# OTIF Usage Guide

Andras Lang
andras.lang@vanderbilt.edu

Institute for Software-Integrated Systems
Vanderbilt University
March 2004

# Table of Contents

**Table of Contents** ..... 2

**1. Architectural description** ..... 3

**2. Manager application** ..... 4

**3. Workflow paradigm and interpreter** ..... 6

**4. Generic ToolAdaptor application** ..... 10

**5. UdmCORBA API** ..... 12

**6. TA_MFC_Support API** ..... 20

**7. TA_XSLT_Support API** ..... 21

**8. Multiple-part document support** ..... 22

**9. How to create a simple ToolAdaptor application** ..... 24

**10. How to create a simple Translator application** ..... 29

**11. UdmCORBA exception handling model** ..... 30

**Appendix A.** ..... 32
**BackPlane_config.xml parameter file** ..... 32

**Appendix B.** ..... 34
**1. Example of a "*document publish*" mechanism.** ..... 34
**2. Example of a "*document fetch*" mechanism.** ..... 35

**Appendix C.** ..... 36
**Simple Translator application example** ..... 36

**Appendix D.** ..... 37
**How to create the *.cpp, .dtd and .h* files by the Udm.exe** ..... 37

**Appendix E.** ..... 38
**Troubleshooting** ..... 38

# 1. Architectural description

The main components of the OTIF system:

- Backplane application
- Manager application
- Workflow paradigm and interpreter
- Generic ToolAdaptor application
- UdmCORBA API
- ToolAdaptor_DLL API

The **Backplane** is the server application. It runs the translator applications to translate the different documents. It uses a configuration parameter file called *BackPlane_config.xml* [Appendix A.].

Backplane can be managed by the **Manager** application and by the workflow GME interpreter. One can design a tool chain in GME with the workflow paradigm (WFL) and register it in the Backplane server with the interpreter. A tool chain specifies the translators, tool adaptor types and paradigms used in the workflow tool chain.

**Generic ToolAdaptor** application is used to publish documents or subscribe to paradigms which are available in the Backplane server. The goal of the generic tooladaptor application is to prevent the users form implementing their tool adaptors. If a paradigm is available in the Backplane that paradigm can be used to fetch or publish documents.

There are three APIs to create *ToolAdaptor* [Appedix B.] or *Translator* [Appendix C.] applications. The **UdmCORBA API** contains the basic interfaces, the **TA_MFC_Support API** extends it with pre-defined MFC dialog boxes and the **TA_XSLT_Support API** extends it with XSLT-based transformation methods.

You may find a more detailed description and concepts of the OTIF framework in the *OTIF.pdf* file.

You may find a detailed description about the OTIF communication protocol in the *OTIF_Protocol_Description.pdf*.

You should be familiar with the latest UDM package and the GME2000 application to use this framework effectively. You may find detailed information about these on the ISIS website.

# 2. Manager application

After the Manager application started, one has to login to the active Backplane server with the correct password of the server. This password can be changed anytime with the Manager application. The default password is blank.

After authentication one is able to view the registered paradigms and translators, get information about the available documents and destroy a document, shutdown the Backplane server or change the authentication password.

**Paradigm:** Paradigm is a metamodel. Paradigms can be created with GME application and exported to an xml type file. These xml files can be registered in the Backplane server.

**Translator:** Translator can be any applications, which are able to read a document of a paradigm and translate it to another document of another paradigm. Translators can be created with the OTIF static library and header files. There are a couple of templates in this document which are good to start with. Translators should be placed in the translator directories of the Backplane server.

### 1. Uploading a paradigm

One can upload a paradigm into the Backplane server which can be used later by the translators and tool adaptors. It won't be registered at once. It only will be stored into the paradigms directory of the Backplane and can be used in any workflow model later.

The name and version parameters and the name of the file of the paradigm (in the *URL* field) have to be specified. The paradigm files may be in the {OTIF demo directory}/bin/paradigms directory.

There are two ways to register a paradigm:

- With the **http://** prefix one is able to download an xml type paradigm file from a specified location and register it.
- With the **local://** prefix one is able to select a local, xml type paradigm file and register it.
  A local file may be selected by the button (labeled with "…") next to the URL field.

### 2. Locked checkbox

A paradigm can be locked or unlocked. A locked paradigm could not be unregistered.

### 3. Refresh datanetworks

Get information about the available documents in the Backplane server. It lists the name, version, remarks, keepAlive parameter and the paradigm of the document.

### 4. Destroy datanetwork

One is able to destroy any kind of document in the Backplane server. The document should be selected in the *Available datanetworks* list.

### 5. Shutdown backplane

One is able to shutdown properly the Backplane application. After shutdown the manager application exits.

### 6. Change password

One is able to change the authentication password of the Backplane. Both the old and the new passwords are necessary.

# 3. Workflow paradigm and interpreter

The OTIF framework supports workflows. One is able to define a so called tool chain which may contain any kind of translators, different types of tool adaptors and connections between them. This tool chain is a specific workflow model of any system based on the OTIF framework.

The OTIF workflow modeling engine leans on

- the workflow paradigm and
- the workflow interpreter.

The **workflow paradigm** is a GME paradigm which defines the different elements and connections of the workflow.

These elements are:

- *Backplane* denotes the Backplane server
- *Translator* denotes any kind of translator application
- *ToolAdaptor* denotes any kind of tool adaptor application
- *Paradigm* denotes any kind of paradigm used by the translator and tool adaptor applications
- *Link* between any ToolAdaptor or Translator denotes the connections between the tool adaptor and translator applications.

One is able to define and visualize any kind of OTIF workflow model in GME with the registered workflow paradigm.

The workflow models based on the workflow paradigm can be interpreted by the **workflow interpreter**.

The interpreter is a so called GME component which is able to process a visualized workflow model and send this XML based configuration information [Appendix A.] to the Backplane server.
When a new configuration arrives to the Backplane, it resets itself and loads it.

*Note: For more information about registering the workflow paradigm and interpreter, please read the readme_core.txt file.*

Let's look at through a creation of a simple workflow model how one is able to design a workflow model.

*Starting the modeling environment*

- Start the GME application and create a new project with the WFM (workflow modeling) paradigm.

- Insert a new diagram under the root object of the workflow model. Everything which is visualized and defined on this diagram is a part of our new model.

- Open the newly created diagram and let's start to design a new model.

*Designing our new workflow model*

- First of all put a *Backplane* object onto the diagram. As we can see there are a few attributes of the *Backplane* object. Let's go through them:
  - *name* denotes the name of the running Backplane server.
    For example write "TEST" here.
  - *version* denotes the version of the running Backplane server.
    For example write "1.0" here.
  - *umlParadigm* denotes the name of the xml file which stores the UML paradigm (Note: in the default installation of the OTIF framework it is the "uml.xml" file).
    For example write "uml.xml" here.
  - *logLevel* denotes the level of the logging procedure in the running Backplane server. For more information [Appendix A.].
    For example write "3" here.
  - *onlyOneManager* specifies that only one or multiple Manager application can connect to the Backplane at one time.
    For example just leave the default.
  - *translatorDocument* specifies the keepAlive value for the documents published by any translator applications to the Backplane server.
    For example write "300" here.

- Let's start with the beginning of our workflow model so put a *ToolAdaptor* object onto the diagram. It has only the type attribute.

  You can distinguish the different tool adaptor applications through the type attribute. Every tool adaptor application has to log as a specific type in the Backplane server.

  For example write "MATLAB_PUBLISHER" here.

  Every tool adaptor which logged in the Backplane server with this type will act in our workflow model like this *ToolAdaptor* object.

- Let's put a transformation mechanism into our workflow model. So put a *Translator* object onto the diagram. It has three attributes. Let's go through them:
  - *name* denotes the name of the translator application in the OTIF environment.
    For example write "Matlab2ECSL" here.
  - *version* denotes the version of the translator application in the OTIF environment.
    For example write "1.0" here.
  - *translatorAction* identifies the name of the file of the translator application.
    For example write "translator_matlab2ecsl.exe" here.

    The translator application files should be in the {OTIF demo directory}/bin/translators or in the {OTIF demo directory}/bin/xsl_translators. It depends on the type of the translator.

    There are two types of translator:
    1. **EXE** type translators are applications developed by the user with the OTIF libraries.
    2. **XSL** type translators are standard xsl files created by the user. These xsl files are used in the translation process by the generic xslt translator application.

- Let's connect these objects together. Switch to "Connect Mode" in GME. Select the ToolAdaptor object first as the source of the *Link* object than select the Translator as the destination of the *Link* object.

  In the workflow model these *Links* object denotes the specific paradigms which are used in the communication between the tool adaptor and translator applications.

- So define the paradigm of this *Link* object. To do this the first step is to put a *Paradigm* object onto the diagram. It has three attributes:
    - *paradigmName* denotes the name of the paradigm in the OTIF environment.
      For example write "Matlab" here.
    - *paradigmVersion* denotes the version of the paradigm in the OTIF environment.
      For example write "1.0" here.
    - *paradigmFile* identifies the name of the xml file of the paradigm.
      For example write "matlab.xml" here.

  The paradigm files should be in the {OTIF demo directory}/bin/paradigms directory.

- The second step is to tell the *Link* object that this is your paradigm and the *ToolAdaptor* and *Translator* object will communicate through this paradigm.
  So switch "Set Mode" in GME and click on the *Paradigm* object with the right mouse button and select the *Link* object by clicking on it.

*So what have we done?*

We have just created a workflow model, in which any kind of tool adaptor application logged in as a "MATLAB_PUBLISHER" type can publish a document. The paradigm of this document is "Matlab 1.0" defined by matlab.xml paradigm file. Every documents based on this paradigm published to the Backplane server will be translated by the "Matlab2ECSL" translator.

Naturally at this status we can not get the result of this tool chain until we define a new type of *ToolAdaptor* object and use it as an output of the translator. This can be done the same as in the case of defining of the previous *ToolAdaptor* object.

*Interpret our new workflow model*

- Let's use our newly created workflow model so interpret it. Click on the interpret icon. A dialog box will show up. By the default installation you may just push the OK button and send the new configuration to the Backplane server.

    - "Backplane password" option lets you supply the correct password of the Backplane server if it is not blank.

# 4. Generic ToolAdaptor application

The Generic ToolAdaptor application can be used as any kind of ToolAdaptor type application. One has to choose a specific type of ToolAdaptor after starting the application.

One is able to publish or fetch any kind of documents which are supported by the chosen ToolAdaptor type.

The available backends are the memory, XML and GME to fetch (save) or publish (load) a document.

One has three options after choosing a specific type of ToolAdaptor:

1. **Publish**

   One has to specify the paradigm (metamodel) of the document. The supported paradigms are defined by the type of the ToolAdaptor.

   One has to specify the name, version, remarks and keepAlive parameters of the document. After specifying the correct parameters one has to select the document file from the local disk.

   The application gives the opportunity to transform an XML file with an XSLT script right before sending it by clicking on the XSLT transform checkbox. It will ask for the specific XSLT file to do the transformation.

   It is useful in that case when a non-UDM conform XML type document should be published into the Backplane, so the document can be transformed into a UDM conform XML type document with a specific XSLT script.

   Note: only XML files can be transformed with the XSLT preprocessing.

   One may attach unstructured (not UDM comfort) documents (files) to any documents.

   One may update a previously published and still existing document in the Backplane server by checking the corresponding checkbox. It will give a list of documents which can be updated.

   The updated document will be translated by the translators as the former document was. Every ToolAdaptor will get a new notification if an updated document will be available.

## 2. Subscribe

One has to specify a paradigm (metamodel). The supported paradigms are defined by the type of the ToolAdaptor.

After the subscription one will get a notification when a document of that paradigm is available.

If a document is available one is able to fetch and save the document on the local disk. Both of the memory, xml and gme backend can be used to save the document.

If the gme backend is used to save the document, one has to specify a metamodel of the mga layer. This metamodel will be used to save the document.

The application gives the opportunity to transform the fetched XML file with an XSLT script. One has to select the XSLT file to do the XSLT postprocessing.

It is useful in that case when a specific, non-UDM XML type document should be fetched from the Backplane, so after fetching the document can be transformed into any kind of XML type document with a specific XSLT script.

Note: only XML files can be transformed with the XSLT postprocessing.

## 3. Unsubscribe

One has to specify a paradigm (metamodel) among from the subscribed paradigms. After the unsubscription one will not get any notification when a document of that paradigm is available.

# 5. UdmCORBA API

This API contains the basic interfaces to implement a ToolAdaptor or a Translator application.

The API provides 3 class interfaces:

- **`class CORBADataNetwork : public Udm::SmartDataNetwork`**

  - `static void Initialize(const char *paradigm_name, const char *paradigm_version);`

    This static method should only be used if one wants to use CORBADataNetwok alone in the application without using the ToolAdaptor or the Translator classes. It initializes the CORBADataNetwork with specifying the name and version of the remote paradigm. After initialization one is able to create a new CORBADataNetwork and use it in the same way as the SmartDataNetwork.

    The difference between the two datanetworks is that SmartDataNetwork works with “local” paradigm and CORBADataNetwork works with “remote” paradigm.

    *Note: CORBADataNetwork should not be created without initialization. Every time a CORBADataNetwork is created with a different remote paradigm the initialization should be processed with the correct name of the paradigm.*

  - `string getParadigmName();`

    This method returns the name of the paradigm of the datanetwork.

  - `string getParadigmVersion();`

    This method returns the version of the paradigm of the datanetwork.

  - `long publishDocument(const char *docName, const char *docVersion, const char *remarks, long keepAlive, const long previous_docId=NO_PREVIOUS_DOCUMENT);`

    This method publishes the CORBADatanetwork to the Backplane server. One has to specify the document name, version, remarks and keepALive parameter. The keepAlive parameter specifies until how many seconds will be kept the document in the server.

    The previous_docId should be NO_PREVIOUS_DOCUMENT whenever a completely new document is published, otherwise it is the backplane-generated ID of an existing, previously published document. If it is a valid ID the document will be updated with the new information.

    The method returns the backplane-generated ID of the published document.

- `void fetchDocument(long docID, const long offset=0);`

  This method fetches a document from the server. One has to specify the document ID which identifies the document in the server.

- `void release();`

  This method releases all memory usage of a CORBADatanetwork.

- `long GetDatanetworkObjectID(long unique_id);`
- `long GetDatanetworkObjectID(Udm::Object &obj);`

  These methods return an ID of an object in the CORBADatanetwork. The first one uses the unique_id parameter of an Udm::Object.

- `void SetDatanetworkObjectID(long unique_id, long dn_id);`
- `void SetDatanetworkObjectID(Udm::Object &obj, long dn_id);`

  These methods set the ID of an object in the CORBADatanetwork. The first one uses the unique_id parameter of an Udm::Object.

- `Udm::Object GetObjectById(long dn_id);`

  This method returns the corresponding udm object.

- `long GetObjectHostInfo(Udm::Object &obj);`

  This method returns the IP address of an object in the CORBADatanetwork. The IP address of an object is the IP address of a host, which created that object.

- `long GetObjectProcessInfo(Udm::Object &obj);`

  This method returns the ID of a process, which created the object specified by the input parameter.

- `long GetObjectThreadInfo(Udm::Object &obj);`

  This method returns the ID of a thread, which created the object specified by the input parameter.

- `void attachDocument(const long docId, const char *paradigm_name, const char *paradigm_version, const long offset=0);`

  This method attaches a structured element to the publishable document. It will not process the structured document, just add it to the existing document.

- `void addAdditionalFile(const char *filename, const char *directory="");`

  This method addes an unstructured element/file to the publishable document.

- `void getAdditionalFileList(multimap<string, string> &additional_files);`

  This method returns the list of the additional elements/files of the document. The first string is the real name, the second string is the physical name of the file (the additional files are stored in temporary files after the fetching process).

- `string getPhysicalFilename(const char *filename);`

  This method returns the physical name of the additional file. The filename is the real name of the file.

- `void copyAdditionalFilesFrom(CORBADataNetwork *source);`

  This method copies the additional files from the source document.

- **`class ToolAdaptor`**

  - `ToolAdaptor(int argc, char **argv);`

    This method creates the “UML paradigm” type ToolAdaptor. This ToolAdaptor works with the UML paradigm.

  - `ToolAdaptor(const char *tooladaptor_type, int argc, char **argv);`

    One has to specify the type of the ToolAdaptor. This type has to be registered in the Backplane server.

  - `string getToolAdaptorType();`

    This method returns the type of the ToolAdaptor.

  - `void getWorkingParadigms(map<long, PARADIGM_STRUCT> &paradigms);`

    This method returns the supported paradigms of the ToolAdaptor.

  - `void getTAConfig(list<CONFIG> &config);`

    This method returns the list of the paradigms of the publishable document. An item of the list identifies the paradigms of a publishable document.

  - `void getTASubscribable(list<CONFIG> &config);`

    This method returns the list of the subscribable paradigms of the ToolAdaptor.

  - `vector<DOCUMENT_STRUCT> getDocuments();`

    This method returns the vector of the iformation of the documents existing in the Backplane server.

  - `CORBADataNetwork* getDatanetwork(const Udm::UdmDiagram &metainfo);`

    One has to specify the diagram object of the paradigm. This method returns a pointer of the CORBADatanetwork. It can be used only if the specific type of ToolAdaptor support only one paradigm.

- `CORBADataNetwork* getDatanetwork(const Udm::UdmDiagram &metainfo, const char *paradigm_name, const char *paradigm_version);`

  One has to specify the diagram object of the paradigm, the name and version of the paradigm. This method returns a pointer of the CORBADatanetwork. It should be used if the specific type of ToolAdaptor support more than one paradigm.

- `bool isRegisteredDatanetwork(const char *paradigm_name, const char *paradigm_version);`

  This method returns true if a datanetwork (corresponding to a paradigm) is not processed yet.

- `void subscribe(const Udm::UdmDiagram &metainfo);`

  One has to specify the diagram object of the paradigm. This method subscribes to a specific paradigm in the Backplane server, so the server will notify the ToolAdaptor if a document in this paradigm arrives. This method can be used only if the specific type of ToolAdaptor support only one paradigm.

- `void subscribe(const Udm::UdmDiagram &metainfo, const char *paradigm_name, const char *paradigm_version);`

  One has to specify the diagram object of the paradigm, the name and version of the paradigm. This method subscribes to a specific paradigm in the Backplane server, so the server will notify the ToolAdaptor if a document in this paradigm arrives. It should be used if the specific type of ToolAdaptor support more than one paradigm.

- `long publishMultipleDocuments(list<CORBADataNetwork*> dns, const char *docName, const char *docVersion, const char *remarks, long keepAlive, const long previous_docId=NO_PREVIOUS_DOCUMENT);`

  One is able to publish a multiple-part document to the Backplane with this method. All parts of the document should be inserted into the CORBADataNetwork* list. The document name, version, remarks and keepALive parameter should be spcified. The keepAlive parameter specifies until how many seconds will be kept the multiple-part document in the server.

  The previous_docId should be NO_PREVIOUS_DOCUMENT whenever a completely new document is published, otherwise it is the backplane-generated ID of an existing, previously published document. If it is a valid ID the document will be updated with the new information.

  The method returns the backplane-generated ID of the published document.

- `void process();`

  This method checks only one time that a document is arrived to the ToolAdaptor.

- `void run();`

  This method checks in an infinite loop that a document is arrived to the ToolAdaptor until it terminates.

- `void terminate();`

  This method terminates the ToolAdaptor run method.

- `void logout();`

  This method logs out the ToolAdaptor from the Backplane server, so the ToolAdaptor will not get any notifications.

- `string getHostInfoString();`

  This method returns the IP address of the host in “A.B.C.D” format.

- `long getHostInfoLong();`

  This method returns the IP address of the host in long type format.

- `long getProcessInfo();`

  This method returns the ID of the process.

- `long getThreadInfo();`

  This method returns the ID of the thread.

- `virtual void getPublishStatus(long offset, long length);`

  One may implement this method to get information about the publishing process. This method called when the offset is incremented.

- `virtual void getFetchStatus(long offset, long length);`

  One may implement this method to get information about the fetching process. This method called when the offset is incremented.

- `virtual void getPublishStatus(long percent);`

  One may implement this method to get information about the publishing process. This method called when the percent is incremented.

- `virtual void getFetchStatus(long percent);`

  One may implement this method to get information about the fetching process. This method called when the percent is incremented.

- `virtual void notify(const char *paradigm_name, const char *paradigm_version, const char *name, const char *version, const char *remarks, long docId, long keepAlive)=0;`

  This abstract method should be implemented by the actual ToolAdaptor. It is invoked automatically when a documents arrives to the ToolAdaptor. The paradigm_name and paradigm_version store the name and version of the paradigm of the arrived document.

- `virtual void changedExistingDocument(const char *name, const char *version, const char *remarks, long docId, long keepAlive);`

  This method should be overridden by the actual ToolAdaptor to get notification if an available datanetwork is changed. It is invoked automatically when a parameter of an available document is changed. If the available document is destroyed the keepAlive value will be -1.

  *Note: The following ToolAdaptor interface definitions are only needed if one wants to create generic ToolAdaptor or Translator applications.*

- `Uml::Diagram remoteFetchMyParadigm(const char *name, const char *version);`

  This method fetch a remote paradigm from the Backplane server. One has to specify the name and version parameter of the paradigm.

- `static void getBackplaneToolAdaptorTypes(list<string> &types, int argc, char **argv);`

  This method returns all the types of the ToolAdaptors which are available in the Backplane server.

- `static void getBackplaneParadigms(map<long, PARADIGM_STRUCT> &paradigms, int argc, char **argv);`

  This method returns all the paradigm which are available in the Backplane server.

- `Uml::Class getDocumentRootobjectMeta(long docId, const Uml::Diagram &meta_diagram);`

  This method fetch the name of the meta of the rootobject (udm object) of the corresponding remote document in the Backplane server. The docId identifies the remote document.

- **`class Translator`**

  - `CORBADataNetwork *fromDN;`

    This variable points to the input CORBADatanetwork, which was created by the constructor of the Translator. Note: this variable may not be used if a multiple-part document arrives to the Translator.

  - `CORBADataNetwork *toDN;`

    This variable points to the output CORBADatanetwork, which was created by the constructor of the Translator.

- `Translator(const Udm::UdmDiagram &from_metainfo, const char *from_name, const char *from_version, const Uml::Class &from_rootclass, const Udm::UdmDiagram &to_metainfo, const char *to_name, const char *to_version, const Uml::Class &to_rootclass, int argc, char **argv);`

  One has to specify the diagram object, the name and version parameter and the root object of the input and the output paradigm. These paradigms have to be registered in the Backplane server.

- `void addFromDN(const Udm::UdmDiagram &from_metainfo, const char *from_name, const char *from_version, const Uml::Class &from_rootclass);`

  This method add a specific CORBADataNetwork to the working datanetwork list of the Translator. It may be used when the Translator is working with more than two (an input and an output) paradigm.

- `CORBADataNetwork *get_fromDN(const char *name, const char *version);`

  This method returns a pointer to a CORBADataNetwork specified by its name and version parameter. If the Translator is working with a multiple-part document one can get the multiple parts of the document with this method.

- `CORBADataNetwork *get_fromDN(const char *name, const char *version, const long offset=0);`

  This method returns a pointer to a CORBADataNetwork specified by its name and version parameter. The offset parameter specifies exactly the CORBADataNetwork with the offset value if multiple parts have the same paradigm.

- `CORBADataNetwork *get_fromDN();`

  This method returns a pointer to the input CORBADatanetwork, which was created by the constructor of the Translator.

- `CORBADataNetwork *get_toDN();`

  This method returns a pointer to the output CORBADatanetwork, which was created by the constructor of the Translator.

- `void run();`

  This method checks in an infinite loop that a document is arrived to the Translator.

- `void publish(char *doc_name, char *doc_version, char *doc_remarks);`

  This method publishes the CORBADatanetwork specified by the toDN variable.

- `void publish(CORBADataNetwork *dn, const char *doc_name, const char *doc_version, const char *doc_remarks);`

  This method publishes the CORBADatanetwork specified by the dn input parameter.

- `void publishMultipleDocuments(list<CORBADataNetwork*> dns, const char *doc_name, const char *doc_version, const char *doc_remarks);`

  This method publishes a list of CORBADatanetworks specified by the dns input parameter.

- `virtual void notify(const char *name, const char *version, const char *remarks, const unsigned long numOfDocs)=0;`

  This abstract method should be implemented by the actual Translator. It is invoked automatically when a document arrives to the Translator. If it is a multiple document, the numOfDocs parameter will contain the number of the structured elements in the document.

- `Uml::Class getDocumentRootobjectMeta(long docId, const Uml::Diagram &meta_diagram);`

  This method fetch the name of the meta of the rootobject (udm object) of the corresponding remote document in the Backplane server. The docId identifies the remote document.

# 6. TA_MFC_Support API

It is a convenient interface to use the OTIF built-in MFC dialog boxes with the UdmCORBA API.

This API gives the following methods:

- `int showFetchDlg(const char *label);`

  This method shows the OTIF built-in fetch dialog box. The return value can be the MFC defined IDOK or IDCANCEL, the return value of the dialog box.

- `void addStringFetchDlg(const char *text);`

  This method adds a string to the listbox of the OTIF built-in fetch dialog box.

- `void clearFetchDlg();`

  This method clears the listbox of the OTIF built-in fetch dialog box.

- `int showPublishDlg(CString *name, CString *version, CString *remarks, long *keepAlive, CString *filename, const char* fileextension="*");`

  This method shows the OTIF built-in publish dialog box. The parameters can be used as the return values of the fields of the dialog box. The return value can be the MFC defined IDOK or IDCANCEL, the return value of the dialog box.

- `int showPublishAttachmentDlg(CString *name, CString *version, CString *remarks, long *keepAlive, CString *filename, std::map<CString, CString> *attachments, const char* fileextension="*");`

  This method shows the OTIF built-in publish with attachment support dialog box. The parameters can be used as the return values of the fields of the dialog box. The return value can be the MFC defined IDOK or IDCANCEL, the return value of the dialog box.

- `int showDatanetworkArrivedDlg(const char *name, const char *version, const char *remarks, long keepAlive, CString *filename, const char* fileextension="*");`

  This method shows the OTIF built-in datanetwork arrived dialog box. The parameters are displayed in the dialog box. The filename parameter can be used as a return value of the filename field of the dialog box. The return value can be the MFC defined IDOK or IDCANCEL, the return value of the dialog box.

# 7. TA_XSLT_Support API

From the 1.2.1 version OTIF gives XSLT transformation support. The XSLT transformation support may be used in those cases when a non-UDM specific XML file wants to be published or after the fetching process one wants to automatically convert the fetched XML file with XSLT transformation to another XML file.

This API gives the following methods:

- `XSLTransformer();`

  This method creates the class which gives the XSLT support.

- `int executeTransform(const char *input, const char *xslt, const char *output, const bool generate_dtd, const char *dtd_file, const char *rootclass);`

  This method does the transformation on a specific XML file. The input parameter identifies the name of the input XML file. The xslt parameter identifies the name of the file which stores the XSLT script. The output parameter identifies the name of the output XML file.

  The generate_dtd parameter should be true if one wants to generate a UDM specific DTD file for the output XML file.

  If one generates a DTD file one has to specify the name of the DTD file and the name of the root class of the paradigm of the output XML file.

- `void GenerateDTDfile(const Uml::Diagram &diagram, const char *dtd_file);`

  This method generates a DTD file from a specific paradigm. One has to specify the diagram object of the paradigm and the name of the DTD file.

Note: The methods are in the OTIF_XSLT namespace.

Here is a simple usage example:

```
      ...

      // fetch and save the available document in a temporary file
      dn->CreateNew("temp_test.xml", "test", Root::meta, Udm::CHANGES_PERSIST_ALWAYS);
      dn->fetchDocument(docId);
      dn->CloseWithUpdate();

      try {
         // create the XSLT translation support object
         OTIF_XSLT::XSLTransformer xslt;
         // do the XSLT transformation which is specified by the test.xsl file
         int result = xslt.executeTransform("temp_test.xml", "test.xsl", "test.xml",
false, "", "");

      // catch any XSLT transformation exceptions
      } catch (OTIF_XSLT::xslt_exception &e) {
         cerr << "XSLT translation error: " << e.what() << endl;
      }

      // after the transformation delete the temporary file
      DeleteFile("temp_test.xml");

      ...
```

XSLT transformation example

# 8. Multiple-part document support

From the 1.2.1 version OTIF gives multiple-part document support. Multiple-part document is a document which contains two or more single documents.

There are cases when a Translator does not just get a document and creates another one from it. For example, a Translator may merge two documents into one or may update a document with another one.

In these cases it is necessary for the Translator to fetch multiple documents and for the ToolAdaptors to publish multiple documents.

In the OTIF environment the Translator can fetch a document which can be a single document or a document with multiple parts. These multiple parts stand by themselves as different single documents.

### Multiple-part documents in the workflows

Let's see how multiple-part documents can be defined in a workflow. As we see there can be a connection between a ToolAdaptor type and a Translator which defines that this type of ToolAdaptor can publish a document to this Translator. The paradigm of the document can be defined by the containment in one of the set of a defined paradigm.

If we create two links between a type of ToolAdaptor and a Translator, it means that this type of ToolAdaptor will publish a multiple-part document which contains two single documents. The two links will specify the paradigms of the two single documents.

*Note: The generic tools of the OTIF framework currently do not support to define a ToolAdaptor type which may publish more multiple-part documents or a multiple-part document and single documents, too.*

### Multiple-part documents in the Translators

Here is a simple usage example:

```
   ...

   // Define the myTranslator class in the original way
   class myTranslator : public UdmCORBA::Translator {

   ...

   public:

      void notify(const char *name, const char *version, const char *remarks, const
unsigned long numOfDocs) {

         // Create the root object of the document with the ESCM_UDM v1.0 paradigm
         ESCM_UDM::ESCM top = ESCM_UDM::ESCM::Cast(get_fromDN("ESCM_UDM", "1.0")-
>GetRootObject());

         // Create the root object of the document with the ESML v1.0 paradigm
         ESML::RootFolder rootFdr = ESML::RootFolder::Cast(get_fromDN("ESML", "1.0")-
>GetRootObject());

         // Update the document with the ESML v1.0 paradigm
         ...

         // Publish the updated document to the Backplane
         publish(get_fromDN("ESML", "1.0"), name, version, "after the ESML update");

      }

   ...
```

Multiple-part document in a Translator example

### Multiple-part documents in the ToolAdaptors

Here is a simple usage example:

```
   ...

   // Create the ToolAdaptor object with the type of ESML_UPDATER
   myToolAdaptor ta("ESML_UPDATER", argc, argv);

   // Create the two datanetworks which we want to publish later
   UdmCORBA::CORBADataNetwork *dn = ta.getDatanetwork(ESCM_UDM::diagram, "ESCM_UDM",
"1.0");
   UdmCORBA::CORBADataNetwork *dn2 = ta.getDatanetwork(ESML::diagram, "ESML", "1.0");

   // Open the two files and load into the two datanetworks
   dn->OpenExisting("escm_udm_sample.xml", "ESCM_UDM");
   dn2->OpenExisting("esml_sample.mga", "ESML");

   // Create a list of CORBADataNetwork*
   list<UdmCORBA::CORBADataNetwork*> docs;

   // Put the two datanetwork into the list
   docs.push_back(dn);
   docs.push_back(dn2);

   // Publish a multiple-part document which contains the two datanetwork
   ta.publishMultipleDocuments(docs, "esml doc updated", "1.0", "", 300);

   ...
```

Multiple-part document in a ToolAdaptor example

# 9. How to create a simple ToolAdaptor application

Let's create a simple ToolAdaptor application with MFC support, which will publish a document in Matlab 1.0 paradigm and fetch documents in ECSL 1.0 paradigm.

We suppose that there is a ToolAdaptor type called "MATLAB_PUBLISHER" which supports the Matlab 1.0 paradigm and a ToolAdaptor type called "ECSL_RECEIVER" which supports the ECSL 1.0 paradigm and there is a translator application which does the translation between those paradigms.

1. Create a new *MFC AppWizard(exe)* project. The project type may be dialog based.

2. Add the following directories to the *Additional include directories*:
   - {OTIF demo directory}/Include/ToolAdaptor
   - {OTIF demo directory}/Include/UdmCORBA
   - {UDM directory}/Include/Udm
   - {3rd Party directory}/Include/STL

3. Add the following files to the *Object/library modules*:
   - TA_MFC_Support.lib
   - ace.lib
   - TAO.lib
   - TAO_PortableServer.lib
   - UdmBase.lib
   - UdmCORBA.lib
   - UdmUtil.lib
   - xerces-c_2.lib

   Optional (it depends on which backend one wants to use in the project):
   - UdmDOM.lib
   - UdmGME.lib

   Important: The name of the files is different in Debug compilation!
   Important: Do not use precompiled headers!

4. Add the following directories to the *Additional library path*:
   - {OTIF demo directory}/Lib/ToolAdaptor
   - {OTIF demo directory}/Lib/UdmCORBA
   - {OTIF demo directory}/Lib/tao
   - {UDM directory}/Lib/Udm
   - {3rd Party directory}/Lib/xerces

5. Put the (e.g. matlab and ecsl) paradigm *.cpp, .h* and *.dtd* files into the project directory and add the *.cpp* to the source files.
   **Udm creates these files for you. [Appendix D.]**

   *Note: Please pay attention to create these generated files as the Appendix D. says. Without it your application will not work correctly.*

6. At this point we are ready to put our source code in the project. Only the *{project_name}.cpp* file will be modified.

Put the following *includes* after the *stdafx.h include*:

```
#include " TA_MFC_Support.h"
#include "{paradigm_name}.h"
#include "UdmCORBA.h"

// These are only optional
#include "UdmDom.h"
#include "UdmGME.h"
```

Put the following code in the `BOOL C{project_name}App::InitInstance()` method:

```
	try {


		// ****************
		// publish Document
		// ****************
		// define some variables
		CString name, version, remarks, filename;
		long keepAlive;

		// show the publish dialog, this dialog is in the ToolAdaptor_DLL library
		// the values from the dialog will be in the corresponding variables
		int result = showPublishDlg(&name, &version, &remarks, &keepAlive,
&filename);

		// if the OK button was pushed we will send a document
		if (result==IDOK) {
			// instantiate a tooladaptor, login with the specific type of
			// ToolAdaptor into the Backplane
			ta = new myToolAdaptor("MATLAB_PUBLISHER", __argc, __argv);

			// Get the datanetwork from the tooladaptor
			// (here the matlab paradigm)
			// the matlab::diagram variable is defined in the
			// {paradigm_name}.h (here, matlab.h)
			UdmCORBA::CORBADataNetwork *dn =
				ta->getDatanetwork(matlab::diagram);

			// open the file and load in the datanetwork
			dn->OpenExisting((LPCSTR)filename, "matlab");

			// publish this datanetwork to the server
			dn->publishDocument((LPCSTR)name, (LPCSTR)version,
(LPCSTR)remarks, keepAlive);

			// close the tooladaptor session
			ta.logout();
		}



		// ****************
		// fetch Document
		// ****************
		// begin a new thread, this thread is implemented by the user
		working_thread = AfxBeginThread(myWorkingThread, NULL);

		// show the fetch dialog, this dialog is in the ToolAdaptor_DLL library
		showFetchDlg();

		// close the tooladaptor session
		ta->logout();

	} catch (udm_exception &e) {
		MessageBox(NULL, e.what(), "Error", MB_OK);
	} catch (udmcorba_exception &e) {
		MessageBox(NULL, e.what(), "Error", MB_OK);
	}
```

The first part is needed for a ToolAdaptor with publish mechanism and the second part for a ToolAdaptor with fetch mechanism.

Also put the following codes to in the *{project_name}.cpp* file:

```
// Have to create your own tooladaptor to implement the notify abstract function
class myToolAdaptor : public UdmCORBA::ToolAdaptor {
public:

	myToolAdaptor(const char *type, int argc, char **argv)
		: ToolAdaptor(type, argc, argv) {};

	// this function is called, when a datanetwork arrives
	void notify(const char *par_name, const char *par_version, const char *name, const
char *version, const char *remarks, long docId, long keepAlive) {

		try {
			CString filename;

			// check if the filename is correct
			while (filename.GetLength()<5 || filename.Right(4)!=".mga") {
				// show the datanetwork arrived dialog, this dialog is in the
				// ToolAdaptor_DLL library
				// the values from the dialog will be in the corresponding
				// variables
				int result = showDatanetworkArrivedDlg(name, version, remarks,
keepAlive, &filename);
				if (result!=IDOK) return;
				// check if the filename is correct
				if (filename.GetLength()<5 || filename.Right(4)!=".mga")
					MessageBox(NULL, "Invalid filename! Filename must end
with .mga", "Error", MB_OK);
			}

			// create a new datanetwork
			dn->CreateNew((LPCTSTR)filename, "ecsl", ecsl::RootFolder::meta,
Udm::CHANGES_PERSIST_ALWAYS);
			// fetch the document and put in the datanetwork
			dn->fetchDocument(docId);
			// method in the ToolAdaptor_DLL library. Add a string to the fetch
			// dialog box status listbox
			addStringFetchDlg("Datanetwork fetched.");
			// save the datanetwork in the local disk
			dn->SaveAs((LPCSTR)filename);
			addStringFetchDlg("Datanetwork saved as " + filename);
			// close the datanetwork, we don't need it anymore
			dn->CloseNoUpdate();
			// free up the datanetwork memory allocations
			dn->release();
			addStringFetchDlg("");
			addStringFetchDlg("Waiting for Datanetworks...");

		} catch (udm_exception &e) {
			addStringFetchDlg(e.what());
		} catch (udmcorba_exception &e) {
			addStringFetchDlg(e.what());
		} catch (...) {
			addStringFetchDlg("Fatal Error!");
			return;
		}
	};
};

myToolAdaptor *ta;
```

*myToolAdaptor* source code

```
CWinThread *working_thread;

UINT myWorkingThread( LPVOID pParam ) {
	try {

		// instantiate a tooladaptor, login with the specific type of
		// ToolAdaptor into the Backplane
		myToolAdaptor *ta = new myToolAdaptor("ECSL_RECEIVER", __argc, __argv);

		addStringFetchDlg("Subscribe to ECSL paradigm v1.0");

		// subscribe to the paradigm (metamodel) (here the ecsl paradigm),
		// the ecsl::diagram variable is defined in the {paradigm_name}.h
		// (here, ecsl.h)
		ta->subscribe(ecsl::diagram);

		addStringFetchDlg("Waiting for Datanetworks...");

		// run the tooladaptor, this is an infinite loop to check the arrived
		// documents
		ta->run();

	} catch (udm_exception &e) {
		MessageBox(NULL, e.what(), "Error!", MB_OK);
		return true;
	} catch (udmcorba_exception &e) {
		MessageBox(NULL, e.what(), "Error", MB_OK);
		return true;
	}

	return 0;
}
```

*myWorkingThread* source code

The *myToolAdaptor* and *myWorkingThread* source code is only needed for a ToolAdaptor with fetch mechanism.

If one writes a ToolAdaptor with publish mechanism, an instance of the ToolAdaptor class always has to be implemented, but its *notify* method should not be implemented.

# 10. How to create a simple Translator application

Let's create a simple Translator application. A Translator application does not need MFC support so we only need the basic UdmCORBA API.

1. Create a new *Win32 Console Application*.

2. Add the following directories to the *Additional include directories*:
   - {OTIF demo directory}/Include/UdmCORBA
   - {UDM directory}/Include/Udm
   - {3rd Party directory}/Include/STL

3. Add the following files to the *Object/library modules*:
   - ace.lib
   - TAO.lib
   - TAO_PortableServer.lib
   - UdmBase.lib
   - UdmCORBA.lib
   - UdmUtil.lib
   - xerces-c_2.lib

   Optional (it depends on which backend one wants to use in the project):
   - UdmDOM.lib
   - UdmGME.lib

   Important: The name of the files is different in Debug compilation!
   Important: Do not use precompiled headers!

4. Add the following directories to the *Additional library path*:
   - {OTIF demo directory}/Lib/UdmCORBA
   - {OTIF demo directory}/Lib/tao
   - {UDM directory}/Lib/Udm
   - {3rd Party directory}/Lib/xerces

5. Put the (e.g. matlab and ecsl) paradigm *.cpp, .h* and *.dtd* files into the project directory and add the *.cpp* to the source files.
   **Udm creates these files for you. [Appendix D.]**

   *Note: Please pay attention to create these generated files as the Appendix D. says. Without it your application will not work correctly.*

6. Put the source code in Appendix C. into the {translator_name}.cpp file.

# 11. UdmCORBA exception handling model

When one creates his own *ToolAdaptor* or *Translator* application, one can get error messages from the *UdmCORBA* layer through exceptions.

One has to catch the *udmcorba_exception* and can get a code and a description of the error.

Here is a simple usage example:

```
try {

        ...

} catch (udmcorba_exception &e) {
        if ( e.code()<11 ) {
                cerr << endl << "CORBA exception: " << e.what() << endl;
        } else if (e.code()<21) {
                cerr << endl << "Paradigm exception: " << e.what() << endl;
        } else if (e.code()<41) {
                cerr << endl << "UdmCORBA exception: " << e.what() << endl;
        } else if (e.code()<91) {
                cerr << endl << "ToolAdaptor exception: " << e.what() << endl;
        }
}
```

Exception example

The possible exceptions with code and description parameters are in the following list:

**0 - Fatal exception: Fatal failure**

**1 <-> 10 CORBA layer exceptions**

1 - Fatal exception: Caught a CORBA::Exception exception
2 - Failed to narrow root POA
3 - Failed to narrow root POA manager
6 - Probably there is no active Backplane server.
7 - Fatal error.

**11 <-> 20 Paradigm related exceptions**

11 - Paradigm is not found
12 - A composition was not found in the paradigm
13 - An association was not found in the paradigm
14 - Paradigm exists
15 - Paradigm is locked
16 - Paradigm is in use
17 - Paradigm file is invalid
18 - Paradigm is unknown
19 - A class was not found in the paradigm. Maybe the paradigms are different.

**21 <-> 40 UdmCORBA layer related exceptions**

21 - CORBAObject's setStringAttr method not implemented
22 - CORBAObject's setBooleanAttr method not implemented
23 - CORBAObject's setIntegerAttr method not implemented
24 - CORBAObject's setRealAttr method not implemented
25 - CORBAObject's setParent method not implemented
26 - CORBAObject's detach method not implemented
27 - CORBAObject's setChildren method not implemented
28 - CORBAObject's createChild method not implemented
29 - CORBAObject's setAssociation method not implemented
30 - CORBAObject's __getdn method not implemented
31 - Invalid session
32 - Invalid document
33 - fetchDocument error: Could not get the document
34 - Subscribe failed: ToolAdaptor is not found
35 - Failed to narrow to TranslatorFeed
36 - Failed to narrow to TranslatorSink
37 - publishDocument error: Could not publish the document. There is no active ToolAdaptor session.

**61 <-> 70 Manager related exceptions**

61 - Could not connect to the Backplane server
62 - The password is incorrect
63 - There is an other active Manager

**71 <-> 80 Translator related exceptions**

71 - Translator exists
72 - Translator is in use
73 - Invalid translator
74 - Invalid translator arguments
75 - Translator get_fromDN function is ambiguous, because two or more paradigms are available.
76 - Translator get_fromDN function is ambiguous, because two or more paradigms with the same specified name are available.

**81 <-> 90 Tooladaptor related exceptions**

81 - Invalid ToolAdaptorType
82 - ToolAdaptor createDatanetwork function is ambiguous, because two or more paradigms are available.
83 - ToolAdaptor has to create a datanetwork first.
85 - ToolAdaptor subscribe function is ambiguous, because two or more paradigms are available.
86 - The %s v%s paradigm is not allowed for this type of ToolAdaptor.
87 - Session of the ToolAdaptor could not be found.
88 - ToolAdaptor could not fetch a remote paradigm because datanetwork is NULL.
89 - CORBA layer was not properly initialized.

# Appendix A.

### BackPlane_config.xml parameter file

```
<backplane name="TEST" version="1.0">

  <options password="" uml_paradigm="uml.xml" onlyOneManager="true"
translator_documents="300" log_level="3"/>

  <startup>

    <paradigms>
      <paradigm id="1" name="MATLAB" version="1.0" filename="matlab.xml"
locked="true"/>
      <paradigm id="2" name="ECSL" version="1.0" filename="ecsl.xml"
locked="true"/>
    </paradigms>

    <translators>
      <translator id="2" name="Matlab2ECSL" version="1.0"
command="translator_matlab2ecsl.exe" inputParadigms="2"
outputParadigms="1"/>
    </translators>

    <tooladaptors>
      <tooladaptor id="1" type="ECSL_RECEIVER" inputParadigms="1"
outputParadigms=""/>
      <tooladaptor id="3" type="MATLAB_PUBLISHER" inputParadigms=""
outputParadigms="2"/>
    </tooladaptors>

    <workflow>
      <link source="2" destination="1" outputParadigm="1"/>
      <link source="3" destination="2" outputParadigm="2"/>
    </workflow>

  </startup>

</backplane>
```

The xml type file has two parts.

In the first part there are the

- name, version parameter of the Backplane server. (<backplane> node)
- password parameter, which is used when a Manager application want to connect to a running Backplane server. (<option> node)
- uml_paradigm parameter, which identifies the xml file of the uml paradigm. (<option> node)

- onlyOneManager parameter, which specifies that only one or multiple Manager application can connect to the Backplane at one time. (<option> node)
- translator_documents parameter, which specifies the keepAlive value for the documents published by a Translator application to the Backplane server. (<option> node)

- log_level parameter, which specifies how many information will be written out in the BackPlane.log file. This value is used for debug purposes.
  **0 value** disables to write out any debug information.
  **1 value** writes out only error messages.
  **3 value** writes out information about the Backplane and the various translators.
  **5 value** writes out more detailed information about the Backplane and the various translators.

In the second part (<startup> node) there are the defined paradigms, translator applications, tooladaptor types and the workflow links for the Backplane server. These components are registered, when the Backplane server starts.

- a paradigm with the
  - name, version parameter (<paradigm> node)
  - filename parameter, which identify an xml paradigm file in the *paradigms* directory (<paradigm> node)
  - id parameter, which identifies the paradigm (<paradigm> node)

- a translator with the
  - name, version parameter (<translator> node)
  - inputParadigms, outputParadigms parameters, which identify previously defined paradigms (<translator> node)
  - command parameter, which identify a translator application program in the *translators* directory. (<translator> node)
  - id parameter, which identifies the translator (<translator> node)

- a tooladaptor with the
  - type parameter (<tooladaptor> node)
  - inputParadigms, outputParadigms parameters, which identify previously defined paradigms (<tooladaptor> node)
  - id parameter, which identifies the tooladaptor (<tooladaptor> node)

- a link with the
  - source, destination parameters, which identify previously defined translators or tooladaptors (<link> node)
  - outputParadigm parameter, which identify a previously defined paradigm (<link> node)

# Appendix B.

### Simple ToolAdaptor application example

### 1. Example of a "*document publish*" mechanism.

```
// test.h is created with the udm application from the test paradigm
#include "test.h"
#include "UdmDom.h"
#include "UdmCORBA.h"

using namespace test;

// Have to create your own tooladaptor to implement the notify abstract function
class myToolAdaptor : public UdmCORBA::ToolAdaptor {
public:

	myToolAdaptor(const char *type, int argc, char **argv)
		: ToolAdaptor(type, argc, argv) {};

	void notify(const char *par_name, const char *par_version, const char *name,
const char *version, const char *remarks, long docId, long keepAlive) {};

};

int main(int argc, char* argv[]) {
	if (argc!=4) {
		cout << "Usage: Push_TEST <tooladaptor type> <filename.xml> <keepAlive in
second>" << endl;
		exit(1);
	}

	try {
		// instantiate a tooladaptor, login with the specific type of
		// ToolAdaptor into the Backplane
		myToolAdaptor ta(argv[1], argc, argv);

		// Get the datanetwork from the tooladaptor (here the test paradigm)
		// the diagram variable is defined in the
		// {paradigm_name}.h (here, test.h)
		UdmCORBA::CORBADataNetwork *dn = ta.getDatanetwork(diagram, "TEST",
"1.0");
		// open the file and load in the datanetwork
		dn->OpenExisting(argv[2], "test");
		// publish this datanetwork to the server
		dn->publishDocument("test_doc", "1.0", "test document", atol(argv[3]));
		cout << "DataNetwork pushed." << endl;

	} catch (udm_exception &e) {
		cerr << e.what() << endl;
	} catch (udmcorba_exception &e) {
		if ( e.code()<11 ) {
			cerr << endl << "CORBA exception: " << e.what() << endl;
		} else if (e.code()<21) {
			cerr << endl << "Paradigm exception: " << e.what() << endl;
		} else if (e.code()<41) {
			cerr << endl << "UdmCORBA exception: " << e.what() << endl;
		} else if (e.code()<91) {
			cerr << endl << "ToolAdaptor exception: " << e.what() << endl;
		}
	}

	return 0;
}
```

### 2. Example of a "*document fetch*" mechanism.

```
// test.h is created with the udm application from the test paradigm
#include "test.h"
#include "UdmDom.h"
#include "UdmCORBA.h"

using namespace test;

// Have to create your own tooladaptor to implement the notify abstract function
class myToolAdaptor : public UdmCORBA::ToolAdaptor {
public:

       myToolAdaptor(const char *type, int argc, char **argv)
              : ToolAdaptor(type, argc, argv) {};

      // this function is called, when a datanetwork arrives
       void notify(const char *par_name, const char *par_version, const char *name, const
char *version, const char *remarks, long docId, long keepAlive) {

              // create a new datanetwork
              dn->CreateNew("__pull_test.xml", "test", Root::meta,
Udm::CHANGES_PERSIST_ALWAYS);
              // fetch the document and put in the datanetwork
              dn->fetchDocument(docId);
              // save the datanetwork in the local disk
              dn->SaveAs("__pull_test.xml");
              // close the datanetwork, we don't need it anymore
              dn->CloseNoUpdate();
              // free up the datanetwork memory allocations
              dn->release();
       };
};


int main(int argc, char* argv[]) {
       if (argc!=2) {
              cout << "Usage: Pull_Test <tooladaptor type>" << endl;
              exit(1);
       }
       try {
              // instantiate a tooladaptor, login with the specific type of
              // ToolAdaptor into the Backplane
              myToolAdaptor ta(argv[1], argc, argv);
              // subscribe to the paradigm (metamodel) (here the test paradigm),
              // the diagram variable is defined in the {paradigm_name}.h
              // (here, test.h)
              ta.subscribe(diagram, "TEST", "1.0");
              // run the tooladaptor, this is an infinite loop to check the arrived
              // documents
              ta.run();

       } catch (udm_exception &e) {
              cerr << e.what() << endl;
       } catch (udmcorba_exception &e) {
              if ( e.code()<11 ) {
                     cerr << endl << "CORBA exception: " << e.what() << endl;
              } else if (e.code()<21) {
                     cerr << endl << "Paradigm exception: " << e.what() << endl;
              } else if (e.code()<41) {
                     cerr << endl << "UdmCORBA exception: " << e.what() << endl;
              } else if (e.code()<91) {
                     cerr << endl << "ToolAdaptor exception: " << e.what() << endl;
              }
       }

       return 0;
}
```

# Appendix C.

### Simple Translator application example

```
// test.h is created with the udm application from the test paradigm
#include "test.h"
#include "UdmDom.h"
#include "UdmCORBA.h"
#include "UdmCORBA_Logger.h"
Logger logger = Logger("Translator_Test");

using namespace test;

class myTranslator : public UdmCORBA::Translator {
public:

	myTranslator(const Udm::UdmDiagram &from_metainfo, const char *from_name, const
char *from_version, const Uml::Class &from_rootclass, const Udm::UdmDiagram &to_metainfo,
const char *to_name, const char *to_version, const Uml::Class &to_rootclass, int argc,
char **argv, Logger *logger)
	: Translator(from_metainfo, from_name, from_version, from_rootclass, to_metainfo,
to_name, to_version, to_rootclass, argc, argv, logger) {};

       // this function is called, when a datanetwork arrives
	void notify(const char *name, const char *version, const char *remarks, const
unsigned long numOfDocs) {

		// Udm specific functions
		Root from_root = Root::Cast(fromDN->GetRootObject());
		Root to_root = Root::Cast(toDN->GetRootObject());
		to_root.count() = from_root.count();
		set<A> as = from_root.A_kind_children();
		for(set<A>::iterator i=as.begin(); i!=as.end(); i++) {
			A a = A::Create(to_root);
			string name = (*i).name();
			name.append("_copy");
			a.name() = name;
		}

		// publish the datanetwork which is in the toDN Translator variable
		publish(name, version, "After the TEST1.0->TEST1.1 translation");
	}
};

int main(int argc, char* argv[]) {
	try {
		// instantiate a translator, the test::diagram variable is defined in the
		// {paradigm_name}.h (here, test.h), Root::meta is the rootobject's meta
		// defined in the {paradigm_name}.h (here, test.h)
		myTranslator tr(diagram, "TEST", "1.0", Root::meta,
			diagram, "TEST", "1.1", Root::meta,
			argc, argv, &logger);
		// run the translator, this is an infinite loop to check the arrived
		// documents and call the notify method
		tr.run();

	} catch (udm_exception &e) {
		cout << e.what() << endl;
		logger.log(1,  "Error", e.what());
	} catch (udmcorba_exception &e) {
		cout << e.what() << endl;
		logger.log(1,  "Error", e.what());
	} catch (...) {
		cerr << "Fatal exception" << endl;
		logger.log(1,  "Error", "Fatal exception");
	}
	return 0;
}
```

# Appendix D.

### How to create the *.cpp, .dtd and .h* files by the Udm.exe

Udm.exe creates the *.h, .cpp, .dtd* files of the paradigm you want to use in your project. However, if you want to use this files in one of your OTIF project, you have to creates these files with the –c option.

*For example:*

```
Udm.exe paradigm.xml -d {directory of the corresponding dtd files} -c
```

Udm application with the –c option creates the CORBA specific remote paradigm (metamodel) files.

*Note: If you do not specify the –c switch, Udm.exe will create the local version of the paradigm and it should not be used in an OTIF application. Please use always the –c switch if you implement an application based on the OTIF framework.*

# Appendix E.

**Troubleshooting**

1. **I installed one of the extension packages and started the OTIF framework I could not use it because there are no ToolAdaptor types and Translators registered.**

   The registration process of the Translators and ToolAdaptor types is changed from the 1.2.0 version. You have open the workflow file of the extension package in GME and interpret it. While the interpretation process it will register all the Translators, ToolAdaptor types and connections between them in the Backplane. Every time you change something in the configuration you have to interpret it.

   **Solution:** Open the workflow file of the extension package in GME and interpret it while the OTIF framework is running.

2. **When I try to fetch or publish a GME document (.mga file), I get an error message that the paradigm is not registered.**

   Probably you did not register the corresponding mga layer paradigm (for example the ecsl.xmp paradigm of the ECSL extension package under the Documents directory).

   **Solution:** Register the paradigm of your document in GME2000 (for example the ecsl.xmp file of the ECSL extension package under the Documents directory).

3. **When I try to publish the sample "mdl" type document with GenericToolAdaptor application, it said "Cannot deduce Udm backend type from mdl_sample.mdl. Available backends: DOM GME MEM"**

   The GenericToolAdaptor application can only publish "xml", "mem" and "mga" type documents. DOM backend supports "xml" type GME backend supports "mga" type and MEM backend supports "mem" type documents. Right now only these backends are available in UDM.

   **Solution:** Publish the "mdl" type documents with ToolAdaptor_MDL application.