IBM

HealthCare & Life Sciences

HL7 Clinical Document Architecture (CDA) Builder

User's Guide

Version 3 Release 2

Table of Contents

1 INTRODUCTION 5

2 OVERVIEW 7

3 INSTALLING HL7 CDA BUILDER 9

3.1 PREPARING TO INSTALL 9

3.2 RUNNING THE INSTALLATION PROGRAM 9

3.3 SETTING UP THE JAVA ENVIRONMENT 10

3.4 DATA ENTRY MANAGER 10

4 JAVADOC 11

VOCABULARY CODES 12

4.1 AUTOMATING THE VOCABULARY MAPPING 13

5 CDA DOCUMENT CLIENT PROGRAM EXAMPLE 14

5.1 CREATING THE CDA DOCUMENT OBJECT 14

5.2 CREATING AN ORGANIZATION OBJECT 14

5.3 CREATING A PATIENT OBJECT 14

5.4 CREATING AN AUTHOR OBJECT 15

5.5 CREATING A CUSTODIAN OBJECT 15

5.6 SETTING THE VITAL SIGN INFORMATION 15

5.7 GENERATING THE CDA XML MESSAGE 16

6 UPDATEPATIENT EXAMPLE 18

6.1 FAMILY HISTORY 18

6.2 GENERATING THE UPDATEPATIENT XML MESSAGE 18

APPENDIX A: EXAMPLE CLIENT PROGRAM 19

APPENDIX B: EXAMPLE OF UNRESOLVED TERMS FILE 21

APPENDIX C: EXAMPLE OF CLIENT PROGRAM CALLING TERMRESOLVER AND TERMMANAGER API 22

APPENDIX D: EXAMPLE OF RESOLVED TERMS FILE GENERATED BY TERMRESOLVER 23

1 Introduction

This document is intended for developers who plan to write client programs that use the IBM HL7 CDA Builder Java APIs. The purpose of the IBM HL7 CDA Builder (referred to in this document as *CDA Builder*) is to enable Java developers to build HL7 CDA documents and HL7 UpdatePatient messages with consistent content and organization, without being burdened with the complexity generally associated with using the HL7 application program interfaces (APIs*[1]) directly. The CDA builder does this by providing a set of APIs that wrap the existing HL7 APIs to simplify the data collection process.

The Java client program that uses the CDA Builder APIs could be a user interface that prompts for data, or a program that reads legacy data from existing databases, spreadsheets, etc. Once the data that is to be included in the HL7 message is collected and the appropriate Java object populated, the CDA builder uses the HL7 APIs to build up the complex data types and internal object map that represents the HL7 document and renders the XML message as a string.

This document will show coding examples illustrating the use of the CDA Builder APIs, and provide explanations of each example. The intent is that the reader will become familiar enough with the CDA Builder that they can write their own client programs.

[1] The HL7 APIs are an open source project developed by the Java SIG of the HL7 organization. The APIs are also distributed by NCICB under the name caAdapter.

2 Overview

In order write an effective client program that utilizes the CDA builder APIs, an understanding of the major components is necessary. Consider the following diagram, which is explained in more detail below:

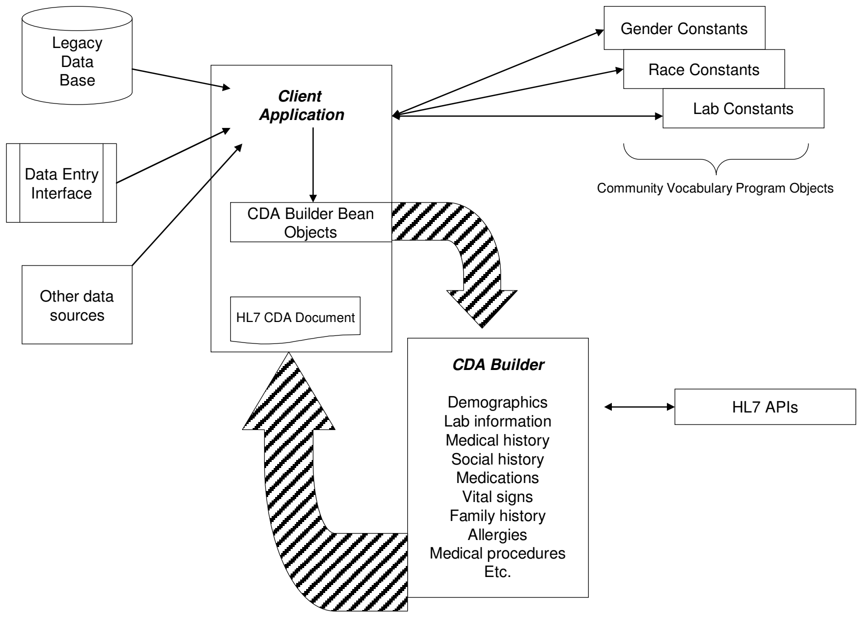

Figure 1

The client application, shown in the upper center of Figure 1, is specific to the environment in which it is running and must be designed to complete, at a minimum, the following core steps:

1. Data Access

Gather the data that will be used to populate the CDA document or `UpdatePatient` message. This can be accomplished using any method of input, including reading records from a legacy database, reading records from a federation of databases, or input from a data entry user interface.

2. Population of CDA Builder objects

The client program must use the data it has accessed to populate CDA builder objects. There is a set of Java beans in the **`com.ibm.lifesci.hl7.builder.core.beans`** package. Using the setter methods of these beans (i.e. `setGender()`), the client program should set all of the data it wants included in the message.

3. Invoking the CDA Builder

Once the CDA builder objects are populated with data, the client calls a method on the CDA builder to build the XML file and return it to the client program.

The CDA Builder is capable of generating two types of HL7 XML messages, `UpdatePatient` and `CDA`. `UpdatePatient` is used to introduce a new patient to the system, or update demographic information on an existing patient. `CDA` is used to record medical information for a patient.

3 Installing HL7 CDA Builder

This Clinical Genomics component consists of several jar files, properties files, and one war file (web archive) which is an example WebSphere application that uses the HL7 CDA Builder APIs.

3.1 Preparing to install

The only prerequisites for using HL7 CDA Builder are caAdapter 3.2 from NCICB and the MIFs from HL7.

Note: Previous versions of caAdapter are not supported with HL7 CDA Builder.

To download and install caAdapter 3.2:

1. Download the caAdapter package from http://ncicb.nci.nih.gov/download/index.jsp .
2. Click on the caAdapter download link and, if you agree, accept the license agreement.
3. Unzip the caAdapter package into a directory on the system where you will be installing HL7 CDA Builder.

To download and install MIFs:

1. Create a new directory in the `caAdapter` top-level directory called `mif`.
2. Download the Java SIG API Demo Release from the HL7 members website: http://www.hl7.org/Memonly/downloads/javasig/jsigDemo20050915.zip
3. Unzip the Java SIG package on your system.
4. Copy the contents of the Java SIG subdirectory called `javaSIG/data/mif207` into the newly created `mif` directory in the caAdapter package.

3.2 Running the installation program

Perform the following steps to install HL7 CDA Builder:

1. Run the install program:
   - Installing on Windows:
     a. If you have downloaded the product, go to the file where you unzipped the download. Locate and run `CGV3.2_HL7CDABuilder_WinInstall.exe.`.
     b. If you have a CD, insert the CD and run the installer from the CD drive. If you prefer, copy the CD contents into a folder on your system and run the installer from there.
   - Installing on AIX:
     a. Mount the CD on the system or transfer the AIX install program named `CGV3.2_HL7CDABuilder_AIXInstall.bin` to a directory on your system.
     b. From the directory where the file is located, type: `./CGV3.2_HL7CDABuilder_AIXInstall.bin` to run the install program.
2. Read the Welcome page and click **Next**.
3. Read the license agreement. If you agree, select **I accept the terms of the license agreement** and click **Next**.
4. Provide the **Install Directory Name**. This is the name of the directory where you want HL7 CDA Builder installed. Accept the default directory name given, or enter a new name for the directory. Click **Next**.
5. Read the summary information and click **Next** to continue the installation
6. Click **Finish** to exit the wizard.

3.3 Setting up the java environment

To build CDA messages with the HL7 CDA Builder in a java application several jars and directories must be included in the java classpath. The following jars are included with HL7 CDA Builder:

Table 1. HL7 CDA Builder jars

Jar	**Description**
HL7BuilderCore.jar	This is the core HL7 CDA Builder jar that must be in the classpath to use the HL7 CDA Builder APIs.
HL7BuilderLupus.jar	This jar contains extensions to HL7 CDA Builder to include Lupus-specific information in a CDA message. It is only needed if these Lupus extensions are used. Additional disease-specific extensions may be added in future releases.
HL7BuilderTerms.jar	This jar contains all of the terminology objects used in the API. It must be in the classpath unless your own term objects are generated and used with HL7 CDA Builder.
MetaFileAPI.jar	This jar contains classes for persisting CDA messages to a database and is only needed when used in conjunction with the provided sample application, Data Entry Manager.

Directories that must be included in the classpath are:

- The `conf` directory. This contains several properties files that, in most cases, will not need to be modified. (On Windows, the default path is `C:\IBM\CG\HL7CDABuilder\conf`)
- The directory containing the mifs from HL7.

In addition to the above jars and and the `conf` directory provided with HL7 CDA Builder, you will also need to include several jar files provided by caAdapter. The required caAdapter jar files are:

- javasig.jar
- saxon8.jar
- jdom.jar
- dom4j-1.6.jar
- serialMif.jar
- xbean.jar
- xbean_xpath.jar
- hibernate3.jar

3.4 Data Entry Manager

Included with HL7 CDA Builder is a sample WebSphere application called **Data Entry Manager,** which is a web interface for entering a patient's medical information. It uses HL7 CDA Builder to build CDA and Update Patient messages to be sent to the CG Server.

Installation: Use the WebSphere Administration console to install the `DataEntryManager.war` file, which is included in the `lib` directory.

Modifications: Before using Data Entry Manager, you will need to modify `DataEntry.properties` and `repo.dsd`. These properties files are located in the `WEB-INF/classes` directory of the war file.

For more information on using Data Entry Manager, please contact an IBM Service representative.

4 Javadoc

As you read through this document, you may find it beneficial to refer to the Javadoc for the CDA Builder. To access the documentation, navigate to the `doc\javadoc` path relative to the installation directory, and open `index.html`.

> **Note:** To speed development time, use an Eclipse-based development environment, version 3.2 or newer, and attach the Javadoc to the HL7 builder JARs to provide the parameter names when using the auto complete feature.

Vocabulary Codes

The HL7 messages generated by the CDA builder represent much of the data as coded values. Using a coded vocabulary provides consistent format and meaning to the data. By representing the data consistently, automated analysis and document sharing are more easily achieved.

The approach used with the CDA builder was to create static program objects (i.e Java constants) for vocabulary codes. All of the vocabulary program objects are found in the `com.ibm.lifesci.hl7.builder.core.beans.terms` package.

As an example of how to create and use one of these static program objects, consider how a clinical patient's gender could be represented. If you decide that gender should be represented by two distinct values, you would create a program in the `com.ibm.lifesci.hl7.builder.core.beans.terms` package named `GenderConstants` that defined those two values.

> **Note:** The class MUST be named `GenderConstants` because classes that accept a gender parameter, such as `com.ibm.lifesci.hl7.builder.core.beans.person.BasicPerson`, take a parameter of type `com.ibm.lifesci.hl7.builder.core.beans.terms.GenderConstants` (refer to the Javadoc for more details).

You might choose to create a program that would look something like this:

```
package com.ibm.lifesci.hl7.builder.core.beans.terms;

import java.io.Serializable;

public class GenderConstants extends Term implements Serializable
{
      public final static GenderConstants FEMALE = new
      GenderConstants("N/A", "Female", "F",
      "2.16.840.1.113883.5.1", "AdministrativeGender");

      public final static GenderConstants MALE = new
      GenderConstants("N/A", "Male", "M",
      "2.16.840.1.113883.5.1", "AdministrativeGender");

      private GenderConstants(String term, String displayName,
      String code, String codeSystem, String codeSystemName,
      String synonym, boolean isLocal) {
         super(term, displayName, code, codeSystem,
      codeSystemName);
      }
}
```

With this program you are creating two constants named `MALE` and `FEMALE`, and defining the vocabulary characteristics for each. The characteristics represent the information that is placed in the HL7 XML message by the CDA builder during message generation. (See the Javadoc for a description of the characteristics.) As shown, the program must inherit from the `com.ibm.lifesci.hl7.builder.core.beans.terms.Term` class and must implement the `java.io.Serializable` class.

The Java constants that are distributed with the CDA Builder are provided as an example. Some of the terms are defined from HL7 vocabularies, but many are just fabricated sample terms that don't refer to any standard coding system. The terms can be used as is, but most likely you will want to create your own terms that map to a standard vocabulary such as SNOMED CT, ICD9, or LOINC.

4.1 Automating the Vocabulary Mapping

The CDA Builder provides the TermResolver and TermManager program application interfaces (APIs) as example mechanisms that can assist with resolution of raw terms to vocabulary system codes. Figure 2 shows a high-level view of how unresolved terms flow through the TermResolver and TermManager APIs to become resolved terms in the form of program constants:

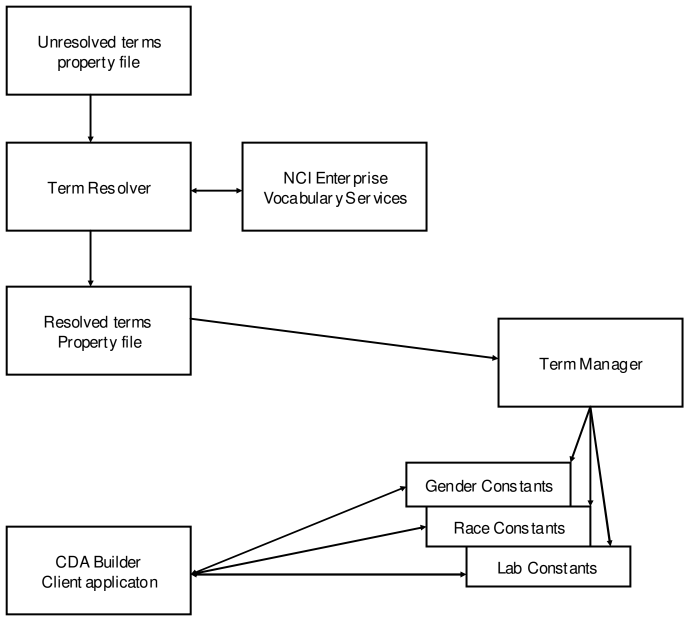
Unresolved terms property file
Term Resolver
NCI Enterprise Vocabulary Services
Resolved terms Property file
Term Manager
Gender Constants
Race Constants
Lab Constants
CDA Builder Client applicaton

Figure 2

Property files can be created to specify terms that need to be resolved to vocabulary codes. The property files can then be resolved by passing that property file into the TermResolver API (see Appendix B: Example of Unresolved Terms Filefor examples of a properties file containing the raw terms to be resolved and Appendix C: Example of Client Program calling TermResolver and TermManager API for a client program calling the TermResolver API).

The `TermResolver` makes a call out to the National Cancer Institute (NCI) Enterprise Vocabulary Services (EVS), passing the term to be resolved. The NCI EVS will attempt to look up the term and pass back the associated code information. This process is repeated once for each term specified in the unresolved terms property file. Once completed, the results should be reviewed to ensure that the processed terms were resolved to acceptable codes, and that there are no unresolved terms that require additional processing.

Following resolution, the resolved term property file contains the terms identified by the group along with their associated vocabulary codes. This property file can then be processed using the `TermManager` API to create Java source files that can be compiled into executable programs. Appendix C: Example of Client Program calling TermResolver and TermManager API shows examples of calling `TermManager` and passing it the property file. The result of what the `TermManager` generates should be similar to the `com.ibm.lifesci.hl7.builder.core.beans.terms.GenderConstants` example program listed above.

See the Javadoc documentation for the CDA builder for more information about the `TermResolver` and `TermManager` APIs.

5 CDA Document Client Program Example

To illustrate how to use the CDA builder, the following sections describe the steps required by a client program to construct and render a CDA document that contains vital signs information for a patient. (The complete program is shown in Appendix A: Example Client Program.) Typically, the values would be retrieved from whatever data source the program author needs to access, such as a database; however, for simplicity, the sample shown uses literal values.

Please refer to the Javadoc documentation for further description of the program interfaces.

5.1 Creating the CDA Document Object

The first step in writing a program to generate a CDA document is to instantiate a `CDADocument` object. This will be the top-level object that contains all of the data that is to be included in the HL7 XML file. To instantiate a `CDADocument` object, call its constructor and pass it an ID:

```
CDADocument cda = new CDADocument("CDADoc1");
```

5.2 Creating an Organization Object

Building an organization requires, at a minimum: the organization name, their assigned object identifier (OID), and a URL to provide a namespace for the enclosed identifiers.

To determine if an organization has already been assigned an OID, visit the **ASN.1 Object Identifier (OID) Repository** at http://asn1.elibel.tm.fr/oid/index.htm. This site provides tools to search the repository by name or look up information by OID. If an existing OID cannot be located, see http://asn1.elibel.tm.fr/en/tools/oid/faq.htm#8 for information on having one assigned.

Note: A valid OID is not required to create an organization. Any string can be used for the organization OID as long as it is unique.

The site URL is used to establish namespace identifiers that ensure uniqueness across different document sources. If the identifiers supplied *are* guaranteed to be unique across the entire organization, supplying just the top-level domain as the site URL would be sufficient. However, if the identifiers *are not* guaranteed to be unique within the organization, a more specific URL should be supplied, for example: `ibm.com/industries/healthcare` or `rchland.ibm.com`.

To build an organization object representing IBM Global Services:

```
Organization ibmGs = new Organization("1.3.6.1.4.1.5766",
      "www.ibm.com/services", "IBM Global Services");
```

5.3 Creating a Patient Object

Building a `patient` object requires two pieces of information:

1. the provider organization, which is used to provide the namespace, *and*
2. an identifier that is unique to the patient within the namespace provider organization.

Below is a sample patient from the IBM Global Services organization object created above:

```
Patient patient = new Patient(patientId, ibmGs);
patient.setDateOfBirth(HL7Date.build(1951, 6, 6));
patient.setGender(GenderConstants.MALE);
patient.setMaritalStatus(MaritalStatusConstants.MARRIED);
patient.setName(new Name("Mr.", "John", null, "Doe", null));
```

5.4 Creating an Author Object

The `author` object consists of the information used to uniquely identify the hardware device and software platform that created the document, along with the organization the authoring device represents. For example, if IBM Global Services was generating documents for a research institution, the `author` would be created as:

```
Author author = new Author("192.168.1.1", "IBM xSeries 306m",
      "IBM Healthcare & Life Sciences HL7 Builder 2.0", ibmGs);
```

`Author` also allows the specification of an address that represents the physical location of the device, as well as multiple contact methods that can list available telecommunication addresses.

```
ContactMethod contact = new contactMethod(
      TelecommunicationAddressUseConstants.WORK_PLACE,
      TelecommunicationSchemeConstants.TELNET, "192.168.1.1");
author.addContactMethod(contact);

Address address = new Address(
      PostalAddressUseConstants.WORK_PLACE, "3605 Highway 52 N",
      "Building 15-1 Lab", "Rochester",
      StateProvConstants.MINNESOTA, "55901",
      CountryConstants.UNITED_STATES)
author.setAddress(address);
```

5.5 Creating a Custodian Object

The `custodian` is the person or institution that is charged with maintaining the original document. In this example, the `custodian` is set to a newly created organization object:

```
cda.setCustodian(new Organization("1.3.6.1.4.1.5766",
"www.ibm.com/services", "IBM Global Services"));
```

5.6 Setting the Vital Sign Information

Vital sign information is included by instantiating a `VitalSign` object, setting the appropriate values, and then adding it to the `CDADocument` object:

```
VitalSign vs = new VitalSign("1", VitalSignConstants.WEIGHT, new
      PhysicalQuantity(170, UnitConstants.POUNDS));
vs.setEffectiveTime(new EffectiveTime(HL7Date.build(2006, 07, 19,
      11, 14, 35)));
cda.addVitalSign(vs);
```

5.7 Generating the CDA XML Message

To generate the `CDA` message, call the static method `com.ibm.lifesci.hl7.builder.core.CDABuilder.build(CDADocument)`, which outputs the CDA XML message as a string. For example:

```
String xmlMessage = CDABuilder.build(cda);
```

The content of the new `CDA` document now exists in the form of the Java String object `xmlMessage`. The client program is then free to store or transmit the document in whatever manner the client's workflow process demands.

This is just a small example of how to use the CDA builder interfaces to produce CDA documents. Following is a complete list of the types of information that the CDA builder interfaces can accept:

Note: Please refer to the Javadoc for details of the actual object names and parameters.

- Demographics
- Encounter
- Labs
- Medications
- Medical Diagnoses/Symptoms
- Allergies
- Family History
- Lifestyle (smoking/drinking/exercise/etc.)
- Medical Procedures (surgeries, etc.)
- Pregnancy History
- Vital Signs

6 UpdatePatient Example

6.1 Family History

Note: The `Relationship` object is only outputted from `UpdatePatientBuilder`.

The family history of a patient can be modeled using a combination of two methods: explicitly adding relationships between patients using `Relationship` objects and the `addRelationship` method when generating a `Patient` object for use with `UpdatePatientBuilder` message, or by adding `FamilyHistoryDiagnosis` objects using `addFamilyHistoryDiagnosis` when generating a `CDADocument` for use with `CDABuilder` message.

The preferred method is to explicitly link patients using relationships, allowing the family history to be determined dynamically by retrieving the last information on the related subjects. In contrast, when using `FamilyHistoryDiagnosis` objects, the family history consists *only* of those diagnoses that were provided by the given patient's encounters. Should a family member be diagnosed with a disease between a patient's last visit and the time of querying, that diagnosis would not be available until a new encounter with the updated information is supplied. A combination of the two methods may be used if some relatives are patients and some are not.

Adding a `Relationship`:

```
Person father = new Person("1", new Name("John", "L.", "Doe"));
Father.setGender(GenderConstants.MALE);
patient.addRelationship(new Relationship("1", father,
      PersonalRelationshipTypeConstants.FATHER));
Person father = new Person("2", new Name("Brenda", "", "Doe"));
Father.setGender(GenderConstants.FeMALE);
patient.addRelationship(new Relationship("2", mother,
      PersonalRelationshipTypeConstants.FATHER));
```

Adding a `FamilyHistoryDiagnosis:`

```
FamilyHistoryDiagnosis diag = new FamilyHistoryDiagnosis("1",
      FamilyHistDiagnosisConstants.SCLERODERMA,
      PersonalRelationshipTypeConstants.FATHER);
cdaDocument.addFamilyHistoryDiagnosis(diag);
```

or

```
Person father = new Person("1", new Name("John", "L.", "Doe"));
Father.setGender(GenderConstants.MALE);
FamilyHistoryDiagnosis diag = new FamilyHistoryDiagnosis("2",
      FamilyHistDiagnosisConstants.SCLERODERMA,
      PersonalRelationshipTypeConstants.FATHER);
cdaDocument.addFamilyHistoryDiagnosis(diag);
```

6.2 Generating the UpdatePatient XML Message

To generate the `UpdatePatient` message, create and populate a `Patient` object and call the static method `com.ibm.lifesci.hl7.builder.core.UpdatePatientBuilder.build(Patient)`, the output of which is the `UpdatePatient` XML message as a string. For example, using the `patient` object built above:

```
String xmlMessage = UpdatePatientBuilder.build(patient);
```

Appendix A: Example Client Program

```
package myPackage;

import com.ibm.lifesci.hl7.builder.core.CDABuilder;
import com.ibm.lifesci.hl7.builder.core.beans.Author;
import com.ibm.lifesci.hl7.builder.core.beans.CDADocument;
import com.ibm.lifesci.hl7.builder.core.beans.Organization;
import com.ibm.lifesci.hl7.builder.core.beans.datatypes.Confidentiality;
import com.ibm.lifesci.hl7.builder.core.beans.datatypes.EffectiveTime;
import com.ibm.lifesci.hl7.builder.core.beans.datatypes.HL7Date;
import com.ibm.lifesci.hl7.builder.core.beans.datatypes.Name;
import com.ibm.lifesci.hl7.builder.core.beans.datatypes.PhysicalQuantity;
import com.ibm.lifesci.hl7.builder.core.beans.observation.VitalSign;
import com.ibm.lifesci.hl7.builder.core.beans.person.Patient;
import com.ibm.lifesci.hl7.builder.core.beans.terms.AnatomicalSiteConstants;
import com.ibm.lifesci.hl7.builder.core.beans.terms.GenderConstants;
import com.ibm.lifesci.hl7.builder.core.beans.terms.UnitConstants;
import com.ibm.lifesci.hl7.builder.core.beans.terms.VitalSignConstants;

public class ExampleCDA {

  private static Organization rtp = new
     Organization("1.3.6.1.4.1.21367.2005.1.1", "w3.rtp.ibm.com/hospital",
     "Research Triangle Park Hospital");

  public static void main(String[] args) throws Exception {
     CDADocument cda = new CDADocument("1153307675766");

     Patient patient = new Patient("EB98352768", rtp);
     patient.setName(new Name("Henry", "Crossing"));
     patient.setGender(GenderConstants.MALE);
     patient.setDateOfBirth(HL7Date.build(1976, 04, 29));

     cda.setCreationTime(HL7Date.build(2006, 07, 19, 11, 14, 35));
     cda.setConfidentiality(Confidentiality.NORMAL);
     cda.setPatient(patient);
     cda.setAuthor(new Author("EB99352768", "IBM Thinkpad T41", "IBM HL7 Builder
     v2.0", rtp));

     cda.setCustodian(rtp);

     VitalSign vs = new VitalSign("1", VitalSignConstants.WEIGHT, new
     PhysicalQuantity(170, UnitConstants.POUNDS));

vs.setEffectiveTime(new EffectiveTime(HL7Date.build(2006, 07, 19, 11, 14, 35)));

cda.addVitalSign(vs);

     vs = new VitalSign("2", VitalSignConstants.DIASTOLIC_BLOOD_PRESSURE, new
     PhysicalQuantity(70, UnitConstants.MM_HG));

     vs.setAnatomicalSite(AnatomicalSiteConstants.ARM);
     vs.setEffectiveTime(new EffectiveTime(HL7Date.build(2006, 07, 19, 23, 12,
     02)));

cda.addVitalSign(vs);

     vs = new VitalSign("3", VitalSignConstants.SYSTOLIC_BLOOD_PRESSURE, new
     PhysicalQuantity(120, UnitConstants.MM_HG));

     vs.setAnatomicalSite(AnatomicalSiteConstants.ARM);
     vs.setEffectiveTime(new EffectiveTime(HL7Date.build(2006, 07, 19, 23, 12,
     02)));

cda.addVitalSign(vs);

     System.out.println(CDABuilder.build(cda));
  }
}
```

Appendix B: Example of Unresolved Terms File

Here are the contents of an example file that contains tab delimited terms to be resolved. The ordering of the data is: group name, term to be resolved, and synonym.

Group Name	**Term**	**Synonym**
gender	male	male
gender	female	female

Appendix C: Example of Client Program calling TermResolver and TermManager API

```
package com.ibm.lifesci.test;

import java.io.*;
import com.ibm.lifesci.hl7.builder.term.*;

public class TermResolutionExample {

  public static void main(String[] args) throws Exception {
      TermResolver tr = new TermResolver("unresolvedTerms.txt",
      "resolvedTerms.txt");
      tr.resolveTerms();
      tr.writeTerms();

      TermSource[] sources = new TermSource[] { null };
      sources[0] = new TermSource(new File("resolvedTerms.txt"),
      TermSource.SOURCE_COMMUNITY);

      TermManager tm = new TermManager(sources);
      tm.buildConstants("application/src/ ");
  }
}
```

Appendix D: Example of Resolved Terms file generated by TermResolver

gender	F-03CE5	2.16.840.1.113883.6.96	SNOMED_CT	Female (finding)	female	female
gender	F-03CE6	2.16.840.1.113883.6.96	SNOMED_CT	Male (finding)	male	male