

CDA Tools Guide 1.0

John T.E. Timm, IBM Research
David A. Carlson, Veterans Health Administration
9/30/2009

Installation and Configuration

The purpose of this section is to provide installation and configuration instructions.

Prerequisites

Eclipse 3.4 Ganymede or IBM Rational Software Modeler (RSM) 7.5.3
Java 1.5+

OHT Web Site Account

Please go to <http://www.openhealthtools.org> and sign up for an account. You will need the username/password to access the source code repository.

Eclipse Users

Documentation for Eclipse can be found here:

<http://www.eclipse.org/documentation/>

Eclipse users must have the following plug-ins installed:

Eclipse UML2
Eclipse Modeling Framework (EMF)
Eclipse OCL
Eclipse Papyrus

Eclipse MDT Update Sites (containing UML and OCL plug-ins is here):

<http://www.eclipse.org/modeling/mdt/updates/>

Eclipse EMF Update Sites:

<http://www.eclipse.org/modeling/emf/updates/>

A build of Eclipse Papyrus can be downloaded here:

Important configuration note: Papyrus requires Eclipse 3.5 Galileo release

<https://mdht.projects.openhealthtools.org/servlets/ProjectDocumentList?folderID=85&expandFolder=85&folderID=0>

Rational Software Modeler (RSM) Users

Documentation for RSM can be found here:

<http://publib.boulder.ibm.com/infocenter/rsmhelp/v7r5m0/index.jsp>

When installing Rational Software Modeler, make sure to include the “Eclipse Extensibility” features during installation component selection.

Subversion Client

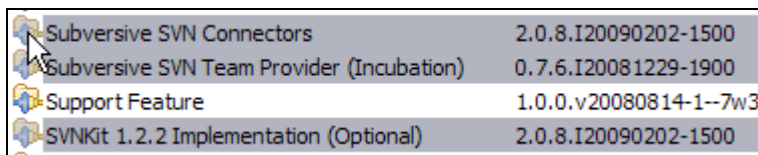
Subversion is a software version control system. In order to access and download CDA models for use in model development and runtime generation, a subversion client must be installed. We recommend the Subversive client which can be downloaded from the following update site:

Subversive: <http://download.eclipse.org/technology/subversive/0.7/update-site/>

Additionally, the Polarion connectors plug-in must be downloaded and installed from here:

Connectors: <http://www.polarion.org/projects/subversive/download/eclipse/2.0/update-site/>

Make sure to include the SVNKit feature when installing from the update site. Your environment should contain Subversive SVN Connectors, Subversive SVN Team Provider and SVNKit features as shown below.



Subversive SVN Connectors	2.0.8.I20090202-1500
Subversive SVN Team Provider (Incubation)	0.7.6.I20081229-1900
Support Feature	1.0.0.v20080814-1--7w3
SVNKit 1.2.2 Implementation (Optional)	2.0.8.I20090202-1500

CDA Tools Feature

Once the prerequisites have been met from the previous section, the CDA Tools Feature can be installed from the Update Site.

- Either RSM or Papyrus should be selected, NOT both

CDA Tools Update Site: <http://oht-modeling.sourceforge.net/cda/updates>

Open Health Tools MDHT CDA Update Site	
UML Modeling Platform for Clinical Document Architecture (CDA)	
UML Tools for CDA	0.1.0.200909291856
UML Tools for CDA in RSM	0.1.0.200909291857

RSM Users: make sure to select the “UML Tools for CDA in RSM” option.

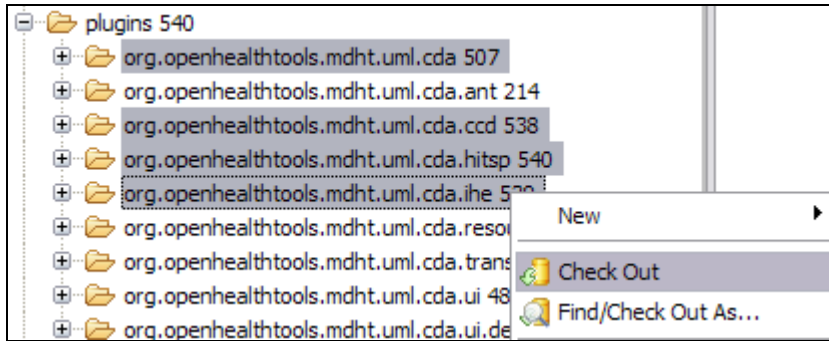
CDA Models and Runtime

The next step is to download the CDA models and runtime from the software repository at MDHT. Using the subversion client switch to the “SVN Browsing Perspective” and create a new repository location as show below:

Repository Location: <http://mdht.projects.openhealthtools.org/svn/mdht>

Where the username and password fields are the login used to sign in at the OHT website. Check out the following plug-ins into your workspace (as shown below):

core/plugins/org.openhealthtools.mdht.uml.hl7.datatypes
core/plugins/org.openhealthtools.mdht.uml.hl7.vocab
cda/plugins/org.openhealthtools.mdht.uml.cda
cda/plugins/org.openhealthtools.mdht.uml.cda.ccd
cda/plugins/org.openhealthtools.mdht.uml.cda.ihe
cda/plugins/org.openhealthtools.mdht.uml.cda.hitsp

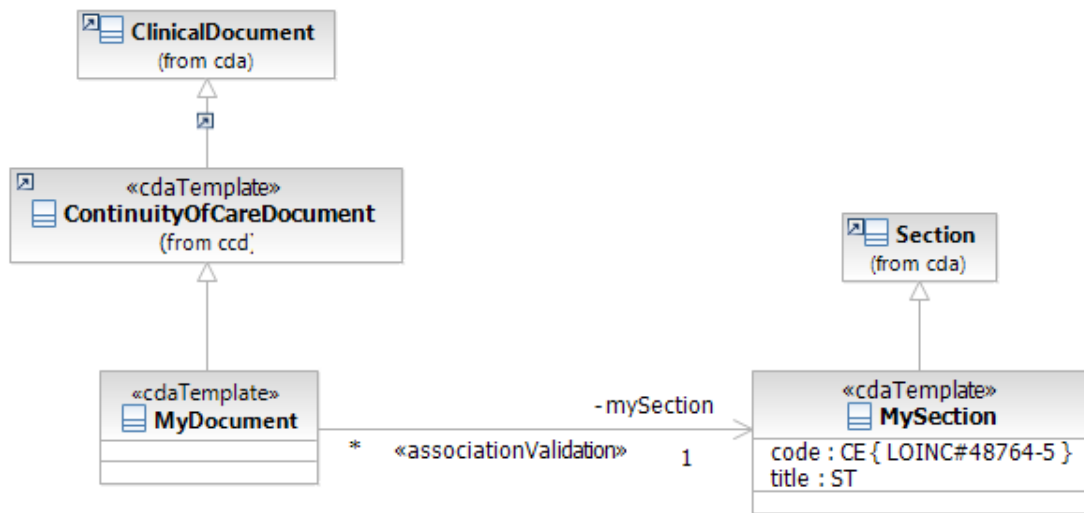


Getting started with the Example project

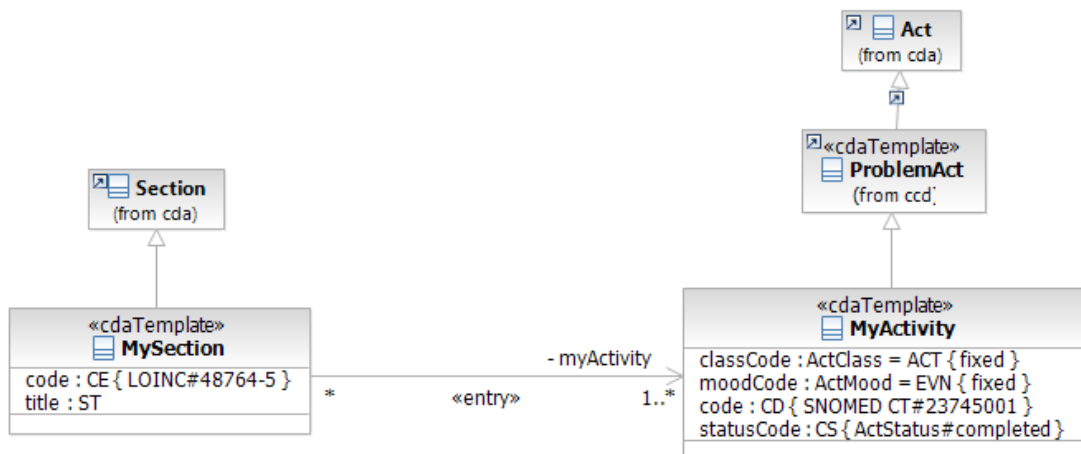
The purpose of this section is to describe the example project. The example project that is available for download from the MDHT software repository at:

cda/examples/org.openhealthtools.mdht.uml.cda.example

The example project contains a simple template model that further constrains the CCD (Continuity of Care Document) model. The model contains three templates: MyDocument, MySection, and MyActivity. The following class diagram captures the relationships between the MyDocument and MySection classes and classes in CCD and the base CDA model. Additionally, it displays the relationship between MyDocument and MySection. The MyDocument and MySection templates both contain the <<cdaTemplate>> stereotype. This indicates, in the diagram, that they further constraint the ClinicalDocument and Section classes, respectively, from CDA. The <<cdaTemplate>> stereotype also holds the value of the template identifier in standard HL7 OID format.



The diagram below captures the relationships between MySection and MyActivity.



These diagrams should give you a feel for the various classes, properties, and relationships in a CDA template model. You will notice that the attributes defined in **MyDocument**, **MySection**, and **MyActivity** mirror those defined in their parent classes from CCD and CDA. These attributes are used to further constrain the attribute definitions from the parent. For example, if a code is optional in the parent, defining an attribute with the same name and type as the parent but specifying a minimum cardinality of 1 makes the attribute required. This type of *property redefinition* is used during the model-to-model transformation step to generate OCL constraints that are attached to the template model classes.

Relationships between classes that specialize `cda::ClinicalDocument` and classes that specialize `cda::Section`, use directed associations to indicate a relationships between two templates in an implementation guide. This type of relationship models conformance rules such as:

MyDocument SHALL contain exactly one MySection

and

MySection SHOULD contain one or more MyActivity

Additional information about the conformance rule can be specified in property values for the `<<associationValidation>>`, `<<entry>>`, and `<<entryRelationship>>` stereotypes. The CDA Tools property tab is a convenient mechanism for entering such data:

General	Entry Type Code: <input type="text" value="COMP (component)"/>
Stereotypes	Validation
Documentation	Severity: <input type="text" value="Warning"/>
Constraints	Message: <input type="text" value="My Section SHOULD contain one or more My Activity."/>
Relationships	
CDA Tools	
Advanced	

The tab is context-sensitive and will only display the data fields available for the element selected in the model diagram or tree view.

Validation *severity* (a property of the <<associationValidation>> stereotype) is used to differentiate whether the relationship is strictly required for an instance to be considered valid. Validation severity may take on one of three possible values ERROR (strictly required, SHALL), WARNING (recommended/best practice, SHOULD), and INFO (optional, MAY).

UML properties defined in the template model are used to place further constraints on those defined in the parent template model and base CDA model. For example, MySection.code defines a fixed value for the section code using stereotype values from the <<vocabSpecification>> stereotype. The CDA Tools property tab for MySection.code is shown below. This tab can be used to specify the Code, Code System, Code System Version, etc. for a particular coded attribute.

General	Vocabulary Specification
Stereotypes	Code System ID: <input type="text" value="2.16.840.1.113883.6.1"/>
Documentation	Code: <input type="text" value="48764-5"/>
Constraints	Validation
Relationships	Severity: <input type="text" value="Error"/>
CDA Tools	Message: <input type="text" value="The value for My Activity code SHALL be 48764-5 'Summary purpose' 2.16.8"/>
Advanced	

In addition to constraints on coded attributes, structural attributes such as classCode, typeCode, and moodCode can be defined with a fixed or default value in the template model. MyActivity.classCode and MyActivity.moodCode are used to illustrate this type of constraint. Fixed or default value constraints can also be applied to attributes that use text-based datatypes such as ED or ST as illustrated by the MySection.title attribute.

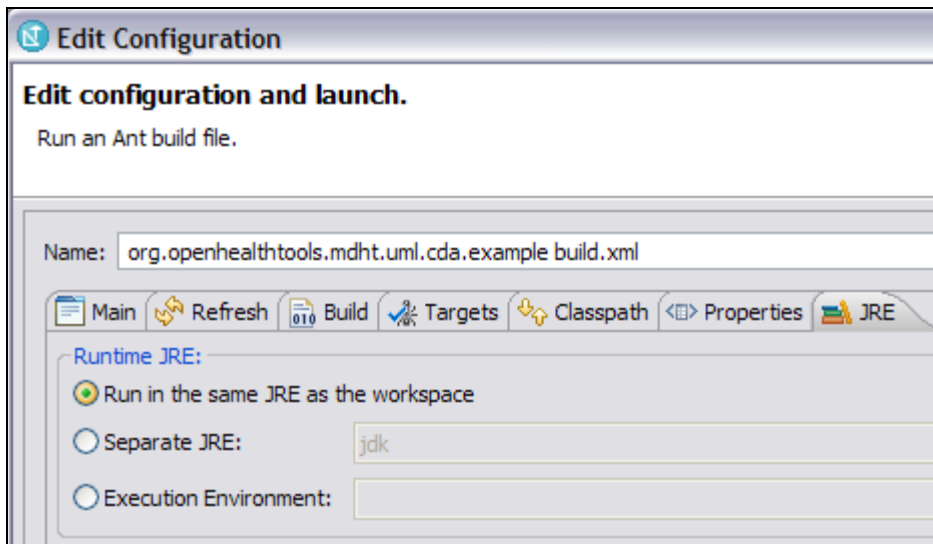
General	Text Value Constraint: My Section Title Goes Here.
Stereotypes	Validation
Documentation	Severity: Error <input type="button" value="v"/>
Constraints	Message: My Section SHALL contain a title.
Relationships	
CDA Tools	

Generating Java from a CDA Template Model

Summary of required steps:

1. Run build.xml to transform model
2. Reload example.genmodel to create EMF Ecore model
3. Generate Model Code

Now that we have explored the two diagrams associated with the example template model, we will begin to look at how this model can be used to generate a Java-based runtime implementation for producing, consuming, and validating instances. The first step in the process is to transform the template model into an implementation model. The model-to-model transformation is invoked from an Ant build task. In order to run the transformation, right-click on the build.xml file in the project root directory and select “Run as > Ant Build...”



Make sure that “Run in the same JRE as the workspace” is selected from the “JRE” tab. Then click the “Apply” button at the bottom of the dialog and then “Run”. The console output should look something like this:

```

all:
convertEmxToUml:
    [delete] Deleting: C:\Documents and Settings\j...
    [xslt] Processing C:\Documents and Settings\j...
    [xslt] Loading stylesheet C:\Documents and S...

refresh:

transformModel:
    [cdatools] Loaded model: example
    [transformToEcoreModel] Saving model: file://C:/Docume...

refresh:
BUILD SUCCESSFUL
Total time: 2 seconds

```

The original template model is converted into an implementation model with “_Ecore.uml” appended to the name. In this example, the implementation model is named “example_Ecore.uml”. Constraints defined as associations and property redefinitions in conjunction with metadata specified in stereotype property values are transformed into OCL constraints and annotations that are used for validation and instance population at runtime. For example, the directed association relationship that was defined between MyDocument and MySection gets transformed into the following OCL constraint:

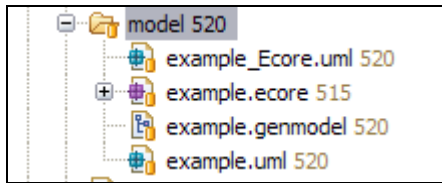
```

self.getSection()->one(section : cda::Section |
section.ocIsKindOf(example::MySection))

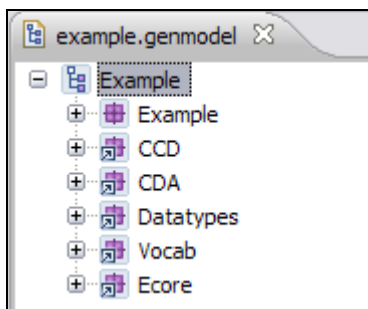
```

The getSection() method is an OCL query method that is defined in the base CDA model to return all of the sections under a clinical document by traversing the component.structuredBody.component association between the ClinicalDocument and Section classes. The rest of the constraint specifies that exactly one section is of type MySection (or one of its subclasses).

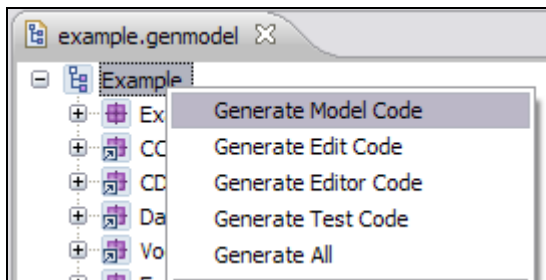
The second step in creating a Java runtime implementation is to create the EMF model that will be used to generate Java source code. The EMF model is provided for you in this example. The EMF model is composed of two parts: the Ecore model and the generator model. The Ecore model is very similar to the UML class model that was created during the template modeling step. The generator model contains information about how to transform the Ecore model into Java source code. As you can see in the diagram below, the Ecore model is named example.ecore and the generator model is named example.genmodel.



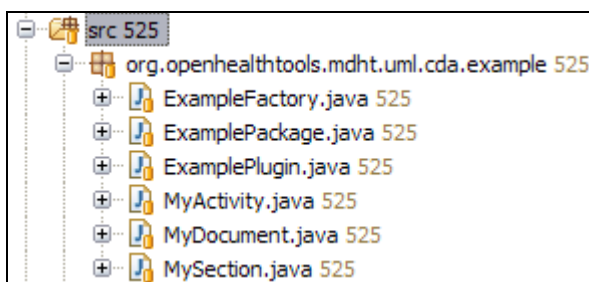
In order for changes to the template model to be reflected in the EMF model after the model-to-model transformation, the EMF generator model must be *reloaded*. In order to reload the generator model, right click on “example.genmodel” and select “Reload...” from the context menu. After the model import dialog pops up, click on the “Next” button twice and then click on the “Finish” button. The new updated generator model will display in editor view as shown:



The third step is to right click on the root element named “Example” and select “Generate Model Code”. This will generate all of the required Java packages and source files under the src folder of the project.



You can explore the generated code by browsing packages such as `org.openhealthtools.mdht.uml.cda.example` under the src folder.



You will notice a Factory class is generated. This class is used to create instances of the classes defined in your template model (e.g. MyActivity) using the following syntax:

```
MyActivity activity =  
ExampleFactory.eINSTANCE.createMyActivity().init();
```

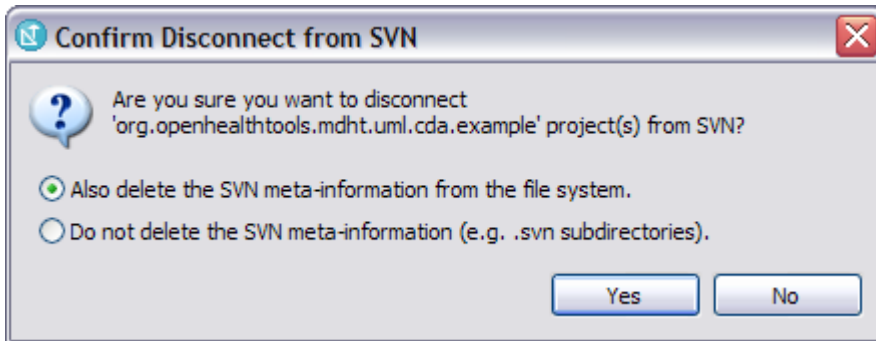
Notice the `init()` method is called after the `createMyActivity()` method. This is used to populate the instance with any fixed or default values that were specified in the template model. This reduces the number of method calls used to construct a valid instance.

We have provided a working test driver for this example in the `org.openhealthtools.mdht.uml.cda.example.tests` package to demonstrate the construction of a basic instance using classes from the template model. Additionally, the driver demonstrates how to use the CDA utility class to serialize and validate the document.

Refactoring the Example project into your own project

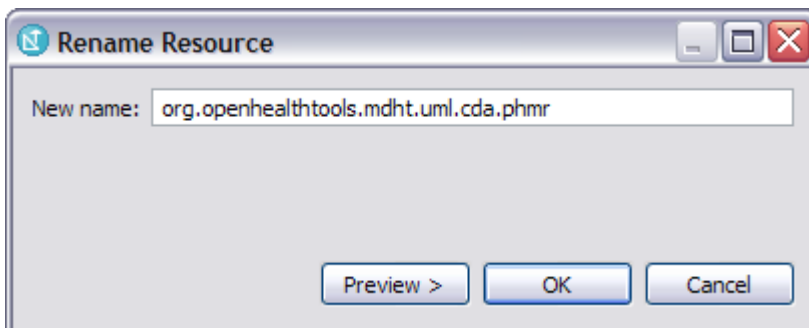
There is an Ant refactor script that can be used to convert the example project into a project for your specific needs. This can be accomplished in three steps:

1. Download and disconnect the example project from the MDHT SVN:
 - a. Check out cda/example/org.openhealthtools.mdht.uml.cda.example
 - b. Right-click on Project and select Team > Disconnect from SVN and select “Also delete SVN meta-information from file system”



NOTE: It is recommended that you check out a clean copy of the example to use in this step.

2. Rename the project:
 - a. Right-click on Project and select Refactor > Rename (example: org.openhealthtools.mdht.uml.cda.phmr)



3. Edit and run the refactor.xml script:
 - a. Open the refactor.xml script in the Ant or text editor
 - b. Change the property values for the following properties:
 - i. basePackage (e.g. org.openhealthtools.mdht.uml.cda)
 - ii. packageName (e.g. phmr)
 - iii. prefix (PHMR)
 - iv. nsURI (e.g. <http://www.openhealthtools.org/mdht/uml/cda/phmr>)

- c. Run the refactor.xml script as an Ant build using the same instructions found in the previous section. Make sure to run in the same JRE as the workspace.

```
<project name="Refactor Example CDA project" basedir="." default="refactor">
  <property name="basePackage" value="org.openhealthtools.mdht.uml.cda" />
  <property name="nsURI" value="http://www.openhealthtools.org/mdht/uml/cda/phmr" />
  <property name="packageName" value="phmr" />
  <property name="prefix" value="PHMR" />
  <property name="package" value="${basePackage}.${packageName}" />
```