Theses and Dissertations   Student Publications

5-2010

Implementing the Intelligent Mail Barcode in the N-Tiered Service Library of a Print Mail Enterprise

Christopher E. Bunch
Columbus State University, bunch_christopher@columbusstate.edu

www.manaraa.com

IMPLEMENTING THE INTELLIGENT MAIL
BARCODE IN THE N-TIERED SERVICE
LIBRARY OF A PRINT MAIL ENTERPRISE

Christopher Bunch

Columbus State University

The College of Business and Computer Science

The Graduate Program in Applied Computer Science

Implementing the Intelligent Mail Barcode in the N-Tiered Service Library of a Print Mail Enterprise

A Thesis in

Applied Computer Science

by

Christopher E Bunch

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2010

I have submitted this thesis in partial fulfillment of the requirements for the degree of Master of Science
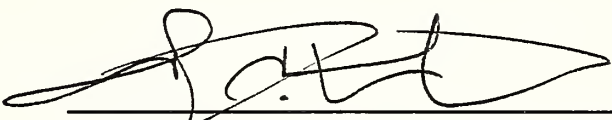
May 6, 2010
Date

Christopher E. Bunch

We approve the thesis of Chris Bunch as presented here.

May 6, 2010
Date
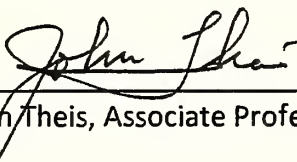
Christopher C. Whitehead, Assistant Professor of Computer Science, Thesis Advisor

05/06/10
Date

V. Zaniv
Vladimir Zaniv, Professor of Computer Science

May 6, 2010
Date

John Theis, Associate Professor of Finance

Acknowledgements

I could never have completed the work resulting in this document without the assistance of those mentioned below.

Table of Contents

List of Figures

Abstract

Starting in autumn, 2009, the Intelligent Mail Barcode fully replaced the PostNet barcode for the United States Postal Service. This barcode enables a sender of a mailpiece to track the mailpiece through the entire mail stream, as well as track any remit mail returned to the sender. This thesis explains how the Intelligent Mail Barcode was implemented in the n-tiered Windows Communication Foundation service architecture of the Emdeon, Inc. print-mail engine. To help provide a full understanding of the environment, this document, also, explains the operation of the print mail engine at Emdeon.

Introduction

In 2006 the United States Postal Service (USPS) released Intelligent Mail Barcode (IMB) as is a service designed to both improve the speed of mail delivery and provide functionality for a mail sender to track the delivery of the mailpiece. A user of IMB is required to meet the specifications set forth by the USPS. This thesis demonstrates an implementation of IMB in a print-mail engine where the operations are processed by an n-tier service library. The print-engine in the demonstration is owned by Emdeon, Inc.

The first chapter of this thesis is devoted to the history, purpose, and requirements of the Intelligent Mail Barcode (IMB). These requirements were established by the USPS and must be met by any person or enterprise intending to use the services offered by IMB.

The second chapter of this document surveys WCF to establish the working environment. The chapter defines WCF, and illustrates how WCF services differ from web services. Each WCF service requires three components: contracts, endpoints, and hosting the service. Each of these components will be covered in the first chapter.

The third chapter is divided into two sections. The first section describes the Emdeon PBPS print-mail engine, and how *Patient Connect*, the application used at Emdeon to administer print mail jobs, helps create and administer the print-mail jobs. The second section describes the print-mail job execution managed by a BizTalk orchestration and processed by WCF services. The intent of both sections is to explain the business requirements of the Emdeon print mail engine.

The fourth chapter demonstrates how IMB was implemented at Emdeon. The n-tiered architecture of the print mail engine allowed IMB to be easily implemented. Only two items needed to be changed. The first item is a stored procedure, and the second item is a class in the core framework used by all Emdeon PBPS systems.

Chapter 1 – Intelligent Mail Barcode

Using direct advertising to further the pursuits of commerce has been in use since at least 1000 B.C. when an Egyptian landowner might write an advertisement for the return of a runaway slave on a piece of papyrus.[1] In the times of Babylonia, messages were often placed on bricks and sent directly to an intended recipient.[1] Though it was present early in the history of civilization, since only a small portion of the population was able to read, mail was not a prevalently used service until much later.[1] Even after mail became a common service, since each message had to be hand written, it was still difficult for a business to use the service effectively.[1] Around 1434, Gutenberg invented movable type, giving birth to the printing industry.

In the 20th century, the print-mail industry improved marketability with the addition of computers to manage the printers.[14] Computers allowed for a client to send the mailer a file in a format agreed upon, and the mailer could then parse the file and send the proper instruction to the print machine. When the machine completed its work, an envelope containing the mailpiece was ready to be shipped.[1]

Today, the list of technologies that enable the efficient production of print-mail is extensive. For example, a printer may develop an in-house system to parse incoming client files and save the files in an extensible mark-up language (XML) format to a specified shared drive. A second program can parse the XML based on rules supplied by the client (i.e. determine which fields go where on the mailpiece and the envelope). A third program can transform the finished XML into a format readable by the printers

---

[1] History of Direct Mail. <http://www.direct-mail.org/history.htm>

and send the document to the printers. The print-mail engine at Emdeon, discussed in follow on chapters, operates similarly to this.

Any of the steps in the example can be expanded or even removed based on the service level of the print company. The higher the level of service, the more options are available to the customer. The availability of additional options requires the print-mail company to have an ability to fulfill the requirements of the option. Although starting a small print-mail company is relatively easy, making the company become a competitor for the larger print mailers requires a well-designed architecture for managing and processing the print requests.

Intelligent Mail Barcode Evolution and Specifics[2]

With print mail, there is no way to know when the customer received the document(s), and, unless certified mail is used, there is no defense if the recipient claims to have never received the document(s). The Intelligent Mail Barcode not only solves this problem for mailers, but also provides a way for the mailer to anticipate the arrival of any remit mail. This section will explore the specifications of the Intelligent Mail Barcode.

The USPS has long provided a postage discount to mass mailers if they used a barcode in the mail-to address.[16] Previously, the barcode used was the PostNet barcode. The PostNet eased the processing requirements of the USPS in delivering the

---

[2] Intelligent Mail Barcode
<https://ribbs.usps.gov/intelligentmail_mailpieces/documents/tech_guides/SPUSPS-B-3200E001.pdf>

mailpiece, and increased dependability that the mail would be delivered accurately and swiftly. But the PostNet did not provide tracking information to the sender. PostNet stands for **Post**al **N**umeric **E**ncoding **T**echnique. It is a method to encode numeric values in a 2-dimensional barcode (see below). This numeric encoding made it simple for the USPS to scan a letter to determine the mail-to zip code.

Value	Encoding
1	..‖
2	.‖.
3	.‖.
4	‖..
5	‖.‖.
6	‖..
7	‖..‖
8	‖.‖.
9	‖.‖..
0	‖...

Figure 1: 2-Dimensional Barcode Digit Encoding

The original PostNet encoded only six digits: the five digit zip-code and a single check digit used for validation, but the most recent version encoded 12 digits (ZIP+4, the delivery point, and the check digit).[3] There have been four revisions, and the last revision allowed the USPS to sort mail into a delivery point sequence. Every mailbox is a separate delivery point.[4] Sorting the mail in delivery point sequence improves the speed of delivery. This PostNet contains all the information needed to get the mail to the mail-to address.

---

[3] POSTNET <http://en.wikipedia.org/wiki/POSTNET>.
[4] Delivery Points < <http://en.wikipedia.org/wiki/Delivery_point >

The check digit is a single digit encoded to the end of the barcode that creates a method for the USPS to verify the validity of the barcode before using it to send the mailpiece. The check digit is generated by first adding all the digits in the zip code and delivery point.[18] For instance, if the ZIP+4 and delivery code were 37071140711, then the first step in finding the check digit would be finding the sum 3+7+0+7+3+1+4+0+7+1+1 or 34. Next, find the mod10 of the sum. The Mod10 of the sum is the remainder when the sum is divided by 10; in the previous example, the mod10 of 34 would be 4. The last step is to subtract the value from 10. In the example, the final check digit would be 10 – 4 or 6. Since there are only ten possible check digits, it does not provide complete validation, but it helps ensure no digits were mistyped by the shipping department and the barcode has not been corrupted during transit.

After the fourth revision, the PostNet barcode contained 62 bars. As the USPS competed with private carriers, the demand grew to find functionality in the barcode that rivaled the services of other carriers that allowed senders to track packages while they are in route to their destination.[2] Meeting this demand would mean adding many more digits to the PostNet. It could not stand to grow any larger so another method of encoding the digits had to be found. This prompted the arrival of Intelligent Mail Barcode.

In 2003, the United States Post Office (USPS) announced the Intelligent Mail Barcode. In 2006, the barcode became available for use. The initial version of Intelligent Mail Barcode was an improvement, but mailers were initially unsatisfied with the process, so, in 2007, Intelligent Mail Barcode was enhanced, and the USPS released

the final version.[5] The final version of Intelligent Mail Barcode offers more reasons than

ever to use a barcode and, starting in Fall, 2009, the USPS required use of Intelligent

Mail Barcode to receive the barcode postage discount.[16]

Intelligent Mail Barcode is a 4 dimensional barcode (see Figure 2). More than

twice as much data can be contained in a four dimensional barcode than a two

dimensional barcode of equal length. Every new bar in a four dimensional barcode

increases the possible values for the entire code fourfold. So a four dimensional

barcode with five bars has 4^5 or 1024 possible values, whereas a two dimensional

barcode of the same length only has 2^5 or 32 possible values. After the bar length

constraint of the barcode was removed, adding the functionality demanded by the

public became a possibility.

Figure 2: 4-Dimensional Intelligent Mail Barcode

After improving the barcode became a possibility, the next logical step was to

determine how to improve it and what needed to be done to meet the demands of the

improvement. The USPS determined that five items would be used to generate an

Intelligent Mail Barcode: a barcode identifier, a service type identifier, a mailer

---

[5] Intelligent Mail Barcode <http://en.wikipedia.org/wiki/Intelligent_Mail_Barcode>

identifier, a serial number, and a routing code, which is the same value previously encoded in the PostNet.[16]

Tracking Code Components

- Barcode Identifier

  This is used to identify the type and purpose of the barcode. In the print-mail industry, '00' is most commonly used.

- Service Type Identifier

  The service type identifier identifies what service the USPS is to provide for the mailpiece, standard mail, first-class mail, etc.

- Mailer Identifier

  This value identifies the sender of the mail. The importance of this field in the USPS Mailpiece Tracking section will be evident later.

- Serial Number

  This identifies the individual piece of mail. There is no rule forcing this value to be unique, but if a mailer intends to reap any of the benefits offered by the Intelligent Mail Barcode, it would be negligent to allow for two identical serial codes to be in the mail stream simultaneously.

Routing Code Components

- Delivery Point Zip Code

  This is the entire value previously contained in the PostNet barcode. Ironically, this section of the barcode is optional.

Type	Field	Digits
	Barcode Identifier	2 (2nd digit must be 0 - 4)
	Service Type Identifier	3
	Mailer Identifier	6 or 9
		9 (when used with 6 digit Mailer ID)
Tracking Code	Serial Number	6 (when used with 9 digit Mailer ID)
Routing Code	Delivery Point ZIP Code	0, 5, 9, or 11
Total		31 Maximum

Figure 3: Intelligent Mail Barcode Field Specifications

If a package address is labeled with the Intelligent Mail Barcode, then the USPS tracks its progress to its destination by scanning the barcode at every facility it stops at on route to the destination.[16]   The USPS offers to send a log back to the mail sender that contains the zip codes of the facilities that scanned the mailpiece and a datetime stamp for each entry. It is the responsibility of the sender to develop their own systems that parse and interpret the logs.[16]

The most appreciable purpose of Intelligent Mail Barcode is mailpiece tracking. By parsing the USPS log, a company can track the mailpiece all the way to the destination to know the date the recipient received the messenger. Then, by including Intelligent Mail Barcode on the remit envelope, the sender can parse the log to discover the date the customer replies to the message. This is very important from an accounting perspective since it allows accounts receivable to anticipate funds from bill payments.

Chapter 2 - Windows Communication Foundation

In procedural oriented languages the emphasis is on the procedure, and all logic executed sequentially. For most of the history of computer programming, most mainstream applications were written using procedural oriented languages, but in the 1960s the seed was planted for object-oriented programming. This seed was planted when Kristen Nygaard and Ole-Johan Dahl in Norway created *Simula 67*, the first language to use object-orientation.[6]

Object –orientation brought many new concepts to the industry. Object-oriented programming brought many new concepts like encapsulation, inheritance, and interface implementation. It was not long before its introduction that object-oriented programming revolutionized the mainstream.[6]

A complete discussion of object-oriented programming is beyond the scope of this thesis, but a basic understanding of the allure of object-oriented programming is important in understanding the birth and rise of service-oriented architecture. In turn, understanding the birth and rise of service-oriented architecture is necessary in any mention of the history of Windows Communication Foundation.

The ideas to group procedures and functions in sets that exist in synergy first came to programmers in the 1960s.[6] They used the term *classes* for collections of procedures and functions, and object-oriented programming was born. Object-

---

[6] "The History of Object Oriented Programming" *Exforsys, Inc*
<http://www.exforsys.com/tutorials/oops/the-history-of-object-oriented-programming.html>

orientation gained a little more ground in the 1970s when *object-oriented techniques* were introduced to developers via the Lisp machine.[7]

In its second decade, in the 1980s, object-orientation finally took flight. There are many reasons the paradigm finally shifted enough in the development world to allow the explosion of object-orientation. The reasons include in the number of applications that operated in a server-client environment, the rising demand for a more user-friendly user interface for every program, the need for heightened security and encapsulation, the desire for a way to reuse code without copy/paste, and requirements to decouple sections of a program. Whatever the reasons, by the turn of the century object-orientation was the tool of choice in most development environments.

Unlike objects alone, services enable a developer to create an application, host it on a server and share the functionality with anyone who wants to use it without making them download and install any new products specific to the service.[8] Within an enterprise with a service-oriented architecture, applications can be light and easy. All of the functionality can be decoupled from the client interface and exist in services hosted elsewhere throughout the enterprise.[8] This fully encapsulates and decouples all the pieces of a program and lets servers perform all the heavy work.

If all of the services and clients in an architecture exist in the same environment (same operating system, same frameworks, applications developed with the same

---

[7] "Object-oriented Programming -." *Wikipedia, the Free Encyclopedia.* Web. 14 Mar. 2010.
<http://en.wikipedia.org/wiki/Object-oriented_programming>
[8] "Decoupled Contract (Erl)" SOA Patterns Web. 22 Feb. 2010
http://www.soapatterns.org/decoupled_contract.php

programming languages, etc.), then many of the hurdles commonly encountered during development are easily surmounted.[9] In the shrinking world of today, it is a mistake for a service-oriented architecture to assume anything about the environment and requirements of service consumers. These concerns inspired Microsoft's desire to expand the tools offered for *web services*. The culmination of this desire is Windows Communication Foundation (*WCF*). In the Microsoft Development Network (*MSDN*), Microsoft defines *WCF* as:

> The typed programming model (called the *service model*) is designed to ease the development of distributed applications and to provide developers with expertise in ASP.NET Web services, .NET Framework remoting, and Enterprise Services, and who are coming to WCF with a familiar development experience. The service model features a straightforward mapping of Web services concepts to those of the .NET Framework common language runtime (CLR), including flexible and extensible mapping of messages to service implementations in languages such as Visual C# or Visual Basic. It includes serialization facilities that enable loose coupling and versioning, and it provides integration and interoperability with existing .NET Framework distributed systems technologies such as Message Queuing (MSMQ), COM+, ASP.NET Web

---

[9] Service Oriented Architecture (SOA) and Specialized Messaging Patterns
http://www.adobe.com/enterprise/pdfs/Services_Oriented_Architecture_from_Adobe.pdf

services, Web Services Enhancements (WSE), and a number of other functions.[10]

Prior to WCF, in an effort to maximize interoperability, Microsoft had to create separate Application Programming Interfaces (API) for web services, .Net remoting, message queues, and distributed transactions, even though all four APIs are based on the Standard Object Access Protocol (SOAP).[11] SOAP is a XML-based messaging protocol that defines a set of rules for structuring messages. [12] By using SOAP messages between processes, WCF can make any SOAP-based application interoperate with any other SOAP based application or process.

Development of a service using WCF requires three things: contract definitions, endpoint definitions, and hosting the service.[13] Each of these three components is discussed briefly below. To learn more than is found in this document, each of the three components can be studied at length in many of the references of this thesis.

Windows Communication Foundation – Contract Definition

In object-oriented programming, many classes will implement an interface that has been supplied to users of the class. This interface will contain public methods within the class, and one of the purposes of interfaces is to supply consumers with the

---

[10] What is Windows Communication Foundation: <http://msdn.microsoft.com/en-us/library/ms731082.aspx>

[11] Sharp, John. *Microsoft Windows Communication Foundation Step by Step.*

[12] Soapuser.com <http://www.soapuser.com>

[13] What Is Windows Communication Foundation? <http://msdn.microsoft.com/en-us/library/ms731082.aspx>

signatures of any methods they may call. WCF contracts form a similar purpose and are often built using an interface.

Microsoft states[14] that a service contract must specify the five following items:

- *The operations a service exposes.*

- *The signature of the operations in terms of messages exchanged.*

- *The data types of these messages.*

- *The location of the operations.*

- *The specific protocols and serialization formats that are used to support successful communication with the service.*

To change an interface into a service contract, two important attributes must be added to the definition: *ServiceContract* and *OperationContract*. The service contract identifies the interface as a service, and the operation contract identifies the operations of the service. An example of WCF service code is below, for the purpose of the example the contract attributes have been bolded.

```
[ServiceContract]
public interface IHelloWorldService
{
    [OperationContract]
    string HelloWorld (string parameter);
}
```

Figure 4: WCF Service Code

Once the contract is set, the class functionality must be added. In the example above, a class would be developed that implements IHelloWorldService and must

---

[14] Designing and Implementing Services <http://msdn.microsoft.com/en-us/library/ms729746.aspx>

contain a method named HelloWorld that accepts a string parameter and returns a string result. Of course, just as an interface is not mandatory to develop a class, an interface is not mandatory to develop a WCF service. The contract can be set only in the class itself, but many enterprises prefer to use interfaces since interfaces better allow future changes to code and add flexibility to implementation.[15]

Windows Communication Foundation – Endpoint Definition

Before communication with the service can start, the client has to know where to go, how to get there, what will be there, and how it will behave. Endpoints are defined to enable the client to do these things. Microsoft states this precisely in MSDN: *"All communication with a Windows Communication Foundation (WCF) service occurs through the endpoints of the service. Endpoints provide clients access to the functionality offered by a WCF service".*[16]

There are four things contained in every endpoint:

- Address-*where to go*

- Binding information – *how to get there*

- Contract – *who will be there*

- Local implementation details – *how it will behave*

Every service requires a unique address. The address is used to reach a specific service and no other. The WCF object model contains an EndpointAddress class. The URI property and the Identity of this class are used to specify how the service can be

---

[15] Microsoft Development Network <http://msdn.microsoft.com/en-us/library/3b5b8ezk(VS.71).aspx>
[16] Microsoft Developer Network: <http://msdn.microsoft.com/en-us/library/ms733107.aspx>

reached. The address of the service is in the Uri property, and the security identity and any optional headers are contained in the identity class.

A WCF service can use several forms of transport protocol, message encoding, and security requirements. Information about acceptable forms can be found in the binding information of the service. This specification is used to determine how exactly to communicate with the service (i.e. does the service use TCP or HTTP, SSL or SOAP message security, text or binary messaging, etc.). Through the services binding information, all of these properties of the service can be passed to the client.

If the client knows where the service can be found and how it communicates, it is almost ready to consume the service. However, before it can, it has to know the signatures of all available methods. Each message signature contains the form of the message, the number, order, and data type of all required parameters, and what kind of processing or response message is to be expected. All of this information is contained in the contract in the endpoint.

With the three items above (Address, Binding, and Contract) a client can find and communicate with a WCF service. However, another property of the endpoint further extends WCF functionality. The local implementation details of the service are used to customize the local behavior of the endpoint. The endpoint uses these settings to participate in the construction process of a WCF runtime. For example, the address used to access the service may be different than the address used to listen to the service. This can be handled through the ListenUri property.

Windows Communication Foundation – Hosting

After the service is created, it is not active until it is hosted within a run-time environment. The run-time environment manages the service by controlling the context and lifetime. Any Windows process that supports managed code can host a WCF service. Developers can choose the hosting environment based on the service deployment requirements, such as the application platform, transport requirements, or the process management requirements to ensure availability. The four potential hosting environments are Self-Hosting in a Managed Application, Managed Windows Service, Internet Information Services (IIS), and Windows Process Activation Service (WAS). Microsoft's description[17] of each of these hosting environments is below.

Any managed application can host WCF services. Since this requires the least deployment infrastructure, it is considered the most flexible. The service code is embedded inside the managed application, and then, to make the service available, simply create and open an instance of the ServiceHost.

Two common scenarios are enabled by the approach above: WCF services running inside console applications and rich client applications. Typically, the most use for hosting a WCF service inside a console application manifests during the development phase of an application. This is because hosting the service inside a console application makes the service easy to debug, easy to trace, and easy to copy the application and paste it in new locations.

---

[17] Hosting Windows Communication Foundation Services <http://msdn.microsoft.com/en-us/library/ms730158.aspx>

A WCF service can also be hosted by managed windows services. In this case, the process lifetime of the service is controlled by the service control manager for Windows services. This type of hosting also requires hosting code be written as part of the application. The service inherits from the ServiceBase class as well as the WCF service contract interface, since it is implemented as both a WCF service and a Windows service. An overridden OnStart method creates and opens the ServiceHost, and an overridden OnStop method closes it. An installer class must be implemented that inherits from Installer to allow the program to be installed as a Windows Service. The operating system controls the lifetime of the service, and all versions of Windows support this hosting option.

The Internet Information Services (IIS) hosting option is integrated with ASP.Net. These technologies offer additional features such as process recycling, idle shutdown, process heath monitoring, and message-based activation.[17] This is the preferred solution for hosting highly available and highly scalable Web service applications. IIS must be properly configured to use this hosting method, but the service code does not have to be written as part of the application.

The new process activation mechanism for Windows Server 2008 and Windows Vista is Windows Process Activation Service (WAS).[17] It retains the application pools, message-based process activation, failure protection, health monitoring, and recycling found in IIS 6.0, but it removes the dependency on HTTP from the activation architecture.[17]

After the WCF services have been developed, contracts set, endpoints defined, and the services are hosted properly, the only work remaining for the service to be consumed and used is on the part of the client. One of the best features of WCF services is their ease of implementation. To implement a WCF service in Microsoft Visual Studio, a developer simply creates a new project, right clicks the project name and selects Add New Service Reference, and supplies the endpoint address (URI) for the service. Then the Integrated Development Environment (IDE) of Visual Studio uses the endpoint address to review the contracts and makes all the available methods visible to the client.
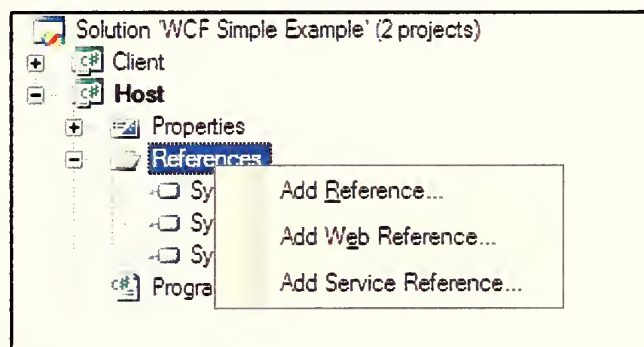
Figure 5 : Adding Service Reference to a >Net Project in Visual Studio 2008

WCF eases development, configuration, and implementation of business-to-business communication and business-to-consumer communication, improves interoperability with other platforms and technologies, lessens the restrictions on messaging formats, and improves message routing. Many service-providing and service-consuming enterprises are reaping the benefits offered by WCF. Later, in the following

sections, this thesis will explore how WCF powers the print-mail engine for Emdeon, and how the services were enhanced to implement Intelligent Mail Barcode.

Chapter 3 - Print Mail Operations at Emdeon Patient Billing and Payment Solutions Business Unit

The Emdeon Patient Billing and Payment Solutions business unit is responsible for the creation and mailing of over ten million products per month. The service level agreement (SLA) between Emdeon and most of its print-mail clients requires products be in the mail stream within 24 hours (not including Sundays) of receipt of the client's request. This SLA could not be met without an effective and robust print–mail engine.

The print engine at Emdeon uses the following subsystems, each existing on its own server cluster.

- Informatica Data Transformation Studio™

- Database

- Job Design

- WCF Services

- First Logic™

- StreamServe™

This chapter focuses on the database, job design, and WCF services, but the role of the other three components listed above is worthy of note.

Informatica Data Transformation Studio (IDTS)

This third party tool is used to write parse jobs to parse files sent to Emdeon from print-mail customers. After parsing the files jobs created with IDTS insert the parsed data into the print-mail database covered in a follow on section of this chapter.

First Logic

First Logic is a third party service used to validate addresses. Employing the First Logic service further prevents loss of resources for both Emdeon and its print-mail clients.

StreamServe

StreamServe is another third party tool, and it is the last tool to touch a product before it gets to the printer. StreamServe uses the finished product XML to create printer command language that can be read by the printers.

Database

Emdeon employs Microsoft SQL Server for the database needs. SQL Server 2005 was chosen because of the ease of stored procedure development using SQL Server Management Studio, the full compatibility with the .Net framework, and its relative low cost compared to Oracle, another powerful database server. In this section this thesis explores the data model powering the Emdeon Patient Billing and Payment Solutions print-mail operations.

The complete data model uses more than fifty tables, but the object of this thesis is concerned specifically with nine of these tables, listed below.

- Company
- Job
- Rule_Set
- Rule_Map

- Rule

- Session

- Product

- Xml_Data

- Xml_Path

The first seven tables contain the necessary execution data for the print-mail engine. The latter two contain the core data that determines what is printed on the documents.

 The top-level table, *Company*, contains, as its name suggests, company data. Each record in this table relates to a specific print-mail customer of Emdeon. Each company has one or more print-mail jobs, and the header data needed to perform these jobs is contained in the *Job* table.

After the company and the job have been created, tasks need to be added to the job, and that is done with the next three tables. The function of these tables is not quite as evident as the first two.
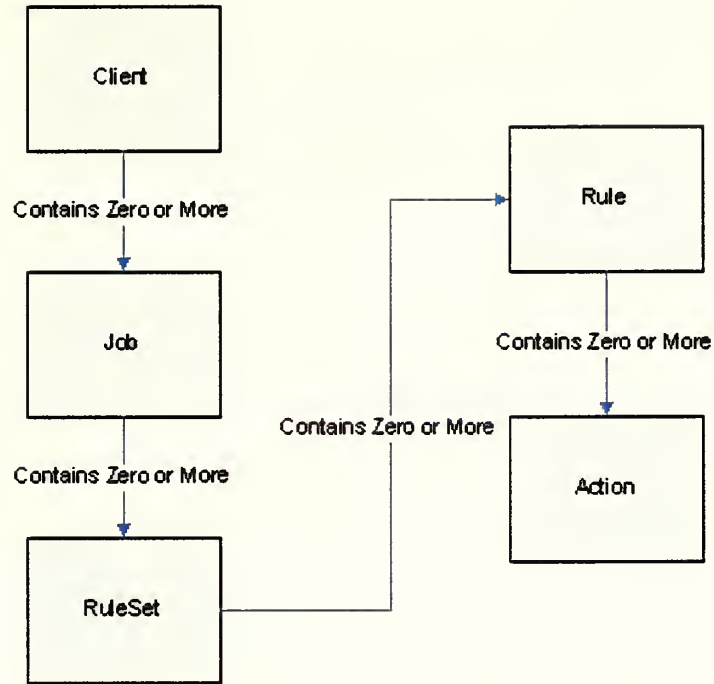
Figure 6: Parent Child Hierarchy from Client to Action

The different tasks are called rules, and, not surprisingly, these rules are stored

in the *Rules* table. The rules are contained within rulesets, stored in the *Rule_Set* table,

and each job contains rulesets. The purpose of the *Rule_Map* table is the least evident,

but it is of supreme importance for reusability. As can be seen in the data model in

Figure 4, the tables are joined by foreign keys that tie a table to its parent. If rule data

contained a foreign key for a rule set, then a rule could exist in one, and only one, rule

set. Many rules can be used in multiple jobs, and, therefore, must be contained in

multiple rule sets. The *Rule_Map* table allows for this by containing a foreign key for

rule and another foreign key for rule set.

The first five tables are static. Data is only inserted when a new company, new

job, new rule set, or new rule is created. The data is only updated in the semi-rare

occurrence that a company, job, or rule needs to change. But the data in the next two

tables, *Session* and *Product,* are more dynamic. Every time a job executes, it creates a

session for the job. Every document created in a job session is a product.

The *Xml_Path* table is the most static table of the nine. This table stores a

relational database translation of the XML schema definitions of the printed documents.

The *Xml_Data* table contains all of the document data, and it is the largest of all the

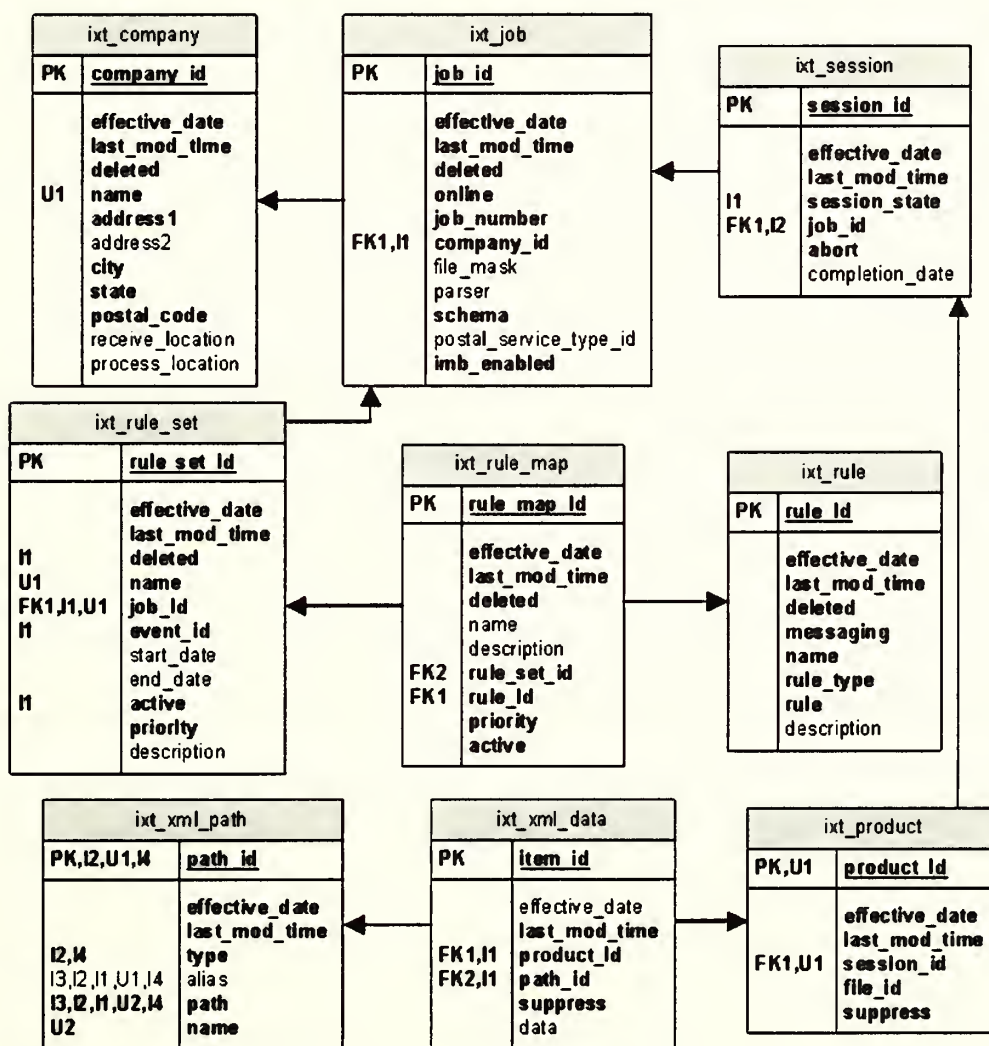tables. When a job executes a session, the rules transform the data contained in this

table.

Figure 7: Emdeon PBPS Print Mail Data Mo

Job Design

Job rules need to be created and configured, when a job is created. There are two separate applications at Emdeon that can be used to complete job configuration. The first application, *Enterprise Studio*, was developed in 2005 using .Net 1.0, and it is nearing depreciation. The first complete version of the second application, *Patient Connect*, was released in January of 2010.

Enterprise Studio contains functionality that allows for lower level configuration than *Patient Connect*, and, for this reason, many more technical users still use the application. *Patient Connect* is built using an n-tier architecture. In a 3-tier architecture, the presentation layer, business rules layer, and the data layer of an application are all encapsulated. These three tiers are mandatory in n-tier architecture. The sections below will cover how client, jobs, and rules are created using *Patient Connect*.

Job Design – Patient Connect Overview

The user interface (UI) classes contain most of the code for *Enterprise Studio*. The tight coupling of the UI and the code presented several problems for updating, testing, and deploying any code changes. During the design phase of *Patient Connect* the decision was made to decouple the code, and to use the Model-View-Presenter (MVP) design architecture as the method to address this decoupling.

MVP addresses UI code separation and facilitates automated unit testing.[18] The three elements of MVP and their responsibilities are:

- Model: An interface that defines the data to be used in the UI; the model contains any database commands to be executed by the application.

- View: The UI forms; the view contains all the interactive controls (buttons, textboxes, labels, etc.) used by the UI.

- Presenter: The bridge between the model and the view; all commands initiated by the view are passed to the model through the presenter, and the presenter *presents* data to the view after it is returns from the model.[19]

In *Patient Connect* the views all implement interfaces. The presenters contain a reference to the view interfaces. This allows for deployment and testing of the presenter logic while it is absent from the view.

The following section uses Client Creation as an example to demonstrate how *Patient Connect* implements the MVP architecture design to facilitate client, job, and rule creation and viewing of finished product XML data.

Job Design – Client Creation

Figure 4 shows the first screen in the *Patient Connect* client creation wizard. As a user completes the wizard, they are required to enter name and address information for the client and other account information that is beyond the scope of this thesis.

---

[18] Design Patterns: Model View Presenter <http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S1>

[19] Model-view-presenter <http://en.wikipedia.org/wiki/Model-view-presenter>
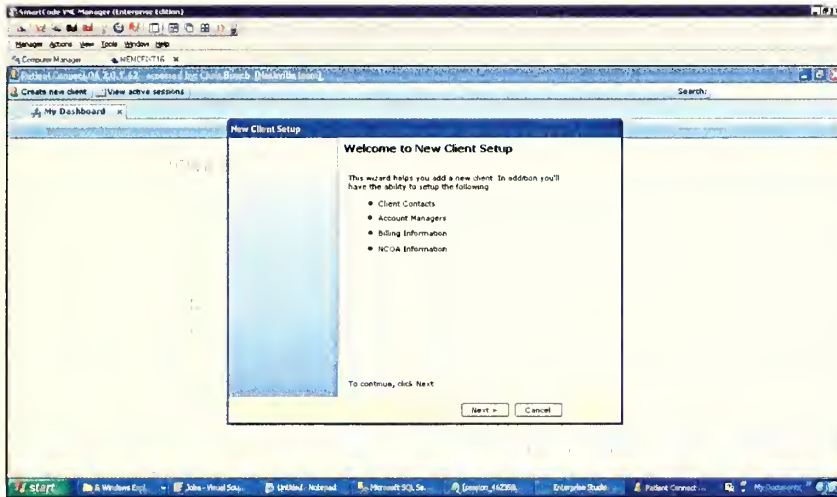
Figure 8: Patient Connect New Client Setup Wizard

The code in Figure 4 is executed when a user clicks *finish* in the wizard.

```
private void wizardControl_FinishButtonClick(object sender, EventArgs e)
{
    // Keep user from clicking wizard buttons after finish is clicked.
    wizardControl.BackButtonEnabled = false;
    wizardControl.NextButtonEnabled = false;
    wizardControl.CancelButtonEnabled = false;

    if (_presenter.OnFinish())
    {
        DialogResult = DialogResult.OK;
    }
    else
    {
        DialogResult = DialogResult.Abort;
    }
        Hide();
}
```

Figure 9: Patient Connect _view.WizardControl_OnFinishClick

The _presenter object referenced in the condition in the above code is a modular level

variable. It is declared as: `private CreateClientPresenter _presenter;`.

```
public bool OnFinish()
{
    bool success = true;
    string message = null;

    try
    {
        var newClient = new                         .Client
        {
            Name = _view.ClientName,
            Address1 = _view.Address1,
            Address2 = _view.Address2,
            City = _view.City,
            State = _view.State,
            Zip = _view.Zip,
            URL - _view.URL,
            TeamId = _view.TeamId,
            OracleId = _view.OracleId,
            PafName = _view.pafName,
            PafTitle = _view.pafTitle,
            PafPhone = _view.pafPhone,
            PafDate = _view.pafDate,
            PafLicenseDate = _view.pafLicenseDate,
            MoveMonths = _view.MoveMonths,
            TaxId = _view.TaxId,
            Contacts = _contacts,
            AccountManagers = _view.SelectedAccountManagers,
            CreatedBy = _view.CreatedBy
        };

        _view.ClientId = _model.AddNewClient(newClient);
    }
    catch (Exception ex)
    {
        success = false;
        message = ex.Message;
    }

    if (!success)
    {
        ThreadingException = message;
    }

    return success;
}
```

Figure 10: Patient Connect _presenter.OnFinish

The _view object, referenced in the code above, like the _presenter object

mentioned earlier, is a modular level object. Unlike the _presenter object, the _view

object is an interface (`private readonly ICreateClient _view;`). The

CreateClient view implements ICreateClient, and during instantiation of the _presenter

object within the view, a reference to the view is passed as a parameter (`_presenter`

= new `CreateClientPresenter(this, _client);`). This reference

becomes the _view_ object within the presenter, and provides full control of the view to

the presenter.

The call to the model in the code above (`_view.ClientId =`

`_model.AddNewClient(newClient);`) prompts the model to perform all work

necessary to create the client in the database. In most MVP applications it is within this

call to the model that all database commands would be generated and the call to the

database completed, but _Patient Connect_ also implements a service oriented

architecture (SoA). So, instead of calling the database, the model calls the service tier.

The service then uses a method within an infrastructure library to communicate with

the database.

```
public int AddNewClient(Client newClient)
{|
    return _service.AddNewClient(newClient);
}
```

Figure 11: Patient Connect _model.AddNewClient

```
public int AddNewClient(Client newClient)
{
    try
    {
        var systemId = IXTWrapper.NewClientSystemId;

        var location = new SystemRepository().GetProcessingLocations();

        string formattedName = Functions.FormatAsAlphaNumeric(newClient.Name);

        newClient.ProcessLocation = Functions.CreateClientFolder(location.ProcessRoot, formattedName);
        newClient.ReceiveLocation = Functions.CreateClientFolder(location.StageRoot, formattedName);
        newClient.ReportLocation = Functions.CreateClientFolder(location.ReportRoot, formattedName);

        var clientId = new ClientRepository().AddNewClient(newClient);
        new WebSiteRepository().CreateCustomerPortal(systemId, clientId);

        LogChange(systemId, Enums.AuditType.Company, clientId, newClient.CreatedBy, "Created " + newClient.Name);
        return clientId;
    }
    catch (Exception ex)
    {
        //Back out any folder creation that succeeded.
        if (newClient.ProcessLocation != null && Directory.Exists(newClient.ProcessLocation))
        {
            Directory.Delete(newClient.ProcessLocation);
        }
        if (newClient.ReceiveLocation != null && Directory.Exists(newClient.ReceiveLocation))
        {
            Directory.Delete(newClient.ReceiveLocation);
        }
        if (newClient.ReportLocation != null && Directory.Exists(newClient.ReportLocation))
        {
            Directory.Delete(newClient.ReportLocation);
        }

        LogError(ex);
        throw;
    }
}
```

Figure 12:Patient Connect _service.AddNewClient

In the _service.AddNewClient method above there are items worthy of mention.

First, the ProcessLocation and ReceiveLocation, and ReportLocation properties of the

Client denote where the Emdeon receives files from customers, and saves the XML

produced containing the file data, and the location to store any reports that will be sent

back to the customer.

Second, the line of code following the creation of the client (new

WebSiteRepository().CreateCustomerPortal(systemId,

clientId);) creates a web site specific for the client where the client's customers

can go online to pay the bills they receive in the mail via Emdeon's print-mail engine.

The automated website generation is worthy of mention but is beyond the scope of this thesis.

After the ClientRepository completes its work, the new client exists in the database, and control of the client steps back through the application returned to the view. Because the print-mail engine does not update or insert into the client table, transactions with the table are very fast. Some of the other tables, such as the product, session, and rules tables, are selected from and/or transacted with by the print-mail engine very frequently at all times.

One of the customer service complaints about *Enterprise Studio* was the amount of time a user waited for a response from the application before moving to their next task. They complained the speed was not bad, but the application did not allow them to complete any more work until it finished. Many customer service agents would have multiple instances of *Enterprise Studio* open. They would execute a command in one, then, while they waited, they would open another instance to perform other work. To address this, *Patient Connect* introduced extensive multi-threading. Now, when the application executes a command for the user, it often does so with a new thread. The user can continue to work in the application while it performs the requested work.

Job Design – Rule Configuration

After the client is created, jobs and rules can be created in the same manner following the MVP design methodology. Instead of demonstrating *Patient Connect's* use

of MVP when creating jobs and rules, it would be more beneficial to take a closer look at

the configuration of the rules for a job and how they work.
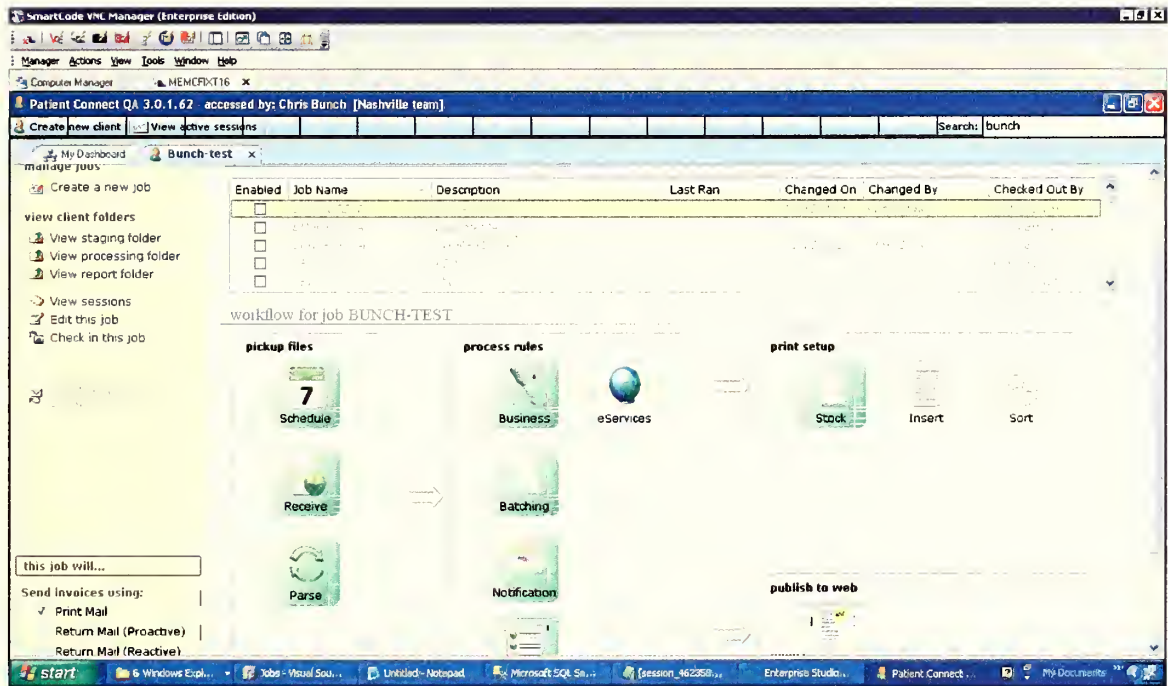
Figure 13: Patient Connect Job Header form

As evident in the example job in the Figure 9, there are five classes of rules:

- Business

    The business rules would contain all rules that control the variation of

text or images that are printed on the products. As an example, if a client file

contains a list of all customer charge details, but the client expected Emdeon to

include the aggregate sum of the charges in the mail item, this summation would

be handled within a business rule.

- Batching

If a job session contains enough products with enough pages to bring the page count too high, the system cannot print the entire job session in one print session. Some jobs have specific rules about how to partition (batch) the large sessions into smaller groups so printing can complete. These rules are contained in the batching rules.

- Notification

    Many clients and customer service representatives may wish to receive notification when a job session is complete. Any such notifications would be added to the job via a notification rule.

- eServices

    The prior section demonstrating the MVP design method mentioned how a web site for collecting payments is created for every client by default. The eServices rules publish job session data required for collecting these payments to the web database.

- Advanced

    The rules in all of the other rule sections occur during a specific event contained in the job execution. For example, every business rule executes in the *After Mail-To Address Is Loaded* event, but there are several events that occur during job processing. When viewing advanced rules, every rule is visible. The difference is that the rules are segregated based on the event in which they execute. If a job needs a specific rule to execute in an event not used by any of

the other rule groups, that rule would need to be added as an advanced rule.

These rules are called "Advanced" because they require an understanding of the

events that occur during job processing, and the rules visible as an advanced rule

but not visible in any other rule category are usually based more on technical

specifications and less on business logic.

Job creation requires supplying several configuration items that will rarely or

never be changed. These items, such as the regular expression used to identify files

belonging to the job, are supplied during the new job wizard. After completing the

wizard, the job exists, but until the rules are created, the job will do very little. In a

following section, this thesis will demonstrate how to use *Patient Connect* to create

rules for a job.

It is interesting to note at this point that even a job with no rules would still

create an addressed document to be mailed. All jobs share parsers, and all jobs inherit

from the master job. With a parser, print batching rules from the master job, and

StreamServe to format the printed documents a job can run and complete, successfully.

But Emdeon has a long list of customers, and few of the customers handle their data in

exactly the same way. The ability to custom fit a job to a client based on job rules gives

Emdeon a competitive advantage.

Job Design – Rule Creation

The best way to understand *Patient Connect* rule creation is to use the rule

creation wizard in *Patient Connect* to create a rule. One of the most common and most

important print-mail rule categories at Emdeon is *Product Suppression*. These rules suppress products that shouldn't print. The decision to not print a product may be made based on incorrect address information, values not matching conditions supplied by the customer, or any other reason.

The first step to creating a new suppression rule is starting the new rule wizard. This wizard walks the user through the entire rule creation process. The wizard created the rule with only header data. The work the rule is to perform must be configured after the rule is created.

Figure 14: Patient Connect Rule Creation Wizard - Step 1

The first form of the wizard, in Figure 10, is a place holder for rule creation options that may be present in a future version of the application. In the current version, this form does not serve an operational purpose. The only optional available to a user, in the current version are click next, click cancel, or close the form.

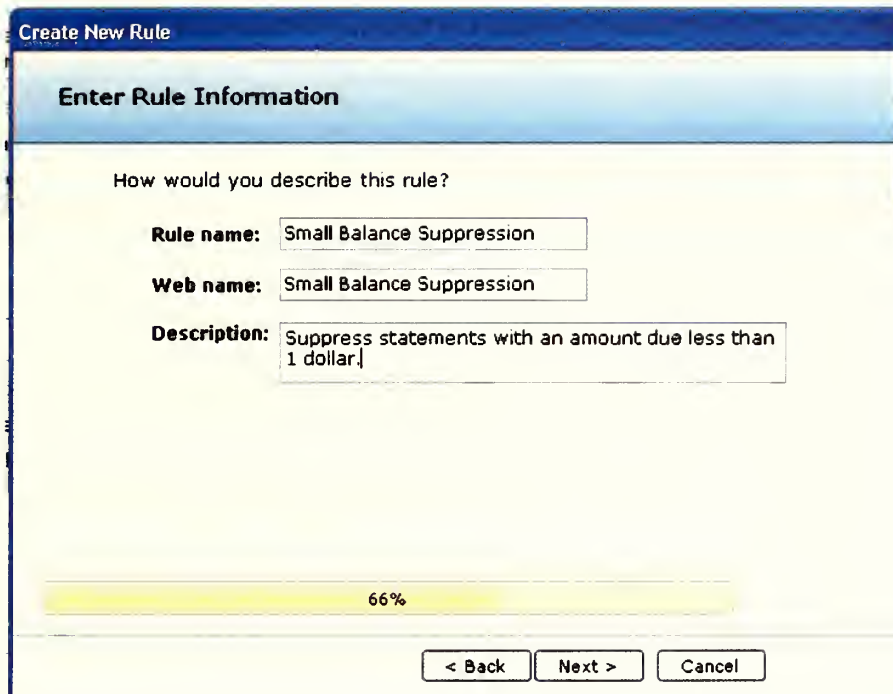**Figure 15: Patient Connect Rule Creation Wizard - Step 2**

After selecting a rule category in the second form in the wizard, all of the rules in that category are displayed. The user must select the rule to use before clicking next. The example in Figure 11, shows the rules in the *Product Suppression* category. If a product is suppressed, it is not added to the print queue. Applying suppression rules to a job ensures products will not be printed when certain criteria are met. The two rules are *Global Suppression* and *Product Suppression*. A *Global Suppression* rule suppresses all the products in a job session if certain criteria are met. For example, the client may not want any products to print if they send the file on a Tuesday. A *Global Suppression* rule would handle such a request. A *Product Suppression* rule is used to suppress specific products from printing. For example, if the print job produces medical billing statements, and, if the client does not want to print any statements that contain an

amount due less than one dollar, the suppression would be handled with a *Product Suppression* rule.

Figure 16: Patient Connect Rule Creation Wizard - Step 3

The third wizard form, shown in Figure 12, is used to assign a name to the specific instance of the rule. The name entered here will be the name of the rule in this specific job. This is the rule name that will be displayed when viewing the rules for the job in *Patient Connect*. The web name for a rule is not currently used. The plan is to develop a less robust version of the application that can be exposed via the web. Once that is done, clients can build their own print-mail rules, the web name of the rule will be the name used when the rule is viewed in the web version of the application. The description field is used to store a description of exactly what the rule dies.
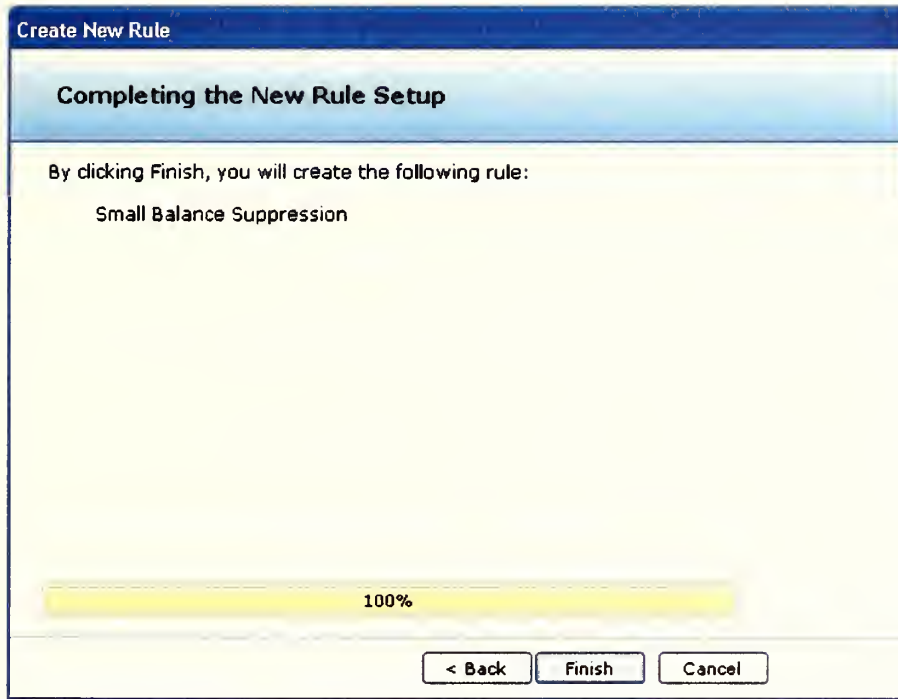
Figure 17: Patient Connect Rule Creation Wizard - Step 4

The last form in the rule creation wizard, shown in Figure 13, displays the name that had been entered for this instance of the rule. If a problem with the rule set up is discovered later, a user can edit the rule, but, after clicking finish on this form, any time the user edits the rule, their actions are saved in an audit table.

A SQL Server stored procedure is assigned to every rule. The stored procedure used depends on the rule. Most rule stored procedures, require specific parameters, *@session_id* and *@message*. The first parameter contains the value of the current session calling the rule, and the latter parameter contains instructions for the rule. The rule often, as it will in this example, uses information in the *@message* parameter to filter the session products to a subset to which the rule will be applied. Obtaining this

subset is the only purpose of the messages for *Suppression Rules*, but for some other

rules, such as *Copy Text* rules, the message can contain specific instructions for the rule.

As an example, a *Copy Text* rule may have an action instructing the rule to set the

address city to "Nashville, TN", or it might even instruct the rule to set the field *Support5*

to the sum of the values in fields *AmountDue* and *Support16*. There is almost no limit to

the power of a rule on processing a job.

Job Design – Creating Rule Actions

Patient Connect helps users create rules correctly with the rule creation wizard,

but rule actions cannot be created by a wizard. After clicking *Create an Action* on the

rule form, a dialog box requests a name for the action. Then the action is added to the

rule action list. The only thing left to do is design the message this action will use when

processing the rule. As evident in Figure 14, in the case of *Suppression Rules,* designing

the message has only one step.

when the following conditions are met:

Search Column		Condition		Search Value		
GuarantorCharges		is less than		1		+

Figure 18: Patient Connect Rule Action Search Design

Designing the action to contain only the search condition above means the rule will

suppress any product where the XML field (*xml_data* table) *GuarantorCharges* is less

than 1. Since this is a suppression rule, no other information is required.

The client has been successfully created. A job was created to handle client documents. A rule was designed for the job, and an action was applied to the rule. Now, when the client sends a file to be processed and mailed, a parser will format the document data into a predefined XML schema. Then the data will be imported into SQL server. The import will create a job session that will contain all of the products to be printed. Before the documents are sent to the printer, the job will use its only rule and rule action to check the value of the *GurantorCharges* XML field for all the products in the session. If the value of this field is less than one, the job will suppress the product. After suppression, the job will ignore the product; it will not be printed and perform no additional work on the specific product.

Job Execution

Two things control print-mail job execution for Emdeon: BizTalk Server orchestration and the WCF service library. The first orchestrates the work, and the latter executes all the rules for the jobs. An extensive exploration of the power and functionality included in BizTalk Server is beyond the scope of this thesis, but this section provides a high-level overview of how BizTalk server orchestrates print-mail job completion for Emdeon. After the BizTalk server overview, this section covers Emdeon's use of WCF services to execute the tasks involved in completing print-mail jobs.

Job Execution BizTalk

There are seventeen different BizTalk server orchestrations that control job process flow for Emdeon.[20] All of the orchestrations were built using BizTalk Orchestration Designer 3.0 several years ago. These orchestrations rarely change and they work well, so there has never been an interest in upgrading the version.

The print-mail jobs all begin as a file sent to Emdeon by the customer. BizTalk orchestrates the control flow of this file as it is processed. As an example, the *Evaluate Variables* orchestration, does as its name suggests and evaluates variables for the job session. The variables include things like page count, which needs to be evaluated to determine if the session is too large for the printers and needs to be split into smaller batches.

This orchestration is one of the simpler orchestrations, but the image in Figure 15 reveals that even a simple orchestration can look fairly involved. All of the items displayed on the left side of the page divider are decisions and calls to separate processing steps for the session document. The items atop the page divider represent ports that pass the session document. Finally, the items on the right side of the page divider represent other BizTalk orchestrations.

---

[20] *BTS Services*. Version 4.0. Emdeon
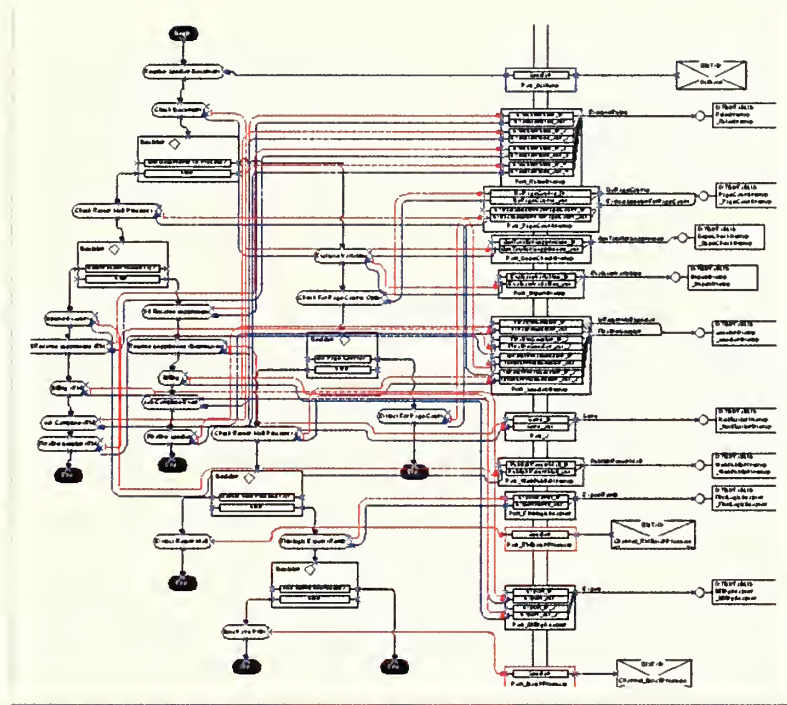
Figure 19: BizTalk Orchestration - Eval Variables

Job Execution – WCF Services

BizTalk server controls the flow of every job session, but it performs no real work. The WCF Service library performs all the real work. The library contains seventeen services, but only nine of the services are devoted to print mail. Out of the other eight services, three enable Emdeon's online medical bill pay, and the other five each complete a specific task unrelated to print mail.
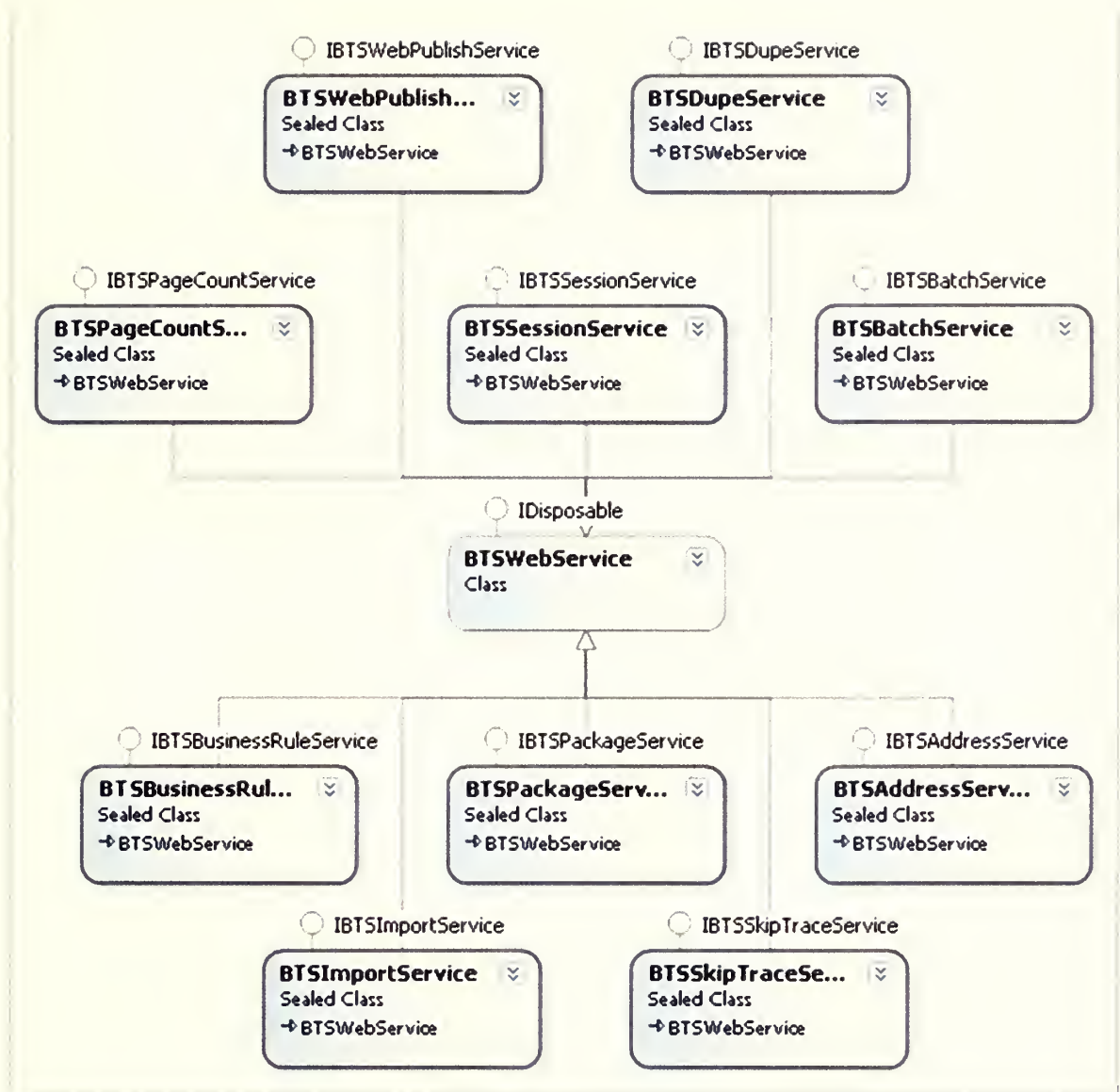
Figure 20: Emdeon Print Mail Services Diagram

In this section, the purpose of each of the nine services in Figure 17, is explained.

The following section of this thesis will cover implementation of Intelligent Mail

Barcode, and that implementation requires changes to the BTSAddressService and the

XmlInserter class used by the service to insert address information into the product XML

document.
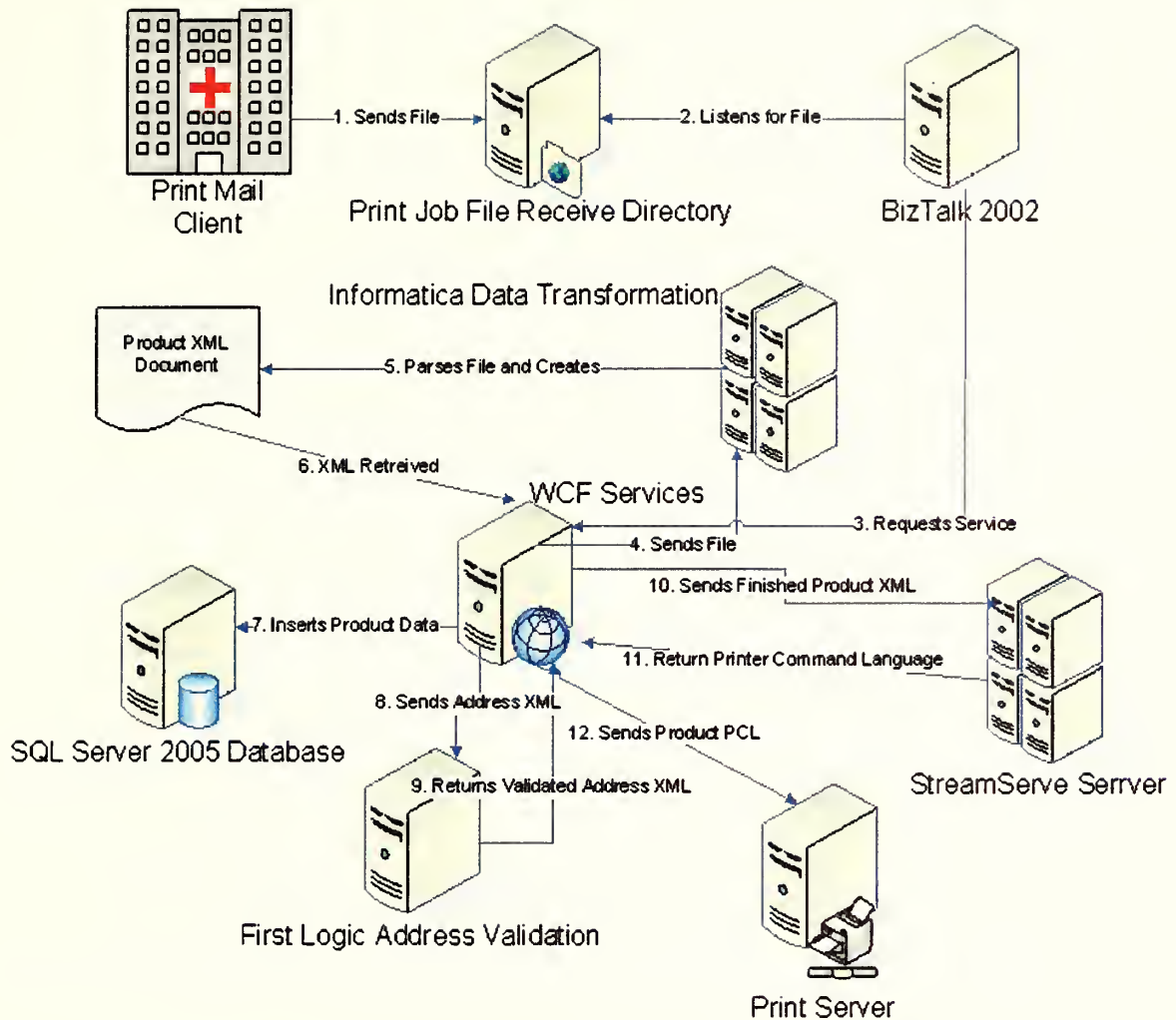
Figure 21: Emdeon Print-mail Engine Process Flow

BTSWebService – The only inheritable service. This parent service is inherited by every other service and contains all of the common service methods and properties.

BTSImportService – When a client sends a file containing the data to be printed to Emdeon, the parser and preprocessor transform the file to XML that matches one of the expected schemas. This service then imports the XML data into the SQL Server database where the data can be processed.

BTSBusinessRuleServices – The previous section explained print-mail business rules. This is the class that executes the business rules. It is called during each print-mail event, and it looks up the rulesets for the job that are to be completed in the event. If it finds any rulesets, it executes the rules.

BTSPageCountService – StreamServe looks at the products and sessions to determine the number of pages that need to be printed. This value is used to determine if a product needs to be batched into smaller subgroups (batches). This service communicates with StreamServe to assist with the page count.

BTSBatchService – Job sessions are divided into batches, based on page count and inventory, before they are sent to the printer. This service creates the batches to print.

BTSDupeService – In some cases a file could be mistakenly imported or processed multiple times. This service looks for such cases and suppresses any duplicate products.

BTSPackageService – After the session is divided into batches, it is ready for packaging. After a batch has been packaged, it is printed and mailed. Many times it is necessary to run a sample session for a job for testing purposes, and these sample sessions should not be printed. This service checks to see if the session should be printed before it is sent to the printer.

BTSSessionService – This service completes session execution by cleaning any open queues and setting the session complete date and time. It also hashes the session identifier. This hash value is used to prevent duplicate sessions from printing.

BTSAddressService – This service communicates with the First Logic™ tool to validate the addresses.

BTSSkipTraceService – Sometimes the client does not know the correct address for the recipient of a mail item. These mail items are often medical statements, and finding the recipient is important to the client. This service works with a third party service to find the current address for a person based on information such as social security number.

BTSWebPublishingService – Most of the documents processed and mailed by Emdeon are medical statements. Emdeon offers a service where the patients can go online to pay the bill. This service publishes the print-mail data to the web database to enable the pay online service.

Chapter 4 – Implementation of Intelligent Mail Barcode at Emdeon

The decision was made in June, 2009, to start implementing Intelligent Mail Barcode (IMB) at Emdeon.[21] Chapter one covered the requirements of IMB, and it also discussed Emdeon's previous barcode, the PostNet barcode. Prior to IMB implementation, the PostNet barcode was added to the product XML during the batching event when batches of products are created and sent to the printer.

The WCF service, *BTSBatchService*, executes product batching. In order to batch the products, the service has to first extract the product details from the SQL Server database and create the product XML document. This extraction is completed in the *ExtractDocuments* method of the service seen in Figure 22.

```
/// <summary>
/// Extracts XML from the database.
/// </summary>
private int ExtractDocuments(int[] products, string extractFile, IxtPrintFile printFile, string streamserve)
{
    XmlExtractor extractor = new XmlExtractor(ProdObjectManager);
    extractor.PrintFile = printFile;
    extractor.StreamServe = streamserve;
    extractor.CommandTimeout = ProdObjectManager.CommandTimeout;
    extractor.Transaction =              BeginTransaction(ProdObjectManager);

    if (File.Exists(extractFile))
    {
        File.Delete(extractFile);
    }

    if (Job.OutMap != DBNull.Value && !IsReturnMailSession)
    {
        string sOutMap = Config.GetStyleSheetLocation(Job) + @"\" + Job.OutMap.       ();
        extractor.ExtractDocuments(products, extractFile, sOutMap, Config.Envelope, Config.ExtractThreadCount);
    }
    else
    {
        extractor.ExtractDocuments(products, extractFile, null, Config.Envelope, Config.ExtractThreadCount);
    }

             .Commit(ProdObjectManager);

    Publish(extractFile);

    return extractor.TotalExtracted;
}
```

Figure 22: BTSBatchService.ExtractDocuments

---

[21] Emdeon. "Intelligent Mail Barcode Project Plan" Developed June 2009

In the *ExtractDocuments* method. An instance of *XmlExtractor*, from the Emdeon

core library, is created. The *ExtractDocuments* method of the instance is called by

BTSBatchService and the instance of *XmlExtractor* extracts the data and creates the

product XML. This chapter demonstrates the changes that had to made to the

XmlExtractor to implement IMB.

To retrieve the product data from the SQL Server database not contained in the

xml_data table, *XmlExtractor* calls the *up_XmlExtractor_GetProductInfo* stored

procedure. One of the data returned by this stored procedure is the information

needed to generate the barcode. Since the IMB barcode contains more data than the

previously used PostNet barcode, this stored procedure needed to be updated to supply

XmlExtractor with all the required data. Figure 23 shows the updates to the select

portion of the stored procedure.

```
convert(varchar(2),def.barcode_id) +
     isnull(pst1.postal_service_code, pst2.postal_service_code) +
     convert(varchar(6),def.mailer_id) +
     right(replicate('0',9) + convert(varchar,p.product_id)
     + convert(varchar(2),'01')   ,9)
     as imbTrackingCode,
convert(varchar(2),defRemit.barcode_id) +
     isnull(pst3.postal_service_code, pst3.postal_service_code) +
     convert(varchar(6),defRemit.mailer_id) +
     right(replicate('0',9) + convert(varchar,p.product_id)
     + convert(varchar(2),'01')   ,9)
     as imbRemitTrackingCode,
```

Figure 23: imbTrackingCode and imbRemitTrackingCode

As evident in Figure 23, two new elements were required to implement IMB: imbTrackingCode and imbRemitTrackingCode. Every IMB consists of a routing code and a tracking code (see Figure 3 in Chapter 1). The routing code component of the IMB barcode has the same value as the PostNet (see Figure 3 in Chapter 1). The decision was made to transform the PostNet into the IMB by concatenating the tracking code to the in front of the PostNet.

The imbTrackingCode is the tracking code used for the send-to address and the imbRemitTrackingCode is the tracking code used for the remit address. In most instances these values would be the same. The only time the values could differ is when a client wants the mail-to and remit to use different postal services. The new elements returned by the stored procedure for both the imbTrackingCode and the imbRemitTrackingCode are:

- Postal_service_code

  This represents the level of service to use for the mail piece. The United States Postal Service offers seventeen service levels for IMB2[2], but only three are used by Emdeon's print-mail engine. The three service levels employed and their associated codes are visible in Table 3.

Description	Code
First-Class Mail with Destination Confirm	41
Standard Mail with Destination Confirm	43
First-Class Mail with Destination Confirm (Remit)	700

Figure 24: Postal service levels

- Mailer_id

  The Mailer Id is a specific identifier distributed by the postal service to identify the sender of the mailpiece. This value is important for tracking purposes.

- Product_id

  For tracking purposes, the IMB specifications require every mailpiece contain a serial number.[16] The Product Id is the primary key of the product table. Every item mailed by Emdeon is a specific product. Rather than use a new serial number to uniquely identify mailpieces, the decision was made to use this already existing identifier.

  After *XmlExtractor* extracts the product information, it transforms the data from a data table to an XML document. During this extraction process, the *XmlExtractor* inserts all of the product address information, including the mail-to and remit IMBs, into the product XML. To implement IMB the *InsertAddress* method of the *XmlExtractor* had to be modified to inject the IMB value into the product XML (see Figure 25).

```
string sImbTrackingCode = null;
string sImbComplete = null;
if (addressType == AddressType.MailTo)
{
    sImbTrackingCode = dt.Rows[0]["imbTrackingCode"].ToString();
}
else
{
    sImbTrackingCode = dt.Rows[0]["imbRemitTrackingCode"].ToString();
}
sImbComplete = !String.IsNullOrEmpty(sImbTrackingCode) &&
               !String.IsNullOrEmpty(address.PostNet)
                   ? string.Concat(sImbTrackingCode, address.PostNet)
                   : null;
if (!String.IsNullOrEmpty(sImbComplete))
{
    //CreateImbElement(sImbComplete, xpath, ref document);
    ReplacePostNetWithIMB(ref document, xpath, sImbComplete);
}
```

Figure 25: Injecting product XML with IMB value

The code in Figure 25 is used when injecting both the mail-to address information and the remit address information. When the code is used to inject mail-to data, the imbTrackingCode data returned by *up_XmlExtractor_GetProductInfo* is used, when injecting the remit data, the imbRemitTrackingCode is used. The complete IMB is generated by concatenating the tracking code in front of the PostNet value. As long as the IMB has a value after the concatenation, the ReplacePostNetWithIMB method is called. This method's only function is to insert the IMB value into the PostNet node of the product XML. After the code in Figure 25 completes, the IMB is in the product XML and ready to be used.

Conclusion and Areas for Additional Research

This thesis communicated the requirements of Intelligent Mail Barcode (IMB), provided a high-level explanation of Windows Communication Foundation (WCF), explained the print-mail architecture at Emdeon, Inc., and, lastly, demonstrated how IMB was implemented using the WCF services that process Emdeon's print-mail system. By following all of the steps outlined in this document, IMB has been implemented successfully. But work still remains to reach full implementation of IMB.

Chapter 1 explained that the demand for IMB stemmed in large part from the need for the United States Postal Service (USPS) to compete with private carriers in the area of mailpiece tracking. After IMB is implemented as described in this document, USPS sends a file containing a numeric identification of the mailpiece, a datetime stamp, and the zip code of the location of the mailpiece at that time. Processing the information contained in this file enables an enterprise to track products as they travel the mail stream.

Research remains to find the most efficient and effective way to process and use this tracking data. Emdeon has plans to parse and extract this data as it is received. The intent is to generate reports of this data. The reports are to be supplied to the print-mail client so they can know the status of everything they have mailed using Emdeon's services. The client will also be able to see when the remit mail is on the way back to them. In the case of remit mail containing bill payments, these reports will help the print-mail client's accounts payable department better anticipate funds received.

Potential uses other than reporting may be discovered during the additional research. Possibilities include data mining to learn ways to improve receipt of remit mail. For instance, an enterprise may learn that if they send bills near the middle of the month, they have a faster rate

of return. The best way to integrate the mailpiece tracking data into existing systems also needs

to be discovered. For instance, processing of the tracking data could be added to an accounting

system to improve the accuracy of revenue forecasting.

Bibliography

"15 Seconds : Windows Communication Foundation - Part 1." *15 Seconds : Asp Tutorials, Asp.net Tutorials, ASP Programming Sample Code, and Microsoft News from 15Seconds.* Web. 14 Feb. 2010. <http://www.15seconds.com/Issue/061130.htm>.

BTS Services. Vers. 4.0. Nashville, TN: Emdeon, 2009. Computer Software.

"Decoupled Contract (Erl)" SOA Patterns Web. 22 Feb. 2010 <http://www.soapatterns.org/decoupled_contract.php>

"Delivery Points" *Wikipedia, the Free Encyclopedia* s Web. 1 Apr. 2010 < <http://en.wikipedia.org/wiki/Delivery_point >

"Design Patterns: Model View Presenter." *MSDN: Microsoft Development, MSDN Subscriptions, Resources, and More.* Web. 14 Feb. 2010. <http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S1>.

Designing and Implementing Services Web. 31 Mar, 2010 <http://msdn.microsoft.com/en-us/library/ms729746.aspx>

Emdeon | The Largest Healthcare Revenue and Payment Cycle Network. Web. 14 Feb. 2010. <http://www.emdeon.com/>.

Emdeon Core Print Mail Library. Vers. 5.0. Nashville, TN: Emdeon, 2009. Computer Software.

Emdeon. *Patient Connect.* Vers. 3.0. Computer Software.

Emdeon. *Enterprise Studio.* Vers. 3.0. Nashville, TN: 2009. Computer Software.

Emdeon. *Core Framework.* Vers. 5.0. Nashville, TN: 2009. Computer Software.

Emdeon. *BTS Web Services.* Vers. 4.5. Nashville, TN: 2009. Computer Software.

Emdeon. "Intelligent Mail Barcode Project Plan" Developed June 2009

Essential Windows Communication Foundation; Resnick, Steve, Richard Crane, and Chris Bowen.. Boston, MA: Pearson Education,, 2008. Print. Microsoft .Net Development.

"History of Direct Mail." *Direct Mail.* Web. 14 Mar. 2010. <http://www.direct-mail.org/history.htm>.

"The History of Object Oriented Programming" *Exforsys, Inc.* Web. 31 Mar. 2010. <http://www.exforsys.com/tutorials/oops/the-history-of-object-oriented-programming.html>.

"Hosting Windows Communication Foundation Services." *MSDN: Microsoft Development, MSDN Subscriptions, Resources, and More*. Web. 14 Feb. 2010. <http://msdn.microsoft.com/en-us/library/ms730158.aspx>.

"Intelligent Mail Barcode -." *Wikipedia, the Free Encyclopedia*. Web. 14 Feb. 2010. <http://en.wikipedia.org/wiki/Intelligent_Mail_Barcode>.

Intelligent Mail Barcode (4-State Customer Barcode) USPS 3200.B (2005). *United States Postal Service Specification*. United States Postal Service. Web. <https://ribbs.usps.gov/intelligentmail_mailpieces/documents/tech_guides/SPUSPS-B-3200E001.pdf>.

" Johannes Gutenberg" *Wikipedia, the Free Encyclopedia*. Web. 16 Dec.. 2010. <http://en.wikipedia.org/ wiki/Johannes_Gutenberg >.

Microsoft Windows Communication Foundation Step by Step; Sharp, John.. Redmond, Wash.: Microsoft, 2007. Print.

"Microsoft Development Network" Web <http://msdn.microsoft.com/en-us/library/3b5b8ezk(VS.71).aspx>

"Model-view-presenter." *Wikipedia, the Free Encyclopedia*. Web. 14 Mar. 2010. <http://en.wikipedia.org/wiki/Model-view-presenter>.

"Object-oriented Programming -." *Wikipedia, the Free Encyclopedia*. Web. 14 Mar. 2010. <http://en.wikipedia.org/wiki/Object-oriented_programming>.

"POSTNET -." *Wikipedia, the Free Encyclopedia*. Web. 14 Mar. 2010. <http://en.wikipedia.org/wiki/POSTNET>.

"Print Mail Manufacturing Transformations." Industry Technical Whitepaper; Hodges, Craig, Vice President Emdeon Patient Billing and Payment Solutions Paper

"SOAP Basics" Soapuser.com Web. 31 Mar 2010 <http://www.soapuser.com>

"Service Oriented Architecture (SOA) and Specialized Messaging Patterns"; Duane Nickul, Laurel Reitman, James Ward, Jack Wilber; Technical White

"Service-orientation -." *Wikipedia, the Free Encyclopedia*. Web. 14 Jan. 2010.
<http://en.wikipedia.org/wiki/Service-orientation>.

USPS National Customer Support Center. Web. 14 Feb. 2010.
<https://ribbs.usps.gov/index.cfm?page=intellmailmailpieces>.

"WCF (Windows Communication Foundation) Introduction and Implementation -
CodeProject." *Your Development Resource - CodeProject*. Web. 14 Mar. 2010.
<http://www.codeproject.com/KB/WCF/WCFServiceSample.aspx>.

"What Is Windows Communication Foundation?" *MSDN: Microsoft Development, MSDN
Subscriptions, Resources, and More*. Web. 15 Jan. 2010.
<http://msdn.microsoft.com/en-us/library/ms731082.aspx>.

"Windows Communication Foundation Endpoints: Addresses, Bindings, and Contracts."
MSDN: Microsoft Development, MSDN Subscriptions, Resources, and More. Web. 14
Mar. 2010. <http://msdn.microsoft.com/en-us/library/ms733107.aspx>.