Chapter 1. Introduction

Chapter 1 seeks to introduce the Everest Framework, its components , and to provide an overview of their interrelations. Installation options for the Everest Framework are reviewed and discussed in detail.

Everest in a Nutshell

In short, the Everest Framework is designed to ease the creation, formatting, and transmission of HL7v3 structures with remote systems.

The "framework" provides a series of consistent, well documented components that, when used together, provide a flexible mechanism for supporting HL7v3 standards within the application. Through a combination of automatically generated code and carefully constructed handwritten modules, Everest has the ability to serialize, validate, and transmit structures.

Before discussing the architecture of Everest, it helps to understand the driving principles behind the framework itself. The pillars of Everest are:

- **Intuitiveness:** All components within Everest are designed to be intuitive to developers. Great care has been taken to reduce the complexity of the Framework and allow developers to focus on HL7v3 messaging.
- **Standards Compliance:** Being a standards-based framework, one of the foundational pieces is standards compliance. The Everest Framework is more than just a serialization engine; it will generate messages, transport them, and validate instances in a standards-compliant manner.
- **Quality:** Everest code is held to the highest standard of quality in terms of regression testing and documentation. All changes made to the framework are reviewed for their quality and are subject to over 8,000 tests.
- **Performance**: Everest has been designed with long-term performance in mind. Many of the methods within Everest (especially formatting) have the ability to "learn" and become faster the more they are used.
- **Flexibility**: Everest has been designed to be flexible enough to support new HL7v3 standards.

The tooling required to extend the framework is also included. Using these tools, it is possible to derive new Everest modules from MIF files published by HL7 International or other jurisdictions to extend functionality. The structures, validation rules, and derivation information from these models are carried 1-to-1 to code. It is also possible to derive other deliverables using these tools such as HTML documentation, XSL transforms, and a variety of other formats.

Everest Architecture

Everest has been designed to be loosely coupled from the wire level format. Typically, a v3 system is architected as pictured in Figure 1.

Figure 1 - Everest at a high level

Many applications will store data in an internal data model or "canonical format". This format is usually designed to solve the business needs of the environment and not necessarily meant to be interoperable.

In order to communicate with another application in an interoperable way, systems normally map their canonical form data to a standard structure. In the Everest world, this standard is HL7v3. Through a process of creating a RIM graph, canonical data is mapped to RMIM structures generated from MIF files.

After performing this task, the developer needs to transform these RIM structures to an interoperable messaging format as not all systems use Everest. In HL7v3 this is typically XML but could also be EDI, JSON, or other data exchange formats.

This interoperable message can now be sent via a transport to the remote system where the same process is reversed to store the data.

Components of the Framework

Everest has been designed as a larger entity made up of smaller components. There are four major components to the Everest Framework:

1. **Core Library**: The core library represents a series of interfaces that all other components implement, and acts as the "entry point" for the framework. It also contains the handwritten data type classes. These data type classes are a combination of HL7v3 R1 and R2 data types and provide functionality for validating, arithmetic and conversion.
2. **RMIM Structures**: The RMIM structures component is used to group all generated code from MIF files. MIF files are passed through GPMR and compiled to a series of RMIM libraries which, when combined with data types, can be used to create message instances. More information about GPMR and the use of GPMR can be found in General Purpose MIF Renderer on page 213.

3. **Formatters**: Formatters are responsible for the serialization and parsing of structures to/from interoperable wire formats and for the validation of instances during that process. Formatters are typically written to a specified ITS (Implementable Technology Specification) and apply the rules of a particular ITS to generate the message wire format.
4. **Connectors**: Connectors are responsible for the transmission of RMIM structures over a communications channel to/from a variety of endpoints. Connectors in the .NET version of Everest can be used to receive messages which allow Everest .NET to be used on server infrastructure.

These components and their relation to one another are illustrated in Figure 2.

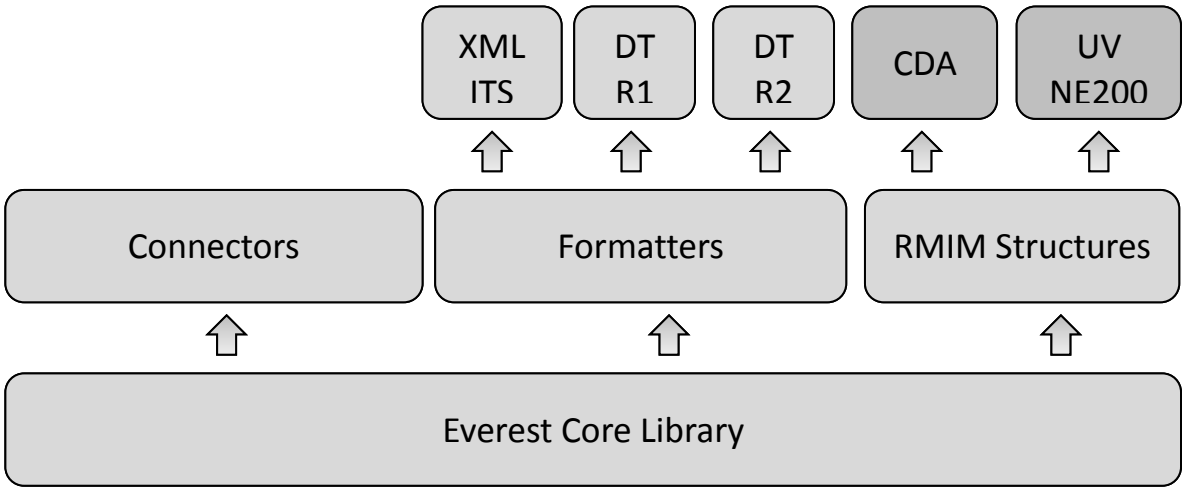

Figure 2 - Everest Components

Generated Code Libraries

Everest 1.0 has the ability to generate and consume messages from any one of the standards listed in Table 1.

Table 1 - Supported Standards

Standard	Realm	SDO	Assembly
MR2009 Delta 1	Canada	Canada Health Infoway	Not Bundled. Can be generated from MIF
MR2009 Delta 2			MARC.Everest.RMIM.CA.R020402
MR2009 Delta 3			MARC.Everest.RMIM.CA.R020403
Normative Ed. 2008	Universal	HL7 Intl.	MARC.Everest.RMIM.UV.NE2008
Normative Ed. 2009			Not Bundled. Can be generated from MIF
Normative Ed. 2010			MARC.Everest.RMIM.UV.NE2010
Clinical Doc.			MARC.Everest.RMIM.UV.CDAr2

Arch. R2			

All RMIM assemblies reside in the org.marc.everest.rmim.* package in the Java version of the Everest Framework. Documentation for these assemblies can be found in the zip file carrying the same name as the RMIM jar file.

Installation of the Framework

The MARC-HI Everest Framework is distributed using a series of packages which can be installed based on the requirements of your environment.

Table 2 illustrates the distribution components that come bundled with each installer available at http://everest.marc-hi.ca. Everest 1.0 installers are distributed in the naming convention everest-**bundle**-*version*. The SDK distribution is available as either a windows installation package, or a tarball.

Table 2 - Installation Package Contents

Bundle	.NET Libraries	JAR Files	GPMR	Dev. Tools
SDK	X		X	X
RT	X			
GPMR			X	
Java Edition		X		X
Compact	X			X

When installing any of the .NET libraries, it is recommended that you use the Windows based installer as it will generate native assemblies (ngen) for each of the Everest components which will enhance the performance of the components.

Software Prerequisites

Runtime Environments

The MARC-HI Everest Framework is officially supported on the Windows® operating systems; however, it has been tested and verified to work within the environments listed in Table 3 when the specified prerequisites are installed.

Table 3 - Operating System Support

Operating System	Microsoft .NET Framework	Mono
Windows® XP	.NET Framework 3.5 or 4.0	2.10.8
Windows Server® 2003		
Windows Vista®		
Windows Server® 2008		

Windows 7®		
Windows Server® 2008 R2		
SUSE® Linux Enterprise Server/Desktop 11	Not Supported	2.10.8
Ubuntu™ 11.04+		2.10.5
Mac OS®		2.10.8

The Everest installer will automatically install all necessary prerequisites for the .NET version of the Everest Framework (.NET 3.5).

The Java edition of the Everest Framework will require the Java Development Kit version 1.7 or greater.

Development Environments

Table 4 outlines the development environments that have been verified to work with Everest.

Table 4 - Tested Development Environments

IDE	C# Development	VB Development
Microsoft Visual Studio 2008 Express/Std/Pro	X	X
Microsoft Visual Studio 2010 Express/Pro/Ult.	X	X
MonoDevelop 2.x	X	
SharpDevelop 3.x	X	

Redistributing the Framework

When packaging software developed with the MARC-HI Everest Framework, it is recommended that you bundle the runtime installer with the installer for your application. The runtime installer can be executed from your setup program using "everest-rt-1.0.exe /silent". This will ensure that Everest assemblies have native generation performed on your user's machines.

Redistribution of the framework can also be achieved by bundling the Everest assemblies directly into your installer package.

Any organization wishing to redistribute RMIM structure assemblies should check with the copyright holder of the standard to ensure they have proper right to do so. This only applies to assemblies in the MARC.Everest.RMIM.* namespace (see Appendix C).

Developer Installation Options

When installing the SDK package using the installer provided on the MARC-HI Technology Exchange, you will be presented with many types of install. Typically a complete install is recommended (uses approximately 160 MB of hard drive space) but the installer will provide options to customize the installation.

The SDK installation for developers has five feature groups:

1. **Tool Files**: Installs GPMR to the developer's hard drive. This is recommended if your developers will be adding new standards to Everest in support of your product development.
2. **Documentation**: Installs the Reference Guide, Everest Samples and a copy of the Everest Library document which contains class-by-class documentation of features found in Everest.
3. **Everest Core**: Installs the core assemblies (such as Formatters, Connectors and the Core library) and optionally pre-generated standards libraries for NE2010, NE2008, CDAr2, and Canadian Specifications.
4. **Visual Studio Integration**: Provides the necessary data files to integrate Everest documentation within Visual Studio or other .NET development environments (such as SharpDevelop or MonoDevelop).
5. **Source**: Installs the source code files for the Everest core components (those which are licensed under the Apache 2 license found in Appendix C of this document).

Chapter 2. Creating Instances

Creating instances within Everest refers to the process of instantiating RMIM or Data Types structures in memory and populating them with data. These objects, although serializable, are not intended to be sent to remote systems as is; rather, they are expected to be passed through a formatter and connector.

Formatters and connectors are used in this chapter for illustrative purposes only and are covered in more detail in Chapter 5 on page 161.

IGraphable

All classes implemented in the Everest Framework implement the *IGraphable* interface. *IGraphable* is an empty "marker" interface that is used by formatters, connectors, and other utility classes to ensure that instances can be represented using an ITS.

Implementers of *IGraphable* are expected to have the following characteristics:

- The class will have a *StructureAttribute* (see "Advanced Topics" on page 213 for more information) that defines the structure from an HL7 point of view, and
- Properties that are to be graphed will have the *PropertyAttribute* attribute which defines the RMIM structure as it appeared in MIF, and
- A validate method that will ensure that the structure is valid according to its HL7v3 constraints.

Your First Message

Everest contains a 1:1 mapping of .NET/Java classes to the underlying HL7v3 message structure. This decision makes debugging easier and more intuitive (i.e.: if a remote system tells you something is wrong with X, it is called X in the class model).

What's in an Instance?

An instance is the term used to describe an instantiation of an RMIM class in memory. RMIM classes are generated from MIF files provided by various standards development organizations. RMIM classes in Everest are more than just empty class structures; they contain vital meta-data about how the instance "looks" in the wire.

Your RMIM instance contains the data that will ultimately be serialized and sent somewhere, be it a file, remote system or another function. HL7v3 instances contain all the necessary data for a remote system to process the message in a reliable manner.

Complex structures in Everest are comprised of simple data types, .NET/Java collections (such as *List*/*ArrayList* respectively) and other complex structures. The following properties are pre-populated for you whenever you create a structure with Everest:

- Any .NET/Java collection is pre-initialized so you can simply call the *Add* method on the property. You can also overwrite this property with your own instance of a collection.
- Any fixed values in the message instance are populated to their fixed value and getters are provided (they are hidden in Everest .NET as to not confuse developers). Fixed values can be reset (i.e.: the *Code* property can be set to something different); however, these are usually not interpretable by remote systems.
- Any properties that have a default value assigned in the MIF will be populated with the default value. Developers can override the default by setting a new value.

RMIM structures also contain a variety of constructors, creator methods and utility functions that make your life easier. Each of these concepts is covered in the "Everything about RMIM Structures" section (page 121) of this chapter.

Choose a Structure

Any structure in Everest can be the "root" of serialization. What does that mean? Well any class in the MARC.Everest.RMIM.X namespace can be graphed to the wire and parsed by the Everest Framework. However, not that doesn't mean that all structures should be serialized in this manner.

There are two types of classes that should be used as the root of graphing:

- **HL7v3 Interactions** – These are messages that are triggered in reaction to an event that has occurred. These classes can be found in the RMIM.XX.YYYY.Interaction namespace and have names like REPC_IN000000UV (..rmim.xx.yyyy.interaction package in Java).
- **Entry Point Structures** – These are special RMIM structures that have been marked as appropriate for the start of serialization. An example of this is the *ClinicalDocument* class in the MARC.Everest.RMIM.UV.CDAr2 assembly (org.marc.everest.rmim.uv.cdar2.jar for Java). These are identified by the *IsEntryPoint* property on the Structure attribute attached to the class definition.

Of course, choosing a structure depends entirely on the task at hand. If serializing some data for future reference, then any class may be selected. If the application is attempting to send a CCD, Referral or other clinical document, the *ClinicalDocument* class from the CDA assembly might be appropriate.

This chapter will use the PRPA_IN201305UV02 message (find candidates) to illustrate the construction of an instance.

This guide doesn't make an assumption about the development environment being used, so all examples provided will simply assume a program with one class named Program in .NET or Java.

Code the Instance

Add References

Using any development environment, create a new project. The first step to creating an instance in Everest is to let your program know where the Everest code is located on your hard drive.

In most programming environments this is known as a reference. Everest libraries are installed in the "lib" folder of your Everest installation. In order to use Everest, the libraries listed in Table 5 will need to be referenced by your project.

Table 5 - Sample project references

Component	Assembly (.NET)	Jar (Java)
Everest Core	MARC.Everest.dll	org.marc.everest.jar
HL7v3 Structures	MARC…RMIM.UV.NE2008.dll	org.m...t.rmim.uv.ne2008.jar
Data Types R1 Formatter	MARC...ters.XML.DataTypes.R1.dll	org.m...ters.xml.datatypes.r1.jar
XML ITS 1.0 Formatter	MARC...Formatters.XML.ITS1.dll	org.m…formatters.xml.its1.jar

Once these references are created for your project, you'll need to import the necessary namespaces into your program file. Example 1 illustrates the necessary code for referencing namespaces.

Example 1 - Referencing namespaces for Everest

```
using MARC.Everest.DataTypes; // Everest Data Types
using MARC.Everest.RMIM.UV.NE2008; // Normative Ed. 2008
using MARC.Everest.RMIM.UV.NE2008.Interactions; // Interactions
using MARC.Everest.RMIM.UV.NE2008.Vocabulary; // Vocabulary
using MARC.Everest.Connectors.File; // File Connector
using MARC.Everest; // Everest Namespace
using MARC.Everest.Formatters.XML.Datatypes.R1; // DT R1 formatter
using MARC.Everest.Formatters.XML.ITS1; // XML ITS1 Formatter
```

```
import org.marc.everest.datatypes.*;
import org.marc.everest.rmim.uv.ne2008.interaction.*;
import org.marc.everest.rmim.uv.ne2008.vocabulary.*;
import org.marc.everest.datatypes.*;
import org.marc.everest.datatypes.generic.*;
import org.marc.everest.formatters.xml.datatypes.r1.DatatypeFormatter;
import org.marc.everest.formatters.xml.its1.XmlIts1Formatter;
```

Visual Studio Wizards

When creating a new project in Microsoft Visual Studio 2008 or 2010, the manual adding of references and namespaces can be skipped. The Everest Framework contains several project templates that can greatly simplify the creation of an Everest project.

To create a new Everest project in Visual Studio, start a new project and select the Everest template parameters (see Figure 3).

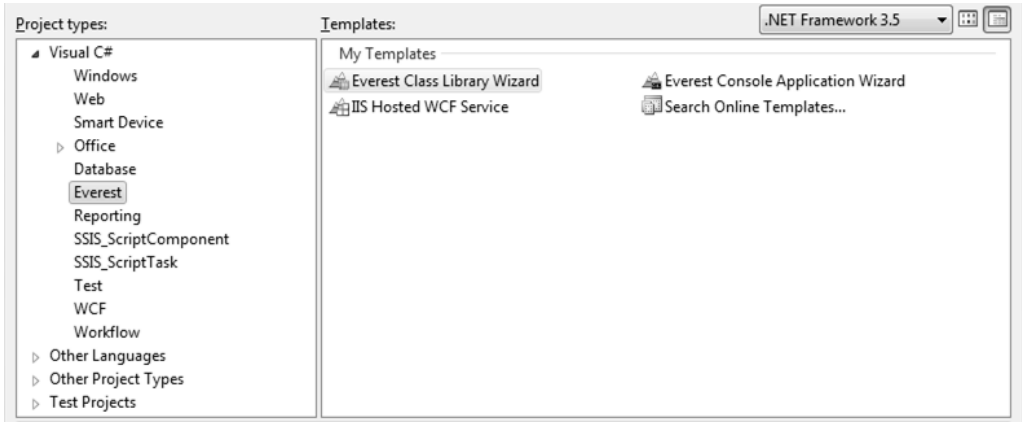

Figure 3 - Everest templates dialog

Select the "Everest Console Application Wizard" option and name your project. After clicking the "create" button, an Everest Wizard will appear which will let you select the release and supporting assemblies to reference (see Figure 4).

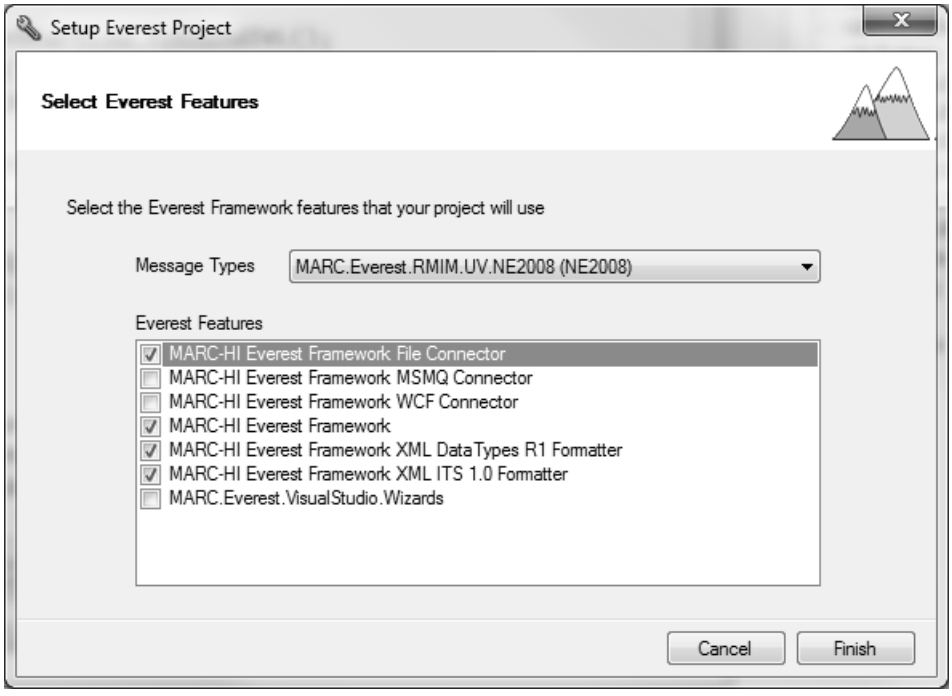

Figure 4 - Everest project wizard

After selecting the message type assembly and supporting libraries, click finish. You'll notice that Everest's wizard has automatically selected the most appropriate assemblies and added them to your project (see Figure 5).

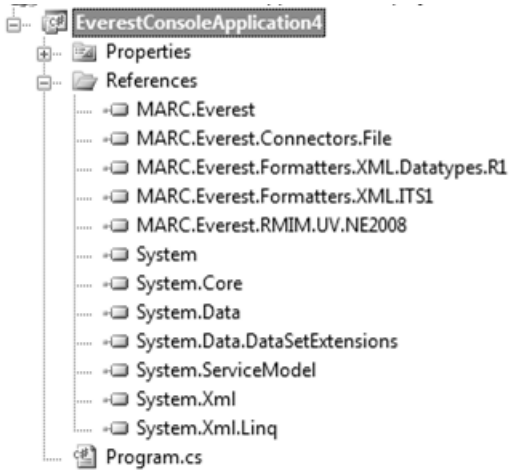

Figure 5 - Everest project setup by the wizard

Create the Instance

After the references and imports are completed the instance needs to be constructed. The interaction that is being created is a PRPA_IN201305UV02 interaction which should be shown in the help window as a find candidate query.

To create an instance of the interaction, a simple call to the constructor can be used. The "Everything About RMIM Structures" chapter (page 121) discusses more advanced methods of creating RMIM structures). Example 2 illustrates using the full constructor to create the instance.

Example 2 - Instantiating the Instance

```
PRPA_IN201305UV02 instance = new PRPA_IN201305UV02(
    Guid.NewGuid(), // Every message needs a unique identifier
    DateTime.Now, // Identify when the message was created
    PRPA_IN201305UV02.GetInteractionId(), // Identify the interaction
    ProcessingID.Training, // Identify the process mode for the receiver
    "I", // Identify the receive mode for the interaction
    AcknowledgementCondition.Always // Identify when we want a response
);
```

```
PRPA_IN201305UV02 instance = new PRPA_IN201305UV02(
  new II(UUID.randomUUID()), // A unique id for the message
  TS.now(),  // The timestamp of the message
  PRPA_IN201305UV02.defaultInteractionId(), // The interaction id
  ProcessingID.Production,  // Processing mode on recv
  new CS<String>("I"),  // Immediate processing
  AcknowledgementCondition.Always // Always respond to messages
);
```

Setup a Formatter

The process of formatting in Everest is flexible and permits the mixing of formatters into hosts and graph aides. In short, a host formatter is a stand-alone formatter that is used to graph any type of instance to the wire and has the capability to classify a type based on wire format data (i.e.: if root element is X then parse into Y). An example of a stand-alone formatter is the XML ITS 1.0 formatter.

Graph aides are formatters that cannot be used on their own and exist only to help the primary formatter. Graph aides are usually intended for data types or very few types that require special graphing considerations.

Formatters are covered in more depth in Chapter 5 (see page 161). Example 3 illustrates the setting up of the XML ITS1 formatter with the Data Types R1 formatter aiding in the graphing process of data types.

Example 3 - Setting up the formatter

```
var xmlIts1 = new XmlIts1Formatter();
xmlIts1.ValidateConformance = false;
xmlIts1.GraphAides.Add(new DatatypeFormatter());
```

```
try(XmlIts1Formatter xmlIts1 = new XmlIts1Formatter())
{
   xmlIts1.setValidateConformance(false);
   xmlIts1.getGraphAides().add(new DatatypeFormatter());
}
catch(Exception e)
{
   e.printStackTrace();
}
```

Save the Instance

Everest is flexible in the facilities it provides for saving instances. Formatters can be used directly to save instances using streams, or they can be appended to a connector and instances can be "sent" via the connector to a file store (connectors are covered in more detail in Chapter 6).

Example 4 shows how to use the formatter to graph directly to the console window.

Example 4 - Graphing the instance

```
Stream output = Console.OpenStandardOutput();
try
{
    xmlIts1.Graph(output, instance);
}
finally
{
    output.Close();
    xmlIts1.Dispose();
}
```

```
xmlIts1.graph(System.out, instance);
```

In the example, the finally block is used to dispose the formatter on the formatter. It is good practice to dispose all connectors and formatters when they are no longer needed. This functionality can be accessed via the *using* statement in .NET or the *try-with-resource* pattern in Java.