SDN Contribution

Why to use the Common Model Interface (CMI)?

Applies to:

Models in Web Dynpro Runtime for Java .The article is largely release independent.

Summary

The intention in writing this document is to clarify the doubts and ambiguities which one thinks of on what's the benefit of doing a CMI implementation while using Web Dynpro models. The documents tries to answer why it should be used and what is it that it has to offer. I personally had several doubts on this when I started off and always felt the urge to know under what scenarios would a model need to use the Common Model Interface (CMI) and what is it that is achieved by it. This article tries to explain that with some simple scenarios. The article is not an official CMI documentation and doesn't explain all the features of CMI. For this please refer to the official CMI documentation.

Author(s): Arun Bhat J

Company: SAP Labs India Private Ltd.

Created on: 22 August 2006

Author Bio

Arun Bhat works at **SAP Labs** in the Web Dynpro Development area. He earlier worked in the Mobile Web Dynpro Client area for two years since he joined SAP in Feb 2004. Currently he has been working on Web Dynpro models and Context and has been supporting the NW04 and NW04S releases.

Perquisite

You have a good understanding of Web Dynpro. You have been exposed to Web Dynpro models and context and built applications using these.

Table of Contents

Role of CMI

The Common Model Interface or CMI as it is more commonly known is an abstraction to the model layer of a UI technology. However until now it has been used only with Web Dynpro. The role of CMI is three fold

1. To help in having a generic UI programming technique which is unaffected by the usage of different models. This would mean that the UI on its own can get complete necessary information from business logic layer(which can be interchangeably used) using only interfaces of CMI .Hence in ideal conditions it should be possible to say switch such an application from say an Adaptive RFC model to Web Services model with little or no modification at all.

   The CMI layer is a very thin layer consisting of just some interface definitions. It has dependencies only with the dictionary package. Hence, the implementing layer only has dependency on CMI itself and the dictionary runtime.

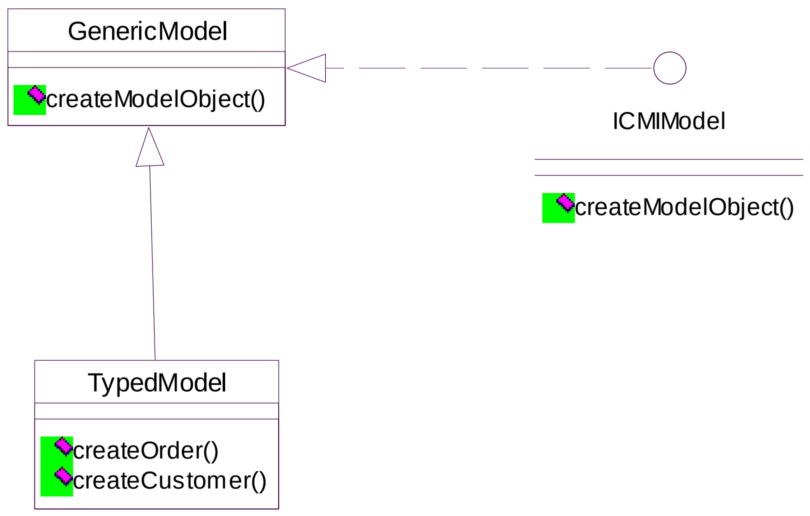
   CMI can be used by any UI implementation conceptually but currently only Web Dynpro seems to be making use of this as already mentioned.

2. CMI also makes it possible to capitalize on the dynamic programming capabilities from a UI framework like Web Dynpro. The scenario being, you don't know until runtime what your model would look like. With the ICMI-Info objects it's possible for you to build your UI and your context entirely from the runtime model metadata support. The UI and the context are created bit by bit by reading the metadata information from the CMI Metadata. For example it is possible to read the properties from the metadata info classes and build the context attributes entirely at runtime.

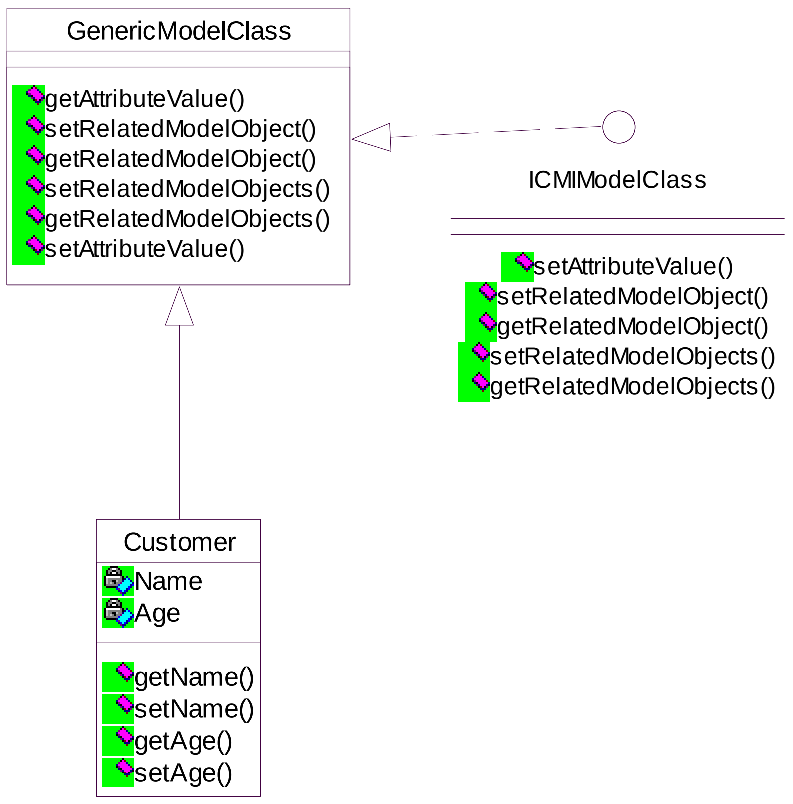
3. It makes it easier for you to work with Web Dynpro, with reduced coding in the application. For example when you bind a context to model consisting of a parent and a child, the child context is automatically populated when the lead selection of the parent changes. This is because the context framework would call some pre-defined methods to call and populate the child records. For example the CMI methods like "*getRelatedModelObjects*" would automatically be fired through the context.

Use case Scenario

Let's consider a use case scenario where in we have a Customer and Order. We will talk about 2 different scenarios. One is the "Typed" scenario where each model class is itself represented as a class. The other one is the "Generic" Scenario where there is one class that represents any model class and is constructed dynamically at runtime. Similarly there is a "*TypedModel*" and a "*GenericModel*". The CMI interfaces ICMIModel and ICMIGenericModelClass are also shown. We restrain from discussing all methods and all interfaces here. The below scenario, however, shows a *Typed* model/modelclass extending a generic implementation. This makes it possible to be used both as a Generic and Typed model.

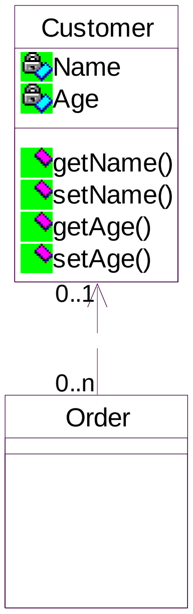


Dig 1. Typed and Generic Model with ICMIModel



Dig 2. Typed and Generic ModelClass with ICMIModelClass

The below diagram shows how Order has a aggregation relationship with Customer



Dig 3. Order related to customer by aggregation

How does it help in Generic UI Programming?

Let's look at how Generic UI Programming is facilitated from Web Dynpro. Model implementations like the Adaptive RFC and the Adaptive Web Service model all implement the CMI to make use of the powerful capabilities of Web Dynpro.

Let's look at some examples to understand this.

Consider an example for say a model class "Customer" constructed in the generic implementation. This generic class (*GenericModelClass*) would implement the interface "*com.sap.tc.cmi.ICMIModelClassExecutable*" .The "*ICMIModelClassExecutable*" in turn extends the "*com.sap.tc.cmi.ICMIModelClass*". Let's look at this conceptually.

The entry to the model is generally from the "*ICMIModel*" implementation. The model in a generic way can usually be read from a "*WDModelFactory*". For example

```
IWDModel model = WDModelFactory.getModelInstance(Model.class,
APPLICATION_SCOPE);
```

The above snipped would return an instance of the model in a generic way.

```
public class Model implements ICMIModel{

public ICMIModelClass createModelObject(String modelClassName){

//Create a ICMIModelClass type and return

return (ICMIModelClass)modelclass;

}
```

At the application level the user may just need to call an object of the model and call "*createModelObject*" to create the instances of the model class - Customer. See below for a way the model classes can be created once you have the model instance.

```
ICMIModel mymodel  = (ICMIModel)new Model();

ICMIModelClass customer = 
(ICMIModelClass)mymodel.createModelObject("Customer");
```

As you can see with this you can plug in any model that you want to use in your application anytime, maybe even your own model implementation. This is because your application code is independent of the model implementation. It's only your model and model Classes that implements the interfaces ("*Model*" in this example). This will be the only thing that might have to be changed. The next most important ICMI interface is the ICMIGenericModelClass interface. Refer to the code below which implements this interface. So hence it is possible to replace the model implementation in the application with little or no changes. This is because most of the coding in your application would be generic.

```
public class GenericModelClass implements
com.sap.tc.cmi.ICMIModelClassExecutable {

public ICMIModelClass getRelatedModelObject(String targetRoleName) {

///Write the implementation and Get the relatedmodelclass for lets say
“Order”

…return relatedClassesMap(targetRoleName);

}

public void setRelatedModelObject(String targetRoleName, ICMIModelClass o) {

// Write a implementation to set the related object

relatedClassesMap.put(targetRoleName,o)

}

public boolean addRelatedModelObject(String targetRoleName, ICMIModelClass
o) {

//Add o to the relation corresponding to targetRoleName

}

public boolean removeRelatedModelObject(String targetRoleName,ICMIModelClass
o) {

//Remove the o from the targetrolename

}

public Object getAttributeValue(String name) {

return attributesmap.get(name);

}

public void setAttributeValue(String name,Object val) {

attributesmap.put(name,val);

}

}
```

The specific model classes can themselves be created from the GenericModelClass as shown below.

The implementations for each of the methods need to have the handling to do the required operations. Like for example, you can set relationships (references at the data level, not metadata level) between this modelclass customer and order for example with the “*setRelatedModelObject*”. The “*getRelatedModelObject*” call would return the relating object. If the cardinality is 0..n or 1..n, then you would have the plurals with a suffix “s”. These would be “*setRelatedModelObjects*” and “*getRelatedModelObjects*” respectively. Model Class properties of the model class can be accessed with getAttributeValue and setAttributeValue methods. For actual details on the methods for all methods please refer to the CMI Documentation.

So as you can observe it's possible to work with all these model/modelclass objects in a generic way.

```
ICMIModelClass customer =
(ICMIModelClass)mymodel.createModelObject("Customer");

//Assuming we have an ord of type ICMIGenericModelClass

customer.setRelatedModelObject("Order",ord);

ICMIModelClass ord = (ICMIModelClass)mclass.getRelatedModelObject("Order");

customer.setAttributeValue("Name","ABC");
```

Helpful in Dynamic Programming

In addition as I mentioned earlier, CMI facilitates dynamic programming. With the Info metadata interfaces of CMI the application can access the metadata at runtime and entirely construct the context and the UI at runtime dynamically. All Interfaces which have the name "Info" stand for metadata at runtime.

Look at the example below. This tries to construct a context node Info entirely at runtime using the metadata interfaces from CMI. Look at the below interfaces. Between every two model classes we have roles. These roles are Source Roles and Target Roles. So between two model classes lets say "Customer" and "Order" we have a source role "CustSource" and target Role "OrderTarget". If the metadata interfaces are implemented then "cust. iterateSourceRoleInfos()" should provide an iterator reference to "CustSource" and the "getOtherRoleInfo()" should yield "OrderTarget", which is a target role. As you can see in the code below you can see that this can help construct a context node entirely at runtime.

```
    for(Iterator r = cust.iterateSourceRoleInfos(); r.hasNext())
    {
    ICMIRelationRoleInfo sourceRole
        = (ICMIRelationRoleInfo)r.next();

    ICMIRelationRoleInfo targetRole = sourceRole.getOtherRoleInfo();

    ICMIModelClassInfo targetMClass = targetRole.getModelClassInfo()
    CMICardinality c = targetMClass.getCardinality();
    //Assume you have a reference to Customer's node Info , you can now create
    // a Node Info for order dynamically.
    IWDNodeInfo orderNodeInfo = custNodeInfo.addChild
      (
        targetMClass.getName(),
        ICMIGenericModelClass.class,
        false,    false, c.isMultiple(), true, c.isMultiple(),  true, null,
    null,  null);
```

In this way a child node can be dynamically created for a node "Customer" by capitalizing on the ICMI Info Classes. Remember you need to provide your implementations for the Info Classes like

“*ICMIModelClassInfo*”, “*ICMIRelationInfo*” and “*ICMIModelClassPropertyInfo*”. For details about the actual interfaces and their methods please refer to the CMI documentation.

The entry point to reading the metadata generally happens through the model. Reading the metadata of a model can be done in the following way

Modelclassinfos can be accessed in the following two ways

Through the model Info like this

```
ICMIModelInfo modelInfo = model.associatedModelInfo();
ICMIModelClassInfo modelClassInfo =
modelInfo.associatedModelClassInfo(modelclassname);
```

Through the model class instance

```
Customer cust = new Customer();
ICMIModelClassInfo mClassInfo = cust.associatedModelClassInfo();
```

Hence as you can see with the dynamic metadata you can create contexts, also use them to generate tables with columns, bind them to the context etc all dynamically.

In the actual scenario in a dynamically programmed application, you begin with an empty context and an empty view; you then use the model metadata implementation to read the metadata and then use this metadata to build a dynamic context. You also use this to build the UI say a Web Dynpro table with the metadata. It’s then possible to bind the created context to the table and run the application. Hence everything in the application is created dynamically at runtime.

Reduced application coding due to Web Dynpro models and the context?

Web Dynpro helps reduce application coding to a large extent because you get so many functionalities for free. When you import a model such as the Adaptive RFC model the “*supplyingRelationRole*” is setup between every relating model classes (when these relating contexts are bound to UI elements like table in the application). This is generated for the typed model classes created in the IDE.

With implementation of CMI, the context sees to it that all related model objects are automatically updated at runtime based on the relations. The effect being that when ever the lead selection changes for a parent the corresponding child records corresponding to the parent automatically change without any user code. This is because of the call to the CMI interface “getRelatedModelObjects” within the context code. Below is the context implementation for this. If you look at the code in bold the CMI interface methods are called for this functionality to work by the Context Framework. Conceptually the “supply function” between relating nodes is called in case of “Value Nodes” .In case of the model nodes the “supply function” is the relating role between the model classes. You don’t need to understand the complete code here but the concept.

```
 protected void doSupplyElements() {

 // Get the supply relation role on this NodeInfo. NodeInfo is metadata for
// node

    ICMIRelationRoleInfo role = nodeInfo.getSupplyingRelationRole();

    if (role != null) {

      if (getParentElementInternal().model() instanceof
ICMIGenericModelClass) {

        ICMIGenericModelClass parentModelClass =
(ICMIGenericModelClass)getParentElementInternal().model();

        String rolename = nodeInfo.getSupplyingRelationRoleName();

        if (role.getCardinality().isMultiple()) {

          bind(parentModelClass.getRelatedModelObjects(rolename));

        } else {

          ICMIModelClass related =
parentModelClass.getRelatedModelObject(rolename);

          if (related != null) {

            bind(Collections.singletonList(related));
```

Let’s take another example of an update to a model class attribute.

First let’s consider the case of a Generic model class implementing the CMI. There is a customer model class mapped to a table row (dynamically mapped to the context) and the customer name needs to be updated. When the user fills the name of a customer in the row of the table the *SetAttributeValue(name, value)* method gets invoked in the Java Class corresponding to the Attribute (this is evident from the context code seen below). The actual code for the implementation of the *SetAttributeValue(name, value)* on the GenericModelClass is called. This is with respect to the GenericModelClass.

In the case of typed model Classes (those that don’t implement CMI), the context code is written in such a way that it calls all the setter methods of the attributes/modelclass properties by reflection. Hence when you update into a context attribute the method corresponding to the setter for the attribute is automatically called. The different handling for the Generic and Typed model classes are clearly evident from the code.

```
  protected void ContextFunctionXXXX(AttributeInfo attrInfo, Object value) {
    String attrName = attrInfo.getOriginName();
    if (attrInfo.getOriginName() == null) {
      super.wdpSetObject(attrInfo, value);
    } else if (reference instanceof ICMIGenericModelClass) {
      ((ICMIGenericModelClass)reference).setAttributeValue(attrName, value);
    } else {
// It is a typed model class
      try {
        Method accessor = getAccessor("set", attrInfo.getName(), new
Class[]{attrInfo.getDataType().getAssociatedClass()});
        accessor.invoke(reference, new Object[]{value});
      } catch (Exception e) {
        throw new ContextException(ContextException.CANNOT_SET_ATTRIBUTE,
```

Relations are similarly handled. The context would look for setter methods corresponding to a role between source and target and call that setter method. For example if the rolename between *Customer* and *Order* is "*Order*" it would look to call

cust.set<<Role-name>> or cust.setOrder(ord);

So if you consider "Typed Classes" you might feel that CMI needn't be implemented here. It's possible to have a model without a CMI implementation but this brings with it several restrictions and doesn't help capitalize entirely the power of Web Dynpro.

Several model implementations such as the Adaptive RFC are supported to be both "*Typed*" as well as "*Generic*". This generally calls for an architecture where the Typed Classes extend the Generic Implementations which are part of the framework. In this way, an application developer is free to use the "Typed" classes and also the "Generic Classes" to represent model classes.

Related Content

Please include at least three references to SDN documents or web pages.

CMI Documentation

https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/157262c8-0b01-0010-95aa-ae094f42acb0

Web Dynpro Models

http://help.sap.com/saphelp_nw04/helpdata/en/7a/74d43fb9490e65e10000000a114b1d/content.htm

Web Dynpro Context

http://help.sap.com/saphelp_nw04/helpdata/en/75/93d43ff9490d65e10000000a114b1d/content.htm

Acknowledgements

Thanks to Frank Weigel, Patric Ksinsik and Uwe Reeder for their help in understanding the topic.