and a simple JSP example.

```
<html>
<body>
<h1>What is your name?</h1>
<form action="SimpleFormHandler.jsp" method="get">
Name: <input type="text" name="firstName">
<br><br>
<input type="submit">
</form>
</body>
</html>
```

Listing 2.2: A simple HTML form

```
<html>
<body>
<%
// Grab the variable from the form.
  String firstName = request.getParameter("firstName");
%>
<%-- Print out the variable. --%>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
<h1>Hello, <%=firstName%>!</h1>
</body>
</html>
```

Listing 2.3: A simple JSP handler for the Basic HTML form

The downfall to arranging a JSP page in this manner is that it makes for possible tedious maintenance. For instance, if a programmer has been hired to maintain the JSP page but does not have access to all of the files then they will have to ask their supervisor for approval. This could very well become a waste of time if the supervisor is out of the office. Also, if multiple pages are using the same server side script then one must be very careful when updating the script. A slight change could alter other pages in a negative way making them inoperable. Now that a couple of different scenarios have been introduced. Let's look at three different methods in more detail.

2.4.1 An Enhanced SMSC Model

There are many different ways to compose a JSP page. We already know how a JSP page is compiled and we know partly how it is deployed. Well, then there is only one more question that needs to be answered. How can one develop a generic JSP page?

A JSP page is a combination of XHTML and server side script. In this case, the server side script will be Java. The actual page can be composed in several different manners. For instance, one could separate the XHTML and Java by placing all of the Java in a separate file. Or, another example would be to, place all of the XHTML and Java in the same file. Each of these methods has various different perks but they're only advantageous if the JSP page is simple [Seshadri, 1999]. The reason for this is that these methods encourage the developer to mix business logic with presentation logic. If the JSP page is complex then it can become a delicate job to modify it.

Other developing languages that encourage this style of employing dynamic pages are ASP and Hypertext Preprocessor (PHP) [Mercay and Bouzeid, 2002]. This style of creating dynamic webpages is known as Model 1 Architecture. The characteristic of this model is that of a collection of JSP files. In this model the JSP page is responsible for processing the incoming request and replying back to the client [Seshadri, 1999]. This model is notorious for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mixing business logic with presentation logic. Thus, the final JSP page will be infested with a combination of scriplets and Java code. This leads to a complicated and tedious job of maintaining the JSP page [Unger, 2000].

2.4.2 An Enhanced SMSC Model Based on MVC Design Pattern

The first model is effective at sending a SMS message, however, it lacks any type of structure. This second model proposes to gain immense amounts of structure in the design of the application by applying a design pattern such as, MVC. Applying the MVC design pattern to a JSP page is often referred to as rescuing the JSP and servlet world [Mercay and Bouzeid, 2002].

How is this MVC design pattern represented inside the design of this SMSC model? Well, in this case there is only one Model, one Controller and one View. The Model is created through the use of JavaBeans. The View is simulated with JSP pages. The Controller is a servlet. Why would one want to use a design pattern on a JSP page? Well, design patterns are one of the key elements in making an application generic [Sundsted, 1998]. They provide structure for current development and they offer an abstract view to solve a logical problem.

How does this all work? The Controller servlet is the front-end that handles all of the HTTP requests. It will also create any necessary JavaBeans or objects for the JSP. Finally, the Controller servlet will also determine which JSP page to forward the request to [Seshadri, 1999]. The Controller servlet is also used for computation intensive tasks. The JavaBeans are used for storing information. The information might be user information from a form or requested information. The JSP page will have absolutely no processing logic with itself [Mercay and Bouzeid, 2002]. It is used as a presentation layer. The responsibilities of the JSP page are to retrieve any objects created by the Controller servlet. It should only extract the dynamic content and place it within the static page.

This model is definitely an improvement from the Model 1 Architecture. It separates the business logic from the presentation logic. It also has a design pattern to base the intended construction around. The design pattern will help eliminate random logic in the presentation layer. Or in other words, as the site grows and expands this problem will effectively be eliminated if the design pattern is followed. The drawback to this model of the SMSC is that the MVC

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

design pattern does not have to be followed. It still leaves the door open for a lot of misplaced code [Unger, 2000].

2.4.3 An Enhanced SMSC Model Based on MVC Design Pattern using Struts

Struts is an open source initiative sponsored by the Apache Software Foundation. It is based around the MVC design pattern. It was actually developed to encourage the MVC design pattern within a web application's presentation layer [Coen and Nanduri, 2003]. Struts uses JSP to help achieve the presentation layer. Struts is often referred to as a generic Controller servlet in the MVC design paradigm.

The Controller servlet is built right into the Struts framework. The Struts API comes with several classes that are created by the Controller servlet. The generic Controller servlet will provide the initial entry point for all HTTP requests routed to Struts [Mercay and Bouzeid, 2002]. As with the previous model, the generic Controller servlet will automatically create JavaBeans based on request parameters. One of the things that distinguish Struts from the previous model is that the Struts framework has a built-in implementation of the Controller servlet. In the previous model, the developer has to build the Controller servlet.

In Struts, the model is represented as one or more JavaBeans. As with the generic Controller servlet, Struts provides a wide variety of built-in classes for handling the Model, e.g., the Action, ActionForm or ActionError classes. The JavaBeans of the Model are typically represented by one of the following forms [Mercay and Bouzeid, 2002]:

1. Form Beans: Holds any attribute that was passed either on URL or in a POST.
2. Request Beans: Holds information needed to generate HTML, XHTML, XML, etc.
3. Session Beans: Holds session information that persists between two HTTP requests by the same user.

In the previous model's case, the developer is not provided with any predefined JavaBeans and must build these objects from scratch.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The View is basically represented the same as in the last implementation's case because both use JSP. The only minor change is that Struts provides tag libraries for inserting the dynamic content. The libraries will assist in making the presentation layer well-formed and avoid using any Java code in the View. The libraries are [Mercay and Bouzeid, 2002].

1. HTML: Helps create well-formed HTML tags
2. Beans: Assists with manipulating Beans
3. Logic: Implements logic constructs based on bean values
4. Template: Handles page templates

Struts handles all of its request-to-action mappings by reading a configuration file called, "struts-config.xml." Each mapping defined in this file causes an instance of the ActionMapping class to be constructed and loaded [Agerwai, 2003]. A mapped object is related to an Action class that implements it. Optionally an ActionForm bean can be used to store the request's data form.

Struts offers an innovative way of representing the MVC design pattern. The actual Struts framework makes it unarguably easy to develop and maintain Enterprise Applications. The tag library comprises the most robust presentation described [Unger, 2000], which makes developing a generic application safe. However, Struts does have a few drawbacks. For instance, if the Enterprise Application is complex then the struts-config.xml file can become complex and difficult to maintain. It has been suggested that Microsoft Visio or StrutsGUI will help organize the struts-config.xml [Coen and Nanduri, 2003].

2.5 Summary

This chapter proposed several ways to deploy an SMSC. It investigated how design patterns could be applied to the SMSC to make it more robust and tidy. The first version proposed was very unkempt and it required some work. In the following version the MVC design pattern was applied to the SMSC architecture and this helped immensely with the organization of the SMSC. The third and final version suggested Struts to aid in the development. Struts is very useful since it automatically applies the MVC design pattern. It also proved to be useful because the components of the MVC design pattern are already

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

developed in a generic way and it is manipulated to meet one's needs. For instance, Struts comes with action classes to handle the "Controller" part of the MVC design pattern. This prevents the developer from starting the MVC design pattern from scratch and quite possibly making a critical design error. It also makes the MVC more object-oriented since this design pattern can be quickly applied through the use of objects.

On top of developing the SMSC in three different methods this chapter also discussed feasible servers for the SMSC to live in. Two servers were discussed and both are suitable for the "Generic Web-Based SMSC". Both servers discussed are Java based servers that will support JSP and servlets.

Lastly, this chapter discussed a Java API called, SimpleWire. SimpleWire is the server that will be delivering the SMS message to and from the SMSC. SimpleWire was chosen because it was found to be effective and it met the SMS standards. A service had to be chosen and this is a good choice. SimpleWire supports Java and it guarantees service to the major global leading mobile devices.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. Implementing a Generic Faculty Short Message System Center Using Various MVC Design Pattern Implementations

With so many SMSC's already built and deployed the need for another almost isn't there. However, it is still evident that the need for a new SMSC is required [Harron, 2002]. The previous SMSC suffer from a lack of generality, see Section 1.7. This chapter develops a generic SMSC based on MVC design pattern. There are many reasons why the MVC design pattern promotes generality.

In this chapter, three different implementations to the generic SMSC application are presented. One that utilizes just JSP technology and two that take advantage of the MVC design pattern. Their consequences are discussed in the chapter conclusion.

3.1 SMSC using JSP

In this model, the business logic has been mixed with the presentation logic. It makes the SMSC incredibly easy to develop, however, it is not the greatest for readability. Appendix A.1 illustrates the server-side JSP page that is used for viewing and sending the message. As one can see, the XHTML code is intertwined with the Java code.

The JSP page in Appendix A.1 is laid out as follows. First, the title is defined and outputted to the screen. Then error-checking is executed amongst the XHTML output. Further down in the interpretation the logic behind sending a message has been inserted and is in fact processed. To send a message, one simply creates a new SMS object, which is in fact a SimpleWire object. SimpleWire requires that a Subscribers ID, password, the receiving phone number, and lastly the message. After the SMS object has been initialized then the SMSC makes a call to msgSend and the SimpleWire server will deliver the message. Upon sending the message the SMSC verifies that it was sent successfully by making a call to isSuccess. The SMSC will display a verification of the message's status.

This method of deploying the SMSC has lots of generality. As we already know, JSP and XHTML are standards that cover multiple platforms. Is the XHTML in Appendix A.2 produced by this SMSC well-formed? It is considered

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to be well-formed by W3C XHTML validator's standards and therefore can be considered as generic as XHTML [McLaughlin, 2000]. This model is a generic version of the SMSC, however, it suffers from a lack of structure. It simply places all of the business logic and presentation logic into one jumbled up mess. The "Generic Faculty SMSC" has been captured in Figure 3.1. This method of composing the SMSC lacks structure because it doesn't have a design pattern [Seshadri, 1999].

Figure 3.1: Screen shot of the SMSC using just JSP

3.2 MVC SMSC using JSP

This model attempts to learn from the previous model's flaws while attempting to build off of its pros. Once again, the strong points of the first model were it uses a universal developing language JSP, the produced output is the strictly standardized XHTML and its generality. It lacked structure. How can structure be added to this SMSC? Well, the most efficient way is to apply a design pattern, such as, MVC. The MVC design pattern specifies where the business

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

logic and presentation logic should be placed [Sundsted, 1996]. It also frowns upon mixing the two. Figure 3.2 illustrates a block-diagram of this method and it will be discussed in subsequent paragraphs to follow.

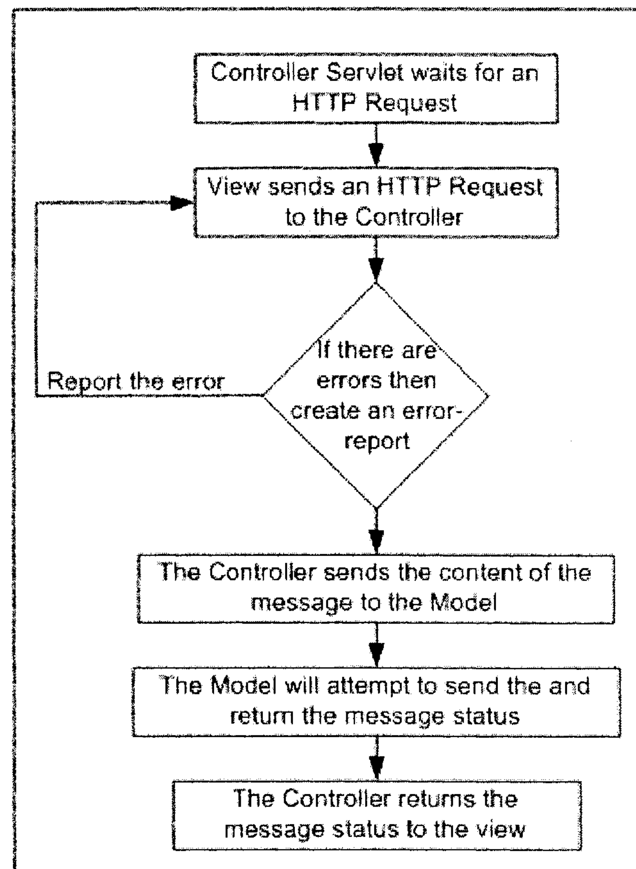

Figure 3.2: The MVC Flow Diagram of the SMSC

The MVC design pattern is broken into three parts, that is, the Model, View and Controller. This version of the SMSC naturally uses the same distinctions. The Model, View and Controller were called "Model.java", "View.jsp" and "Controller.java", respectively. The three can be located in Appendix B.1, Appendix B.2 and Appendix B.3, respectively. In this model, the SMSC starts with the Controller. The Controller checks for any http requests, if none then it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

forwards the browser to the View and idly waits for a request. Once there is a request, then it does some error-checking and it will attempt to send the message by informing the Model to send the message. If there was an error, either with the phone number or with sending the message, then the Controller will forward the error onto the View so that the user can correct the error. Once again, if there was no error and the message was sent successfully then the Controller will inform the View to display that. A UML diagram has been provided in Figure 3.3 to show the associations between the different segments of code.

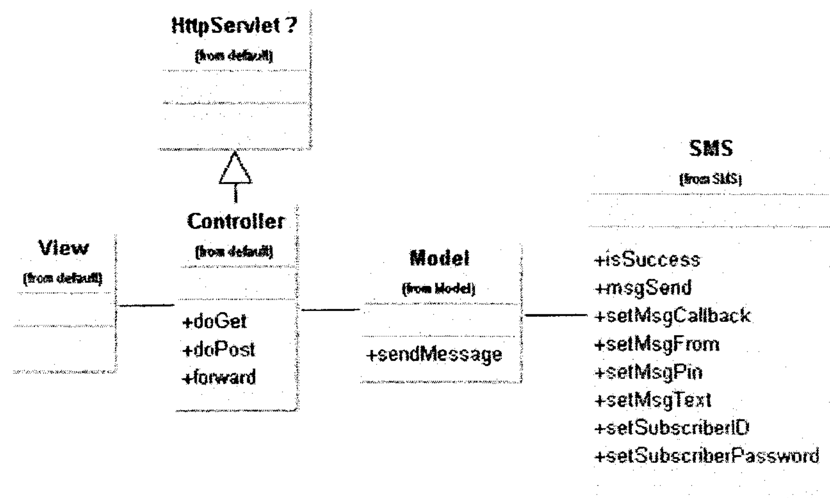

Figure 3.3: The MVC UML Diagram of the SMSC

The Controller's job is a really important one and everything in the SMSC revolves around it. In this model, the Controller is a servlet and it's superclass, or parent, is the HttpServlet class. The HttpServlet class is part of the J2EE bundle. The Controller overrides two of the parent's methods, both of which are for retrieving the user information from the View. The methods are called, "doGet" and "doPost". The latter simply makes a direct call to "doGet". doGet will do some error-checking to make sure that the user inputted a correct phone number and a message. If an error is found then it will store the error-message in the error-variable and call the "forward" method. The "forward" method will redirect the browser back to the View. However, if there were no errors detected then the Controller will proceed to give all of the heavy processing to the Model. The Model will return a message stating if the message was sent

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

properly or not and the Controller will pass this information onto to the View so that the user can see it.

The View looks similar to the previous model. There are a few significant changes though that should be pointed out. First, the error-checking is no longer executed from the JSP page. In terms of error-checking, the View checks for an error in the error-variable and displays one if there is one found. The XHTML-form is almost the exact same except for the value of the action-attribute in the form-tag. The previous model required this attribute to be the same JSP because it had all of the error-checking and sending logic. In this model, the value of the action-attribute should be the Controller servlet. The last change is that the sending logic has been moved to the Model.

The Model is a JavaBean. This JavaBean creates a new SMS object that will be used to communicate with the SimpleWire server. The JavaBean will inform the SimpleWire server of the Subscribers ID, password, the receiving phone number and the message. This differs from the previous model where the JSP page did all of this. The Model will inform the Controller of the sent message's status.

This model builds off of the previous models generality. It uses the same generic developing languages, such as, Java and JSP. It also has a distinctive generic output of XHTML. This model exceeds the previous attempt because of its structure that has been provided through the use of the MVC design pattern. If the design pattern is followed when developing the SMSC it will have lots of structure [Shin, 2003]. However, how can the design pattern be enforced? It's really optional and if there are multiple developers working on the same project then one of them might not understand MVC. If they don't understand the design pattern then how are they supposed to follow it?

3.3 MVC SMSC Implementation Using Struts

In the previous two models, the flaws were fairly obvious. The first didn't have a design pattern and the second had a marvellous design pattern but no means to enforce it. This model will also use the MVC design pattern and it will enforce this structure with Struts. This implementation models the MVC design pattern with a Model JavaBean, a Controller servlet, and a View JSP page. Struts was built for this very purpose [Cavaness, 2002]. If the SMSC doesn't have a Model JavaBean, Controller servlet and View JSP page then Struts will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not deploy. Thus, this enforces the design pattern. All of this is imposed through the "struts-config.xml" file that can be viewed in Appendix C.4. Struts checks this XML file to see what the Model is called and where it's located. Please refer to Figure 3.4 for the UML diagram that explains the Struts relationships.

As mentioned earlier, this model doesn't start with the Controller. It starts by interpreting the "struts-config.xml" file. This XML file will instruct Struts as to the whereabouts of the Model and the Controller. Struts uses this file to define the servlet and the JavaBean. It also explains what to do when the user clicks "send" from the View. There should be an "action-mapping" for every XHTML-form in the JSP page. The action-mapping will define which Action servlet to use and it also specifies which Form bean to reference. After the interpretation of the XML file, the View is loaded and the SMSC waits idly for an http request.

When there is a request, Struts will use the servlet that was defined in the "struts-config.xml" file to handle the request. In terms of the MVC design pattern, this is known as, the Controller, which can be viewed in Appendix C.1. Unlike the previous model, the Controller doesn't do any error-checking. The Controller simply retrieves the information that the user specified and passes this information onto the Model. If this was all successful then the Controller will forward a "success" to the View, or, a "failure" for the opposite case. In the configuration XML file, different JSP pages can be specified for a "success" or a "failure". Sometimes a special-case has to be handled differently.

The error-checking has been passed onto the form-JavaBean, please see Appendix C.3. In Struts, every XHTML-form must have it's own JavaBean, which should be filled with "get" and "set" methods for every attribute in the XHTML-form [Cavaness and Keeton, 2003]. On top of the "get" and "set" methods this JavaBean can also throw Struts-errors. In the "struts-config.xml" file, one can specify where the errors are located. When a Struts-error has been thrown then the View will check the specified location for the error-code, please see Appendix C.6. Next to every error-code is an appropriate error-message, which the View will automatically display.

The Model, in terms of sending the message, is the exact same as the previous SMSC. Since, the Model is a JavaBean, there was no need to make any modifications to it. After all, one of the main features of a JavaBean is that they are pluggable [Deitel and Deitel, 1999].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

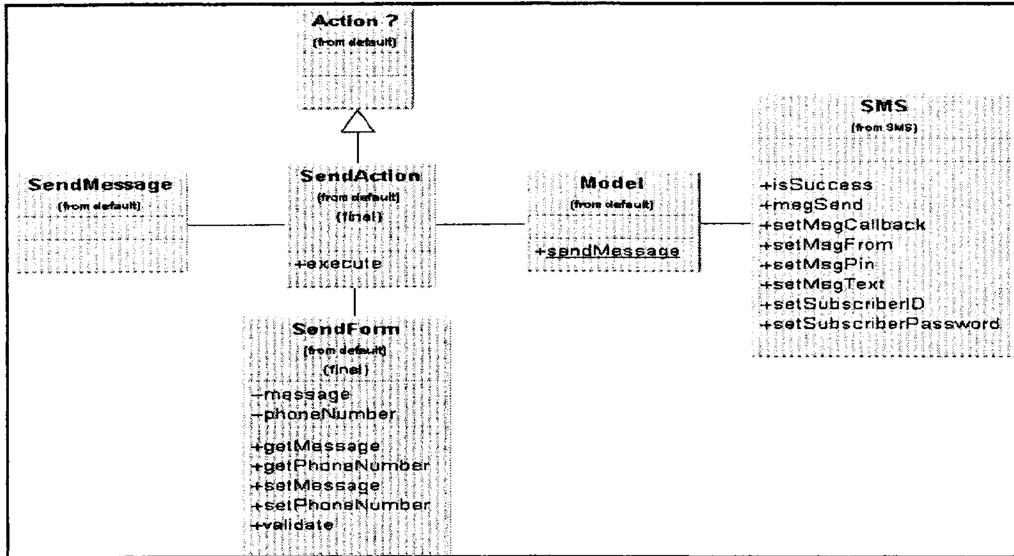

Figure 3.4: JSP Struts UML Diagram

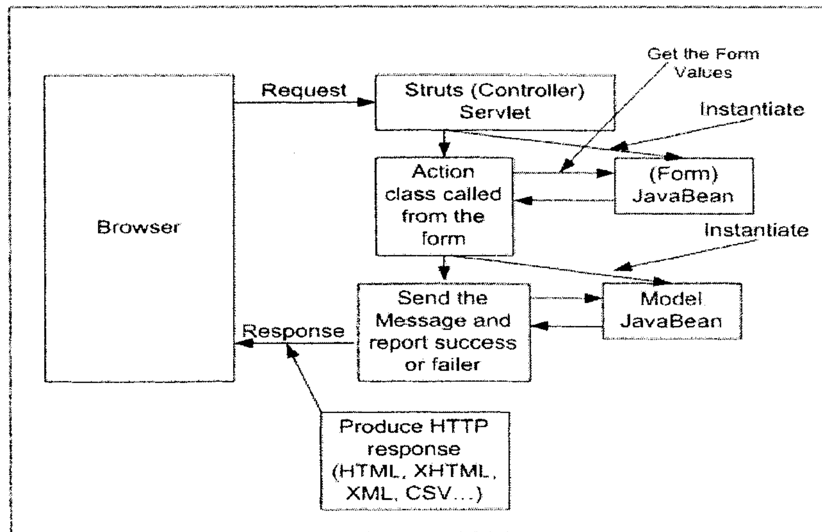

Figure 3.5: JSP Struts Flow Diagram

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Not only does Struts enforce the use of the MVC design pattern through the use of "struts-config.xml" file but it also has a few catches for the View too. In order for a form to operate properly in Struts, the Struts' tags must be used [Hightower, 2004. The DTD to the Struts tags are defined in the opening lines of the JSP page. The only drastic change between a Struts-tag and an XHTML tag is that the word "html" is placed at the beginning of every tag. One other minor change is that Struts has changed the tag-attribute-name from "name" to "property". Other than that, the View looks just like the previous model.

How does the Struts implementation fare up against the others in terms of generality? Well, the most important thing to observe is that Struts is operating on the same server that the other two methods operate on too. The other thing is to see if the output from the View is well-formed or not. This is especially crucial because the JSP page was using personal Struts tags that would be considered not well-formed. One will notice by checking Appendix C.5 that Struts does a conversion and the output is most certainly well-formed by W3C's standards [Dudney and Lehr, 2004]. Struts is just as general as the other two implementations.

Struts has the generality, and it not only uses a design pattern but it enforces one. Is this a sufficient enough implementation for the "Generic Web-Based SMSC? It most certainly is, however, one could enhance their SMSC further by using XSLT with Struts.

3.4 Summary

This chapter implemented three versions of the "Generic Web-Based SMSC". All three versions are extremely generic and function well so one gets the chance to become fussy about design issues.

The first version illustrated shows the JSP mingling with the XHTML. This has been proven to be bad for current and future maintenance. Following the model of the MVC design pattern, the presentation logic, i.e., XHTML, should be separate from the business logic, i.e., JSP. In other words, this version should not be followed because it promotes poor design architecture.

The second version implemented used the MVC design pattern to help organize the architecture. The design goals of the MVC design pattern are to separate the presentation logic from the business logic. This proved to be quite feasible and the outcome was extremely readable. Providing that the developer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

knows about MVC and understands it then this version is quite feasible. Even if the developer does understand MVC this version still suffers from the disadvantage that the developer has to continually build the MVC from scratch. Why can't MVC be bundled in objects so that one can quickly implement it?

The third and final SMSC version implemented used Struts. As mentioned earlier, Struts automatically applies the MVC design pattern. It not only applies it but it also enforces it. This aides immensely in the amount of time it takes to apply the MVC design pattern. Plus since most of the design pattern is bundled in the Struts' objects, it can be quickly applied to the SMSC. Struts is a very object-oriented package that offers the MVC design pattern as objects that can be implemented quickly.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Developing a Generic Mobile Station

4.1 Introduction

The wireless device is most often referred to as the Mobile Station (MS). An MS could be a cell-phone, PDA, Pocket PC, etc. Our general concern is for MS's that are capable of receiving and sending SMS message. An MS is now the preferred term for describing the mobile device that a subscriber uses to communicate with a mobile network. The MS can also be described as a wireless terminal that is capable of receiving and sending alphanumeric messages [Malhotra, 2001].

There are many ways to design MS stations, however, any generic MS application should pay close attention to design patterns. These patterns play a crucial role in developing any application and this definitely includes MS applications. Some of the traditional design patterns can still be applied to an MS application but MS applications have slightly different design issues than regular applications. This is because mobile devices suffer from different drawbacks like limited viewing space and/or limited power. Therefore, if there are different problems then various different patterns will exist for the MS. Where there are patterns there are also design patterns to help correct these structural problems. Some famous MS design patterns consist of the Cascading Menu pattern, the Wizard Dialog pattern, or the Slide Show pattern [Hui, 2002].

In addition to the usage of design patterns, the programming language used for developing MS stations is also crucial for having a generic mobile platform.

4.2 MS General Architecture

The MS's responsibility is to send and receive SMS messages. To start the process, the MS must be turned on and within broadcast range. When the MS is initially turned on it will register with the network. Network registration takes place by a text-message going out to the Visitor Location Register (VLR), which in-turn will contact the Home Location Register (HLR). The HLR will verify that the MS should have network coverage or not. At this point, the HLR will inform the VLR of the MS's network coverage status. It will also check to see if the MS has unsent text-messages that need to be sent to the MS. If there are any then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it will inform the SMSC that the MS is now recognized by the mobile network to be accessible and thus the message(s) can be delivered [Harron, 2002].

After the MS is registered with the network it's ready to send and receive text-messages. First off, to send a message the MS transfers the message to the Mobile Station Center (MSC) via it's operating frequency. The send process is demonstrated in Figure 4.1. All MS operating frequencies are outlined in the wireless technologies document [Alphonse and Rajkotia, 2002]. The MSC is used for switching connections between MS, or between MS and the fixed network [Malhotra, 2001]. The fixed network could in fact be any IP address. The MSC will ask the VLR for confirmation that the message does not violate any restrictions, i.e., the MS could have some restrictions placed on it or a country might not allow text-messages. If the VLR okays the message then the MSC will forward the message onto the SMSC. The SMSC will deliver the message, and send an acknowledgement back to the MSC that the message was delivered successfully [Harron, 2002]. The MSC will forward the acknowledgement to the MS so the user can see the message's status.

The MS receives a text-message from the SMSC but there is a lot more going on than a simple message being received. All of the steps can be viewed in Figure 4.2. The SMSC starts this process by contacting the HLR for the whereabouts of the MS. The HLR will have routing-information stored in it's database. After the SMSC has the routing-information it will send the message to the MSC using the forward short message operation [Harron, 2002]. It is the responsibility of the MSC to retrieve the subscriber information from the VLR. Quite often the VLR will force the MSC to authenticate itself. Once the subscriber information has been retrieved the MSC will send the message to the MS. Finally, the MSC will send the message status back to the SMSC. If all of the steps were successful then the SMSC will remove the message from its queue, otherwise it will attempt to resend it at a later date.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

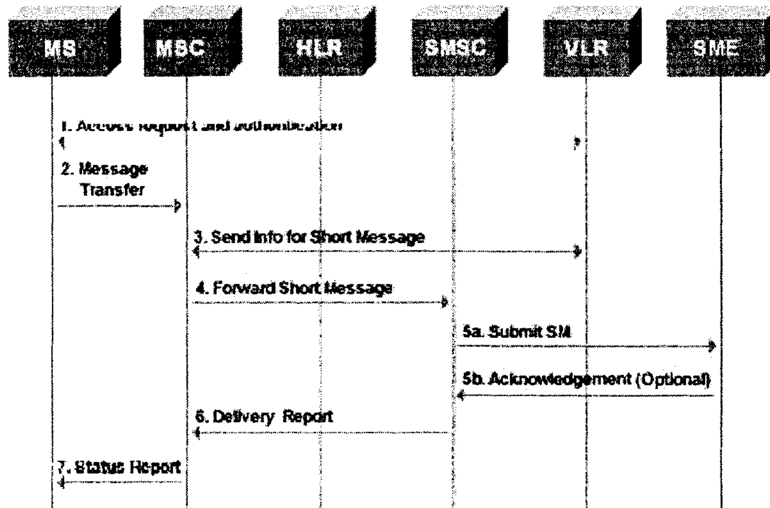

Figure 4.1: Sending a SMS message from an MS to an SME

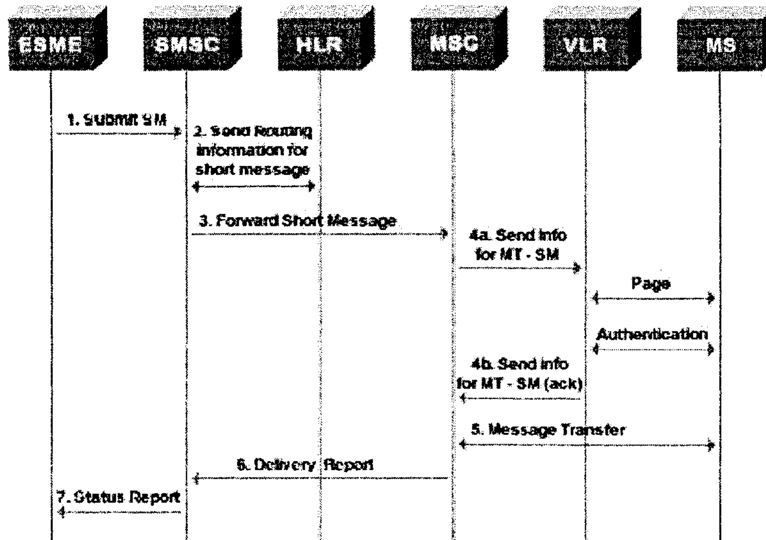

Figure 4.2: Sending a SMS message from an ESME to an MS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3 Choosing the Right Test-beds for MS Stations

Test-beds and their toolkits play an important role in developing successful MS applications. There are numerous MS available on the Internet. With so many MS already out on the market it becomes hard to find the proper one. There is at least one available for each mobile device. This chapter selected two MS's that claimed to be effective test-beds and they are the Nokia MS http://www.forum.nokia.com/main/ and Microsoft's Pocket PC MS http://www.microsoft.com/downloads/details.aspx?FamilyID=359ea6da-fc5d-41cc-ac04-7bb50a134556&displaylang=en for comparison.

Nokia's MS is built from J2ME technology. This gives them the advantage of knowing that any software built that is developed for the Nokia MS will also operate on over 50 company's MS [Knudsen, 2002]. The reason for this is that over 50 mobile companies support the J2ME runtime environment, MIDP, which gives it the advantage of being very stable on a wide variety of mobile devices. The Nokia MS supports MIDP 2.0 and MIDP 1.0. MIDP has already been adopted as the platform of choice for mobile applications and is deployed globally on millions of mobile devices [Knudsen, 2002][Ciucci et. al, 2002].

In comparison, the Microsoft Pocket PC's MS is built from .Net CF Framework technology. This implies that it will only operate on mobile devices that support Microsoft Windows CE and just this operating system. This is a huge drawback since the Microsoft Windows CE only has a small portion of the market [Yuan, 2003] [CM20143]. Thus, when an application is developed for the Microsoft Pocket PC platform it cannot be highly deployable.

If one were to develop a MS then it would be wise to develop it from J2ME technology. This not only assists with making the MS wider spread and more deployable but a developer has the confidence of knowing that he doesn't have to test his MS application on multiple mobile devices. If an MS application operates properly on one mobile device that supports the J2ME runtime environment, MIDP, then it will operate properly on all mobile devices that support MIDP. This is definitely a generic feature about J2ME that one cannot ignore when they are developing an MS or MS applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4 Traditional MS SMS Standards

With millions upon millions of SMS messages being sent and received the demand for a SMS standard is almost required. All of the SMSC's shouldn't vary by much or the VLR, HLR, MSC or MS wouldn't know what to do with the message. Or if the MS's all had their own version of a SMS message then two different MS's couldn't send SMS messages between each other. Luckily, there are standards that are enforced. The SMS standard document is called, "TIA/EIA-637-B" [TIA, TIA/EIA-637-B] and is often referred to as the technical requirements that form a standard for a SMS message [Alphonse and Rajkotia, 2002]. This document only deals with sending and receiving a SMS message and does not concern itself with the quality or reliability of the SMS message.

A SMS can send and receive message in either the analog or the spread spectrum (CDMA) mode. Figure 4.3 shows a simplified view of the network carrying SMS, including only a single SMS message relay point [Alphonse and Rajkotia, 2002]. The "TIA/EIA-637-B" [TIA, TIA/EIA-637-B] article clearly outlines in detail the three different SMS network-layers that were explained in Chapter 1.2.

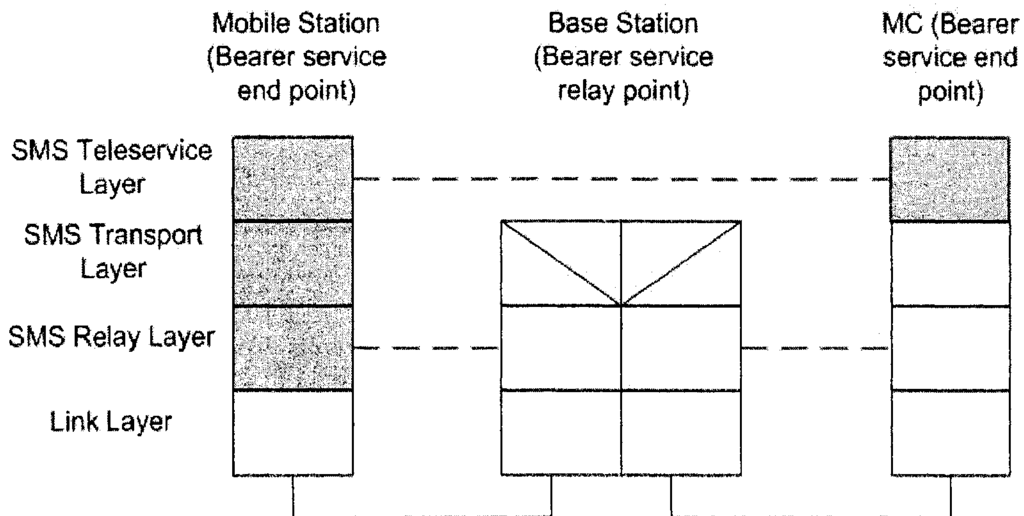

Figure 4.3: SMS Protocol Stack [Alphonse and Rajkotia, 2002].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.5 Developing the MIDP of the MS Station

The Mobile Information Device Profile (MIDP) is a crucial part of the J2ME. It is the backbone when running a J2ME application on a mobile device. It is, in fact, a standard Java runtime environment for today's most popular mobile information devices [Knudsen, 2002]. The MIDP does not deploy a J2ME application all-alone. MIDP was designed to run on top the Connected Device Configuration (CLDC), which is described in JSR-139 [Ciucci et. al, 2002]. An overview of CLDC was presented in Chapter 1.3.5. As with most developing Java platforms, the MIDP was defined through the Java Community Process (JCP) under JSR-118 and JSR-037. This gives MIDP the advantage over other runtime environments because over 50 companies contributed in the production of this [Ciucci et. al, 2002].

MIDP has been around long enough to have two versions developed. This is the reason why there are two Java specifications in the JCP for MIDP. The first version just offered the required core functionality so that a mobile application could work properly. It wasn't very sophisticated but it was still highly deployable on multiple mobile devices. The second version was based on the first version and fortunately it's also backwards compatible. This implies that all of the applications that were written for the first version are automatically deployable on a mobile device that supports the second version of MIDP. The second version offers an enhanced user interface, multimedia , more extensive connectivity, and end-to-end security. These features have been outlined in greater detail below:

- User Interface: The second version of MIDP enhanced the user interface by offering a better foundation to graphics. Naturally, it had to be optimized for the small display size, have varied input methods, and offer other native features of modern mobile devices [Knudsen, 2002]. MIDP is capable of offering navigation through the use of the mobile device's keypad, touch screens, and small keyboards. The enhanced user interface also manages local data securely and MIDP applications are executed from the mobile device.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Multimedia — MIDP is quite efficient at handling Multimedia applications [Ciucci et. al, 2002]. Through the use of a high-level UI API, developers have utmost control over graphics when they need it. MIDP multimedia comes with built-in audio so it makes it easier for a MIDP application to have sound. This will make it straightforward for a user to notice an error if the MIDP application "beeps" at them.
- Extensive Connectivity — The extensive connectivity that was offered in the second version of MIDP was really crucial in order for MIDP and J2ME to survive. It not only offers leading connectivity standards that include HTTP, HTPPS, server sockets, serial port, etc. [Knudsen, 2002] but it also made it possible for a MIDP application to send and receive a SMS or CBS message. With so many applications requesting the transmission of a text-message via a mobile device it's no wonder that the MIDP specification was enhanced to offer this service.
- End-to-end security — End-to-end security offers MIDP applications the chance to use existing secure options when required. For example, a MIDP application that needs to send or receive encrypted data can now do this because MIDP now supports https and thus it would also support SSL.

4.6 Reviewing some Essential J2ME API

Why was J2ME chosen for developing a generic Mobile Station application for sending and receiving SMS text-messages? Well, J2ME is in a class of its own. There are only a limited number of choices when selecting a programming language to develop a mobile application. While, there are competitors to J2ME but J2ME surpasses them in generality and in the case of SMS, functionality too. J2ME's opponent, .Net CF, doesn't quite have the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

generality that J2ME has because J2ME is platform independent. .Net CF is not platform independent because .Net CF applications only operate on Windows CE Operating System. This gives J2ME more versatility and also an advantage over other mobile programming languages.

In order to create a generic MS application the J2ME developing language must be utilized. What does J2ME have to offer in terms of providing the developer control over a mobile device? The J2ME API is quite extensive and very thorough when it comes to a handling a mobile device. There are certain classes that have to be utilized in order for a J2ME application to operate a mobile device. The classes will be outlined below:

- MIDlet — The MIDlet class is used by all of the applications because this allows the application management software to control the MIDlet. Once the application management software has control then this allows it to create, start, pause and destroy a MIDlet. A MIDlet is a set of classes designed to be run and controlled by the application management software via this interface [Sun, J2ME-API]. It is the responsibility of the application management software to maintain which MIDlets are active. This is achieved by using the start and pause states. It is possible for a MIDlet to change it's own state but it must inform the application management software of such a change.

- Alert — The Alert class is used for relaying information back to the user. Typically, a message, or an alert, would only be displayed on the screen for a short period of time. Alert should only be used for displaying error(s) or other exceptional conditions. If the Alert is very important and it is crucial that the user reads it and acknowledges it then the Alert's display time could be set to infinity. In this case, it would be the responsibility of the user to dismiss it.

- Command — The Command class is purely a construct class that merely encapsulates the action but the actual action is handled through an interface [Sun, J2ME-API]. If a J2ME application wishes to control a button on a mobile device then it would use this class. Please refer to the example provided in Appendix D for a further example.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Display — Use the Display class to handle all input and output to the mobile device. There can only be one instance of the Display class per a MIDlet. Since the Display class doesn't have a constructor, one obtains the instance of Display by calling, "getDisplay" method. Typically, the user interface objects that are viewed on a display device are contained within the Displayable object.

- Displayable — The Displayable class encompasses all objects that can be displayed. A Displayable object have a title, a ticker, zero or more commands and a listener associated with it [J2ME API]. By default, a new Displayable object is not visible, the title is null, there are no commands present and lastly, there isn't a listener associated with it.

The classes mentioned above can be used to create a MIDlet but that MIDlet doesn't have any text-messaging capabilities. In order for a J2ME application to offer SMS capabilities the following classes have to be utilized:

- Textbox — First off, we need to provide means for the user to enter a destination address and a text-message. The Textbox class is introduced for this purpose. It displays a screen on the mobile device that allows the user to enter and edit text. The developer has the ability to specify a maximum size, which is the maximum size a destination address or text-message can be. This maximum will be enforced by the MIDlet. The mobile device will decide how many rows and columns to display depending on its display size.

- MessageConnection — After the user has composed the message then we need means of sending it. To start this process MessageConnection is required because this class provides the basic functionality for sending and receiving message [Sun, J2ME-API]. In order to send or receive a message the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Message class is required because this class has the means to store the message. The MessageConnection will automatically calculate how many message segments to break the text-message up into. The developer might set a Textbox maximum at a 1000 characters but the actual size of a SMS text-message cannot be that large. Therefore, a large text-message might need to be sent in separate segments so that it can meet the standards.

- MessageListener — To receive messages, the MessageListener interface should be implemented and it should be registered with the MessageConnection. Thus, when new messages are received the MessageConnection will be notified and the developer must use MessageConnection's "receive" method. Another rule that must be followed is that the receive method should never be called inside MessageListener because it can never occupy itself with receiving messages.

- Message — The Message class is the base interface for messages of varying types [Sun, J2ME-API]. This implies that the message stored in this object is not necessarily a text-message. The Message class can be used for storing generic non-specific messages. These messages could be plain text or binary.

4.7 Developing a Generic Mobile Station SMS application using J2ME

There are any number of J2ME SMS applications already pre-built and pre-deployed. Sun Microsystems even provides a pre-built J2ME SMS application that can be deployed on any mobile device that supports J2ME. Their J2ME SMS application will be presented here in this Chapter. The SMS demo application is an example MIDlet suite that demonstrates SMS features. The demo uses MIDP 2.0. On top of the Sun SMS application a further enhanced

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SMS application will be presented that has been restructured and made more generic by using design patterns.

4.7.1 Sun Microsystem's MS SMS Application

In the J2ME toolkit the SMS application is called, "SMSDemo." The SMS demo is broken into 3 different classes.

- SMSReceive Used for displaying received SMS messages
- SMSSend Prompts for a destination address and a message
- SMSSender This class will do all of the sending and receiving of SMS messages

This particular demo was built for the Java 1.4 SDK. To simulate the SMS demo on an emulator use the "run" script located in the "bin" directory. The emulator that is displayed will be considered the mobile device for the duration of this demo. It can be viewed in Figure 4.4.

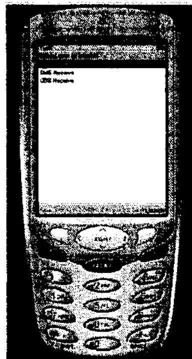

Figure 4.4: Sun MicroSystem's Emulator

The application management software, i.e., the emulator, can utilize the SMSReceive object. It will specify when to start, pause and destroy the MIDlet. The SMSReceive object does override these methods so that they will operate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

properly. When a SMSReceive object is initiated it creates an instance of the SMSSender object. This object will be used later if the user chooses to reply to a received message. The SMSReceive object has the ability to handle user actions, i.e., the user might press the "exit" key and the SMSReceive should do the appropriate action.

When the SMSReceive object starts it will designate a message listener and it will also start the receive message thread. In this particular application, the SMSReceive object handles the thread. The thread will check for a received message with the aide of the MessageConnection object. If it finds a received message then it will display the message on the screen and continue to check for more incoming messages. At this point, the user has the option to reply and the thread will add this action command to the user's display area.

SMSReceive is destroyed when the SMS connection is closed. Also, the receive-thread is stopped and the object is nullified.

The next object that will be discussed is the SMSSend object. This is similar to the previous MIDlet except that it does the reverse role. This object handles retrieving the destination address and text-message from the user. It too handles user commands and it will perform the appropriate actions. When this object is initiated it creates a destination address object and it too creates an SMSSender object. When the user is ready to compose a message they are prompted for a destination address. The destination address is tested to ensure that it's valid. If it is valid then the message is sent through the use of the SMSSender object.

The SMSSender object is the only object in this application that doesn't extend the MIDlet class. When an instance of this is created a TextBox object is also created that will be used for inputting the message. The SMSSender object has a prompt and send method. This method will display the input TextBox object and SMSSender will wait idly for the user to click the "send" button. When the "send" button is pressed it will be detected and controlled by creating a new Thread that SMSSender will handle. To send the message the SMSSender object creates a new MessageConnection object. The body of the message will be passed to the MessageConnection object and it's corresponding "send" method will be called. If there are any errors then they will be caught by the SMSReceive object.

This SMS application that was developed by Sun with mobile devices in mind works quite efficiently. It achieves its goal of sending and receiving SMS text-messages all through the use of J2ME technology. It was built for MIDP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.0 and the latest version of CLDC. Since it utilizes J2ME, it also comes with certain guarantees. For instance, it will be easily deployable on any mobile device that supports MIDP 2.0, which is currently supported by over 50 companies [Knudsen, 2002].

How could the provided SMS application be enhanced further? After it's been tested and it works properly then a design pattern could be applied. Design patterns are proven to help with the current and future maintenance and architecture of the application. Also they assist in making application generic. With so many design patterns to choose from, one has to be very careful to apply the correct pattern. In order to apply the design pattern, the pattern must exist in the code first.

4.7.2 A Generic MS Application using the MVC and Wizard Design Patterns

This chapter will introduce a generic MS application for sending and receiving SMS applications. The application is built from J2ME technology. J2ME was chosen because it has been proven to be the most generic solution for developing applications for mobile devices. Two design patterns have been applied to the architecture of this application. They are used to assist in developing the application. Through the use of the design patterns and the developing language the proposed MS application is generic.

The first design pattern that was applied to the generic MS application for sending SMS messages was the MVC design pattern. The Cascading Menu pattern was considered first because this particular design pattern is actually a scaled down version of the MVC design pattern. However, the SMS application required the use of the Controller so the entire MVC design pattern was used. The MVC design pattern was implemented with 4 classes. The Controller can be viewed in Appendix E.1. The View can be viewed in Appendix E.4. There are two Models. One Model for sending a message, which can be viewed in Appendix E.3 and the other Model, is for receiving a message, which can be viewed in Appendix E.2.

The Controller does all of the message listening and all of the command listening. Effectively, everything passes through the Controller and it will distribute the work appropriatly. Please refer to the UML diagram in Figure 4.5. In Figure 4.5, the Controller can be viewed as the core component of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

generic MS application. There are associations between the Controller to the Models and vice versa. The same relationship exists between the Controller and the View. The View and the Model never communicate directly with each other.

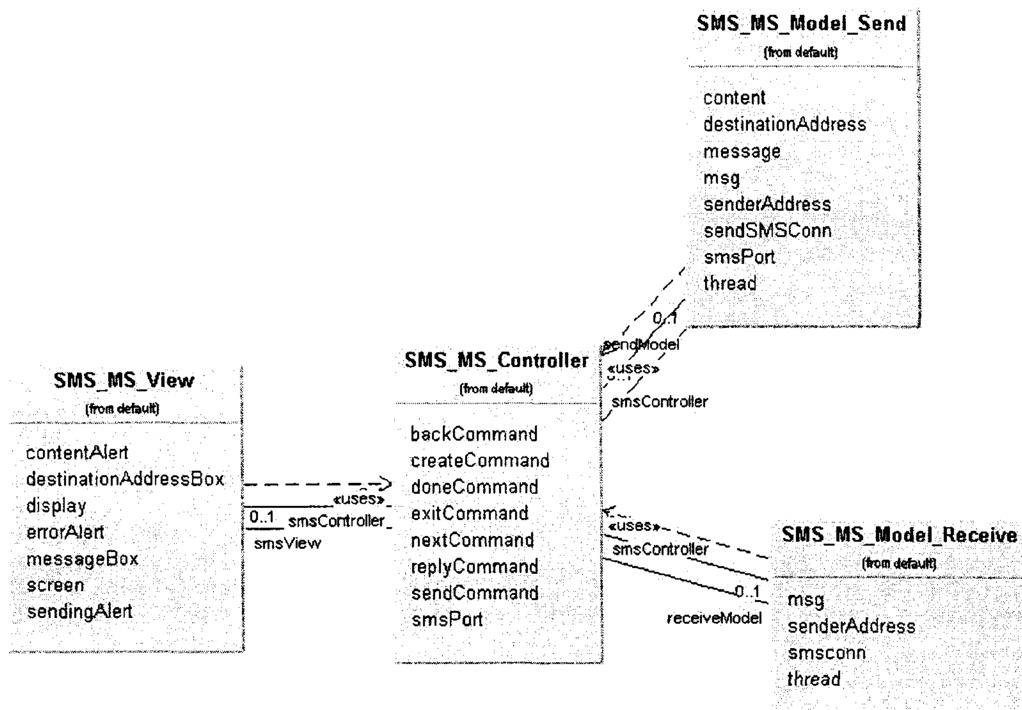

Figure 4.5: The UML Diagram of the Generic MS SMS Application

Let's consider the case when a message is received as in Figure 4.6. The Controller would test to see if there is a new message on the queue. If one exists then it would tell the Model to retrieve it. When the Model is finished retrieving the message it would inform the Controller that it has completed. The Controller would tell the View to display the message on the screen. The Controller also informs the View to display two buttons for the user. One of the buttons is for exiting and the other is for replying. If the user clicks the "exit" button then the Controller exits the application. If the user clicks the "reply" button then the Controller tells the View to display a textbox on the screen for inputting the user's message. In this case, there is no need to input the destination address because we already know it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A similar situation occurs for sending a SMS message in Figure 4.7. When the user clicks "new" button the Controller tells the View to input the destination address and to display two buttons. The first button will exit out of the compose section of the SMS application. The other is for moving to the next stage of the message composition. The Controller will wait idly for the user to input the destination address. After the address has been inputted it tells the Model to test if the destination address was correct. If it was not correct then the Controller tells the View to display an error message on the screen. If it was correct then it would inform the View to display the textbox for inputting the message. When the user finishes inputting the message then it tells the Model to send the message to the destination address. If all of this was successful then the Model informs the Controller and in turn the Controller informs the View. Naturally, the View displays a message sent confirmation on the mobile device's display screen for the user's information.

Obviously the MVC design pattern played the most crucial role in this MS application, however, the Wizard Dialog pattern was used too. This design pattern is for collecting information before performing a task [Cepa and Mezini, 2003]. Therefore, this design pattern was used to retrieve the destination address and text message before sending the message to the intended inbox. Both Figure 4.6 and Figure 4.7 illustrate the Wizard Dialog pattern. The first Figure shows it at the very ending where the user clicks the "Reply" button then the Wizard Dialog plays it's small but important role of collecting the user data. In the second Figure, the Wizard Dialog is used throughout the entire Figure because the entire Figure is collecting user data.

4.8 Summary

In this chapter, the Mobile Station was introduced in greater depth. The process of sending and receiving a SMS message was explained. The SMS standards were also introduced because these standards are crucial in order for a SMS message to conform to the rest of the SMS messages.

J2ME was introduced as the preferred language of choice for developing a mobile device application. It's runtime environment, MIDP, is supported on over 50 companies that produce mobile devices. This makes it highly deployable and very generic. The advantage of J2ME is that it follows Java's motto of "Write once, run anywhere." The J2ME API is quite extensive and programming in J2ME is just as easy as programming in J2SE.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

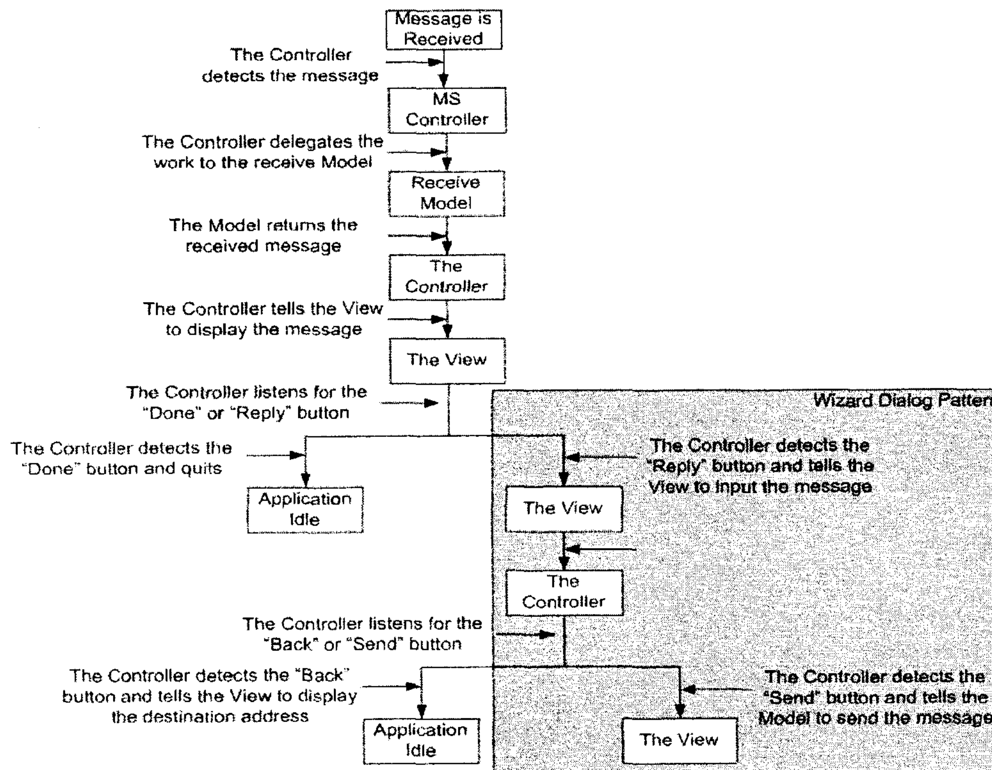

Figure 4.6: A flow diagram for a message that is received. The MVC and Wizard Dialog design patterns can be visibly noted

One of the SMS applications that were discussed in this chapter was distributed by Sun and automatically comes with the J2ME package. It is generic and very dynamic. One could easily deploy this on any mobile device that supports MIDP. The only visible drawback of this MS application is that it's lacking a design pattern. Without a design pattern the application will become tedious to maintain.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

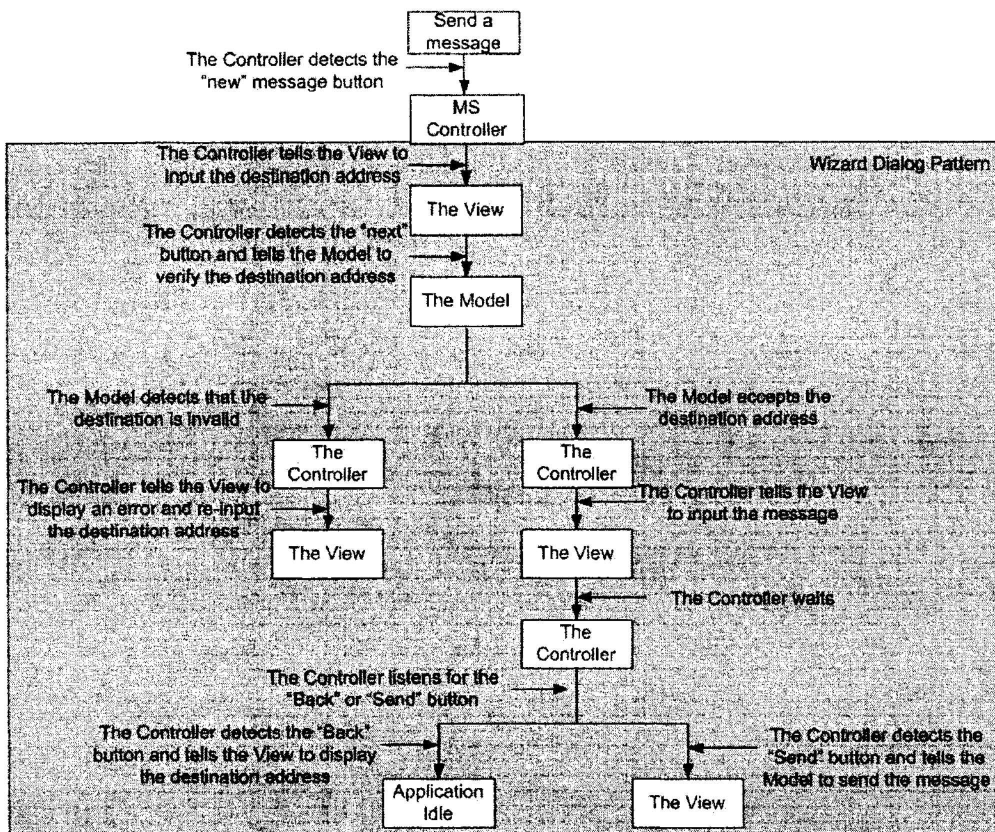

Figure 4.7: A flow diagram for sending a SMS message. The MVC and Wizard Dialog design patterns can be visibly noted

The other SMS application that was discussed in this chapter was the generic MS application for sending and receiving SMS messages. It too is very generic and dynamic. One could easily deploy this on any mobile device that supports MIDP. Obviously, this MS application built upon the previous MS application's advantages, however, this application is slightly more sophisticated because two design patterns were applied to its architecture. MVC is the main design pattern and it affects the entire architecture. The Wizard Dialog design pattern was applied for retrieving user input before sending the SMS message.

Both applications have been tested on the J2ME emulator that Sun provides. They were also further tested on Motorola's emulator and Nokia's

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

emulator. The generic MS application for sending and receiving SMS messages lives up to the test. Since it is deployable on these emulators that support J2ME runtime environment, MIDP, then the developer has the satisfaction of knowing that it will be highly deployable on a wide variety of mobile devices.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Conclusions and Future Research

5.1 Thesis Summary and Findings

This thesis developed two generic applications. Both are used for sending and receiving SMS messages. The first application was a web-based SMSC. The latter was a mobile-application that could be easily deployed on mobile devices. We found that the most ideal way to achieve generality for the web-based SMSC was through the notion of design patterns.

The SMSC utilizes these findings by first building the SMSC with the aide of JSP. JSP is an extraordinaire language to develop with [Dyck, 2000] but it's clearly too easy for a developer to stray from an architectural design, if one is even present. We turned to design patterns to help maintain the generality. The MVC design pattern offered the necessary structure that was required. The MVC design pattern is quite often paired with JSP. The reason for this is that MVC design pattern strives to make the developer's job as abstract as possible [Sauter et al, 2004]. This is achieved by breaking an application into three components. That is, the Model, which does the brute work, the View, which is strictly used for display purposes, and the Controller, which governs the entire program. We found that this implementation worked fine, however, it was still too easy to deviate from the design pattern. Thus, the Struts framework was introduced. Not only does Struts implement the MVC design pattern but also it enforces it. The best part about Struts is that it enforces the MVC design pattern through the use of objects and XML [Sauter et al, 2004]. The Struts' objects are used for quickly applying the MVC design pattern and the XML configuration file is the backbone that verifies the objects were used properly. The Struts MVC SMSC implementation proved to be the most generic. It not only has a good developing language like Java but it also has an enforced design pattern.

The SMS application for sending and receiving text-messages for mobile devices also uses Java technology. However, the mobile device only has limited resources so the Java Community developed J2ME. We found J2ME relatively easy to develop with and it is by far the most generic developing platform for mobile devices [Subramanian, 2001]. It's runtime environment, MIDP, is supported on over 50 companies that produce mobile devices.

There were two implemented versions of the SMS application for the mobile device. The first was an application that is provided by Sun Microsystems. This application was developed with J2ME technology. It naturally works and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

since it was developed using J2ME it could be easily deployable, however, it doesn't use a design pattern. Since design patterns add to current and future maintenance, plus they add to the generality [Sauter et al, 2004] then this application was not satisfactory.

We looked at design patterns for mobile devices and discovered that the Wizard Dialog pattern would be useful when the user is composing a new SMS message. The Wizard Dialog pattern offers the proper structure to collect information via two buttons, i.e., "back" and "next" before performing a task [Brown and Dhaliwal, 2002]. Thus, we applied that design pattern but we were still not satisfied because the rest of the application was lacking structure. We feared that it would lose its generality over time. To maintain the entire application the MVC design pattern proved to be useful once again. For the SMS application for mobile devices, the main design pattern was MVC and within MVC the Wizard Dialog pattern was used.

5.2 Analytical Comparison of the Generic SMS Applications

Although there are numerous other SMS Systems already developed most of them suffer from various different drawbacks. For instance, in Barry Harron's thesis, he used Java technology, which is good because Java is a generic programming language but his application was only concerned with the PC sending a message to the mobile device. Therefore, he didn't develop a generic SMS System but a generic SMSC. Another drawback occurs in Vivek Malhotra paper because the source code will only work on a Microsoft PC or Server. The reason being is that it is written using Active Server Pages and VBScript, each of which may not be supported on other Operating Systems.

In comparison to the generic SMSC and MS applications, the SMSC wasn't written with just Java but JSP. Another plus comes to the generic SMSC when one starts to analyse the architecture of it. The other SMSC did not apply a design pattern and we found this to be a critical error in their designs. With the use of the design patterns and JSP the SMSC is deployable on any server. Thus, the generic SMSC is easily and highly maintainable.

As with the SMSC's, there are lots of MS' applications. We didn't care for a lot of them though. The existing applications were not generic enough or they had poor architectural designs. We chose a nice generic developing platform, J2ME that is highly deployable on countless mobile devices. J2ME offers the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

generality that we were trying to succeed. Any of the SMS applications developed for the mobile device lacked any type of structure. It is hard to achieve generality without structure and structure is gained by utilizing design patterns. Thus, we decided that the generic MS application required a design pattern.

The need for an entirely generic SMS System is evident and not just the SMSC but the whole SMS System. After all this is one of the key attributes to a successful SMS System. The other keys are for it to be cost efficient, easily deployable, and for it to work anywhere at anytime. This can be easily achieved through the use of Java technology. Java has been proven to be a generic language to develop applications for, either a SMSC or a mobile device. Since Java can be developed to run on any platform or almost any mobile device this makes it very cost efficient, easy to deploy and it can work anywhere at anytime.

5.3 Chapter Summary

5.3.1 Chapter 1

This Chapter investigated the idea of having a generic SMS and the traditional SMS system architecture has been explained. For example, it explained the difference between the SMSC and MS. The Chapter also explained current SMS applications for the SMSC and MS. It outlined possible developing platforms for a SMS application, detailing their pros and cons.

5.3.2 Chapter 2

This chapter's focus is on developing a generic SMSC application. The first part of this chapter explores the differences between a local SMS application and a web-based SMS application with the web-based prevailing. The next part of the chapter discusses possible platforms to house the web-based SMS application. We decided upon Java's JSP as the primary developing platform because the JSP technology has the Java motto of "write once, run anywhere." The final part of this chapter investigated how to build the SMSC using JSP technology. There were three implementations that were investigated, all used JSP and the last two used a design pattern. The first was generic but lacked structure. It proposed to place all of the business and presentation logic in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

same JSP page. The second implemented the MVC design pattern, which improved the structure of the SMSC application immensely. However, we grew concerned that the MVC would be lost in future extensions so the third implementation investigated the possibilities of using Struts to enforce the MVC design pattern.

5.3.3 Chapter 3

This chapter developed a generic SMSC application for sending and receiving SMS messages. This chapter outlines the three stages in order to achieve this goal. The first part of this chapter implements the generic SMSC through the use of a single JSP page. This operated properly, however, it could be improved enormously by adding some organization to the logic. The second part of this chapter implemented the generic SMSC application by developing it through the MVC design pattern. This pattern divides the logic and distributes the work. In other words, it offers structure and stability. The last part of this chapter realizes the importance of the MVC design pattern and comprehends how crucial it is that the MVC design pattern is executed exactly. The final implementation uses Struts to impose the MVC design pattern on the structure of the SMSC application. This is achieved because Struts will not deploy if the MVC design pattern has deviated from the norm.

5.3.4 Chapter 4

This chapter explains the architecture of a MS. The first part outlines in detail the necessary steps in order to send and receive a SMS message. It also discusses SMS standards, which are essential if one is to develop a generic MS application. Next it investigates test-beds for developing this generic application. Naturally, the main concern is for the test-bed to support a generic developing platform. Lastly, the first part of this chapter introduces possible design patterns for mobile devices. The second part of this chapter realizes J2ME as a generic developing platform and begins to exam the J2ME API. After the programming language has been fully introduced two MS applications for sending and receiving SMS messages are introduced. The first SMS application was developed by Sun Microsystems and is easily deployable. However, it didn't have a design pattern so the second SMS application

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

proposed two design patterns that will serve the architecture some structure. The two design patterns were the MVC and the Wizard Dialog.

5.3.5 Chapter 5

This chapter presents the summary, findings, and the future work of this research.

5.4 Future Research Directions

Ideally, the applications developed should be easily extensible so that they can grow with the ever-changing mobile world. We aim to extend the applications in the following way.

1. The SMSC could be extended to output XML. This would ensure that the SMSC application's output is in a uniform format [Zimmermann, 2001]. Once the output has been unified then the output can be easily deployed to any program [Jeuring, 2004]. For instance, the SMSC could send a text-message to an XML Messenger. Or possibly send a message to a future messaging system that has not even been developed yet. If the output is offered in XML then it can be further improved with Extensible Stylesheet Language for Transformation (XSLT). XSLT is used for transforming unreadable XML documents into readable data [Laird, 2002]. Thus, if the SMS output ever became unreadable one could use XSLT to translate the XML into a readable format. Therefore, the SMSC will continue to grow with the ever-changing world and not left behind it.

2. As for the MS, this is a generic application that is highly deployable. It's output conforms to the SMS standards. Therefore, it shall remain a readable format as long as the standards don't change. Our future aim is to see the MVC design pattern enforced on the mobile device. Since the MS application's generality is gained largely through the design pattern the SMS application needs means to enforce this. It literally needs a Struts built for the mobile device. Unfortunately there are no current frameworks that offer this type of solution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. Our future research will also explore the idea of extending both SMS applications to handle any type of multimedia, not just plain text. This type of messaging is referred to as Multimedia Messaging Service (MMS) [Brown and Dhaliwal, 2002]. The scope of MMS currently encompasses audio, pictures, streaming video, etc [Hayes, 2004]. An idealistic MMS application should be highly deployable on any operating system and executable on any server [Hayes, 2004]. It should also have the capabilities of handling any type of current and future multimedia format [Shen et al, 2004]. Preferably all of this should be executed in real-time [Shen et al, 2004]. How close are the current SMS applications to achieving these goals and what further enhancements should be made?

   To extend the developed SMS applications to MMS applications we would need to ensure that the network supports the multimedia formats. Then we would create an even more generic JavaBean. In terms of the MVC design pattern, the JavaBean is the Model and it does all of the sending and receiving. Thus, the JavaBean will need to understand the incoming and outgoing multimedia formats. The View component needs to be altered slightly to handle varying display formats and audio playback. Lastly, the Controller would need a few logical controlling statements. Most definitely research would be required to investigate the use of different design patterns. For instance, the Command pattern might be useful because the Controller would receive notice of an incoming message and it wouldn't bother determining which type of message but just forward it to the Command object. Then the Command object would have all the types stored and the proper actions to handle them. A similar case would occur for the View. The Controller would tell the Command object to display a message; the Command object would determine the message's format then pass the command to the View.

   The current SMSC application is fundamentally built around the MVC design pattern and naturally this design would remain throughout and MMSC application. In order for the current SMSC application to handle MMS messages, it would need to be able to handle binary messages as well as plain text messages. The current SMSC architecture for sending a message is displayed in pseudo in Figure 5.1. How much of the current SMSC application would be reusable in the MMSC application? Well, naturally the View has to be altered so that the user can specify a different type of multimedia. The Controller will need an extra control statement to see what type of multimedia the user is trying to send. The current Model would be reusable providing that a second Model is created for transmitting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

binary messages. The new pseudo code is laid out in Figure 5.2. This illustrates the necessary logic in sending a MMS message from a new sophisticated MMSC application. As one can see, the current Model is reused and the Controller is basically reused too. A brand new View would be required but the View is nothing more than cosmetic though so this isn't a huge deal.

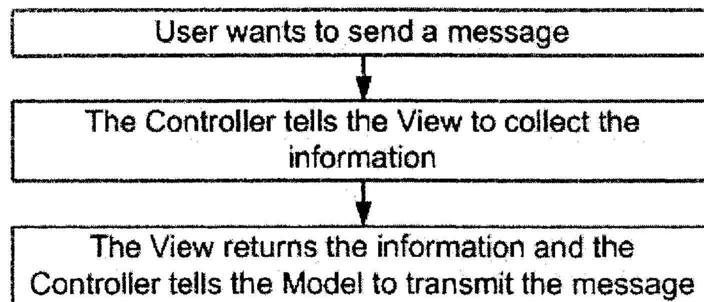

Figure 5.1: Pseudo code for sending a plain text message from the SMSC application.

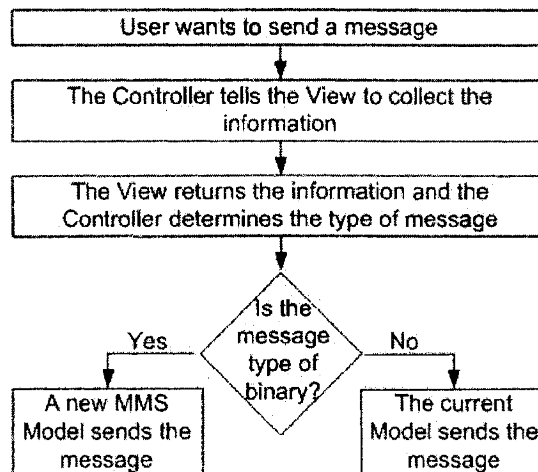

Figure 5.2: Pseudo code for sending a MMS message from a MMSC application.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Lastly, how conveniently easy would it be for these SMS applications to encrypt and decrypt the messages for transmission over public places? In both instances, the encryption and decryption could utilize the latest technology by simply plugging the security into the Transport Layer prior to any public transmission [Brady, 2000]. A flow diagram has been provided in Figure 5.3. For the SMSC one would use the Transport Layer Security (TLS) protocol standard. It offers all of the necessary components required to make a personal SMS message secure and safe. For instance, it offers message integrity, privacy to a transmitted message, and it authenticates a received message. These are all obviously very important features of any security system [Vogler, 2000]. The MS would use a slightly different security system. This one is called the Wireless Transport Layer Security (WTLS) and it offers the same features except that they're built for a mobile device [Brady, 2000]. For instance, a mobile device doesn't support 128 bit encryption because it doesn't have the power to compute numbers that large. Therefore, the WTLS uses a smaller bit-encryption [Saarinen, 2000]. This solution requires no alterations to either of the SMS applications; however, the security issues behind sending an unencrypted message is quite unsafe and this area should be reviewed.

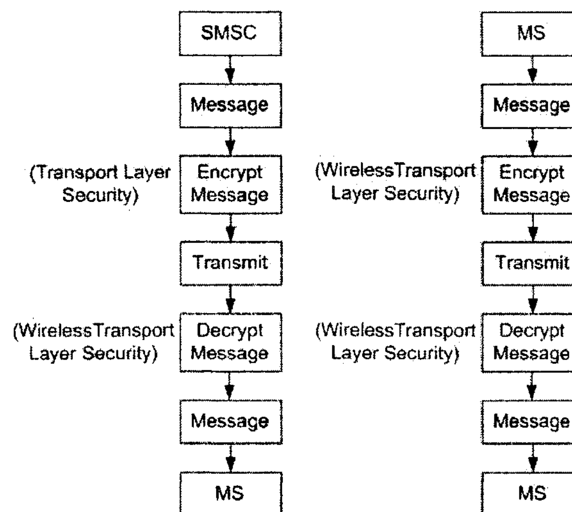

Figure 5.3: A flow diagram for sending a SMS message. For security purposes the message has been encrypted and decrypted.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

[Agarwal, 2004] Puneet Agarwal, "Struts best practices," article presented by http://www.javaworld.com, September 2004.

[Alphonse and Rajkotia, 2005] Jean Alphonse and Purva Rajkotia, "TIA/EIA-637-B," technical document presented by Telecommunications Industry Association, January 2005.

[Apache, 2005] Apache Tomcat, "The Apache Jakarta Project," published by Apache, http://jakarta.apache.org/tomcat/index.html, 2005.

[Balani, 2001] Naveen Balani, "Deliver XHTML applications to mobile devices," Developer Works tutorial, www.ibm.com/developerworks, August 2001.

[Biggs, 2002] Wes Biggs, "Smart and simple messaging," article presented by developerWorks IBM, February 2002.

[Brady, 2000] Dermot Brady, "Development of a WAP-enabled Unit Fund Management Application using WML and ASP," Dissertation from the Faculty of Informatics, University of Ulster, March 2000.

[Brown and Dhaliwal, 2002] Graham Brown and Josh Dhaliwal, "Multimedia Messaging 2002," published by The Wireless World Forum, May 2002.

[Buckingham, 2000] Simon Buckingham, "What is SMS?" Mobile Streams, http://www.gsmworld.com/technology/sms/intro.shtml, 2000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Butts and Cockburn, 2001] Lee Butts and Andy Cockburn, "An Evaluation of Mobile Phone Text Input Methods," Third Australasian User Interfaces Conference, Australian Computer Society Inc, July 2001.

[Cavaness, 2002] Chuck Cavaness, "Jakarta Struts 1.1," article presented by Atlanta Java Users Group (AJUG), August 2002.

[Cavaness, 2003] Chuck Cavaness and Brian Keeton, "Jakarta Struts Pocket Reference," by O'Reilly first edition, 2003.

[Cepa and Mezini], Vasian Cepa and Mira Mezini, "MobCon: A Generative Middleware Framework for Java Mobile Applications," article presented by Darmstadt University of Technology, November 2003.

[Ciucci et al, 2002] Fabio Ciucci, Glen Cordrey, Jon Eaves, David Hook, Myank Jain, Neil Katin, Steve Ma, Ravi Reddy, and Wai Fung, "Mobile Information Device Profile, v2.0 (JSR-118)," article presented by Motorola Inc., Sun Microsystems Inc. and Java Process Community, 2002.

[Coen and Nanduri, 2003] Michael Coen and Amarnath Nanduri, "Jump the hurdles of Struts development," article presented by http://www.javaworld.com, April 2003.

[Deitel and Deitel, 1999] Harvey Deitel and Paul Deitel, "Java: How to Program," published by Prentice Hall, 3rd Edition, 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Dudney and Lehr, 2004] Bill Dudney and Jonathan Lehr, "Jakarta Pitfalls: Time-Saving Solutions for Struts, Ant, JUnit, and Cactus (Java Open Source Library)," published by Wiley, January 2004.

[Dyck, 2000] Timothy Dyck, "Four Scripting Languages Speed Development," published by Ziff Davis Publishing Holdings Inc., February 2000.

[Gamma et al, 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," published by Addison-Wesley Professional Computing Series, August 1994].

[Garcia, 2000] Antonio Garcia, "Command Design Pattern," article presented by Rice University, September 2000.

[Garcia, 2000] Antonio Garcia, "The Strategy Design Pattern," article presented by Rice University, September 2000.

[Geary, 2001] David Geary, "Amaze your developer friends with design patterns," article presented by http://www.javaworld.com, October 2001.

[Geary, 2002] David Geary, "Take command of your software," article presented by http://www.javaworld.com, June 2002.

[Ghosh, 2003] Soma Ghosh, "Extend J2ME to Wireless Messaging," article presented by developerWorks IBM, http://www-106.ibm.com/, February 2003.

[Green, 2004] Roedy Green, "JPhone Design Ideas," article presented by Canadian Mind Products, www.mindprod.com, August 2004.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Grehan, 2002] Rick Grehan, "Get the JavaPhone," article presented by JavaPro, http://www.fawcette.com/javapro/2002_04/magazine/columns/javatogo/, April 2002.

[Hapner, Burridge and Sharma, 1999] Mark Hapner, Rich Burridge, and Rahul Sharma, "Java Message Service," published by Sun Microsystems, November 1999.

[Harron, 2002] Barry Harron, "A Generic Short Messaging Service Application," MSc Thesis from the Faculty of Informatics University of Ulster, September 2002.

[Hayes, 2004] Ed Hayes, "Multimedia Messenger," article presented by Phantom Applications New Zealand Agent, www.phantomapps.co.nz, May 2004.

[Hepper and Hesmer, 2003] Stefan Hepper and Stephan Hesmer, "Introducing the Portlet Specification, Part 1," article presented by http://www.javaworld.com, August 2003.

[Hepper and Hesmer, 2003] Stefan Hepper and Stephan Hesmer, "Introducing the Portlet Specification, Part 2," article presented by http://www.javaworld.com, September 2003.

[Hightower, 2004] Rick Hightower, "Jakarta-Struts Live," published by SourceBeat, http://www.serverside.com, 2004.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Holmes, 2002] Bill Holmes, "SMS (Short Message Service) – Technical Overview," California Software Labs, http://www.cswl.com/whiteppr/tech/sms.html, January 2002.

[Hui, 2002] Ben Hui, "Big designs for small devices," article presented by http://www.javaworld.com, December 2002.

[Hurst, 2002] Walter Hurst, "Design patterns make for better J2EE apps," article presented by http://www.javaworld.com, June 2002.

[Jeuring, 2004] Johan Jeuring, "Generic Programming Introduction XML," article presented by Institute of Information and Computing Sciences, www.cs.uu.nl, September 2004.

[Kluyt, 2002] Onno Kluyt, "Java 2 Platform, Micro Edition (J2ME); JSR 68 Overview," in http://java.sun.com/j2me/overview.html, August 2002.

[Knudsen, 2002] Jonathan Knudsen, "Mobile Information Device Profile (MIDP) Overview," article presented by Sun Microsystems, http://java.sun.com/products/midp/overview.html, November 2002.

[Knudsen, 2003] Jonathan Knudsen, "Introduction to Java Phone API," article presented by Sun Microsystems, http://java.sun.com/products/javaphone/overview.html, March 2003.

[Laird, 2001] Cameron Laird, "SMS: Case study of a Web services deployment," article presented by developerWork IBM, http://www-106.ibm.com/, August 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Laird, 2002] Cameron Laird, "XSLT Powers a New Wave of Web Applications," Linux Journal, Volume 2002 Issue 95, March 2002.

[LCTTP, 2001] LCTTP, "XHTML," W3C Recommendation summary by Michigan State University Libraries, Computing, & Technology Training Program, March 2001.

[Mahmoud, 2001] Qusay Mahmoud, "Web Application Development with JSP and XML Part 1: Fast Track JSP," article presented by Sun Microsystems, http://www.sun.com, June 2001.

[Mahmoud, 2003] Qusay Mahmoud, "Developing Web Applications with Java Server Pages 2.0," article presented by Sun Microsystems, http://www.sun.com, July 2003.

[Malhotra, 2001] Vivek Malhotra, "Introduction to SMS," tutorial presented by developerWorks, IBM, http://www-106.ibm.com/, October 2001.

[McLaughlin, 2000] Douglas McLaughlin, "Clean Up Your Act with XHTML," article presented by Intercom, http://www.intercom.org, November 2000.

[Mercay and Bouzeid, 2002] Julien Mercay and Gilbert Bouzeid, "Boost Struts with XSLT and XML," article presented by http://www.javaworld.com, February 2002.

[Mittal, Moffet, and Wutka, 2003] Kunal Mittal, Alan Moffet and Mark Wutka, "Creating HTML Forms with Java Server Pages," online tutorial put on by Sams

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Publishing, http://www.samspublishing.com/articles/ article.asp?p=102175&seqNum=1, December 2003.

[Muchow, 2002] John Muchow, "J2ME 101, Part 1: Introduction to MIDP's high-level UI," Portions of this tutorial are used with permission from the book Core J2ME Technology and MIDP, http://www-106.ibm.com/, September 2002.

[Pemberton et al, 2000] Steve Pemberton, Daniel Austin, Jonny Alexsson, Tantek Celik, Doug Dominiak, Herman Elenbaas, Beth Epperson, Masayasu Ishikawa, Shi'icki Matsui, Shane McCarron, Ann Navarro, Subramanian Peruvemba, Rob Relyea, Sebastian Schnitzenbaumer and Peter Stark, "XHTML 1.0 – The Extensible HyperText Markup Language (Second Edition)," W3C Recommendation, www.w3.org/TR/2002/REC-xhtml1-20020801, January 2000.

[Ping et al, 2003] Yu Ping, Jianguo Lu, Terence Lau, Kostas Kontogiannis, Tack Tong, and Bo Yi, "Migration of Legacy Web Applications to Enterprise Java Environments – Net.Data to JSP Transformation," Centre for Advanced Studies and Engineering Research Council by IBM, March 2003.

[Saarinen, 2000] Markku Saarinen, "Attacks against the WAP WTLS Protocol," article presented by University of Jyvaskyla, May 2000.

[Sauter et al, 2004] Patrick Sauter, Gabriel Vögler, Günther Specht, and Thomas Flor, "Extending the MVC Design Pattern towards a Task-Oriented Development Approach for Pervasive Computing Applications," Proc. Int. Conf. on Architecture of Computing Systems - Organic and Pervasive Computing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(ARCS 2004), Augsburg, 23.-26. March 2004, Spinger-Verlag, LNCS 2981, 2004, pp. 309-321

[Seshadri, 1999] Govind Seshadri, "Understanding JavaServer Pages Model 2 architecture," article presented by http://www.javaworld.com, December 1999.

[Shen et al, 2004] Jun Shen, Pei Sun, Jianming Zhang and Song Song, "iMMS: Interactive Multimedia Messaging Service," article presented by International Business Machines Corporation, www.ibm.com, November 2004.

[Shin, 2003] Sang Shin, "MVC Pattern & Framework," article presented by JavaPassion, http://www.javapassion.com, November 2003.

[Sivakumar, 2001] Srinivasa Sivakumar, "Building Mobile Web Applications with .Net Mobile Web SDK & ASP.Net," article presented by Wireless Developer Network, www.wirelessdevnet.com, January 2001.

[Subramaniam, 2001] Venkat Subramaniam, "Servlets, JSP, Struts and MVC," article presented by http://www.agiledeveloper.com/download.aspx, March 2001.

[Sun, J2ME-API], Sun Microsystems, "J2ME API," published by Sun Microsystems, java.sun.com/j2me/, 2002.

[Sundsted, 1996] Todd Sundsted, "Observer and Observable," article presented by http://www.javaworld.com, October 1996.

[Sundsted, 1998] Todd Sundsted, "MVC Meets Swing," article presented by http://www.javaworld.com, April 1998.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Tarr, 2000] Bob Tarr, "The Command Pattern," article presented by University of Maryland, September 2000.

[Tarr, 2000] Bob Tarr, "The Strategy Pattern," article presented by University of Maryland, September 2000.

[TIA, TIA/EIA-637-B] TIA, "TIA/EIA Standard: Short Message Services for Wideband Spread Spectrum Systems," published by Telecommunications Industry Association, www.tiaonline.org, May 2002.

[Under, 2000] Kevin Under, "Solve your servlet-based presentation problems," article presented by http://www.javaworld.com, November 2000.

[Vieregger, 2003] Carl Vieregger, "Develop Java Portlets," article presented by http://www.javaworld.com, February 2003.

[Vogler, 2000] Dean Vogler, "Security Issues In Wireless Environments," article presented by Motorola Labs Communication Systems and Technologies Labs, October 2000.

[Xu, Teo, and Wang, 2003] Heng Xu, Hock Hai Teo, Hao Wang, "Foundations of SMS Commerce Success: Lessons from SMS Messaging and Co-opetition," proceedings of the 36th Hawaii International Conference on System Sciences, IEEE, 2003.

[Yuan and Long, 2002] Michael Yuan and Ju Long, "Securing wireless J2ME," Center for Research in Electronic Commerce, University of Texas, http://www-106.ibm.com/, June 2002.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Yuan, 2002] Michael Yuan, "Mobile P2P messaging, Part 1," Center for Research in Electronic Commerce, University of Texas, http://www-106.ibm.com/, December 2002.

[Yuan, 2003] Michael Yuan, "Let the mobile games begin, Part 1," article presented by http://www.javaworld.com, February 2003.

[Zimmermann, 2001] Tilo Zimmermann, "Unified XML Messaging for Business Partners in the Publishing, Print, Pulp & Paper Industries," proceedings by deepX Ltd., October 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendices

Appendix A SMSC source code using JSP

Appendix A.1 A Generic JSP example of the "Generic Web-Based SMSC"

```
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
 <% String title = new String("Generic Faculty SMSC"); %>
 <%@ page import="com.simplewire.sms.*" %>
 <head><title><%= title %></title></head>
 <body bgcolor="white">
  <table width="300" border="1">
   <tr>
    <td align="center">
     <br />
     <h2><%= title %></h2>
     <%
       String strFormMessage[] = request.getParameterValues("txtMessage");
       String strFormIndividual[] = request.getParameterValues("txtFacIndividual");
       String err = "", strMessage = "", strIndividual = "";

       if(strFormMessage != null)
       {
         strMessage = strFormMessage[0];
         if(strMessage.equals(""))
         {
           err = "An error has occurred and one or more errors are highlighted in red";
         }
       }
       else
         strMessage = "";

       if(strFormIndividual != null)
       {
         strIndividual = strFormIndividual[0];
         if(strIndividual.equals(""))
         {
           err = "An error has occurred and one or more errors are highlighted in red";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
else
{
  // valid phone number separations are ' ', or ', ', or ',', or ';' or '\n',
  // my phone number separation will be a space ','

  char[] chars = strIndividual.toCharArray();

  if (chars.length == 0)
  {
    err = "Invalid phone number!!";
    strIndividual = "";
  }

  int startPos = 0;
  boolean newNumber = true;
  String strNumber = new String("");

  if(chars[0] != '+') strNumber = "+";

  for(int i = startPos; i < chars.length; i ++)
  {
    if(chars[i] == ' ' || chars[i] == ';' || chars[i] == '\n' || chars[i] == ',' || chars[i] == '+' || chars[i] == '\r')
    {
      // eat these characters
      while(i < chars.length && (chars[i] == ' ' || chars[i] == ';' || chars[i] == '\n' || chars[i] == ',' || chars[i] ==
'+' || chars[i] == '\r'))
        i++;

      if(i < chars.length)
      {
        if(strNumber.equals(""))
          strNumber += "+";
        else
          strNumber += ",+";
      }
      i--;
    }
    else if(!Character.isDigit(chars[i]))
    {
      err = "Invalid phone number!!";
      for(;i<chars.length; i++)
        strNumber += Character.toString(chars[i]);
      break;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        }
        else
          strNumber += Character.toString(chars[i]);
      }
      strIndividual = strNumber;
    }
  }
  else
    strIndividual = "";

  out.println("<font color=\"RED\">" + err + "</font><br /><br />");
%>
<table width="250" cellspacing="0" cellpadding="0" border="0">
 <tr>
  <td>
   <% if(strMessage.equals("") && strFormMessage != null) out.println("<font color=\"RED\">"); %>
   Message:
   <% if(strMessage.equals("") && strFormMessage != null) out.println("</font>"); %>
   <form name="test" method="get" action="index.jsp">
    <textarea rows="4" cols="20" name="txtMessage"><%= strMessage %></textarea>
    <br />
    <br />
    <% if(strIndividual.equals("") && strFormIndividual != null) out.println("<font color=\"RED\">"); %>
    <br />Individual's Telephone Number (use a comma to delimit multiple phone numbers):<br /><br />
    <% if(strIndividual.equals("") && strFormIndividual != null) out.println("</font>"); %>
    <center>
     <textarea rows="4" cols="20" name="txtFacIndividual"><%= strIndividual %></textarea>
    </center>
    <br /><br />
    <center>
     <input type="submit" value="Send" />
     <input type="reset" value="Reset" />
     <%
       out.println("<br /><br />");

       if(!strMessage.equals("") && !strIndividual.equals("") && err.equals(""))
       {
         String []strMsgPins = strIndividual.split(",");

              SMS sms = new SMS();
              sms.setSubscriberID("799-948-986-29352");
              sms.setSubscriberPassword("D437A165");
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    sms.setMsgText(strMessage);

                    // Send Message
                    for(int i = 0; i < strMsgPins.length; i++)
                    {
                     sms.setMsgPin(strMsgPins[i]);
                     sms.msgSend();

                     // Check For Errors
                     if(sms.isSuccess())
                     {
                      out.println("Message was sent to " + strMsgPins[i] + "!<br />");
                     }
                     else
                     {
                      out.println("<font color=\"red\">");
                      out.println("Message was not sent to " + strMsgPins[i] + "!<br />");
                      out.println("Error Code: " + sms.getErrorCode() + "<br />");
                      out.println("Error Description: " + sms.getErrorDesc() + "<br />");
                      out.println("Error Resolution: " + sms.getErrorResolution() + "<br />");
                      out.println("</font>");
                     }

                     out.println("<br />");
                    }


               }
               else
               {
                out.println("I will not bother SimpleWire");
               }
                    %>
              </center>
             </form>
            </td>
           </tr>
          </table>
         </td>
        </tr>
       </table>
      </body>
     </html>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A.2 XHTML Output from the SMSC using JSP

```
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
 <head><title>Generic Faculty SMS Center</title></head>
 <body bgcolor="white">
  <table width="300" border="1">
   <tr>
    <td align="center">
     <br />
     <h2>Generic Faculty SMS Center</h2>
     <font color="RED"></font><br /><br />

     <table width="250" cellspacing="0" cellpadding="0" border="0">
      <tr>
       <td>

        Message:

        <form name="test" method="get" action="index.jsp">
         <textarea rows="4" cols="20" name="txtMessage"></textarea>
         <br />
         <br />

         <br />Individual's Telephone Number:<br /><br />

         <center>
          <input type="text" value="" name="txtFacIndividual" />
         </center>
         <br /><br />
         <center>
          <input type="submit" value="Send" />
          <input type="reset" value="Reset" />
          <br /><br />
I will not bother SimpleWire

         </center>
        </form>
       </td>
      </tr>
     </table>
    </td>
   </tr>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
</table>
</body>
</html>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix B MVC SMSC source code using JSP

Appendix B.1 The Model JavaBean as viewed from the SMSC

```
import java.io.*;

import com.simplewire.sms.*;

public class Model {
   public static String sendMessage(String strNumber, String strMessage)
   {
         String []strMsgPins = strNumber.split(",");

         String returnMessage = "";

         SMS sms = new SMS();
         sms.setSubscriberID("799-948-986-29352");
         sms.setSubscriberPassword("D437A165");

         sms.setMsgText(strMessage);

         // Send Message
         for(int i = 0; i < strMsgPins.length; i++)
         {
               sms.setMsgPin(strMsgPins[i]);
               sms.msgSend();

               // Check For Errors
               if(sms.isSuccess())
                     returnMessage += "Message was sent to " + strMsgPins[i] + "!<br />";
               else
               {
                     returnMessage += "<font color=\"red\">";
                     returnMessage += "Message was not sent to " + strMsgPins[i] + "!<br />";
                     returnMessage += "Error Code: " + sms.getErrorCode() + "<br />";
                     returnMessage += "Error Description: " + sms.getErrorDesc() + "<br />";
                     returnMessage += "Error Resolution: " + sms.getErrorResolution() + "<br />";
                     returnMessage += "</font>";
               }
               returnMessage += "<br />";
         }
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
		return returnMessage;
	}

	// Is this a valid phone number?
	public static boolean isPhoneNumberValid(String strNumber, Controller controller)
	{
		// valid phone number separations are ' ', or ', ', or ',', or ';' or '\n',
	   // my phone number separation will be a comma ','
		char[] chars = strNumber.toCharArray();

		if (chars.length == 0)
			return false;

		int startPos = 0;

		strNumber = "";

		if(chars[0] != '+') strNumber = "+";

		for(int i = startPos; i < chars.length; i ++)
		{
			if(chars[i] == ' ' || chars[i] == ';' || chars[i] == '\n' || chars[i] == ',' || chars[i] == '+' || chars[i] == '\r')
			{
				// eat these characters
				while(i < chars.length && (chars[i] == ' ' || chars[i] == ';' || chars[i] == '\n' || chars[i] == ',' ||
chars[i] == '+' || chars[i] == '\r'))
					i++;

				if(i < chars.length)
				{
					if(strNumber.equals(""))
						strNumber += "+";
					else
						strNumber += ",+";
				}
				i--;
			}
			else if(!Character.isDigit(chars[i]))
			{
				return false;
			}
			else
				strNumber += Character.toString(chars[i]);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            }
            controller.strNumber = strNumber;

            return true;
    }
}
```

Appendix B.2 The View JSP implemented from the SMSC

```
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<%@ page import="java.util.Enumeration;" %>

<html>
 <% String title = new String("A Generic SMSC using the MVC Design Pattern"); %>
 <head><title><%= title %></title></head>
 <body bgcolor="white">

   <%
    // populate the error string
    String error = new String((String)request.getAttribute("error"));
    // populate the Message (if applicable)
    String txtMessage = new String((String)request.getAttribute("message"));
    // populate the phone number (if applicable)
    String txtFacIndividual = new String((String)request.getAttribute("numbers"));

   %>

   <table width="300" border="1">
    <tr>
     <td align="center">
      <br />
      <h2><%= title %></h2>
      <table width="250" cellspacing="0" cellpadding="0" border="0">
       <tr>
        <td>
         <%= error %>
         <br />

         <% if(txtMessage.equals("") && !error.equals("")) out.println("<font color=\"RED\">"); %>
           Message:
         <% if(txtMessage.equals("") && !error.equals("")) out.println("</font>"); %>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
<form name="test" method="post" action="Controller">
  <textarea rows="4" cols="20" name="txtMessage"><%= txtMessage %></textarea>
  <br />
  <br />
  <br />
  <% if(txtFacIndividual.equals("") && !error.equals("")) out.println("<font color=\"RED\">"); %>
    <br />Individual's Telephone Number (use a comma to delimit multiple phone numbers):<br /><br />
  <% if(txtFacIndividual.equals("") && !error.equals("")) out.println("</font>"); %>
  <center>
    <textarea rows="4" cols="20" name="txtFacIndividual"><%= txtFacIndividual %></textarea>
  </center>
  <br /><br />
  <center>
    <input type="hidden" value="sent" name="action" />
    <input type="submit" value="Send" />
    <input type="reset" value="Reset" />
  </center>
</form>
</td>
</tr>
</table>
</td>
</tr>
</table>
</body>
</html>
```

Appendix B.3 The Controller Servlet implemented from the SMSC

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Controller extends HttpServlet {

   public String strNumber;
   public String strMessage;

   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
String strFormMessage[] = request.getParameterValues("txtMessage");
String strFormAction[] = request.getParameterValues("action");
String strFormIndividual[] = request.getParameterValues("txtFacIndividual");
String error = "", strAction = "";

// Initialize the variables
strNumber = "";
strMessage = "";

if(strFormAction != null)
{
	strAction = strFormAction[0];
	if(strAction.equals("sent"))
	{
		// then they clicked "submit"
		if(strFormMessage != null && strFormIndividual != null)
		{
			strMessage = strFormMessage[0];
			strNumber = strFormIndividual[0];
			if(strMessage.equals(""))
			{
				error = "<font color=\"red\" size=\"+1\">An error has occurred and one
or more errors are highlighted in red</font> <br /><br />";
				forward(error, request, response);
			}
			else if (!Model.isPhoneNumberValid(strNumber, this))
			{
				error = "<font color=\"red\" size=\"+1\">The given phone number is
invalid</font>";

				forward(error, request, response);
			}
			else
			{
				// call the JavaBean to do the 'heavy' processing
				error = "<font color=\"red\" size=\"+1\">" +
Model.sendMessage(strNumber,strMessage) + "</font><br /><br />";
				forward(error,request,response);
			}
		}
		else if(strFormMessage == null && strFormIndividual == null)
		{
			error = "<font color=\"red\" size=\"+1\">An error has occurred and one or more
errors are highlighted in red</font> <br /><br />";
			forward(error, request, response);
		}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
				else if(strFormMessage == null)
				{
					strNumber = strFormIndividual[0];
					error = "<font color=\"red\" size=\"+1\">An error has occurred and one or more
errors are highlighted in red</font> <br /><br />";
					forward(error, request, response);
				}
				else if(strFormIndividual == null)
				{
					strMessage = strFormMessage[0];
					error = "<font color=\"red\" size=\"+1\">The given phone number is
invalid</font>";

					forward(error, request, response);
				}
			}
			else
			{
				// they haven't submitted anything and the error should be nothing
				forward(error, request, response);
			}
		}
		else
		{
			// they haven't submitted anything and the error should be nothing
			forward(error, request, response);
		}
	}

	public void forward(String error, HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException
	{
		// Set the attribute and Forward to hello.jsp
		request.setAttribute ("error", error);
		request.setAttribute ("message", strMessage);
		request.setAttribute ("numbers", strNumber);

		getServletConfig().getServletContext().getRequestDispatcher("/View.jsp").forward(request, response);
	}

	/**
	 * We are going to perform the same operations for POST requests
	 * as for GET methods, so this method just sends the request to
	 * the doGet method.
	 */
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException,
ServletException
    {
        doGet(request, response);
    }
}
```

Appendix B.4 Output of for the Generic SMSC using the MVC Design Pattern

```
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>

 <head><title>A Generic SMSC using the MVC Design Pattern</title></head>
 <body bgcolor="white">



  <table width="300" border="1">
   <tr>
    <td align="center">
     <br />
     <h2>A Generic SMSC using the MVC Design Pattern</h2>
     <table width="250" cellspacing="0" cellpadding="0" border="0">
      <tr>
       <td>
        <br />
         Message:
        <form name="test" method="post" action="Controller">
         <textarea rows="4" cols="20" name="txtMessage"></textarea>
         <br />
         <br />
         <br />

          <br />Individual's Telephone Number (use a comma to delimit multiple phone numbers):<br /><br />

         <center>
          <textarea rows="4" cols="20" name="txtFacIndividual"></textarea>
         </center>
         <br /><br />
         <center>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          <input type="hidden" value="sent" name="action" />
          <input type="submit" value="Send" />
          <input type="reset" value="Reset" />
         </center>
        </form>
       </td>
      </tr>
     </table>
    </td>
   </tr>
  </table>
 </body>
</html>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C MVC SMSC using Struts Source Code

Appendix C.1 Struts SendAction Servlet

```
//package Controller;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import java.io.*;

public final class SendAction extends Action
{
   public SendForm form;
   public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request,
HttpServletResponse response)
   {

         SendForm f = (SendForm) form; // get the form bean

         // Get the Telephone Number
         String strNumbers = f.getPhoneNumber();

         // Get the Message
         String strMessage = f.getMessage();

         if(strNumbers.equals("") || strMessage.equals(""))
               return (mapping.findForward("failure"));

         String msg = "";
         msg = "<font color=\"red\">" + Model.sendMessage(strNumbers, strMessage) + "</font><br /><br />";

         request.setAttribute("error", msg);
         request.setAttribute("message", strMessage);
         request.setAttribute("numbers", strNumbers);

         // Forward control to the specified success target
         return (mapping.findForward("success"));
   }
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C.2 Struts the View JSP

```
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<%@ page import="java.util.Enumeration;" %>

<html>
 <% String title = new String("Generic SMSC Enforcing the MVC Design Pattern with Struts"); %>
 <%
   // populate the error message
   String strError = (String)request.getAttribute("error");
   if(strError == null)
     strError = "";

   // populate the phone number (if applicable)
   String strNumber = (String)request.getAttribute("numbers");
   if(strNumber == null)
     strNumber = "";

   // populate the Message (if applicable)
   String strMessage = (String)request.getAttribute("message");
   if(strMessage == null)
     strMessage = "";
 %>
 <head><title><%= title %></title></head>
 <body>

  <table width="300" border="1">
   <tr>
    <td align="center">
     <br />
     <h2><%= title %></h2>
     <table width="250" cellspacing="0" cellpadding="0" border="0">
      <tr>
       <td>
        <html:errors/>

        <%= strError %>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            Message:
            <html:form action="SendMessage.jsp">
              <html:textarea rows="4" cols="20" property="message"></html:textarea>
              <br />
              <br />
              <br />
              <br />Individual's Telephone Number (use a comma to delimit multiple phone numbers):<br /><br />
              <center>
                <html:textarea rows="4" cols="20" property="phoneNumber"></html:textarea>
              </center>
              <br /><br />
              <center>
                <html:submit value="Send" />
                <html:reset value="Reset" />
              </center>
            </html:form>
          </td>
        </tr>
      </table>
    </td>
   </tr>
  </table>
 </body>
</html>
```

Appendix C.3 Struts the SendForm JavaBean

```
//package Controller;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

public final class SendForm extends ActionForm
{
   /* Telephone Number */
   private String phoneNumber = "";
   public String getPhoneNumber()
   {
         return (this.phoneNumber);
   }
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
	public void setPhoneNumber(String phoneNumber)
	{
		this.phoneNumber = phoneNumber;
	}

	/* Message */
	private String message = null;
	public String getMessage()
	{
		return (this.message);
	}

	public void setMessage(String message)
	{
		this.message = message;
	}

	public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
	{
		// Log the forms data
		servlet.log("phoneNumber:" + phoneNumber);
		servlet.log("message:" + message);

		// Check for mandatory data
		ActionErrors errors = new ActionErrors();

		if (phoneNumber == null || phoneNumber.equals(""))
			errors.add("Phone Number", new ActionError("errors.phoneNumber"));
		else if(!Model.isPhoneNumberValid(phoneNumber, this))
			errors.add("Phone Number", new ActionError("errors.invalidNumber"));

		if (message == null || message.equals(""))
		  errors.add("Message", new ActionError("errors.message"));

		return errors;
	}
}
```

Appendix C.4 Struts the struts-config.xml file

```
<?xml version="1.1" encoding="ISO-8859-1" ?>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>

 <!-- ========== Form Bean Definitions ================= -->
 <form-beans>

   <form-bean      name="sendForm"
               type="SendForm"/>

 </form-beans>


 <!-- ========== Action Mapping Definitions ============ -->
 <action-mappings>

   <action   path="/SendMessage"
          type="SendAction"
          name="sendForm"
          input="/SendMessage.jsp"
          scope="request">
     <forward name="success" path="/SendMessage.jsp"/>
     <forward name="failure" path="/SendMessage.jsp"/>
   </action>

 </action-mappings>

 <message-resources parameter="MessageResources" />

</struts-config>


 <!-- ========== Action Mapping Definitions ============ -->
 <action-mappings>

   <action   path="/SendMessage"
          type="SendAction"
          name="sendForm"
          input="/SendMessage.jsp"
          scope="request">
     <forward name="success" path="/SendMessage.jsp"/>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    <forward name="failure" path="/SendMessage.jsp"/>
  </action>

 </action-mappings>

 <message-resources parameter="MessageResources" />

</struts-config>
```

Appendix C.5 Struts Output

```
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>

 <head><title>Generic Faculty SMS Center (JSP and Struts)</title></head>
 <body>

  <table width="300" border="1">
   <tr>
    <td align="center">
     <br />
     <h2>Generic Faculty SMS Center (JSP and Struts)</h2>
     <table width="250" cellspacing="0" cellpadding="0" border="0">
      <tr>
       <td>

        Message:
        <form name="sendForm" method="post" action="/Faculty_JSP_Struts/SendMessage.do">
         <textarea name="message" cols="20" rows="4"></textarea>
         <br />
         <br />
         <br />
         <br />Individual's Telephone Number:<br /><br />
         <center>
          <input type="text" name="phoneNumber" value="">
         </center>
         <br /><br />
         <center>
          <input type="submit" value="Send">
          <input type="reset" value="Reset">
         </center>
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          </form>
        </td>
      </tr>
    </table>
    </td>
  </tr>
  </table>
 </body>
</html>
```

Appendix C.6 Struts' Error Codes

```
# -- standard errors --
errors.header=<h4><font color="red">Error:</font></h4><UL>
errors.prefix=<LI>
errors.suffix=</LI>
errors.message=Please specify a message in the text box below
errors.phoneNumber=Please specify a telephone number
errors.invalidNumber=Invalid phone number!!
errors.footer=</UL>
# -- validator --
errors.invalid={0} is invalid.
errors.maxlength={0} can not be greater than {1} characters.
errors.minlength={0} can not be less than {1} characters.
errors.range={0} is not in the range {1} through {2}.
errors.required={0} is required.
errors.byte={0} must be an byte.
errors.date={0} is not a date.
errors.double={0} must be an double.
errors.float={0} must be an float.
errors.integer={0} must be an integer.
errors.long={0} must be an long.
errors.short={0} must be an short.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid e-mail address.
# -- other --
errors.cancel=Operation cancelled.
errors.detail={0}
errors.general=The process did not complete. Details should follow.
errors.token=Request could not be completed. Operation is not in sequence.
# -- welcome --
welcome.title=Struts Blank Application
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

welcome.heading=Welcome!

welcome.message=To get started on your own application, copy the struts-blank.war to a new WAR file using the name for your application. Place it in your container's "webapp" folder (or equivalent), and let your container auto-deploy the application. Edit the skeleton configuration files as needed, restart your container, and you are on your way! (You can find the application.properties file with this message in the /WEB-INF/src/java/resources folder.)

Appendix C.7 The Model object for the Struts MVC Implementation

```
import java.io.*;

import com.simplewire.sms.*;

public class Model {
   public static String sendMessage(String strNumber, String strMessage)
   {
         String []strMsgPins = strNumber.split(",");

         String returnMessage = "";

         SMS sms = new SMS();
         sms.setSubscriberID("799-948-986-29352");
         sms.setSubscriberPassword("D437A165");

         sms.setMsgText(strMessage);

         // Send Message
         for(int i = 0; i < strMsgPins.length; i++)
         {
               sms.setMsgPin(strMsgPins[i]);
               sms.msgSend();

               // Check For Errors
               if(sms.isSuccess())
                     returnMessage += "Message was sent to " + strMsgPins[i] + "!<br />";
               else
               {
                     returnMessage += "<font color=\"red\">";
                     returnMessage += "Message was not sent to " + strMsgPins[i] + "!<br />";
                     returnMessage += "Error Code: " + sms.getErrorCode() + "<br />";
                     returnMessage += "Error Description: " + sms.getErrorDesc() + "<br />";
                     returnMessage += "Error Resolution: " + sms.getErrorResolution() + "<br />";
                     returnMessage += "</font>";
               }
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
				returnMessage += "<br />";
			}

			return returnMessage;
		}

		// Is this a valid phone number?
		public static boolean isPhoneNumberValid(String strNumber, SendForm sendForm)
		{
			// valid phone number separations are ' ', or ', ', or ',', or ';' or '\n',
		  // my phone number separation will be a comma ','
			char[] chars = strNumber.toCharArray();

			if (chars.length == 0)
				return false;

			int startPos = 0;

			strNumber = "";

			if(chars[0] != '+') strNumber = "+";

			for(int i = startPos; i < chars.length; i ++)
			{
				if(chars[i] == ' ' || chars[i] == ';' || chars[i] == '\n' || chars[i] == ',' || chars[i] == '+' || chars[i] == '\r')
				{
					// eat these characters
					while(i < chars.length && (chars[i] == ' ' || chars[i] == ';' || chars[i] == '\n' || chars[i] == ',' ||
chars[i] == '+' || chars[i] == '\r'))
						i++;

					if(i < chars.length)
					{
						if(strNumber.equals(""))
						{
							strNumber += "+";
						}
						else
							strNumber += ",+";
					}
					i--;
				}
				else if(!Character.isDigit(chars[i]))
				{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                        return false;
                }
                else
                        strNumber += Character.toString(chars[i]);
        }
        sendForm.setPhoneNumber(strNumber);

        return true;
    }
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix D An Generic MS J2ME Example

Appendix D.1 The SMSReceive Class

```
/*
 * @(#)SMSReceive.java  1.9 03/06/22
 *
 * Copyright (c) 1999-2003 Sun Microsystems, Inc. All rights reserved.
 * PROPRIETARY/CONFIDENTIAL
 * Use is subject to license terms
 */

package example.sms;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.wireless.messaging.*;

import java.io.IOException;

/**
 * An example MIDlet displays text from an SMS MessageConnection
 */
public class SMSReceive extends MIDlet
   implements CommandListener, Runnable, MessageListener {

   /** user interface command for indicating Exit request. */
   Command exitCommand  = new Command("Exit", Command.EXIT, 2);
   /** user interface command for indicating Reply request */
   Command replyCommand = new Command("Reply", Command.OK, 1);
   /** user interface text box for the contents of the fetched URL. */
   Alert content;
   /** current display. */
   Display display;
   /** instance of a thread for asynchronous networking and user interface. */
   Thread thread;
   /** Connections detected at start up. */
   String[] connections;
   /** Flag to signal end of processing. */
   boolean done;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/** The port on which we listen for SMS messages */
String smsPort;
/** SMS message connection for inbound text messages. */
MessageConnection smsconn;
/** Current message read from the network. */
Message msg;
/** Address of the message's sender */
String senderAddress;
/** Alert that is displayed when replying */
Alert sendingMessageAlert;
/** Prompts for and sends the text reply */
SMSSender sender;
/** The screen to display when we return from being paused */
Displayable resumeScreen;

/**
 * Initialize the MIDlet with the current display object and
 * graphical components.
 */
public SMSReceive() {
  smsPort = getAppProperty("SMS-Port");

  display = Display.getDisplay(this);

  content = new Alert("SMS Receive");
  content.setTimeout(Alert.FOREVER);
  content.addCommand(exitCommand);
  content.setCommandListener(this);
  content.setString("Receiving...");

  sendingMessageAlert = new Alert("SMS", null, null, AlertType.INFO);
  sendingMessageAlert.setTimeout(5000);
  sendingMessageAlert.setCommandListener(this);

  sender = new SMSSender(smsPort, display, content, sendingMessageAlert);

  resumeScreen = content;
}

/**
 * Start creates the thread to do the MessageConnection receive
 * text.
 * It should return immediately to keep the dispatcher
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 * from hanging.
 */
public void startApp() {
    /** SMS connection to be read. */
    String smsConnection = "sms://:" + smsPort;
    /** Open the message connection. */
    if (smsconn == null) {
        try {
            smsconn = (MessageConnection) Connector.open(smsConnection);
            smsconn.setMessageListener(this);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    /** Initialize the text if we were started manually. */
    connections = PushRegistry.listConnections(true);
    if (connections == null || connections.length == 0) {
        content.setString("Waiting for SMS on port " + smsPort + "...");
    }
    done = false;
    thread = new Thread(this);
    thread.start();

    display.setCurrent(resumeScreen);
}

/**
 * Notification that a message arrived.
 * @param conn the connection with messages available
 */
public void notifyIncomingMessage(MessageConnection conn) {
    if (thread == null) {
        done = false;
        thread = new Thread(this);
        thread.start();
    }
}

/** Message reading thread. */
public void run() {
    /** Check for sms connection. */
    try {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            msg = smsconn.receive();
            if (msg != null) {
                senderAddress = msg.getAddress();
                content.setTitle("From: " + senderAddress);
                if (msg instanceof TextMessage) {
                    content.setString(((TextMessage)msg).getPayloadText());
                } else {
                    StringBuffer buf = new StringBuffer();
                    byte[] data = ((BinaryMessage)msg).getPayloadData();
                    for (int i = 0; i < data.length; i++) {
                        int intData = (int)data[i] & 0xFF;
                        if (intData < 0x10) {
                            buf.append("0");
                        }
                        buf.append(Integer.toHexString(intData));
                        buf.append(' ');
                    }
                    content.setString(buf.toString());
                }
                content.addCommand(replyCommand);
                display.setCurrent(content);
            }
        } catch (IOException e) {
            // e.printStackTrace();
        }
    }
    /**
     * Pause signals the thread to stop by clearing the thread field.
     * If stopped before done with the iterations it will
     * be restarted from scratch later.
     */
    public void pauseApp() {
        done = true;
        thread = null;
        resumeScreen = display.getCurrent();
    }

    /**
     * Destroy must cleanup everything.  The thread is signaled
     * to stop and no result is produced.
     * @param unconditional true if a forced shutdown was requested
     */
    public void destroyApp(boolean unconditional) {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        done = true;
        thread = null;
        if (smsconn != null) {
            try {
                smsconn.close();
            } catch (IOException e) {
                // Ignore any errors on shutdown
            }
        }
    }

    /**
     * Respond to commands, including exit
     * @param c user interface command requested
     * @param s screen object initiating the request
     */
    public void commandAction(Command c, Displayable s) {
        try {
            if (c == exitCommand || c == Alert.DISMISS_COMMAND) {
                destroyApp(false);
                notifyDestroyed();
            } else if (c == replyCommand) {
                reply();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Allow the user to reply to the received message
     */
    private void reply() {
        // remove the leading "sms://" for diplaying the destination address
        String address = senderAddress.substring(6);
        String statusMessage = "Sending message to " + address + "...";
        sendingMessageAlert.setString(statusMessage);
        sender.promptAndSend(senderAddress);
    }
}
```

Appendix D.2 The SMSSender Class

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```java
/*
 * @(#)SMSSender.java	1.5 03/10/29
 *
 * Copyright (c) 1999-2003 Sun Microsystems, Inc. All rights reserved.
 * PROPRIETARY/CONFIDENTIAL
 * Use is subject to license terms
 */

package example.sms;

import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.wireless.messaging.*;

import java.io.IOException;

/**
 * Prompts for text and sends it via an SMS MessageConnection
 */
public class SMSSender
    implements CommandListener, Runnable {

    /** user interface command for indicating Send request */
    Command sendCommand = new Command("Send", Command.OK, 1);
    /** user interface command for going back to the previous screen */
    Command backCommand = new Command("Back", Command.BACK, 2);
    /** Display to use. */
    Display display;
    /** The port on which we send SMS messages */
    String smsPort;
    /** The URL to send the message to */
    String destinationAddress;
    /** Area where the user enters a message to send */
    TextBox messageBox;
    /** Where to return if the user hits "Back" */
    Displayable backScreen;
    /** Displayed when a message is being sent */
    Displayable sendingScreen;

    /**
     * Initialize the MIDlet with the current display object and
     * graphical components.
     */
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
public SMSSender(String smsPort, Display display,
   Displayable backScreen, Displayable sendingScreen) {
   this.smsPort = smsPort;
   this.display = display;
   this.destinationAddress = null;
   this.backScreen = backScreen;
   this.sendingScreen = sendingScreen;

   messageBox = new TextBox("Enter Message", null, 65535, TextField.ANY);
   messageBox.addCommand(backCommand);
   messageBox.addCommand(sendCommand);
   messageBox.setCommandListener(this);
}

/**
 * Prompt for message and send it
 */
public void promptAndSend(String destinationAddress)
{
   this.destinationAddress = destinationAddress;
   display.setCurrent(messageBox);
}

/**
 * Respond to commands, including exit
 * @param c user interface command requested
 * @param s screen object initiating the request
 */
public void commandAction(Command c, Displayable s) {
   try {
      if (c == backCommand) {
         display.setCurrent(backScreen);
      } else if (c == sendCommand) {
         display.setCurrent(sendingScreen);
         new Thread(this).start();
      }
   } catch (Exception ex) {
      ex.printStackTrace();
   }
}

/**
 * Send the message. Called on a separate thread so we don't have
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
     * contention for the display
     */
    public void run() {
        String address = destinationAddress + ":" + smsPort;

        MessageConnection smsconn = null;
        try {
            /** Open the message connection. */
            smsconn = (MessageConnection)Connector.open(address);

            TextMessage txtmessage = (TextMessage)smsconn.newMessage(
                MessageConnection.TEXT_MESSAGE);
            txtmessage.setAddress(address);
            txtmessage.setPayloadText(messageBox.getString());
            smsconn.send(txtmessage);
        } catch (Throwable t) {
System.out.println("Send caught: ");
t.printStackTrace();
        }
        if (smsconn != null) {
            try {
                smsconn.close();
            } catch (IOException ioe) {
System.out.println("Closing connection caught: ");
ioe.printStackTrace();
            }
        }
    }
}
```

Appendix D.3 The SMSSend Class

```
/*
 * @(#)SMSSend.java    1.5 03/03/02
 *
 * Copyright (c) 1999-2003 Sun Microsystems, Inc. All rights reserved.
 * PROPRIETARY/CONFIDENTIAL
 * Use is subject to license terms
 */

package example.sms;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.wireless.messaging.*;

import java.io.IOException;

/**
 * An example MIDlet to send text via an SMS MessageConnection
 */
public class SMSSend extends MIDlet
    implements CommandListener {

    /** user interface command for indicating Exit request. */
    Command exitCommand  = new Command("Exit", Command.EXIT, 2);
    /** user interface command for proceeding to the next screen */
    Command okCommand = new Command("OK", Command.OK, 1);
    /** current display. */
    Display display;
    /** The port on which we send SMS messages */
    String smsPort;
    /** Area where the user enters the phone number to send the message to */
    TextBox destinationAddressBox;
    /** Error message displayed when an invalid phone number is entered */
    Alert errorMessageAlert;
    /** Alert that is displayed when a message is being sent */
    Alert sendingMessageAlert;
    /** Prompts for and sends the text message */
    SMSSender sender;
    /** The last visible screen when we paused */
    Displayable resumeScreen = null;

    /**
     * Initialize the MIDlet with the current display object and
     * graphical components.
     */
    public SMSSend() {
        smsPort = getAppProperty("SMS-Port");

        display = Display.getDisplay(this);

        destinationAddressBox = new TextBox("Destination Address?",
            null, 256, TextField.PHONENUMBER);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    destinationAddressBox.addCommand(exitCommand);
    destinationAddressBox.addCommand(okCommand);
    destinationAddressBox.setCommandListener(this);

    errorMessageAlert = new Alert("SMS", null, null, AlertType.ERROR);
    errorMessageAlert.setTimeout(5000);

    sendingMessageAlert = new Alert("SMS", null, null, AlertType.INFO);
    sendingMessageAlert.setTimeout(5000);
    sendingMessageAlert.setCommandListener(this);

    sender = new SMSSender(smsPort, display, destinationAddressBox,
        sendingMessageAlert);

    resumeScreen = destinationAddressBox;
  }

  /**
   * startApp should return immediately to keep the dispatcher
   * from hanging.
   */
  public void startApp() {
    display.setCurrent(resumeScreen);
  }

  /**
   * Remember what screen is showing
   */
  public void pauseApp() {
    resumeScreen = display.getCurrent();
  }

  /**
   * Destroy must cleanup everything.
   * @param unconditional true if a forced shutdown was requested
   */
  public void destroyApp(boolean unconditional) {
  }

  /**
   * Respond to commands, including exit
   * @param c user interface command requested
   * @param s screen object initiating the request
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 */
public void commandAction(Command c, Displayable s) {
    try {
        if (c == exitCommand || c == Alert.DISMISS_COMMAND) {
            destroyApp(false);
            notifyDestroyed();
        } else if (c == okCommand) {
            promptAndSend();
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

/**
 * Prompt for and send the message
 */
private void promptAndSend() {
    String address = destinationAddressBox.getString();
    if (!SMSSend.isValidPhoneNumber(address)) {
        errorMessageAlert.setString("Invalid phone number");
        display.setCurrent(errorMessageAlert, destinationAddressBox);
        return;
    }
    String statusMessage = "Sending message to " + address + "...";
    sendingMessageAlert.setString(statusMessage);
    sender.promptAndSend("sms://" + address);
}

/**
 * Check the phone number for validity
 * Valid phone numbers contain only the digits 0 thru 9, and contain
 * a leading '+'.
 */
private static boolean isValidPhoneNumber(String number) {
    char[] chars = number.toCharArray();
    if (chars.length == 0) {
        return false;
    }
    int startPos = 0;
    // initial '+' is OK
    if (chars[0]=='+') {
        startPos = 1;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        }
        for (int i = startPos; i < chars.length; ++i) {
            if (!Character.isDigit(chars[i])) {
                return false;
            }
        }
        return true;
    }
}
```

Appendix E A Generic MS Application for Sending and Receiving SMS messages

Appendix E.1 The Controller for the MS Application

```
import javax.wireless.messaging.MessageConnection;
import javax.wireless.messaging.MessageListener;

import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Display;

import java.io.IOException;

/**
 * An example MIDlet displays text from an SMS MessageConnection
 */
public class SMS_MS_Controller extends MIDlet implements CommandListener, MessageListener
{
    // Create global variables
    SMS_MS_View smsView;
    SMS_MS_Model_Receive receiveModel;
    SMS_MS_Model_Send sendModel;

    Command exitCommand  = new Command("Exit", Command.EXIT, 2);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Command doneCommand = new Command("Done", Command.CANCEL, 2);
Command replyCommand = new Command("Reply", Command.OK, 1);
Command createCommand = new Command("New", Command.OK, 1);
Command nextCommand = new Command("Next", Command.OK, 1);
Command backCommand = new Command("Back", Command.CANCEL, 2);
Command sendCommand = new Command("Send", Command.OK, 1);

String smsPort;

public SMS_MS_Controller()
{
        // Initialize variables
        smsPort = getAppProperty("SMS-Port");
        smsView = new SMS_MS_View(this, exitCommand, createCommand);
        receiveModel = new SMS_MS_Model_Receive(this);
        sendModel = new SMS_MS_Model_Send(this);

        // get the Display object for this MIDlet and pass it to the View
        smsView.setDisplay(Display.getDisplay(this));
}

// Start the application
public void startApp()
{
        // Open a new connection
        receiveModel.openNewConnection("sms://:" + smsPort);
        // Start the receive Thread
        receiveModel.start();

        // Display the screen
        smsView.setCurrentDisplay();
}

// A new message has just been received better retrieve it
public void notifyIncomingMessage(MessageConnection conn)
{
        // A new message was received and we should start the Thread
        receiveModel.start();
}

// A new message has been received and we should inform the View
public void messageReceived(String message, String senderAddress)
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        smsView.setDestinationAdressString(senderAddress.substring(6));
        smsView.displayMessage(message, doneCommand, replyCommand);
}

// Pause the application
public void pauseApp()
{
  receiveModel.pause();
  smsView.setDisplayable(smsView.getDisplayable());
}

// Time to close the application
public void destroyApp(boolean unconditional)
{
        receiveModel.destroy();
}

// The user pressed a button and we should handle it
public void commandAction(Command c, Displayable s)
{
  try
  {
    if (c == Alert.DISMISS_COMMAND || c == exitCommand)
    {
      destroyApp(false);
      notifyDestroyed();
    }
    else if (c == doneCommand)
    {
                        // return to the main screen
                        smsView.setCurrentDisplay();
                }
                else if (c == replyCommand)
                {
                        // we already know the address so just retrieve the message
                        smsView.inputMessage(backCommand, sendCommand);
                }
    else if (c == createCommand)
    {
      smsView.inputDestination(exitCommand, nextCommand);
    }
    else if (c == nextCommand)
    {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    // we better check that the phone number is correct
                    if(!sendModel.isPhoneNumberValid(smsView.getDestinationAddressString()))
                    {
                            smsView.setErrorMessage("Invalid phone number");
                            smsView.setCurrentDisplay(smsView.getError(),
smsView.getDestinationAddress());
                            return;
            }
                    smsView.inputMessage(backCommand, sendCommand);
                }
                else if (c == backCommand)
                {
                    smsView.inputDestination(exitCommand, nextCommand);
                }
                else if (c == sendCommand)
                {
                    // Notify the user that the message will be sent
                    smsView.setSendDisplayString("Sending message to " +
smsView.getDestinationAddressString() + "...");
                    smsView.setCurrentDisplay(smsView.getSendDisplay());

                    // Tell the Model to send the message
                    sendModel.setDestinationAddressString(smsView.getDestinationAddressString());
                    sendModel.setSMSPort(smsPort);
                    sendModel.setMessage(smsView.getMessageString());
                    sendModel.start();
                }
        }
        catch (Exception ex)
        {
          ex.printStackTrace();
        }
      }
    }
```

Appendix E.2 The Model for Receiving SMS messages

```
import javax.wireless.messaging.MessageConnection;
import javax.wireless.messaging.Message;
import javax.wireless.messaging.TextMessage;
import javax.wireless.messaging.BinaryMessage;

import javax.microedition.lcdui.Alert;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
import javax.microedition.io.Connector;

import java.io.IOException;

public class SMS_MS_Model_Receive implements Runnable
{
   // Create global variables
   Thread thread;
   MessageConnection smsconn;
   SMS_MS_Controller smsController;
   Message msg;
   String senderAddress;

   SMS_MS_Model_Receive(SMS_MS_Controller smsController)
   {
         // Initialize variables
         this.smsController = smsController;
         smsconn = null;
         thread = null;
   }

   // Open a new SMS connection for receiving text messages
   public MessageConnection openNewConnection(String smsConnection)
   {
         if (smsconn == null)
         {
               try
               {
                     smsconn = (MessageConnection) Connector.open(smsConnection);
                     smsconn.setMessageListener(smsController);
               }
               catch (IOException ioe)
               {
                     ioe.printStackTrace();
               }
     }

     return smsconn;
   }

   // Close the SMS connection
   public void closeConnection()
   {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
		this.destroy();
	}

	// Check to see if there are any new SMS messages
	public void start()
	{
		// Start receiving messages
		if (thread == null)
		{
			thread = new Thread(this);
			thread.start();
		}
	}

	// Retrieve the new message
	public void run()
	{
		String message = new String("");

		// Check for sms connection
		try
		{
			msg = smsconn.receive();
			if (msg != null)
			{
				senderAddress = msg.getAddress();
				if (msg instanceof TextMessage)
				{
					message = ((TextMessage)msg).getPayloadText();
				}
				else
				{
					StringBuffer buf = new StringBuffer();
					byte[] data = ((BinaryMessage)msg).getPayloadData();
					for (int i = 0; i < data.length; i++)
					{
						int intData = (int)data[i] & 0xFF;
						if (intData < 0x10)
						{
							buf.append("0");
						}
						buf.append(Integer.toHexString(intData));
						buf.append(' ');
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    }

                    message = buf.toString();
                }
                smsController.messageReceived(message, senderAddress);
            }
        } catch (IOException e) {
            // e.printStackTrace();
        }
    }

    // Pause the receiving
    public void pause()
    {
        thread = null;
        return;
    }

    // Close the SMS Connection and stop the thread
    public void destroy()
    {
        thread = null;

        if (smsconn != null) {
            try {
                smsconn.close();
            } catch (IOException e) {
                // Ignore any errors on shutdown
            }
        }

        return;
    }
}
```

Appendix E.3 The Model for Sending SMS Messages

```
import javax.wireless.messaging.MessageConnection;
import javax.wireless.messaging.Message;
import javax.wireless.messaging.TextMessage;
import javax.wireless.messaging.BinaryMessage;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
import javax.microedition.io.Connector;
import javax.microedition.lcdui.Alert;

import java.lang.Thread;

import java.io.IOException;

public class SMS_MS_Model_Send implements Runnable
{
   // Create global variables
   Thread thread;
   MessageConnection sendSMSConn;
   SMS_MS_Controller smsController;
   Message msg;

   String senderAddress;
   String destinationAddress;
   String smsPort;
   String message;

   Alert content;

   SMS_MS_Model_Send(SMS_MS_Controller smsController)
   {
         // Initialize variables
         this.smsController = smsController;
         content = null;
         sendSMSConn = null;
         thread = null;
   }

   // Is this a valid phone number?
   public boolean isPhoneNumberValid(String address)
   {
         char[] chars = address.toCharArray();

         if (chars.length == 0)
               return false;

         int startPos = 0;

         // initial '+' is OK
         if (chars[0]=='+')
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                startPos = 1;

        for (int i = startPos; i < chars.length; ++i)
                if (!Character.isDigit(chars[i]))
                        return false;

  return true;
}

// Set the destination address
public void setDestinationAddressString(String destinationAddress)
{
        this.destinationAddress = destinationAddress;
}

// declare the SMS port to send the message on
public void setSMSPort(String smsPort)
{
        this.smsPort = smsPort;
}

// set the message
public void setMessage(String message)
{
        this.message = message;
}

// Time to send the message so we better start the Thread
public void start()
{
        // Send a message
        if (thread == null)
        {
                thread = new Thread(this);
                thread.start();
  }

}

// Send the message to the destination address
public void run()
{
        String address = "sms://" + destinationAddress + ":" + smsPort;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                sendSMSConn = null;
                try
                {
                        // Open the message connection
                        sendSMSConn = (MessageConnection)Connector.open(address);

                        TextMessage txtmessage =
(TextMessage)sendSMSConn.newMessage(MessageConnection.TEXT_MESSAGE);
                        txtmessage.setAddress(address);
                        txtmessage.setPayloadText(message);
                        sendSMSConn.send(txtmessage);
                }
                catch (Throwable t)
                {
                        System.out.println("Send caught: ");
                        t.printStackTrace();
                }
                if (sendSMSConn != null)
                {
                        try
                        {
                                sendSMSConn.close();
                        }
                        catch (IOException ioe)
                        {
                                System.out.println("Closing connection caught: ");
                                ioe.printStackTrace();
                        }
                }
        }
    }
```

Appendix E.4 The View for displaying received messages and for composing new messages

```
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.TextBox;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
import javax.microedition.lcdui.TextField;

public class SMS_MS_View
{
   // Create global variables
   Alert contentAlert;
   Alert errorAlert;
   Alert sendingAlert;

   SMS_MS_Controller smsController;

   Display display;
   Displayable screen;

   TextBox destinationAddressBox;
   TextBox messageBox;

   SMS_MS_View(SMS_MS_Controller smsController, Command exitCommand, Command createCommand)
   {
         // Initialize variables
         this.smsController = smsController;

         contentAlert = new Alert("A Generic MS SMS Application");
         contentAlert.setTimeout(Alert.FOREVER);
         contentAlert.addCommand(exitCommand);
         contentAlert.addCommand(createCommand);
         contentAlert.setCommandListener(smsController);
      contentAlert.setString("What would you like to do now?");

      errorAlert = new Alert("SMS", null, null, AlertType.ERROR);
      errorAlert.setTimeout(5000);

      sendingAlert = new Alert("SMS", null, null, AlertType.INFO);
      sendingAlert.setTimeout(5000);
      sendingAlert.setCommandListener(smsController);

      destinationAddressBox = new TextBox("Destination Address?", null, 256, TextField.PHONENUMBER);

      // Initialize the viewing screen
      screen = contentAlert;
   }

   // Set the display object for the mobile device
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
public void setDisplay(Display display)
{
        this.display = display;
}

// Set the display object and the display for the mobile device
public void setCurrentDisplay(Displayable screen)
{
        this.screen = screen;
        display.setCurrent(screen);
}

// Set the display for the mobile device
public void setCurrentDisplay()
{
        display.setCurrent(screen);
}

// Set the display for the mobile device
public void setCurrentDisplay(Alert error, TextBox textbox)
{
        display.setCurrent(error, textbox);
}

// Set the display for the mobile device
public void setCurrentDisplay(TextBox textbox)
{
        display.setCurrent(textbox);
}

// Set the display for the mobile device
public void setCurrentDisplay(Alert alert)
{
        display.setCurrent(alert);
}

// Set the display for the mobile device
public void setDisplayable(Displayable screen)
{
        this.screen = screen;
}

// Return the display object for the mobile device
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
public Displayable getDisplayable()
{
        return display.getCurrent();
}

// Display the received message
public void displayMessage(String textMessage, Command doneCommand, Command replyCommand)
{
        Alert message = new Alert("");

        message.setTitle("From: " + getDestinationAddressString());
        message.setTimeout(Alert.FOREVER);
        message.addCommand(doneCommand);
        message.addCommand(replyCommand);
        message.setCommandListener(smsController);
        message.setString(textMessage);

        setCurrentDisplay(message);
}

// Alter the content Alert string
public void setContentString(String string)
{
        contentAlert.setString(string);
}

// Composing part of the View
public void inputDestination(Command exitCommand, Command nextCommand)
{
        destinationAddressBox.addCommand(exitCommand);
        destinationAddressBox.addCommand(nextCommand);
  destinationAddressBox.setCommandListener(smsController);

        setCurrentDisplay(destinationAddressBox);
}

// Specifying an error
public void setErrorMessage(String errorAlertMessage)
{
        errorAlert.setString(errorAlertMessage);
}

// Returning the error object
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
public Alert getError()
{
        return errorAlert;
}

// Returning the destination address object
public TextBox getDestinationAddress()
{
        return destinationAddressBox;
}

// Return the destination address string
public String getDestinationAddressString()
{
        return destinationAddressBox.getString();
}

// Specify the destination address as a String
public void setDestinationAdressString(String addressString)
{
        destinationAddressBox.setString(addressString);
}

// Specify the sending message
public void setSendDisplayString(String sendString)
{
        sendingAlert.setString(sendString);
}

// Retrieve the user's message
public void inputMessage(Command backCommand, Command sendCommand)
{
        messageBox = new TextBox("Enter Message", null, 65535, TextField.ANY);
        messageBox.addCommand(backCommand);
        messageBox.addCommand(sendCommand);
   messageBox.setCommandListener(smsController);

   setCurrentDisplay(messageBox);
}

// Return the user's message
public String getMessageString()
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            return messageBox.getString();
    }

    // Sending Message part of the View
    public Alert getSendDisplay()
    {
            return sendingAlert;
    }
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.