How to develop an interoperability solution in one code base and then use it to generate many individual interfaces

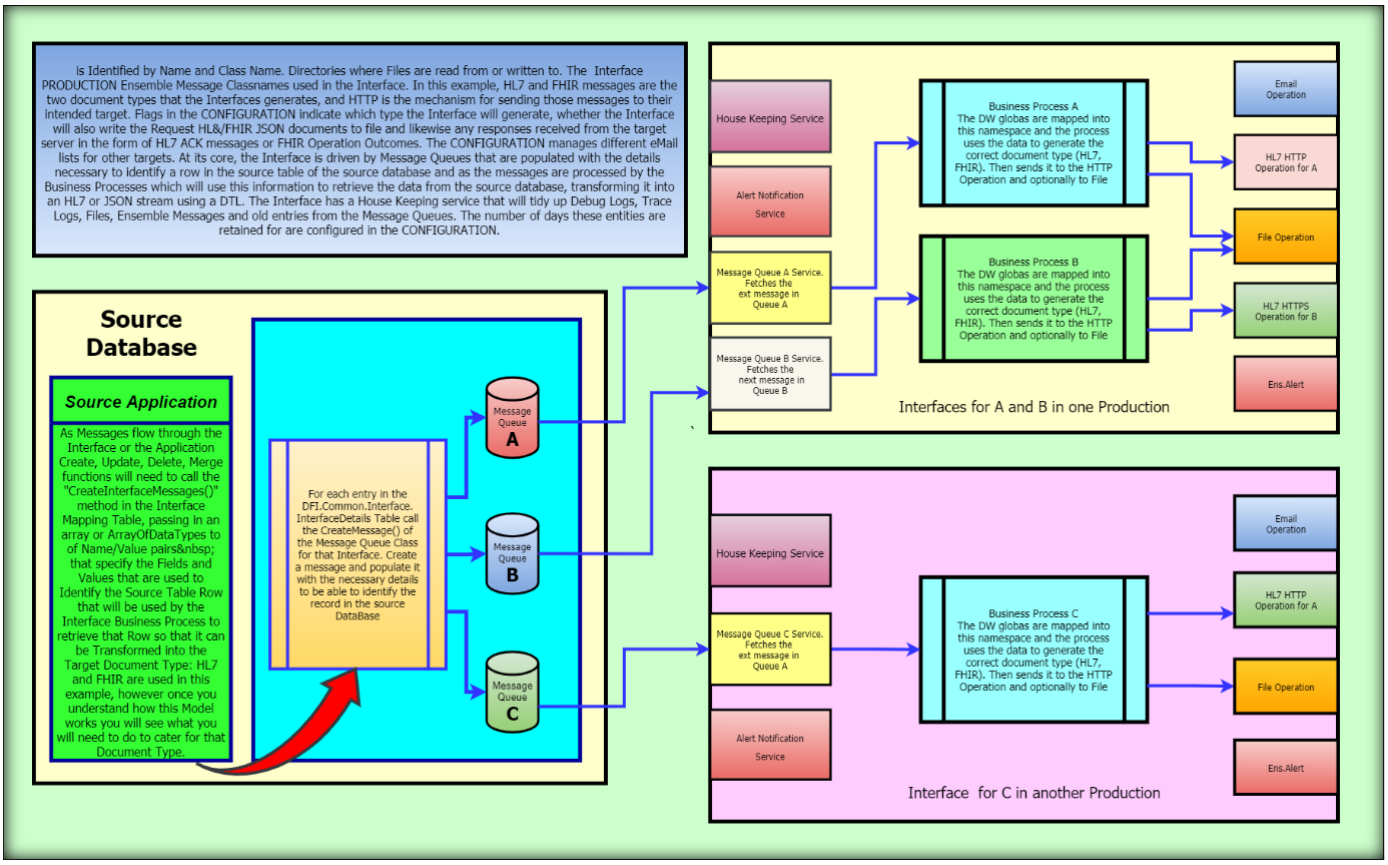
Overview

In 2009 I wrote the first of several Interfaces for LabTrak. The Interfaces allowed existing National Health Laboratory Services (NHLS) clients to send Patient Demographics and Orders to LabTrak. LabTrak would, in turn, send the results of those tests back to the client. I wrote 3 or 4 at the time and an Interface that fed data from LabTrak into the NHLS Corporate Data Warehouse. I moved onto other things, and over the years, I was aware that further Interfaces were written, and most of them copied chunks of my code into those new Interfaces. I know this because I would periodically get calls from the developers asking me to explain how some logic worked. They all commented that they loved my code because it was well written and well documented, and, most importantly, they worked.

I built several other Interfaces over the years, so when I returned to the company as an employee as opposed to a contractor as I had been at the time, I wasn't surprised to find a host of Interfaces that all bore the signature signs of my coding style. The company has been building a FHIR based Master Patient Index and are planning to develop additional modules to cover the regular aspects of a Hospital Information System. This new Application will replace the current HIS that is 30+ years old and has been running in our province for the last twenty. At the beginning of the year, I was asked to write many interfaces that will replace the HL7 interfaces driven by the old Application. I will also write several interfaces that will take data from the Operational Data Store and populate the new Master Patient Index and, down the line, the new HIS modules as well. Once the Patient Master Index goes live and the HIS Modules go live, the data flow will reverse, and the Interfaces will take data from the PMI and HIS modules and push the data into the Operational Data Store. I realized that if I were to follow history, I would write many almost identical interfaces, varying little more than the source database/tables and create messages with little difference other than the specific HL7 Message Type or FHIR Resource type. I decided to write a single code base that could be mapped into the Namespace of each new Interface.

Summary view of the Interface Model

The following diagram describes the basic flow of the Data Flow Interface Model.



Approach

The Message Queue Classes

The Message Queue classes sit at the heart of this Interface Model. The Message Queue classes carry the properties that identify a record in the Source Database.

1. The Interfaces would be based on Message Queues where each message queue contained the fields that would identify the record in the source database that must be processed by the Interface and transformed into an HL7 or FHIR message.
2. The Message Queue classes would hold some common properties as follows:
    a. Created, Processed and Competed TimeStamps
    b. Fields that would store the request and response HL7 and FHIR messages.
    c. File Names and Folder paths into which the request and response messages could be written.
    d. The HTTP Operation Status Code, the File Operation Status code and the overall status of the Message Queue Message.
3. These properties are defined in an Abstract Class and inherited into every Message Queue Class created for each Interface.
4. When you define a new Interface, the best way is to create a new class definition and inherit (extends) this abstract class. You then define the properties populated with data from the source database/table. You may have two or more interfaces that use the same source tables, and you must create a new Message Queue class for each of them. This is so that when the classes are compiled, they will generate different storage definitions, and implicitly, different global names, which is critical as the Interfaces must be able to differentiate between the Message Queue Class Names and the Global Names linked to those classes.

5. There is a %RegisteredClass of class methods that manipulate the data in the Message Queues. These methods include:

    a. CreateMessage()
    b. GetNextMessage()
    c. UpdateMessage()
    d. CompleteMessage()
    e. ResendMessage()
    f. ResendRangeOfMessages()

6. When you create a new Message Queue Class, you inherit this Methods class. The CreateMessage() and UpdateMessage() methods are passed a Name/Value array of property names and values. The Message Queues are aware of the common properties in the Message Queues but do not know what fields the developer added when creating the new Interface. You need to create Indices on the three TimeStamp Fields. They cannot be defined in the Abstract class. These TimeStamp indices are used to identify messages that have not been processed and messages that are complete and, based on the CompletedTS, ready to be purged.

7. Each Interface has a Business Service that will process the Message Queue class created for the Interface. The Service has no Adapter. The OnProcessInput() of the Service calls the GetNextMessage() method to retrieve the next unprocessed Queue Message based on an SQL query that has a WHERE clause: "WHERE ProcessedDate is NULL". If a message is found, the GetnextMessage() method will set the ProcessedTS property of the Message to the current Date/Time.

8. If a Message has to be resent, then the ResendMessage() will set the ProcessedTS and CompletedTS to NULL, which ensures that the GetNextMessage() method will find the Message and resend it.

9. The Business Service gets the Message Queue Class Name from the Interface Configuration Record.

abstract class **DFI.Common.Abstract.MessageQueueBaseProperties**

This is the Template Message Queue Class that contains the core Message Queue Properties that are to every Message Queue used in all DFI Interfaces.

If the Source Namespace is and Operational Data Store (ODS) and we are doing a bulk export of ODS Patients into a FHIR Server, then the DFI Interface will execute an SQL query that selects records from the Patient table and creates messages in the 'Patient List' message queue. Once the List of Patients has been created a different Business Service then processes that list and generates FHIR messages to send to FHIR Server.

At the same time the Application to ODS Trickle feed will be creating messages in the DFI ODS to FHIR trickle feed message queue using data from the Request messages passing the Application to ODS Trickle Feed.

If the data source is an FHIR database and we are running 'Use Case' Tests in the DFI Test Module then we introduce the concept of a Test Manifest which will contain a number of Manifest Records where each Manifest Record reference a FHIR Patient and the Test definition that contains the Test Rules to be applied to those Patients in the Manifest. The Manifest can contain anywhere between 1 and 1000 Patients. Linked to the Manifest is the Test Definition which consists of one or more Test Rules that modify specific properties in that Patient Records and related Patiet Tables such as Address, Contact, Names.

Read the documentation in the Test Module to understand how the Test Module works.

The Message Queue Classes have the following properties in common:
Three time stamps: CreateTS, ProcessTS and CompletedTS.

The **CreateTS** is set to the current Date/Time when a new message is Instantiated.

The **ProcessTS** is set when the Method GetNextMessage() is called and the method finds a message that has not been processed. The ProcessTS is updated with the Current Date/Time.

The **CompletedTS** is updated when the Message has gone through the cycle of Business Service -> Business Process -> Business Operation and back to the Business Process and back to the Business Service.

In order to facilitate this there is a class in the Source Interface Namespace, **DFI.Common.Interface.InterfaceMappingDetails** that is a list of Interface Names, the Namespace in which the Interface is running, the Message Queue Class Name of the message queue that drives the Interfaces in those namespaces. The property IsProduction indicates whether the target Interface is the 'Live' or 'Production' namespace and the property IsActive indicates whether the target Interface is Active or not.

Inventory

Parameters	Properties	Methods	Queries	Indices	ForeignKeys	Triggers
	22					

Summary

Properties		
CompletedTS	CreateTS	FHIRFileDirectory
FHIRRequestFileName	FHIRResponseFileName	FHIRResponseLocation
HL7FileDirectory	HL7RequestFileName	HL7ResponseFileName
ManifestId	MessageStatus	MessageStatusText
ProcessTS	SourceFHIRRequestMessage	SourceHL7RequestMessage
TargetDocumentType	TargetFHIRResponseMessage	TargetHL7ResponseMessage
TargetResponseStatus	TargetResponseStatusText	TimeTakenFromCreateToComplete
TimeTakenFromProcessingToComplete		

Subclasses	
DFI.BulkExport.Queue.BulkExportMessageQueuePRD	DFI.BulkExport.Queue.BulkExportMessageQueu
DFI.Common.Queue.EMCITestMessageQueue	DFI.Common.Queue.EMCIToEMCITestMessageQ
DFI.Common.Queue.ODSMessageQueue	DFI.Common.Queue.ODSTFMessageQueue
DFI.EMCIEMCITestModule.Queue.TestMessageQueuePRD	DFI.EMCIEMCITestModule.Queue.TestMessageQ
DFI.EMCITrickleFeed.Queue.ODSTFMessageQueueQC	DFI.ODSEMCITestModule.Queue.TestMessageQu
DFI.WCGHL7STD.Queue.ODSTFMessageQueuePRD	DFI.WCGHL7STD.Queue.ODSTFMessageQueueQ

Properties

• property **CompletedTS** as %String;

This is a system Property and should not be updated by the developer

• property **CreateTS** as %TimeStamp [InitialExpression = $ZDatetime($Horolog,3)];

This is a system Property and should not be updated by the developer

• property **FHIRFileDirectory** as %String;

This is the file Directory where FHIR Files are written

• property **FHIRRequestFileName** as %String(MAXLEN=400);

This is the FHIR Request Message File Name

• property **FHIRResponseFileName** as %String(MAXLEN=400);

This is the FHIR Response Message File Name

• property **FHIRResponseLocation** as %String(MAXLEN=1000);

If the FHIR HTTP Operation Recieves a Location in the HTTPResponse Header put it in this property

• property **HL7FileDirectory** as %String;

This is the file Directory where HL7 Files are written

• property **HL7RequestFileName** as %String(MAXLEN=400);

This is the HL7 Request Message File Name

• property **HL7ResponseFileName** as %String(MAXLEN=400);

This is the HL7 Response Message File Name

• property **ManifestId** as %String;

If the Test Module has been used then we need to know the ManifestId

• property **MessageStatus** as %Status [InitialExpression = $$$OK];

The Message Status is a system property and will be determined when the Develloper calls the CompleteMessage() method by passing in a vallid %Status value reflecting the Status at the time the method is called.

• property **MessageStatusText** as %String(MAXLEN=5000);

This is the Message Text that gives more information on the Message Status. Typically it will be the contents of the %Status Code in Message Status but there may be many error conditions that can be passed to the UpdateMessage() method and the CompleteMessage() method. These messages are appended to the current Message.MessageStatusText

• property **ProcessTS** as %TimeStamp;

This is a system Property and should not be updated by the developer

• property **SourceFHIRRequestMessage** as %CharacterStream;

The FHIR JSON Request Message Created by the Business Process

• property **SourceHL7RequestMessage** as EnsLib.HL7.Message;

This is the HL7 Request Message that will be created by the Business Process and the Message Object will be updated by the Business Process with the HL7 Message it creates. This must be set by the developer. The 'Source' indicates that this is the Message that is sent from the Source application which is the Interface and the Target Response Message is the Response that comes back from the 'Target' 3rd Party application. The Business Process knows which Operations it is calling and in what sequenceso it is up to the Business Process to set the IsRequest flag in the Request Message it sends to the file operation.

• property **TargetDocumentType** as %String(DISPLAYLIST=",HL7,FHIR,JSON,SQL",VALUELIST=",H,F,J,S");

This may be specifed by the developer or if a copy class has been created where this is known then use [InitialValue] to set the value and on instantiation the property will assume that value. The Response message has the same property and should be set by the developer when he creates the Response Message

• property **TargetFHIRResponseMessage** as %CharacterStream;

The FHIR JSON Response Message received by the HTTP/HTTPS FHIR Business Operation

• property **TargetHL7ResponseMessage** as EnsLib.HL7.Message;

This should be the HL7 Response Message that comes back from the Target system. For the HTTP Operation it is simple in that we send the source to the HTTP opertion and we get a response back. The file operation on the other hand is used primarily to file the messages during testing and is normally disabled in Production. So I need to tell the file operation wheter to getthe data from the source hL7 message or the Target. The Business process knows and in the ensemble request message there are properties for the directory and the filename and a flag to indicate if it is the requestmessage or not. The operation then knows which property to use to get the contents that it will write to file.

• property **TargetResponseStatus** as %String [InitialExpression = $$$OK];

This is the Response returned by the HTTP or File Operation. The Operation will evaluate the ACK Message Status Code aand if it is a CE or AE then the operation will create a Error Status with a Status message describing the the HL7 Status Code and Error Message. The operation will quit with tSC'=OK In the case of FHIR the operation will react to wheher it gets an Operation Outcome or not. If it does it will decode the Operration Outcome and create an Error Status Code and construct a Message based on theOperation Outcome There is no need to differentiate between FHIR or HL7 as the Interface will either be an HL7 Interface or a FHIR Interface

• property **TargetResponseStatusText** as %String(MAXLEN=5000);

The Response Status Text is a Readable interpretation of the Response Status. It is the responsibility of the operation to interpreet the HTTP Status Code, the NACK error code or a code crash and generate an appropriate Error Status Code an a meaning full Error Message so that we know if the error was the result of the HTTP POST itself or the HL7 ACK Code

• property **TimeTakenFromCreateToComplete** as %Integer;

This is a system property and should not be updated by the developer. As the Method name suggests this is the time difference between the Message being Created through to Completion

• property **TimeTakenFromProcessingToComplete** as %Integer;

This is a system property and should not be updated by the developer. As the Method name suggests this is the time difference between the Message being picked up in the GetNextMessage() method. This is effectively the time from the Message being identified in the Business

Service that processes the Message Queue and picks up a Message ID which it puts into a Request Object and sends it to the Primary Business Process which does the work to retrieve the appropriate data, transforms it in an HL7 or FHIR JSON message and in turn is passed to an HTTP operation and optionally a File Operation. Once we have processed the Message we call the CompleteMessage() method that updates the CompleteTS

The Message Queue Methods look like this,

```
Class DFI.Common.Abstract.MessageQueueBaseMethods Extends %RegisteredObject
{

/// The Message Queue Class includes the fields from the TransactionLog Request Message in the Clinicom - ODS Trickle Feed are
/// Action, ActivityDateTime, ClinicCode, DoctorCode, EpisodeNumber, LogType, PatientNumber, TrackingDate, TrackingSequence,
TransactionType
ClassMethod CreateMessage(ByRef pValues As %String(MAXLEN=5000), ByRef pMessageId As %String = "") As %Status
{
	Set tSC = $$$OK,pMessageId=""
	Try {
		set obj = $classmethod($classname(),"%New")
		set tProp="" For {
			set tProp=$O(pValues(tProp)) quit:tProp=""
			// Only set the property value if the node pValues(pProp)'=""
			if $l(pValues(tProp)) set $property(obj,tProp)=pValues(tProp)
			// Unless the node pValues(pProp,"Force") is set to 1 which indicates set the
			// property to NULL.
			if '$l(pValues(tProp)),+$get(pValues(tProp,"Force")) set $property(obj,tProp)=""
		}
		Set tSC = obj.%Save() if 'tSC quit
		Set pMessageId = obj.%Id()
	}
	Catch ex {Set tSC = ex.AsStatus()}
	$$$DebugLog($username,"CreateMessage","Create Message for Message Queue: ("_$classname()_") Status:
"_$s(tSC:$$$OK,1:$$$GetErrorText(tSC)),.dSC)
	Quit tSC
}

/// The Update Message Method allows properties in the Message Object to be updated. The message object
/// for the specified Message ID must exist. If it does not exist use the CreateMessage() method.<br>
/// There are specific Message Properties that can only be set when the Message is first created
/// and cannot be modified or cannot be modified by this method. They are the TimeStamps.
ClassMethod UpdateMessage(pMessageId As %String = "", ByRef pValues As %String(MAXLEN=500000),
pSourceHL7RequestMessage As EnsLib.HL7.Message = "", pTargetHL7ResponseMessage As EnsLib.HL7.Message = "",
pSourceFHIRRequestMessage As %CharacterStream = "", pTargetFHIRResponseMessage As %CharacterStream = "") As %Status
{
	Set tSC = $$$OK,hSSC=$$$OK,hTSC=$$$OK,fSSC=$$$OK,fTSC=$$$OK // hSSC/hRSC=HL7Source/Target Status,
fSSC/fTSC=FHIRSource/Target Status
	Try {
		if '$l($g(pMessageId)) set tSC=$$$ERROR(5001,"Message ID cannot be NULL") quit
		set obj = $classmethod($classname(),"%OpenId",pMessageId) if '$IsObject(obj) set tSC=$$$ERROR(501,"There is no
Mesage for Message ID: "_pMessageId) quit
		// Deal with the Source and Target HI7 or FHIR Request/Response Messages
		if $IsObject(pSourceHL7RequestMessage)
set obj.SourceHL7RequestMessage=obj.SourceHL7RequestMessage.ImportFromString(pSourceHL7RequestMessage.OutputToString(,,.hSS
C)) $$$DebugLog($username,"UpdateMessage","Update Source HL7 Message Status: "_$s(hSSC:"OK",1:$$$GetErrorText(hSSC)),.dSC)
		if $IsObject(pTargetHL7ResponseMessage)
set obj.TargetHL7ResponseMessage=obj.TargetHL7ResponseMessage.ImportFromString(pTargetHL7ResponseMessage.OutputToString(,,.h
TSC)) $$$DebugLog($username,"UpdateMessage","Update Source HL7 Message Status: "_$s(hTSC:"OK",1:$$$GetErrorText(hTSC)),.dSC)
		// Update the Source and Target FHIR Messages
		if $IsObject(pSourceFHIRRequestMessage) set tSC=pSourceFHIRRequestMessage.Rewind() quit:'tSC
set fSTC=obj.SourceFHIRRequestMessage.CopyFrom(pSourceFHIRRequestMessage) $$$DebugLog($username,"UpdateMessage","Update
Source FHIR Message Status: "_$s(fSSC:"OK",1:$$$GetErrorText(fSSC)),.dSC00)
		if $IsObject(pTargetFHIRResponseMessage) set tSC=pTargetFHIRResponseMessage.Rewind() quit:'tSC
set fTSC=obj.TargetFHIRResponseMessageCopyFrom(pTargetFHIRResponseMessage)
$$$DebugLog($username,"UpdateMessage","Update Source FHIR Message Status: "_$s(fTSC:"OK",1:$$$GetErrorText(fTSC)),.dSC)
		// Update any other properties

	for tProp="CreateTS","ProcessTS","CompletedTS","SourceHL7RequestMessage","TargetHL7ResponseMessage","SourceFHIRReq
uestMessage","TargetFHIRResponseMessage" kill pValues(tProp)
		set tProp="" For {
			set tProp=$O(pValues(tProp)) quit:tProp=""
			// Neeed to check if this is the best way to update an HL7 Message Property
			// Update the Source and Target HL7 Messages
			// Only set the property value if the node pValues(pProp)'=""
			if $l(pValues(tProp)) set $property(obj,tProp)=pValues(tProp) continue
			// Unless the node pValues(pProp,"Force") is set to 1 which indicates set the
			// property to NULL.
			if '$l(pValues(tProp)),+$get(pValues(tProp,"Force")) set $property(obj,tProp)=""
```

```
                }
                Set tSC = obj.%Save() if 'tSC quit
        }
        Catch ex {Set tSC = ex.AsStatus()}
        $$$DebugLog($username,"UpdateMessage","Update Message for Message ID: "_pMessageId_" Status:
"_$s(tSC:$$$OK,1:$$$GetErrorText(tSC)),.dSC)
        Quit tSC
}

/// This method finds the next Message in the Interface Message Queue Table where the ProcessTS IS NULL.
/// If a message is found then the method calls the method SetMessageStateToProcessing() which sets
/// the message property ProcessTS to the current Date/Time. To resend a message the properties
/// ProcessTS and CompletedTS to null which effectively sets the messages to un-processed
ClassMethod GetNextMessage(ByRef pMessageID As %String) As %Status
{
        Set tSC = $$$OK
        Try {
                set tTable=$classname() if $l($classname(),".")>2 set tTable=$tr($p($classname(),".",1,$l($classname(),".")-
1),".","_")_"."_$p($classname(),".",$l($classname(),"."))
                set sql="select ID as pMessageID from "_tTable_" where ProcessTS IS NULL"
                set rs=##class(%ResultSet).%New("%DynamicQuery:SQL")
                set tSC=rs.Prepare(sql) if 'tSC quit
                set tSC=rs.Execute() if 'tSC quit
                // find next Message and call the method to set the ProcessTS to the curent Date/Time signifying that
                // the Message is being processed
                set found=rs.Next() if 'found set pMessageID="" quit
                set pMessageID=rs.Data("pMessageID")
                set tSC=..SetMessageStateToProcessing(pMessageID) if 'tSC quit
        }
        Catch ex { Set tSC=ex.AsStatus()}
        $$$DebugLog($username,"GetNextMessage","Get Next Message Status:"_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        if 'tSC set pMessageID=""
        Quit tSC
}

/// This Method sets the ProcessTS to the Current Date/Time and indicates that the Message is
/// currently being processed
ClassMethod SetMessageStateToProcessing(pMessageID As %String) As %Status
{
        Set tSC = $$$OK
        Try {
                Set obj=$classmethod($classname(),"%OpenId",pMessageID)
                If '$IsObject(obj) { Set tSC=$$$ERROR(5001,"Unable to Open Message: "_pMessageID) quit}
                Set obj.ProcessTS = $zdt($Horolog,3)
                Set tSC = obj.%Save() if 'tSC quit
        }
        Catch ex { Set tSC=ex.AsStatus() }
        $$$DebugLog($username,"SetProcessTS","Set ProcessTS for Message: "_pMessageID_" Status:
"_$s(tSC:$$$OK,1:$$$GetErrorText(tSC)),.dSC)
        Quit tSC
}

/// The CompleteMessage() method is called when the Interface Production has finished processing the
/// Message. It should be called by the DFI Process Message Queue Service which is the Business Service
/// that processes the DFI.Common.Oqueue.ODSMessageQueue after a message has been retrieved from the
/// queue and sent Synchronously to the DFI Create Base Message Process which is the Business Process
/// that creates the Base HL7 or FHIR Message. The Ensemble Response Message will contain the Status
/// Code returned from the Target Application API and the MessageStatus.<br>
/// The method updates the overall Message Status, the Status Code returned from the
/// HTTP Operation and is a valid %Status. If the HL7 ACK Message is a NACK then the Status
/// returned by the Operation should be an interpretation of the HL7 ACK Code<br>
/// The TargetResponseStatus is the HTTP Response Status<br>
/// The TargetResponseStatusText should be an interpretation of the HTTP Status Code<br>
/// The method also updates the two Time Calculation Fields that record the time taken:<br>
/// 1) From Created to Completed (in seconds)<br>
/// 2) From In Progress to Completed (in seconds)<br>
ClassMethod CompleteMessage(pMessageID As %String = "", pMessageStatus As %Status = {$$$OK}, pMessageStatusText As %String = "",
pTargetResponseStatus As %String = "201", pTargetResponseStatusText As %String(MAXLEN=500) = "",
pTargetHL7ResponseMessage As EnsLib.HL7.Message = "", pTargetFHIRResponseMessage As %CharacterStream = "") As %Status
{
        Set tSC = $$$OK
        Try {
                if '$l(pMessageID) set tSC=$$$ERROR(5001,"No Message ID specified") quit
                Set obj = $classmethod($classname(),"%OpenId",pMessageID)
                If '$IsObject(obj) { Set tSC=$$$ERROR(5001,"Unable to Open Message: "_pMessageID) quit }
                if $l(obj.CompleteTS) $$$DebugLog($username,"CompleteMessage","Message: "_pMessageID_" in Message Queue:
"_$classname()_" is already Complete",.dSC)
                if $l(pMessageStatus) {
                        set obj.MessageStatus=pMessageStatus
                        if '$l(pMessageStatusText) set pMessageStatusText=$s(pMessageStatus:"Message Completed
OK",1:$$$GetErrorText(pMessageStatus))
```

```
                        set obj.MessageStatusText=obj.MessageStatusText_$s($l(obj.MessageStatusText):"
",1:"")_obj.MessageStatusText
                }
                if $l(pTargetResponseStatus) {
                        set obj.TargetResponseStatus=pTargetResponseStatus
                        if '$l(pTargetResponseStatusText) set pTargetResponseStatusText=$s(pTargetResponseStatus:"Message
Completed OK",1:$$$GetErrorText(pTargetResponseStatus))
                        set obj.TargetResponseStatusText=obj.TargetResponseStatusText_$s($l(obj.TargetResponseStatusText):"
",1:"")_pTargetResponseStatusText
                }
                if $IsObject(pTargetHL7ResponseMessage)
set obj.TargetHL7ResponseMessage=obj.TargetHL7ResponseMessage.CopyFrom(pTargetHL7ResponseMessage)
                if $IsObject(pTargetFHIRResponseMessage) set tSC=pTargetFHIRResponseMessage.Rewind() quit:'tSC
set tSC=obj.TargetFHIRResponseMessage.CopyFrom(pTargetFHIRResponseMessage) quit:'tSC
                Set obj.CompletedTS = $zdt($h,3)
                set obj.TimeTakenFromCreateToComplete=..CalculateTime(obj.CreateTS,obj.CompletedTS)
                set obj.TimeTakenFromProcessingToComplete=..CalculateTime(obj.ProcessTS,obj.CompletedTS)
                Set tSC = obj.%Save() if 'tSC { Quit }
        }
        Catch ex { Set tSC=ex.AsStatus() }
        $$$DebugLog($username,"CompleteMessage","Complete Message: ("_pMessageID_") Status:
"_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        Quit tSC
}

/// The CalculateTime() Method firstly computes the time taken from when the Message was Created and
/// when the Message was Completed. Secondly, it computes the time from when the Message was picked
/// up from the queue through to when the Message was Completed. The time unit is seconds.<br>
/// The method is called from the CompleteMessage() Method.<br>
ClassMethod CalculateTime(pStartTS As %TimeStamp, pEndTS As %TimeStamp) As %Integer
{
        set tSC=$$$OK
        try {
                set tStart=$zdth(pStartTS,3),tEnd=$zdth(pEndTS,3) &sql(SELECT DATEDIFF('ss',:tStart,:tEnd) INTO :return)
        }
        catch ex {set tSC=ex.AsStatus()}
        quit return
}

/// The PurgeMessageQueue() Method will cleardown messages that have been processed or completed
/// that are older than the Number of days messages must be retained. If no value is passed to the
/// method parameter pNumberOfDays then the method will obtain the value from the Interface
/// Configuration Record. This method should be called by the DFI Housekeeping Service.
/// The method will return the Number of Messages purged.<br>
ClassMethod PurgeMessageQueue(pNumberOfDays As %Integer, ByRef pNumberOfMessagesPurged = 0) As %Status
{
        set tSC=$$$OK
        try {
                set tTable=$classname() if $l(tTable,".")>2 set tTable=$tr($p($classname(),".",1,$l($classname(),".")-
1),".","_")_"."_$p($classname(),".",$l($classname(),"."))
                if '$l(pNumberOfDays) {
                        set tConfig=##class(DFI.Common.Configuration.ConfigurationSettings).%OpenId("Settings")
if $IsObject(tConfig) set tSC=$$$ERROR(5001,"Unable to Open Configuration Settngs") quit
                        set pNumberOfDays=$g(pSettings("DFINumberOfDaysToKeepQueueMessages"),90)
                }
                set tDate=$zdt($h-pNumberOfDays,3)
                set sql="delete from "_tTable_" where CompletedTS < '"_tDate_"'"
                set rs=##class(%ResultSet).%New("%DynamicQuery:SQL")
                set tSC=rs.Prepare(sql) if 'tSC quit
                set tSC=rs.Execute() if 'tSC quit
                set pNumberOfMessagesPurged=rs.%ROWCOUNT
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"PurgeMessageQueue","Purge Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This method will reset the ProcessTS and CompletedTS to NULL which mans that it is effectively back to a state of unprocessed.
/// The GetNextMessage() method will find this Message and resend it.
ClassMethod ResendQueueMessage(pMessageID As %Integer = "") As %Status
{
        set tSC=$$$OK
        try {
                if '$l(pMessageID) set tSC=$$$ERROR(5001,"No Message ID specified") quit
                Set obj = $classmethod($classname(),"%OpenId",pMessageID)
                If '$IsObject(obj) { Set tSC=$$$ERROR(5001,"Unable to Open Message: "_pMessageID) quit }

        set obj.ProcessTS="",obj.CompletedTS="",obj.TargetResponseStatus="",obj.MessageStatus=$$$OK,obj.MessageStatusText=""
                set obj.SourceFHIRRequestMessage="",obj.SourceHL7RequestMessage="",obj.TargetFHIRResponseMessage=""

        set obj.TargetHL7ResponseMessage="",obj.TargetResponseLocation="",obj.TargetResponseStatus=$$$OK,obj.TargetResponseSt
```

```
atusText=""
                    set tSC=obj.%Save() if 'tSC quit
            }
            catch ex {set tSC=ex.AsStatus()}
            $$$DebugLog($username,"ResendMessage","Resend Queue Message: "_pMessageID_" Status:
"_$s(tSC:$$$OK,1:$$$GetErrorText(tSC)),.dSC)
            quit tSC
}

ClassMethod ResendDateRange(pFromTS As %TimeStamp = {$zdt($p($h,",",1)_","_($p($h,",",2)-7200),3)}, pToTS As %TimeStamp =
{$zdt($h,3)}) As %Status
{
            set tSC=$$$OK
            try {
                    set table=$p($classname(),".",$l($classname(),".")-1)_"."_$p($classname(),".",$l($classname(),"."))
                    set sql="select ID from "_table_" where ProcessTS > '"_pFromTS_"' and ProcessTS < '"_pToTS_"'"
                    $$$DebugLog($username,"ResendDateRange","Date Range SQL: "_sql,.dSC)
                    set rs=##class(%ResultSet).%New("%DynamicQuery:SQL")
                    set tSC=rs.Prepare(sql) if 'tSC quit
                    set tSC=rs.Execute() if 'tSC quit
                    while rs.Next(.tSC) {
                            quit:'tSC
                            set id=rs.Get("ID"),obj=$classmethod($classname(),"%OpenId",id)
                            if '$IsObject(obj) $$$DebugLog($username,"ResendDateRange","Cannot open Message with ID: "_id,.dSC)
continue

            set obj.ProcessTS="",obj.CompletedTS="",obj.TargetResponseStatus="",obj.MessageStatus=$$$OK,obj.MessageStatusText=""

            set obj.SourceFHIRRequestMessage="",obj.SourceHL7RequestMessage="",obj.TargetFHIRResponseMessage=""

            set obj.TargetHL7ResponseMessage="",obj.TargetResponseLocation="",obj.TargetResponseStatus=$$$OK,obj.TargetResponseSt
atusText=""
                            set tSC=obj.%Save() if 'tSC $$$DebugLog($username,"ResendDateRange","Cannot save Message with ID:
"_id_" Error: "_$$$GetErrorText(tSC),.dSC) continue
                    }
            }
            catch ex {set tSC=ex.AsStatus()}
            $$$DebugLog($username,"ResendDateRange","Resend Message Queue Date Range Status:
"_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
            quit tSC
}

}
```

An Example of a Message Queue Class for a Specific Interface

Note that the Package Name has been modified to indicate that this class is not in the Common Package and the Name indicates which Interface it is related to. Notice that I create the Indices on the three TimeStamp Fields and I have Indexed the Patient Identifiers as I may need to locate a specific Patient to do a ReSendMessage() for example.

```
Include DFIInclude

/// This Message Queue Class is used for the Trickle feed from a Source Database to a
/// Target FHIR Server. The Identifiers for the Patient are the PatientId, PatientInternalNumber
/// and the PatientHID (Unique Hospital Identifier). Note that there are two Message Queue
/// classes, one for UAT and one for PRD (Propduction). Each Interface requires a unique
/// Global Name and Class Name therefore I have differentiated them by appenting the characters
/// PRD or UAT tp the Clas and Global Names
Class DFI.BulkExport.Queue.BulkExportMessageQueuePRD Extends (%Persistent, DFI.Common.Abstract.MessageQueueBaseProperties,
DFI.Common.Abstract.MessageQueueBaseMethods, %XML.Adaptor, %ZEN.DataModel.Adaptor)
{

/// The GlobalName Parameter specifies the Global Names to be used in this class. IRIS
/// will generate a Abstracted Global Name and that complicates Namespace Global Mapping
/// If the storage is ever re-created then the Storage Defination must be modified and
/// the global names changed to use this value for the D, I and S Globals
Parameter GlobalName = "^DFI.BulkExport.MsgQueuePRD*";

Property PatientId As %String;
```

```
Property PatientInternalNumber As %Integer;

Property PatientHID As %String;

Index CTS On CreateTS;

Index PTS On ProcessTS;

Index CPTS On CompletedTS;

Index PID On PatientId;

Index PIN On PatientInternalNumber;

Index pHID On PatientHID;

ClassMethod BulkExportPatientListExport(ByRef filename As %String) As %Status
{
   Set tSC = $$$OK,total=0,completed=0,fileopen=0,totalerror=0,totalok=0
   Try {
      Set file = "/usr/cache/mgr/dfi_general_files/Bulk_Export_Patient_List "_$Translate($ZDatetime($Horolog,3),"-: ","")_".csv"
      Open file:("WNS"):0
      Else  set tSC=$$$ERROR(5001,"Cannot Open File: "_file) quit
      Write !,file,!
      Write !,"Starting",!
      Set fileopen = 1
      Set rs = ##class(%ResultSet).%New("%DynamicQuery:SQL")
      Set sql = "SELECT ID, CreateTS, ProcessTS, CompletedTS, PatientID, PatientInternalNumber, PatientHID, HL7ACKCode,
HL7NACKMessage, HTTPStatus, Status FROM "_tTable
                        write !,"SQL: ",sql
      Set tSC = rs.Prepare(sql) if 'tSC quit
      Set tSC = rs.Execute() if 'tSC quit
      Use file write "RowID",$Char(9),"Created TS",$Char(9),"Processed TS",$Char(9),"Completed TS",$Char(9),"Patient ID",$c(9),"Patient
Internal#",$c(9),"Patient HPRN",$c(9),"HL7 ACK Code",$c(9),"HL7 NACK Message",$c(9),"HTTP Status",$c(9),"Status",!
      While rs.Next(.tSC) {
         Quit:'tSC
         Use file
write rs.Data("ID"),$Char(9),rs.Data("CreateTS"),$Char(9),rs.Data("ProcessTS"),$Char(9),rs.Data("CompletedTS"),$Char(9),rs.Data("PatientId
"),$c(9),rs.Data("PatientInternalNumber"),$c(9),##class(DFI.Common.Utility.Functions).ConvertHIDtoHPRN(rs.Data("PatientHID")),$c(9)
         use file write rs.Data("HL7ACKCode"),$c(9),rs.Data("HL7NACKMessage"),$c(9),rs.Data("HTTPStatus"),$c(9),rs.Data("Status"),!
         Set total = total+1 if $Length(rs.Data("CompletedTS")) set completed=completed+1
         if rs.Data("Status")="OK" {set totalok=totalok+1}
         else {set totalerror=totalerror+1}
      }
      Use file w !
      Use file w !,"Total Records",$Char(9),total
      Use file w !,"Total Completed Records",$Char(9),completed
      use file w !,"Total Records Status OK",$c(9),totalok
      use file w !,"Total Records Status NOT OK",$c(9),totalerror
      use file w !,"<ENDOFFILE>"
   }
   Catch ex { Set tSC=ex.AsStatus() }
   If fileopen close file
   use 0 Write !,"Finished. Status: "_$Select(tSC:"OK",1:$$$GetErrorText(tSC)),!
   Quit tSC
}
}
```

The Business Service

The Business Service that Processes the queue is very simple. All it does is get the next Message from the Message Queue and pass it to the Business Process.

```
Class DFI.Common.Service.ProcessMessageQueue Extends Ens.BusinessService
{

Parameter ADAPTER = "Ens.InboundAdapter";

Property Adapter As Ens.InboundAdapter;

Property MaxNumberOfLoops As %Integer [ InitialExpression = 200 ];

Property DFIQueueClassName As %Persistent;

Parameter SETTINGS = "MaxNumberOfLoops:Basic,DFIQueueClassName:Basic";

Method OnProcessInput(pInput As %RegisteredObject, Output pOutput As %RegisteredObject = "") As %Status
{
	set tSC=$$$OK
	try {
		set tConfigClass=$$$GetConfig(.tSC) quit:'tSC set tOS=$$$GetOS(.tSC) quit:'tSC
		set tSC=$classmethod(tConfigClass,"GetConfigurationSettings",tConfigClass,.tConfig,.tSettings) if 'tSC quit
		$$$TRACE("Message Queue Class: "_$g(tSettings("DFIMessageQueueClassName")))
		$$$TRACE("Production Name: "_$g(tSettings("DFIProductionName")))
		$$$TRACE("DFIRequestMessageClassName: "_$g(tSettings("DFIRequestMessageClassName")))
		$$$TRACE("DFIResponseNessageClassName: "_$g(tSettings("DFIResponseMessageClassName")))
		$$$TRACE("Business Process Name: "_$g(tSettings("DFIPrimaryBusinessProcessName")))
		set tQueueClassName=$g(tSettings("DFIMessageQueueClassName")) if '$l(tQueueClassName)
{set tSC=$$$ERROR(5001,"The Message Queue Class Name is Null") quit}
		$$$TRACE("Message Queue Class Name: "_tQueueClassName)
		set tPrimaryBP=$g(tSettings("DFIPrimaryBusinessProcessName")) if '$l(tPrimaryBP) {set tSC=$$$ERROR(5001,"The
Business Process Name is not Specified") quit}
		$$$TRACE("Primary Business Process: "_tPrimaryBP)
		set tRequestClassName=$g(tSettings("DFIRequestMessageClassName")) if '$l(tRequestClassName)
{set tSC=$$$ERROR(5001,"The Request Message Class Name is Undefined") quit}
		set tResponseClassName=$g(tSettings("DFIResponseMessageClassName")) if '$l(tResponseClassName)
{set tSC=$$$ERROR(5001,"The Response Message Class Name is Undefined") quit}
		$$$TRACE("Request Class Name: "_tRequestClassName_" Response ClassName: "_tResponseClassName)
		for i=1:1:..MaxNumberOfLoops {
			$$$TRACE("Loop Counter: "_i)
			set tSC=$classmethod(tQueueClassName,"GetNextMessage",.pMessageId) if 'tSC quit
			if '$l($g(pMessageId)) {$$$TRACE("There is no message in the Queue") continue}
			// Have the option of directing messages to different Business Processes depending
			// on the Transaction Type and Log Type as per the Main BPL in the ODS Production
			// That class is called: BusinessProcesses.TransactionRouter.BPL and it routes
			// messages from the source to specific Business Processes that handle the PMI,
			// Admissions and Discharges, Transfers and so on.
			$$$TRACE("Queue Message: "_pMessageId_" found")
			set tRequest=$classmethod(tRequestClassName,"%New"),tRequest.MessageId=pMessageId
			set tResponse=$classmethod(tResponseClassName,"%New"),tResponse.MessageId=pMessageId
			set tSC=..SendRequestSync(tPrimaryBP,tRequest,.tResponse,,"Sending Async Request for Message ID:
"_pMessageId) if 'tSC quit
			$$$TRACE("Message: "_pMessageId_" Sent OK")
		}
	}
	catch ex {set tSC=ex.AsStatus()}
	if 'tSC {set tResponse.ResponseStatus=tSC}
	set msg="Service Outcome: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)) $$$DebugLog($username,"OnProcessInput",msg,.dSC)
$$$TRACE(msg)
	quit tSC
}
}
```

Environment and Configuration Classes

I defined two configuration classes, identical in every sense however, as there are many directory references, I decided to create a Windows Version and another for UNIX. The Configuration class has a single object with a Configuration ID set to "Settings". My diagram implies that there could be two Interfaces within one Namespace, and I will discuss that possibility later.

After much thought, I decided to add an Environment Class, which also has a single Object. The Environment class has four fields:

1) The Production Class Name
2) The Production Namespace
3) The Operating System
4) The Configuration Class Name

Apart from the Environment and Configuration classes that enforce a single instance by specifying a unique value for the Environment Object RowId, the Configuration class enforces a single object in the same way. There is not a single class in the DFI model that uses hardcoded values. Every detail that the Interface needs to know is specified in the Configuration Settings Object. The Configuration Object gives the Interface Context.

Every method or class method that needs to know what it is doing starts with the following statement:

```
Set tOS=$$$GetOS(.tSC) quit:'tSC  set tConfigName=$$$GetConfig(.tSC) quit:'tSC
Set tConfig=$classmethod(tConfigName,"%OpenId","Settings") if '$IsObject(tConfig) {set
tSC=$$$ERROR(5001,"Configuration Settings are not Defined" quit}
```

If you are going to reference many properties in the Configuration Object, you can put a #DIM statement into the code as follows:

```
#DIM tConfig as DFI.Common.Configuration.UNIXConfigSettings
```

Once the Interface is deployed, you can strip away the #DIM statement. The reason to include it during development is to allow the studio/VS Code to list the properties of the Configuration.

The Configuration Object tells you the Ensemble Request and Response message class names that pass the MessageId to the Business Process. The Production Item Names and underlying Class Names are defined in the Configuration. Once again, $classmethod() is used to instantiate the Request and Response message objects.

The Business Service use the name of the Business Process from the Configuration to know what value to pass in the ..SendRequestSync() call that passes the Request message to the Business Process. The Business Process gets the HTTP Operation Production Name, and if the flag SendToFile flag is set to TRUE, the File Operation Production Item Name.

Sending the HL7 or FHIR Request Messages to file and likewise, the response messages is helpful during testing so that you can check the output of the Data Transform invoked by the Business Process.

Here is the UNIX configurations Class Properties

```
Class DFI.Common.Configuration.UNIXConfigSettings Extends %Persistent
{

/// For the purposes of Global Mappings the Global Names in the Storage Definition have been modified to
/// be more readable than the Ensemble generated Global Names. This Parameter informs the developer that
/// should they delete the Storage Definition then they should replace the Glabal Names with the Value in
/// the Parameter where the * = D, I or S
Parameter GlobalName = "^DFI.Common.UNIXConfigSettings*";

/// The DFIConfigurationID is indexed to be the Primary Key/ID Key with a value of "Settings". The logic of the UpdateConfiguration()
/// Method ensures that only the object with an ID of "Settings" will be created or updated
Property DFIConfigurationID As %String [ InitialExpression = "Settings", Required ];

/// The namespace in which the Interface is running.
Property DFINamespace As %String [ InitialExpression = {$namespace} ];

/// The Production Name is the Name of the Interface Production. The Production Name is used in the methods in this class that use the
/// Ens.Director class to perform Production Actions Start, Stop and Update. This is a Class Name.<BR>
Property DFIProductionName As %String(MAXLEN = 200) [ InitialExpression = "DFI.Common.Production.DFIInterfaceProduction", Required ];

/// The Version number of the Interface.
Property DFIInterfaceVersion As %String [ InitialExpression = "V1.0.0" ];

/// This flag indicates if the Production is the 'Live' production.
Property DFIIsProductionInterface As %Boolean [ InitialExpression = 0 ];

/// Is the Production Active.
Property DFIIsProductionActive As %Boolean [ InitialExpression = 1 ];

/// The MessageQueueClassName is used by the Production Classes that call the methods in the
/// Message Queue Class.
Property DFIMessageQueueClassName As %String [ InitialExpression = "DFI.Common.Queue.ODSTFMessageQueue", Required ];

/// The Data Source is the Table or Request Message in the Source Database that is
/// used to retrieve the data record we wish to process
Property DFIDataSource As %String(VALUELIST = ",ODSGeneral,ODSPatient,EMCIMaster,EMCICopy,IHIS{Module}") [ InitialExpression =
"ODSPatient" ];

/// The Request Class Name that is created by the Buisness Service.
Property DFIRequestMessageClassName As %String(MAXLEN = 100) [ InitialExpression = "DFI.Common.Messages.TransactionRequest" ];

/// The Response Class Name that is returned to the Business Service.
Property DFIResponseMessageClassName As %String(MAXLEN = 100) [ InitialExpression =
"DFI.Common.Messages.TransactionResponse" ];

/// The Target Message Type indicates the Target Message type that will be created by the Interface. There are essentially 3 types
/// HL7, FHIR JSON and SQL.
Property DFITargetMessageType As %String(DISPLAYLIST = ",HL7,FHIR,SQL", VALUELIST = ",H,F,S") [ InitialExpression = "H", Required ];

/// The Target HTT or TCP or EMail operation uses SSl/TSL using HTTPS. This directs the Business Process
/// to send the Request Message to either the Secure Operation or the normal operation
Property DFIIsSSLTSLOperation As %Boolean [ InitialExpression = 0 ];

/// The list of FHIR Resource(s) that are managed in the Interface if the Interface is an EMCI or IHIS Data Flow
/// The property is a comma delimited list of Resource Names.
Property DFIFHIRResources As %String(MAXLEN = 1000);

/// If Debugging is True then the $$$DebugLog() calls in the class method code will create Debug Log Records. This is usually only
/// required in Development and QC. It should be False in the Live Production.<BR>
/// The class DFI.Common.Debug.Status also holds a flag indicating if Debugging is turned on or off and there are
/// methods in the DFI.Common.Debug.Logging class that Set or Get the Logging Status. So This property is Calculated
/// and calls the GetDebugOnOff() method in the Debug Logging Class.
Property DFIDebugging As %Boolean [ Calculated, InitialExpression = 1 ];

/// HL7 Properties for HL7 Messages<BR>
/// MSH Receiving Aplication
Property DFIReceivingApplication As %String(MAXLEN = 100) [ InitialExpression = "StandardODSPIXInterface" ];

/// MSH Receiving Facility
Property DFIReceivingFacility As %String(MAXLEN = 100) [ InitialExpression = "WCGDOH" ];

/// MSH Sending Application
Property DFISendingApplication As %String(MAXLEN = 100) [ InitialExpression = "WCGSTDPIXInterface" ];

/// MSH Sending Facility
Property DFISendingFacility As %String(MAXLEN = 100) [ InitialExpression = "WCGDOHODS" ];

/// This is the default Trigger Event of the HL7 message from which the HL7 Message Structure will be determined
Property DFIDefaultEvent As %String [ InitialExpression = "A08" ];

/// If True the generated message will be sent to the HTTP Outbound Operation (HL7 or FHIR JSON)
```

```
Property DFISendMessageHTTP As %Boolean [ InitialExpression = 1 ];

/// If True the generated message will be sent to the HTTPS Outbound Operation (HL7 or FHIR JSON)
Property DFISendMessageHTTPS As %Boolean [ InitialExpression = 0 ];

/// If True the generated message will be written to file (HL7 or FHIR JSON)
Property DFISendMessageToFile As %Boolean [ InitialExpression = 1 ];

/// This is the HL7 HTTP Business Operation Name in the Interface Production
Property DFIHTTPHL7OperationName As %String(MAXLEN = 200) [ InitialExpression = "DFI HL7 HTTP Operation" ];

/// This is the HL7 HTTP Business Operation Class Name in the Interface Production
Property DFIHTTPHL7OperationClassName As %String(MAXLEN = 200) [ InitialExpression =
"DFI.Common.Operation.PIXHL7HTTPOperation" ];

/// This is the HL7 HTTPS Business Operation Name in the Interface Production
Property DFIHTTPSHL7OperationName As %String(MAXLEN = 200) [ InitialExpression = "DFI HL7 HTTPS Operation" ];

/// This is the HL7 HTTPS Business Operation Class Name in the Interface Production
Property DFIHTTPSHL7OperationClassName As %String(MAXLEN = 200) [ InitialExpression =
"DFI.Common.Operation.PIXHL7HTTPSOperation" ];

/// This is the FHIR HTTP Business Operation Name in the Interface Production
Property DFIHTTPFHIROperationName As %String(MAXLEN = 200) [ InitialExpression = "DFI FHIR HTTP Operation" ];

/// This is the FHIR HTTP Business Operation Class Name in the Interface Production
Property DFIHTTPFHIROperationClassName As %String(MAXLEN = 200) [ InitialExpression =
"DFI.Common.Operation.FHIRHTTPOperation" ];

/// This is the FHIR HTTPS Business Operation Name in the Interface Production
Property DFIHTTPSFHIROperationName As %String(MAXLEN = 200) [ InitialExpression = "DFI FHIR HTTPS Operation" ];

/// This is the FHIR HTTPS Business Operation Class Name in the Interface Production
Property DFIHTTPSFHIROperationClassName As %String(MAXLEN = 200) [ InitialExpression =
"DFI.Common.Operation.FHIRHTTPSOperation" ];

/// This is the HL7 File Business Operation Name in the Interface Production
Property DFIFileHL7OpertionName As %String(MAXLEN = 200) [ InitialExpression = "DFI HL7 File Operation" ];

/// This is the HL7 File Business Operation Class Name in the Interface Production
Property DFIFileHL7OpertionClassName As %String(MAXLEN = 200) [ InitialExpression = "DFI.Common.Operation.PIXHL7FileOperation" ];

/// This is the FHIR File Business Operation Name in the Interface Production
Property DFIFileFHIROperationName As %String(MAXLEN = 200) [ InitialExpression = "DFI FHIR File Operation" ];

/// This is the FHIR File Business Operation Class Name in the Interface Production
Property DFIFileFHIROperationClassName As %String(MAXLEN = 200) [ InitialExpression = "DFI.Common.Operation.FHIRFileOperation" ];

/// This is the Interface HouseKeeping Business Service Name
Property DFIHouseKeepingServiceName As %String(MAXLEN = 200) [ InitialExpression = "DFI HouseKeeping Service" ];

/// This is the Interface HouseKeeping Business Service Class Name
Property DFIHouseKeepingClassName As %String(MAXLEN = 200) [ InitialExpression = "DFI.Common.Service.HouseKeeping" ];

/// This is the Interface Alert Notification Service Name
Property DFIAlertNotificationServiceName As %String(MAXLEN = 200) [ InitialExpression = "DFI Alert Notification Service" ];

/// This is the Interface Alert Notification Service Class Name
Property DFIAlertNotificationServiceClassName As %String(MAXLEN = 200) [ InitialExpression =
"DFI.Common.Service.AlertNotificationService" ];

/// The Ensemble Ens Alert Monitor Production Item Name.
Property DFIAlertMonitorServiceName As %String(MAXLEN = 200) [ InitialExpression = "Ens Alert Monitor" ];

/// The Ensmeble Ens Alert Monitor Class Name.
Property DFIAlertMonitorServiceClassName As %String(MAXLEN = 200) [ InitialExpression = "Ens.Alerting.AlertMonitor" ];

/// ----------------------------------------------------------------------------------------------------------------<br>
/// Settings for the Bulk Export of Patients from ODS to EMCI Settings<BR>
/// ----------------------------------------------------------------------------------------------------------------<BR><br>
///
/// This is the BulkExport BuildPatientListService ClassName. This is the service that Builds the List
/// of Patients that will be Exported to EMCI before the Patient to EMCI Trckle Feed Interface takes over
Property DFIBuildPatientListServiceName As %String [ InitialExpression = "DFI Build Patient List Service" ];

/// This is the Bulk Export Build Patient List Service Class Name
Property DFIBuildPatientListServiceClassName As %String [ InitialExpression = "DFI.BulkExport.Service.BuildPatientListService" ];

/// This is the Queue Class Name of the Patient List created by the Build Patient List Service in the Bulk Export
/// Interface. This Queue is then processed by the DFI.BulkExport.Service.BulkExportSendPatient Service
/// will use and for each entry it will create an Entry in the BulkExportMessageQueue that tracks the
```

```
/// Sending of the Patients to EMCI.
Property DFIBulkExportQueueClassName As %String [ InitialExpression = "DFI.BulkExport.Queue.BulkExportMessageQueue" ];

/// This is the Service Name of the Bulk Export Send Patients Service that kicks in when the Build Patient
/// List Service is complete.
Property DFIBulkExportSendPatientsServiceName As %String [ InitialExpression = "DFI Send Patients Service" ];

/// This is the Service Class Name for the Bulk Export
/// Send Patient Service Name
Property DFIBulkExportSendPatientServiceClassName As %String [ InitialExpression = "DFI.BulkExport.Service.SendPatientsService" ];

/// This is the name of the Bulk Export Send Patient Business Process Name
Property DFIBulkExportSendPatientProcessName As %String [ InitialExpression = "DFI Send Patient Process" ];

/// This is the Class Name of the Bulk Export Send Patient Process Class Name
Property DFIBulkExportSendPatientProcessClass As %String [ InitialExpression = "DFI.BulkExport.Process.SendPatientProcess" ];

/// This Property is used by the ODS to EMCI Bulk Export. The Bulk Export builds a Message Queue
/// of Patient Internal Numbers. Once the List is Built then the Service that Builds the List disables
/// itself and the Service to process the Bulk Export List is started and will process the Bulk Export List
/// and for each message a request is sent to the Business Process that converts the ODS Patient into
/// a FHIR Patient Document and sends it to the EMCI FHIR Server
/// This property is updated by the Build Patient List Service
Property DFIBulkExportPatientListComplete As %Boolean [ InitialExpression = 0 ];

/// This is the Number of Patients processed by the ODS to EMCI FHIR Bulk Export
/// This property is updated by the Build Patient List Service
Property DFIBulkExportPatientRecordsProcessed As %Integer [ InitialExpression = 0 ];

/// This property can be used to control how many Patients are selected for export in the
/// ODS to EMCI FHIR Bulk Export and is referenced by the Build Patient Lisr Service. Once
/// the Number of Patient Records that have been Processed either hits the end of File or
/// it reaches the Number to be Selected then the List is Complete and the Build Patient List
/// will effectively stop and if the BulkExportStartSendAutomatically is TRUE then it will
/// start sending the Patients to EMCI
Property DFIBulkExportNumberOfPatientsToBeSelected As %Integer [ InitialExpression = 18000000 ];

/// If StartBulkExportAutomatically is TRUE then the Business Service, DFI.EMCIBulkExport.Service.BulkExportSendPatients,
/// that starts transmitting the Patients that the service DFI.EMCIBulkLoad.Service.BulkExportPatientList
/// has added to the DFI.EMCIBulkExport.Queue.BulkExportList, will start transmitting the Patients.<br>
/// If the setting is False then the DFI.EMCIBulkExport.Service.BulkExportSendPatients service will just loop until
/// until the setting DFOBuildListOfPatientsIsComplete is TRUE
Property DFIBulkExportStartSendAutomatically As %Boolean [ InitialExpression = 0 ];

/// The Start Patient is the last patient that was processed by the BuildPaientList Service. This is the seed
/// value from where the next cycle of building the Patient List. The Patient List Build Service can
/// have a Call Interval of no less than 0.1second so the OnProcessInput method has an
/// inner loop and it will process say 200 records in a loop before exiting the service only to be called
/// again 0.1 seconds later.<br>
/// When the Service is invoked again it will pick up from where it left off<br>
/// This property is updated by the Build Patient List Service
Property DFIBulkExportStartPatient As %Integer [ InitialExpression = 0 ];

/// ----------------------------------------------------------------------------------------------------------------------<br>
/// End of Bulk Export of Patients from ODS to EMCI Settings<BR>
/// ----------------------------------------------------------------------------------------------------------------------<BR><br>
/// This is the Default Business Service Name for the Business Service that processes the Message
/// Queue that drives the Interface. Messages are created in the Message Queue based on activity in
/// an Interface running in another Namespace. For example the Clinicom to ODS Trickle Feed creates
/// messages whenever an Data Event occurs in Clinicom and a TransactionLog Request Message is processed
/// in the Trickle Feed Production.<BR><br>
/// Messages can also be created through a UI or programmatically through a method Call.<BR>
/// The Message Queue Classes all have the same generic methods CreatMessage(), GetNextMessage()
/// UpdateMessageQueue(), CompleteMessage(), ResendMessage(), PurgeMesssages().<BR><br>
/// Any class that can be Inherited or Copied must use $classmethod() to run mmethods within
/// that Class and $Property() to Set or Get Property Names and Values. In particular
/// the Properties in a Message Queue Class that are specific to that Message Queue are
/// passed in, or retrieved from, are passed as an array of values to the methods that Create,
/// Update or Retrieve these message specific properties.<BR><br>
/// There are properties in every Message Queue Class that are controlled by the Message Queue Class
/// and cannot be updated by those methods. There are Messge Queue Properties that are generic to
/// every message queue class (and influenced by the type of message and the transport mechanism
/// that sends the message body to a target application. These properties can be updated by a UI
/// or through the {Property Name}/{Property Value} Arrray.<BR><br>
/// These properties are enerally related to the JSON, HL7, Text, XML Body Content and the
/// File Directory and File Name properties used to create the files into which the Body Contents
/// are written. This applies to both Request and Response Messages. An HL7 Request Message will
/// have a complimentary HL7 Response Message in the form of an HL7 ACK Message<BR><br>
/// A FHIR JSON Request Message (if there is one) will have a corresponding FHIR JSON Response Message
/// which will typically be the JSON for a FHIR Resource, a FHIR Bundle of One or More FHIR Resource
/// JSON Objects.<BR>
```

```
Property DFIMessageQueueServiceName As %String [ InitialExpression = "DFI Message Queue Service" ];

/// This is the Message Queue Service Class Name.
Property DFIMessageQueueServiceClassName As %String [ InitialExpression = "DFI.Common.Service.ProcessMessageQueue" ];

/// These Production Items are effectively copies of the Default Message Queue Service and if it helps
/// make the production contents more Readable then use this field.
Property DFIAltODSMessageQueueServiceName As %String [ InitialExpression = "DFI ODS Message Queue Service" ];

/// These Production Items are effectively copies of the Default Message Queue Service and if it helps
/// make the production contents more Readable then use this field.
Property DFIAltODSMessageQueueServiceClassName As %String [ InitialExpression = "DFI.Common.Service.ProcessMessageQueue" ];

/// These Production Items are effectively copies of the Default Message Queue Service and if it helps
/// make the production contents more Readable then use this field.
Property DFIAltEMCIMessageQueueServiceName As %String [ InitialExpression = "DFI EMCI Message Queue Service" ];

/// These Production Items are effectively copies of the Default Message Queue Service and if it helps
/// make the production contents more Readable then use this field.
Property DFIAltEMCIMessageQueueServiceClassName As %String [ InitialExpression = "DFI.Common.Service.ProcessMessageQueue" ];

/// The Main Busines Process Name. The Primary Business Process is called by the "DFI Message Queue Service"
/// The Business Service looks at the configuration settings to get the name of the Message Queue
/// and then calls the GetNextMessage() method of that class using $Classmethod(). If the method returns
/// a message the Service then uses the Configuration Property "DFIRequestMessageClassName" to
/// create a new Request Message which it passes to the DFI Primary Bussiness Process.
Property DFIPrimaryBusinessProcessName As %String(MAXLEN = 100) [ InitialExpression = "DFI WCG HL7 STD Process" ];

/// The Primary Business Process Class Name. This is the underlying class name of the DFI Primary Business
/// Process Production Item Name (above).
Property DFIPrimaryBusinessProcessClassName As %String(MAXLEN = 100) [ InitialExpression =
"DFI.Common.Process.PrimaryBusinessProcess" ];

/// The File Directory where EMCI/IHIS FHIR JSON Files will be written. This is for Files that contain
/// the Request and Response FHIR JSON Content for EMCI and IHIS Interfaces.<br>
/// See the notes on the DFIFileDirectory property for additional notes.
Property DFIFHIRFileDirectory As %String(MAXLEN = 100) [ InitialExpression = "/usr/cache/mgr/dwprd/DFI-Codebase-
PRD/DFI/Files/FHIRFiles/Out/" ];

/// The File Directory where EMCI/IHIS FHIR JSON Files will be written. This is for Files that contain
/// the Request and Response FHIR JSON Content for EMCI and IHIS Interfaces.<br>
/// See the notes on the DFIFileDirectory property for additional notes.
Property DFIHL7FileDirectory As %String(MAXLEN = 100) [ InitialExpression = "/usr/cache/mgr/dwprd/DFI-Codebase-
PRD/DFI/Files/HL7Files/Out/" ];

/// The File Directory where Files will be written. This is for any Files that are generated by
/// the Interface. There other File Directory Property Names specific to FHIR, HL7 and Manifests
/// that can be used to override this setting. It is the responsibility of the Business Process
/// or other Production Item that calls the File Operation to decide whether an alternative directory
/// Name is to be used based on the desired functionality of the Interface Production<br>
/// There are Directories for the Interface QC Namespace and the Production
/// Namespace. So the naming of this field should take into consideration the setting:<br><br>
/// DFIIsProductionInterface=1 (true) or 0 (false)<br><br>
/// that indicates whether the Interface is running in Production or not.<br>
/// Again, the files from both QC and PRD can be written to the same directory if required.
Property DFIFileDirectory As %String(MAXLEN = 200) [ InitialExpression = "/usr/cache/mgr/dwprd/DFI-Codebase-DT-
PRD/DFI/Files/GeneralFiles/" ];

/// The default name for EMCI/ODS HL7 ADT Request Messages.
Property DFIHL7RequestFileName As %String(MAXLEN = 3000) [ InitialExpression = "HL7 Request for Message {MessageId} Patient
{PatientId} run on {TimeStamp}.txt" ];

/// The default name for EMCI/ODS HL7 ACK Response Messages.
Property DFIHL7ResponseFileName As %String(MAXLEN = 3000) [ InitialExpression = "HL7 Response for Message {MessageId} and Patient
{PatientId} run on {TimeStamp}.txt" ];

/// The default name for EMCI FHIR Patient JSON Request Messages.
Property DFIFHIRResourceRequestJSONFileName As %String(MAXLEN = 3000) [ InitialExpression = "FHIR Request for Message
{MessageId} and Patient {PatientId} and ResourceID {ResourceId} run on {TimeStamp}.json" ];

/// The default name for EMCI FHIR Patient JSON Response Messages Typically a Resource, a Bundle or an Operation
/// Outcome.
Property DFIFHIRResourceResponseJSONFileName As %String(MAXLEN = 3000) [ InitialExpression = "FHIR Response for Message
{MessageId} for Patient {PatientId} and ResourceID {ResourceId} run on {TimeStamp}.json" ];

/// The File Directory where Manifest Files will be created. This is the Directory where Test Files
/// are written. The HTTP Request JSON files and HTTP Response JSON files are stored here. There is
/// a Manifest Directory for both a QC Interface Namespace and a PRD Interface Namespace so the setting
/// 'DFIIsProductionInterface' = 1 (true) that indicates that the Interface is running in a Production Namespace
Property DFIManifestFileDirectory As %String(MAXLEN = 200) [ InitialExpression = "/usr/cache/mgr/dwprd/DFI-Codebase-DT-
PRD/DFI/Files/ManifestFiles/Out/" ];
```

```
/// The default File Name for Manifest Files. The fields enclosed in {} are substituted at runtime
/// with specific details for the resultant filename.
Property DFIManifestFileName As %String(MAXLEN = 3000) [ InitialExpression = "Data Load Manifest {ManifestId} from Patient
{FromConsumerId} to Patient {ToConsumerId} for Test {TestId} run at {Date}.csv" ];

/// This is the file name into which the JSON or HL7 or other document type is written when the
/// Business Process that uses Manifests sends the Request Document to the HTTP/TCP/Email/File Outbound
/// Business Operstion. This is configured for a FHIR Request/Response HTTP Request. It will be
/// Modified when the Configuration Settings are created for new DFI Interface,
Property DFIManifestRequestRecordFileName As %String(MAXLEN = 3000) [ InitialExpression = "EMCI Patient {EMCIUId} FHIR Request -
Manifest {ManifestId} - Record {RecordId} - Test {TestId} run on {TimeStamp}.json" ];

/// This is the File Name into which the JSON or HL& or other Document Type that is received back from
/// the Target Application. For example: An HL7 ACK Message, a FHIR Interaction Operation Outcome.
/// This is configured for a FHIR Request/Response HTTP Request. It will be Modified when the
/// Configuration Settings are created for new DFI Interface that uses HL7 for example.
Property DFIManifestResponseRecordFileName As %String(MAXLEN = 3000) [ InitialExpression = "EMCI Patient {EMCIUId} FHIR Response
- Manifest {ManifestId} - Record {RecordId} - Test {TestId} run on {TimeStamp}.json" ];

/// This is the default Email Sender Email addresss. This is used when Notifications or
/// other Emails are sent to targeted lists of Recipients.
Property DFIEMailSenderAddress As %String(MAXLEN = 100) [ InitialExpression = "nigel@healthsystems.co.za" ];

/// This is the default Target List of Email Recipients that will be used to send EMails to. There are
/// specific EMail Lists for different situations and if those lists are not populated then this list
/// will be used instead. The list is a string of Email addresses seperated by a "," or ";'
/// This list should not be used for any of the Alert Notification Lists as they are
/// specified seperately. This list would include people who receive Manifest Reports
/// or other Ssystem Reports
Property DFIDefaultEmailList As %String(MAXLEN = 1000) [ InitialExpression = "nigel.salm@icloud.com", Required ];

/// The DFIAlertNotificationEmailList is the default list of Recipients to receive
/// Alert Notifications. The ProductionAlerts class is a list of Production Items
/// with the Production Item Name as the PrimaryKey in the ProductionAlerts Table. At the
/// Production Item Level the only attributes that are monitored are the Item Queue Size
/// and the ProductionItem Status. If the Queue Size exceeds a certain size then we
/// have the option of sending an Alert Notification. Likewise if the Item Status is
/// 'InError" and likewise an Alert Notification could be sent. This email list
/// is the list of recipients that should receive these notifications.
/// By default this list is inherited into the ErrorAlert Notification List and
/// the ConditionAlert Notification List.
Property DFIAlertNotificationEmailList As %String(MAXLEN = 1000) [ InitialExpression = "nigel@healthsystems.co.za", Required ];

/// This is the Target List of Email Recipients for Alert Error Notirfications.
Property DFIErrorAlertEmailList As %String(MAXLEN = 1000) [ InitialExpression = "nigel.salm@outlook.com" ];

/// This is the Target list of Email Recipients for Alert Condition Notifications.
Property DFIConditionAlertEmailList As %String(MAXLEN = 1000) [ InitialExpression = "nigel.salm@gmail.com" ];

/// The Number of days that the messages in the Message Queue are retained.
Property DFINumberOfDaysToKeepQueueMessages As %Integer [ InitialExpression = 90 ];

/// The Number of days to retain the Ensemble Messages.
Property DFINumberOfDaysToKeepEnsembleMessages As %Integer [ InitialExpression = 90 ];

/// The Number of days that the Ensemble Logs are retained.
Property DFINumberOfDaysToKeepEnsembleLogs As %Integer [ InitialExpression = 90 ];

/// The Number of days that the Debug Logs are retained.
Property DFINumberOfDaysToKeepDebugLogs As %Integer [ InitialExpression = 90 ];

/// The Number of days that files are retained.
Property DFINumberOfDaysToKeepFiles As %Integer [ InitialExpression = 90 ];

Index PK On DFIConfigurationID [ IdKey, PrimaryKey, Unique ];

/// This method returns the 'true'/'false' flag that determines whether Debugging is Turned On or Off.
Method DFIDebuggingGet() As %Boolean
{
	quit ##class(DFI.Common.Debug.Status).GetDebugOnOff(.tSC)
}

/// Thsi method will return an array of Configuration Property Names. If the method is passed an alternative
/// classname then it will get the properties of that class. This is used for any Configuration
/// Property that points to a Code table or other DFI class. Lists and Arrays are also catered for. Refere
/// to the main class documentation to see the specifics of how the the pSettings array is specified.
ClassMethod GetConfigurationSettings(pClassName As %String = {$classname()}, ByRef pClassObject As %RegisteredObject = "",
ByRef pSettings As %String(MAXLEN=3000)) As %Status
{
	set tSC=$$$OK
```

```
	try {
		set pClassName=$$$GetConfig(.tSC) quit:'tSC if '$l(pClassName) set pClassName=$Classname()
		if '$IsObject(pClassObject) {
			set pClassObject=$classmethod(pClassName,"%OpenId","Settings")
			if '$IsObject(pClassObject)
{set pClassObject=$classmethod(pClassName,"%New"),pClassObject.DFIConfigurationID="Settings",tSC=pClassObject.%Save() if 'tSC quit}
		}
		set tCompClass=##class(%Dictionary.CompiledClass).%OpenId(pClassName)
		if '$IsObject(tCompClass) set tSC=$$$ERROR(5001,"Unable to open Compiled Class Definition for '"_$classname()_"'")
quit
		set (tKey,tProp)="" for {
			set tProp=tCompClass.Properties.GetNext(.tKey) quit:tKey=""  if $e(tProp.Name,1)="%" continue
			if '$IsObject(tProp) $$$DebugLog($username,"Configuration:GetProperties","Property for Property Key:
"_tKey_" is not an object",.dSC) continue
			set tName=tProp.Name,tType=tProp.Type
			// Deal with collections
			if tType["%List",tType["%Array",tType["%Collection" {
				if $IsObject(pClassObject) {
					set tCollection=$property(pClassObject,tName)
					set cKey="" for {
						set cItem=tCollection.GetNext(.cKey) quit:cKey=""
						set pSettings(tName,"Values",cKey)=cItem
					}
				}
				else {set pSettings(tName)=""}
				set pSettings(tName,"Type")="%List"
				continue
			}
			elseif $p(tType,".",1)="DFI" {
				kill tSettings

	set tSC=$classmethod($classname(),"GetClassProperties",tName,$s($IsObject(pClassObject):$property(pClassObject,tName),1:""),
.tSettings) if 'tSC quit
				Merge pSettings(tName,"Properties")=tSettings
				set pSettings(tName,"Type")="DFI"
			}
			else {
				set pSettings(tName)=$s($IsObject(pClassObject):$property(pClassObject,tName),1:"")
				set pSettings(tName,"Type")="DT"
				continue
			}
		}
	}
	catch ex {set tSC=ex.AsStatus()}
	$$$DebugLog($username,"GetConfigurationSettigs","The Status of Get Configuration Settings is:
"_$s(tSC:$$$OK,1:$$$GetErrorText(tSC)),.dSC)
	quit tSC
}

ClassMethod GetClassProperties(pClassName As %String = {$classname()}, ByRef pClassObject As %RegisteredObject,
ByRef pSettings As %String(MAXLEN=5000)) As %Status
{
	set tSC=$$$OK
	try {
		set pClassName=$$$GetConfig(.tSC) quit:'tSC if '$l(pClassName) set pClassName=$Classname()
		if '$IsObject(pClassObject) {
			set pClassObject=$classmethod(pClassName,"%OpenId","Settings")
			if '$IsObject(pClassObject)
{set pClassObject=$classmethod(pClassName,"%New"),pClassObject.DFIConfigurationID="Settings",tSC=pClassObject.%Save() if 'tSC quit}
		}
		set tCompClass=##class(%Dictionary.CompiledClass).%OpenId(pClassName)
		if '$IsObject(tCompClass) set tSC=$$$ERROR(5001,"Unable to open Compiled Class Definition for '"_pClassname_"'")
quit
		set (tKey,tProp)="" for {
			set tProp=tCompClass.Properties.GetNext(.tKey) quit:tKey=""
			if '$IsObject(tProp) $$$DebugLog($username,"Configuration:GetClassProperties","Property for Property Key:
"_tKey_" is not an object",.dSC) continue
			set tName=tProp.Name,tType=tProp.Type
			// Deal with collections
			if tType["%List",tType["%Array",tType["%Collection" {
				if $IsObject(pClassObject) {
					set pCollection=$property(pClassObject,tName)
					set cKey="" for {
						set cItem=pCollection.GetNext(.cKey) quit:cKey=""
						set pSettings(tName,"Values",cKey)=cItem
					}
				}
				else {set pSettings(tName)=""}
				set pSettings(tName,"Type")="%List"
				continue
			}
```

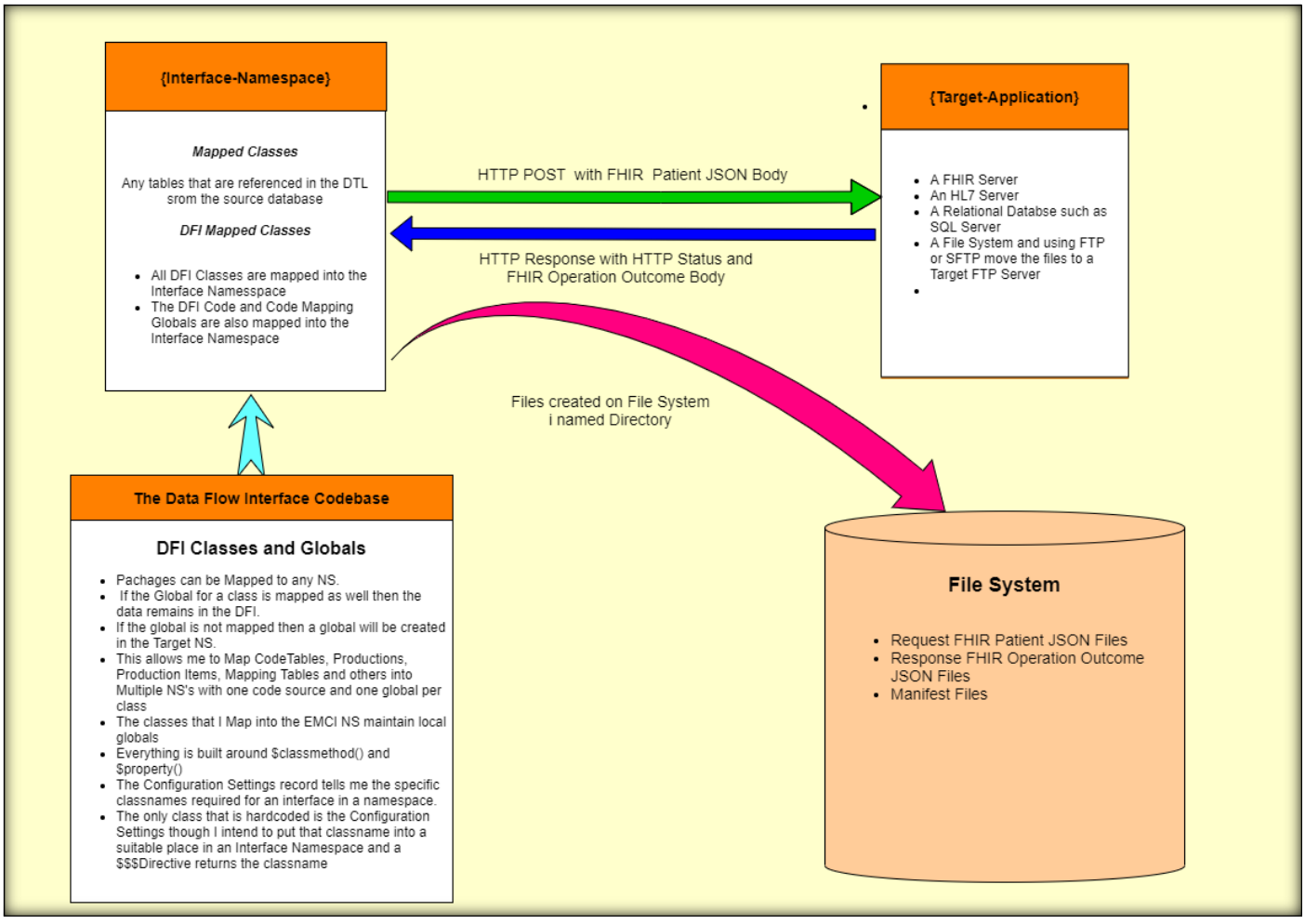
```
                                elseif $p(tType,".",1)="DFI" {
                                        kill tSettings

        set tSC=$classmethod($classname(),"GetClassSettings",tName,$s($IsObject(pClassObject):$property(pClassObject,tName),1:""),.t
Settings) if 'tSC quit
                                        Merge pSettings(tName,"Properties")=tSettings
                                        set pSettings(tName,"Type")="DFI"
                                }
                                else {
                                        set pSettings(tName)=$s($IsObject(pClassObject):$property(pClassObject,tName),1:"")
                                        set pSettings(tName,"Type")="DT"
                                        continue
                                }
                        }
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"GetClassProperties","Get Class Properties Status: "_$s(tSC:"$$$OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This method is called to create or update the "Settings" row in the Configuration Class. Refer to the
/// main class documentation that details the specifics of how Proeprty valus are Updated in this class.
ClassMethod UpdateConfigurationSettings(pClassName As %String = {$classname()}, ByRef pClassObject As %RegisteredObject = "",
ByRef pSettings As %String(MAXLEN=3000), pKillExtent As %Boolean = 0) As %Status
{
        set tSC=$$$OK
        try {
                set pClassName=$$$GetConfig(.tSC) quit:'tSC if '$l(pClassName) set pClassName=$Classname()
                if '$IsObject(pClassObject) {
                        set pClassObject=$classmethod(pClassName,"%OpenId","Settings")
                        if '$IsObject(pClassObject)
{set pClassObject=$classmethod(pClassName,"%New"),pClassObject.DFIConfigurationID="Settings",tSC=pClassObject.%Save() if 'tSC quit}
                }
                if pKillExtent set pClassObject="",tSC=$classmethod(pClassName,"%DeleteExtent") quit:'tSC set pClassObject=""
                set tProp="" for {
                        set tProp=$o(pSettings(tProp)) quit:tProp=""  Continue:tProp="DFIConfigurationID"  continue:$e(tProp,1)="%"
                        write !,tProp
                        // Deal with normal properties
                        if $g(pSettings(tProp,"Type"))="DT" {
                                if $l(pSettings(tProp)) set $property(pClassObject,tProp)=pSettings(tProp)
                                if '$l(pSettings(tProp)),+$g(pSettings(tProp,"Force")) set $property(pClassObject,tProp)=""
                        }
                        elseif $g(pSettings(tProp,"Type"))="DFI" {
                                set tField="" for {
                                        set ttField=$o(pSettings(tProp,"Properties",tField)) quit:tField=""
                                        if $l(pSettings(tProp,"Properties",tField))
set $property($property(pClassObject,tProp),tField)=pSettings(tProp,"Properties",tField)
                                        if '$l(pSettings(tProp,"Properties",tField)),+$g(pSettings(tProp,"Properties",tField,"Force"))
set $property($property(pClassObject,tProp),tField)=""
                                }
                        }
                        elseif $g(pSettings(tProp,"Type"))="%List" {
                                do $classmethod($property(pClassObject,tProp),"Clear")
                                set x="" for {
                                        set x=$o(pSettings(tProp,"Values",x)) quit:x=""
                                        do $classmethod($property(pClassObject,tProp),"InsertAt",pSettings(tProp,"Values",x),x)
                                }
                        }
                }
                set tSC=pClassObject.%Save() if 'tSC quit
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"ConfigurationUpdate","Update Configuration Settings Status: "_$s(tSC:1,1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This Method gets the specific Information about the Interface that is used to create the ODSToInterfaceMapping class
/// that resides in the ODS and is used to list all Interfaces and contains the method that will walk through each Interface
/// definition and from that call the CreateMessage() Method of the DFI.Common.Queue.ODSMessageQueue or
/// a sub-class thereof.
ClassMethod GetMessageQueueData(pClassName As %String = {$classname()}, ByRef pClassObject As %RegisteredObject,
ByRef pProductionName = "", ByRef pNamespace As %String = "", ByRef pQueueClassName As %String = "",
ByRef pIsProduction As %Boolean = 0, ByRef pIsProductionActive As %Boolean = 1) As %Status
{
        set tSC=$$$OK
        try {
                if '$l(pClassName) set pClassName=$$$GetConfig(.tSC) quit:'tSC if '$l(pClassName) set pClassName=$Classname()
                if '$IsObject(pClassObject) {
                        set pClassObject=$classmethod(pClassName,"%OpenId","Settings")
                        if '$IsObject(pClassObject)
{set pClassObject=$classmethod(pClassName,"%New"),pClassObject.DFIConfigurationID="Settings",tSC=pClassObject.%Save() if 'tSC quit}
```

```
                }

        set pProductionName=pClassObject.DFIProductionName,pNamespace=pClassObject.DFINamespace,pQueueClassName=tConfig.
DFIMessageQueueClassName
                set pIsProduction=pClassObject.DFIIsProductionInterface,pIsProductionActive=pClassObject.DFIIsProductionActive
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"GetMessageQueueData","Get Message Queue Data Status: "_$$$GetErrorText(tSC),.dSC)
        quit tSC
}

/// Create entry in the mapped DFI.Common.Interface.ODStoInterfaceMapping class
ClassMethod CreateInterfaceMappingDetails(pClassName As %String(MAXLEN=100) = {$classname()},
ByRef pClassObject As %RegisteredObject) As %Status
{
        set tSC=$$$OK
        try {
                set pClassObject=##class(DFI.Common.Configuration.ConfigurationSettings).%OpenId("Settings")
                if '$l(pClassName) set pClassName=$$$GetConfig(.tSC) quit:'tSC if '$l(pClassName) set pClassName=$Classname()
                if '$IsObject(pClassObject) {
                        set pClassObject=$classmethod(pClassName,"%OpenId","Settings")
                        if '$IsObject(pClassObject)
{set pClassObject=$classmethod(pClassName,"%New"),pClassObject.DFIConfigurationID="Settings",tSC=pClassObject.%Save() if 'tSC quit}
                }

        set tMapping=##class(DFI.Common.Interface.InterfaceMappingDetails).%OpenId(pClassObject.DFIProductionName_"||"_pClassObj
ect.DFINamespace)
                if '$IsObject(tMapping) {
                        set tMapping=##class(DFI.Common.Interface.InterfaceMappingDetails).%New()

        set tMapping.InterfaceName=pClassObject.DFIProductionName ,tMapping.InterfaceNamespace=pClassObject.DFINamespace
                }

        set tMapping.DataSource=pClassObject.DFIDataSource,tMapping.MessageQueueClassName=pClassObject.DFIMessageQueueCl
assName,tMapping.IsProduction=pClassObject.DFIIsProduction,tMapping.IsActive=pClassObject.DFIIsActive
                set tSC=tMapping.%Save() if 'tSC quit
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"CreateMapping","Create Interface Mapping Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This method will Start the Production where the Production Name is derived from the Configuration Settings
ClassMethod StartProduction() As %Status
{
        set tSC=$$$OK
        try {
                set tSC=$classmethod($classname(),"GetConfigurationSettings",$classname(),.tConfig,.tSettings) if 'tSC quit
                if '$l($g(tSettings("DFIProductionName")))
set tSettings("DFIProductionName")=$g(^Ens.Configuration("csp","LastProduction"))
                set tSC=##class(Ens.Director).StartProduction(tSettings("DFIProductionName")) if 'tSC quit
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"Configuration","Start Production Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This method will Stop the Production where the Production Name is derived from the Configuration Settings
ClassMethod StopProduction(pTimeOut As %Integer = 120, pForce As %Boolean = 1) As %Status
{
        set tSC=$$$OK
        try {
                set tSC=##class(Ens.Director).StopProduction(pTimeOut,pForce) if 'tSC quit
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"Configuration","Stop Production Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This method will Update the Production where the Production Name is derived from the Configuration Settings
ClassMethod UpdateProduction(pTimeOut As %Integer = 120, pForce As %Boolean = 1, pCalledByScheduleHandler As %Boolean = 0)
As %Status
{
        set tSC=$$$OK
        try {
                set tSC=##class(Ens.Director).UpdateProduction(pTimeOut,pForce,pCalledByScheduleHandler) if 'tSC quit
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"Configuration","Update Production Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}
```

```
/// This method returns the Production Status.<br>
/// pProductionName: Returns the production name when the status is running, suspended or troubled.<br>
/// pState: Outputs production status. The valid values are:<br>
/// $$$eProductionStateRunning<br>
/// $$$eProductionStateStopped<br>
/// $$$eProductionStateSuspended<br>
/// $$$eProductionStateTroubled<br>
ClassMethod GetProductionStatus(Output pProductionName As %String, Output pState As %Integer, pLockTimeout As %Numeric = 10,
pSkipLockIfRunning As %Boolean = 0) As %Status
{
	set tSC=$$$OK
	try {
		if '$l(pProductionName) {
			set tSC=$classmethod($classname(),"GetConfigurationSettings",.tSettings) if 'tSC quit
			set pProductionName=tConfig.DFIProductionName
		}
		set tSC=##class(Ens.Director).GetProductionStatus(.pProductionName, .pState, pLockTimeout, pSkipLockIfRunning)
if 'tSC quit
	}
	catch ex {set tSC=ex.AsStatus()}
	$$$DebugLog($username,"Configuration","Get Production Status is: "_$s(tSC:"OK",1:"Error: "_$$$GetErrorText(tSC)),.dSC)
	quit tSC
}

/// Call this method to determine which operations are being called and get the
/// filenames for HL7 or FHIR or Manifest Request and Response filenames and
/// Manifest File Name. The array must pass in the substitution values for replacing
/// {Identifier}'s embedded in the file name. In Return you get back the correct HTTP
/// Operation name and the File Operation Name and the file names with the {Identifiers}
/// resolved
ClassMethod GetOperationDetails(pClassName As %String = {$classname()}, pType As %String(VALUELIST=",HL7,FHIR,Manifest,Bulk") =
"", ByRef pOperations As %String(MAXLEN=1000)) As %Status
{
	set tSC=$$$OK
	try {

	set (httpOperation,fileOperation,tClassObject,directory,manifestFileName,requestFileName,responseFileName)="",(sendToHTTP,se
ndToFile,HTTP,HTTPS)=0
		if '$l($g(pType)) set tSC=$$$ERROR(5001,"The Operation Type must be indicated 'HL7,FHIR,Manifest,Bulk'") quit
		#dim tClassObject as DFI.Common.Configuration.ConfigurationSettings
		set pClassName=$$$GetConfig(.tSC) quit:'tSC if '$l(pClassName) {set pClassName=$Classname()}
		if '$IsObject($g(tClassObject)) {
			set tClassObject=$classmethod(pClassName,"%OpenId","Settings")
			if '$IsObject(tClassObject)
{set tClassObject=$classmethod(pClassName,"%New"),tClassObject.DFIConfigurationID="Settings",tSC=tClassObject.%Save() if 'tSC quit}
		}
		$$$DebugLog($username,"GetOperationDetails","Type: "_pType_" Config Class: "_pClassName_" Config Object:
"_tClassObject,.dSC)

	set HTTP=tClassObject.DFISendMessageHTTP,HTTPS=tClassObject.DFISendMessageHTTPS,sendToFile=tClassObject.DFISend
MessageToFile

	set tRequestMessageClass=tClassObject.DFIRequestMessageClassName,tResponseMessageClass=tClassObject.DFIResponseM
essageClassName
		$$$DebugLog($username,"GetOperationDetails","HTTP: "_HTTP_" HTTPS: "_HTTPS_" SendToFile: "_sendToFile,.dSC)
		$$$DebugLog($username,"GetOperationDetails","Request Message Class Name: "_tRequestMessageClass_" Response
Message Class Name: "_tResponseMessageClass,.dSC)
		if pType="HL7" {
			if HTTP {set httpOperation=tClassObject.DFIHTTPHL7OperationName,sendToHTTP=1}
			elseif HTTPS {set httpOperation=tClassObject.DFIHTTPSHL7OperationName,sendToHTTP=1}
			set fileOperation=tClassObject.DFIFileHL7OpertionName
			set directory=tClassObject.DFIHL7FileDirectory
			set requestFileName=tClassObject.DFIHL7RequestFileName
			set responseFileName=tClassObject.DFIHL7ResponseFileName
		}
		if pType="FHIR"!(pType="Bulk") {
			if HTTP {set httpOperation=tClassObject.DFIHTTPFHIROperationName,sendToHTTP=1}
			elseif HTTPS {set httpOperation=tClassObject.DFIHTTPSFHIROperationName,sendToHTTP=1}
			else {set httpOperation="",sendToHTTP=0}
			set fileOperation=tClassObject.DFIFileFHIROperationName
			set directory=tClassObject.DFIFHIRFileDirectory
			set requestFileName=tClassObject.DFIFHIRResourceRequestJSONFileName
			set responseFileName=tClassObject.DFIFHIRResourceResponseJSONFileName
		}
		if pType="Manifest" {
			if HTTP {set httpOperation=tClassObject.DFIHTTPFHIROperationName,sendToHTTP=1}
			elseif HTTPS {set httpOperation=tClassObject.DFIHTTPSFHIROperationName,sendToHTTP=1}
			else {set httpOperation="",sendToHTTP=0}
			set fileOperation=tClassObject.DFIFileFHIROperationName
			set directory=tClassObject.DFIManifestFileDirectory
```

```
                        set manifestFileName=tClassObject.DFIManifestFileName
                        set requestFileName=tClassObject.DFIManifestRequestRecordFileName
                        set responseFileName=tClassObject.DFIManifestResponseRecordFileName
                }
                $$$DebugLog($username,"GetOperationDetails","Directory: "_directory_" ManifestFileName: "_manifestFileName_"
RequestFileName: "_requestFileName_" ResponseFileName: "_responseFileName,.dSC)
                set tSC=$classmethod(pClassName,"ResolveFileNames",.manifestFileName,.pOperations) quit:'tSC
set pOperations("ManifestFileName")=manifestFileName
                set tSC=$classmethod(pClassName,"ResolveFileNames",.requestFileName,.pOperations) quit:'tSC
set pOperations("RequestFileName")=requestFileName
                set tSC=$classmethod(pClassName,"ResolveFileNames",.responseFileName,.pOperations) quit:'tSC
set pOperations("ResponseFileName")=responseFileName
                $$$DebugLog($username,"GetOperationDetails","GetOperationDetails","Directory: "_directory_" ManifestFileName:
"_manifestFileName_" RequestFileName: "_requestFileName_" ResponseFileName: "_responseFileName)
                $$$DebugLog($username,"GetOperationDetails","GetOperationDetails","HTTPOperation: "_httpOperation_" FileOpeation:
"_fileOperation_" SendToHTTP: "_sendToHTTP_" SendToFile: "_sendToFile)

        set pOperations("Directory")=directory,pOperations("ManifestFileName")=manifestFileName,pOperations("RequestFileName")=requ
estFileName,pOperations("ResponseFileName")=responseFileName
                set pOperations("HTTPOperation")=httpOperation,pOperations("FileOperation")=fileOperation
                set pOperations("SendToHTTP")=sendToHTTP,pOperations("SendToFile")=sendToFile

        set pOperations("RequestMessageClassName")=tRequestMessageClass,pOperations("ResponseMessageClassName")=tResponse
MessageClass
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"GetOperationDetails","Get Operation Details Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}

/// This method takes a FileName and substitutes any string enclosed in {} and replaces
/// the string with the equivalent Name/Value pair specified in the array pValues
/// The valid values for substitution are:<br><br>
/// <b>{MessageId}</b> is replaced with pValues("MessageId")<br>
/// <b>{ManifestId}</b> is replaced with pValues("ManifestId")<br>
/// <b>{RecordId}</b> is replaced with pValues("RecordId")<br>
/// <b>{ResourceName}</b> is replaced with pValues("ResourceName")<br>
/// <b>{ResourceId}</b> is replaced with pValues("ResourceId")<br>
/// <b>{PatientId}</b> is replaced with pValues("PatientId")<br>
/// <b>{TimeStamp}</b> is replaced with $tr($zdt($h,3),":- ","")<br>
/// <b>{Date}</b> is replaced with $tr($zd($h,3),":- ","")
/// <b>{FromConsumerId}</b> is replaced with pValues("FromConsumerId")
/// <b>{ToConsumerId}</b> is replaced with pValues("ToConsumerId")
ClassMethod ResolveFileNames(ByRef pFileName As %String(MAXLEN=400), ByRef pValues As %String(MAXLEN=2000)) As %Status
{
        set tSC=$$$OK
        try {

        if pFileName["{PatientId}" set pFileName=$p(pFileName,"{PatientId}",1)_$g(pValues("PatientId"))_$p(pFileName,"{PatientId}",2,99)

        if pFileName["{MessageId}" set pFileName=$p(pFileName,"{MessageId}",1)_$g(pValues("MessageId"))_$p(pFileName,"{MessageId}
",2,99)

        if pFileName["{FromCosumerId}" set pFileName=$p(pFileName,"{FromCosumerId}",1)_$g(pValues("FromConsumerId"))_$p(pFileNa
me,"{FromConsumerId}",2,99)

        if pFileName["{ToConsumerId}" set pFileName=$p(pFileName,"{ToConsumerId}",1)_$g(pValues("ToConsumerId"))_$p(pFileName,"{
ToConsumeId}",2,99)

        if pFileName["{TestDefinition" set pFileName=$p(pFileName,"{TestId}",1)_$g(pValues("TestId"))_$p(pFileName,"{TestId}",2,99)

        if pFileName["{ManifestId}" set pFileName=$p(pFileName,"{ManifestId}",1)_$g(pValues("ManifestId"))_$p(pFileName,"{ManifestId}",
2,99)

        if pFileName["{RecordId}" set pFileName=$p(pFileName,"{RecordId}",1)_$g(pValues("RecordId"))_$p(pFileName,"{RecordId}",2,99)
        if pFileName["{Date}" set pFileName=$p(pFileName,"{Date}",1)_$tr($zd($h,3),": -")_$p(pFileName,"{Date}",2,99)
        if pFileName["{TimeStamp}" set pFileName=$p(pFileName,"{TimeStamp}",1)_$tr($zdt($h,3),": -
","")_$p(pFileName,"{TimeStamp}",2,99)

        if pFileName["{EMCIUId}" set pFileName=$p(pFileName,"{EMCIUId}",1)_$g(pValues("EMCIUId"))_$p(pFileName,"{EMCIUId}",2,99)

        if pFileName["{ResourceName}" set pFileName=$p(pFileName,"{ResourceName}",1)_$g(pValues("ResourceName"))_$p(pFileName
,"{ResourceName}",2,99)

        if pFileName["{ResourceId}" set pFileName=$p(pFileName,"{ResourceId}",1)_$g(pValues("ResourceId"))_$p(pFileName,"{ResourceI
d}",2,99)
        }
        catch ex {set tSC=ex.AsStatus()}
        quit tSC
}
```

```
/// This method is used to define any validation rules that ensure that the configuration
/// is sensible.
Method Validation() As %Status
{
	set tSC=$$$OK
	try {
		set msgType=..DFITargetMessageType
		if ..DFISendMessageHTTP,..DFISendMessageHTTPS set tSC=$$$ERROR(5001,"The flags DFISendMessageHTTP and
DFISendMessageHTTPS cannot both be TRUE") quit
		if msgType="H" {
			if ..DFISendMessageHTTP,(($l(..DFIHTTPHL7OperationName))!($l(..DFIHTTPHL7OperationClassName)))
set tSC=$$$ERROR(5001,"HL7 HTTP Operation Fields are not specified") quit
			if ..DFISendMessageHTTPS,(($l(..DFIHTTPSHL7OperationName))!($l(..DFIHTTPSHL7OperationClassName)))
set tSC=$$$ERROR(5001,"HL7 HTTPS Operation Fields are not specified") quit
		}
		if msgType="F" {
			if ..DFISendMessageHTTP,(($l(..DFIHTTPFHIROperationName))!($l(..DFIHTTPFHIROperationClassName)))
set tSC=$$$ERROR(5001,"FHIR HTTP Operation Fields are not specified") quit

	if ..DFISendMessageHTTPS,(($l(..DFIHTTPSFHIROperationName))!($l(..DFIHTTPSFHIROperationClassName)))
set tSC=$$$ERROR(5001,"FHIR HTTPS Operation Fields are not specified") quit
		}
	}
	catch ex {set tSC=ex.AsStatus()}
	$$$DebugLog($username,"ConfigValidation","Configuration Validation Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
	quit tSC
}

Method %OnBeforeSave(insert As %Boolean) As %Status [ Private, ServerOnly = 1 ]
{
	Quit ..Validation()
}
```



The Business Process

The Business Process in my Model uses code in the OnRequest() and OnResponse() methods. I believe that they could be re-written as BPL's as the logic in the methods has a straightforward flow that would be easy to represent in BPL.

I Open the Message Queue Object using the MessageId passed in the Request Message. As I do in every single method I write, I never assume that the object must exist. Logically speaking, there should be an object. However, I exception handle every possibility. This is just my way of coding. I use Try/Catch in every method, no matter how simple. I check the return status from every method call, and if the status code is in error, I quit the Try/Catch. I have seen too many Interfaces and Applications where this attention to detail is not applied, and it shocks me to see how many of them periodically report <UNDEF> errors in the Ensemble Trace Logs.

I will define a DTL that maps the data in the source database/table record into an HL7 or FHIR message. There is one trick I learned, and that is that the Transform() method has a third parameter called AUX. If the DTL is called from a BPL, this parameter will pass context information to the DTL. As I don't use BPL, I make use of this feature to pass values into the DTL that are not present in the Source Object passed to the DTL. For example: if you are creating an HL7 message, you will specify the SendingApplication, SendingFacility, RecievingApplication, and ReceivingFacility fields in the HL7 MSH segment. These values are defined in the Configuration Settings, and I need a way to pass them into the DTL. I create a class based on the DTL Name with the word AUX appended to the class name, and I define properties for any data I want to pass into the DTL. In the DTL, I assign the properties to the Target Message using ASSIGN statements, TYPE='SET'.

The Business Process may be faced with the scenario that under certain conditions, the HL7 Message may need to be transformed into another Message Structure. For example, most of the ADT messages have the same basic structure. However, if the ADT message is a Merge, then the HL7 structure changes and so I call another DTL that transforms an ADT_A08 (for example) into an ADT_A40 Message Structure. I use the AUX object to pass in the Identifier of the merged Patient in order to populate the MRG segment.

The Business Process stores the Request Message into the Message Class Object. I then call the HTTP Operation using the same Request Message that was passed to the Business Process. I put everything that the operations are going to need into the Message Class Definition and set their values from the Configuration Settings. The Configuration Settings Class contains a Method to GenerateOperationDetails() method, which passes back, by reference, the names of the HTTP and File Operation Production Names. It also resolves the file names into which the HL7 and FHIR messages will be written. In the Configuration settings. I define the file names, and I use various embedded code fields such as {Date}, {Time}, {PatientId}, {Episode} and a host of other possible keywords. The GetOperationDetails() method calls a ResolveFileNames() method which will substitute actual values for these codded fields.

Here is some example code from a Business Process that locates the Patient, invokes the Transform() method to create a FHIR Patient JSON Message and then send it to the HTTP Operation and optionally the File Operation.

```
Method OnRequest(pRequest As DFI.Common.Messages.DataLoadRequest,
Output pResponse As DFI.Common.Messages.DataLoadResponse) As %Status
{
	set tSC=$$$OK
	set pResponse=##class(DFI.Common.Messages.DataLoadResponse).%New(),pResponse.MessageId=pRequest.MessageId
	set pResponse.ResponseStatus=$$$OK
	try {
```

```
                $$$TRACE("Processing Request")
                // Get Patient
                set tOS=$$$GetOS(.tSC) quit:'tSC set tConfigClass=$$$GetConfig(.tSC) quit:'tSC
                $$$TRACE("Operating System: "_tOS_" Config Class: "_tConfigClass)
                set tConfig=$classmethod(tConfigClass,"%OpenId","Settings")
                set tConfig=##class(DFI.BulkExport.Configuration.ConfigurationSettings).%OpenId("Settings") if '$IsObject(tConfig)
{set tSC=$$$ERROR(5001,"Unable to Open DFI Configuration Settings") quit}
                #dim tMessage as DFI.BulkExport.Queue.BulkExportMessageQueue
                set tMessageQueue=tConfig.DFIBulkExportQueueClassName,tMessageId=pRequest.MessageID
                $$$TRACE("Message Queue: "_tMessageQueue_" Message Id: "_tMessageId)
                set tMessage=$classmethod(tMessageQueue,"%OpenId",tMessageId) if '$IsObject(tMsg)
{set tSC=$$$ERROR(5001,"Unable to Open Message queue Item: "_tMessageId_" in queue "_tMessageQueue) quit}
                /// If we want to process this Patient we now retrieve the Patient from the PMI
                set tPatientId=tMessage.PatientId,tPatientInternal=tMessage.PatientInternalNumber,tPatientHID=tMessage.PatientHID
                set pSourceObject = ##class(DW.Modules.Pmi.Patient).%OpenId(tPatientId) if '$IsObject(pSourceObject)
{set tSC=$$$ERROR(5001,"Patient with ID: "_tPatientId_" is not defined") quit}
                // Then check if the Patient has a Unique Identity
                if '$IsObject(pSourceObject.UniqueIdentity) {set tSC=$$$ERROR(5001,"Person: "_tPerson_" as Patient: "_tPatientId_"
has no UniqueIdentity") quit}
                // Create New FHIR Patient Object
                set pDestinationObject = ##class(FHIR.Patient).%New(),pDestinationObject.ResourceType="Patient"
                // Call the Transform from Patient to FHIR.Patient
                set tSC = ##class(DFI.Common.Transforms.DWToFHIR).Transform(pSourceObject,.pDestinationObject) if 'tSC {quit}
                set tSC = pDestinationObject.%Save() if 'tSC {quit}
                $$$TRACE("FHIR: "_pDestinationObject.toJSON())
                do tMessage.SourceFHIRRequestMessage.Write(pDestinationObject.toJSON())

        set tOperations("PatientId")=tPatientId,tOperations("MessageId")=tMessageId,tOperations("ResourceId")=pDestinationObject.%Id(),t
Operations("ResourceName")="Patient"
                set tSC=tConfig.GetOperationDetails(tConfigClass,.tConfig,"Bulk",.tOperations) quit:'tSC
                if $d(tOperations) {
                        $$$TRACE("Send to HTTP: "_$g(tOperations("SendToHTTP"))_" HTTP Opertion Name:
"_$g(tOperations("HTTPOperation")))
                        $$$TRACE("Send to File: "_$g(tOperations("SendToFile"))_" File Opertion Name:
"_$g(tOperations("FileOperation")))
                        set tMessage.FHIRFileDirectory=$g(tOperations("Directory"))
                        set tMessage.FHIRRequestFileName=$g(tOperations("RequestFileName"))
                        set tMessage.FHIRResponseFileName=$g(tOperations("ResponseFileName"))
                }
                else {
                        set msg="Neither HTTP nor File Operation Details exist. Completing Message as
Error",tSC=$$$ERROR(5001,msg)

        set tMessage.MessageStatus=tSC,tMessage.MessageStatusText=$$$GetErrorText(tSC),tMessage.CompletedTS=$zdt($h,3)
                        set tSC=tMessage.%Save()
                        set pResponse.ResponseStatus=tSC
                        quit
                }
                if tOperations("SendToHTTP") {
                        set tRequest=pRequest.%ConstructClone(1)
                        set tSC=..SendRequestAsync(tOperations("HTTPOperation"),tRequest,1,"FHIR HTTP Operation") if 'tSC quit
                }
                if tOperations("SendToFile") {
                        set tRequest=pRequest.%ConstructClone(1)
                        set tSC=..SendRequestAsync(tOperations("FileOperation",tRequest,1,"FHIR File Operation")) if 'tSC quit
                }
        }
        catch ex {set tSC=ex.AsStatus()}
        $$$DebugLog($username,"SendSelectedPatients","Send Patients to EMCI Status: "_$s(tSC:"OK",1:$$$GetErrorText(tSC)),.dSC)
        quit tSC
}
```

The Business Operations

I created 5 operations:

1) HTTP HL7 Outbound Operation
2) HTTPS HL7 Outbound Operation
3) HTTP FHIR Outbound Operation
4) HTTPS FHIR Outbound Operation
5) File Outbound Operation

Obviously, you could add other Business Operations to meet your needs. I know now that I could design the Configuration File differently and use embedded classes for different groups of related data and in the case where there could be many options, I would create these as a List of Embedded objects. This is version one and I there are a few things I would change.

The Business Operations are passed the Message Queue MessageId. The Business Operation OnInit() method gets the Configuration and populates the Operation Properties and Adapter Properties with any information that is derived from the Configuration class. The Main method of the Business Operation opens the Message Queue Object for MessageId, which has all of the details that it needs to perform its task. The HL7 Message or FHIR JSON are stored in the Message Queue Message, and the method reads this into the %Net.HTTPRequest EntityBody.

The Request Message is sent to the Target Server. If you need to use HTTP Methods other than POST, then you need to decide this in the Business Process and pass this in in the Request Message or the Message Queue Class Object.

If you get a response, there will be an HTTP Response Code and an HTTP Response Body. I update the message Queue Message with these values and update the Ensemble Response Message with the Operation Outcome. The Response message is sent back to the OnProcessResponse() Method of the Business Process (if you use an Async call to the HTTP Operation. Otherwise, it will be returned in the tResponse parameter of the Sync call.

I update the Message Queue Message with the HTTP Status and any content and send the Ensemble Response Back to the Business Process which handles the response. If ‘SendToFile’ is true then the Response HL7 or FHIR message is sent to the File Operation, which will write the contents to file.

At this point, I call the CompleteMessage() of the Message Queue Class and update the Ensemble Response Message, which is sent back to the Business Service, thereby competing with the Look,

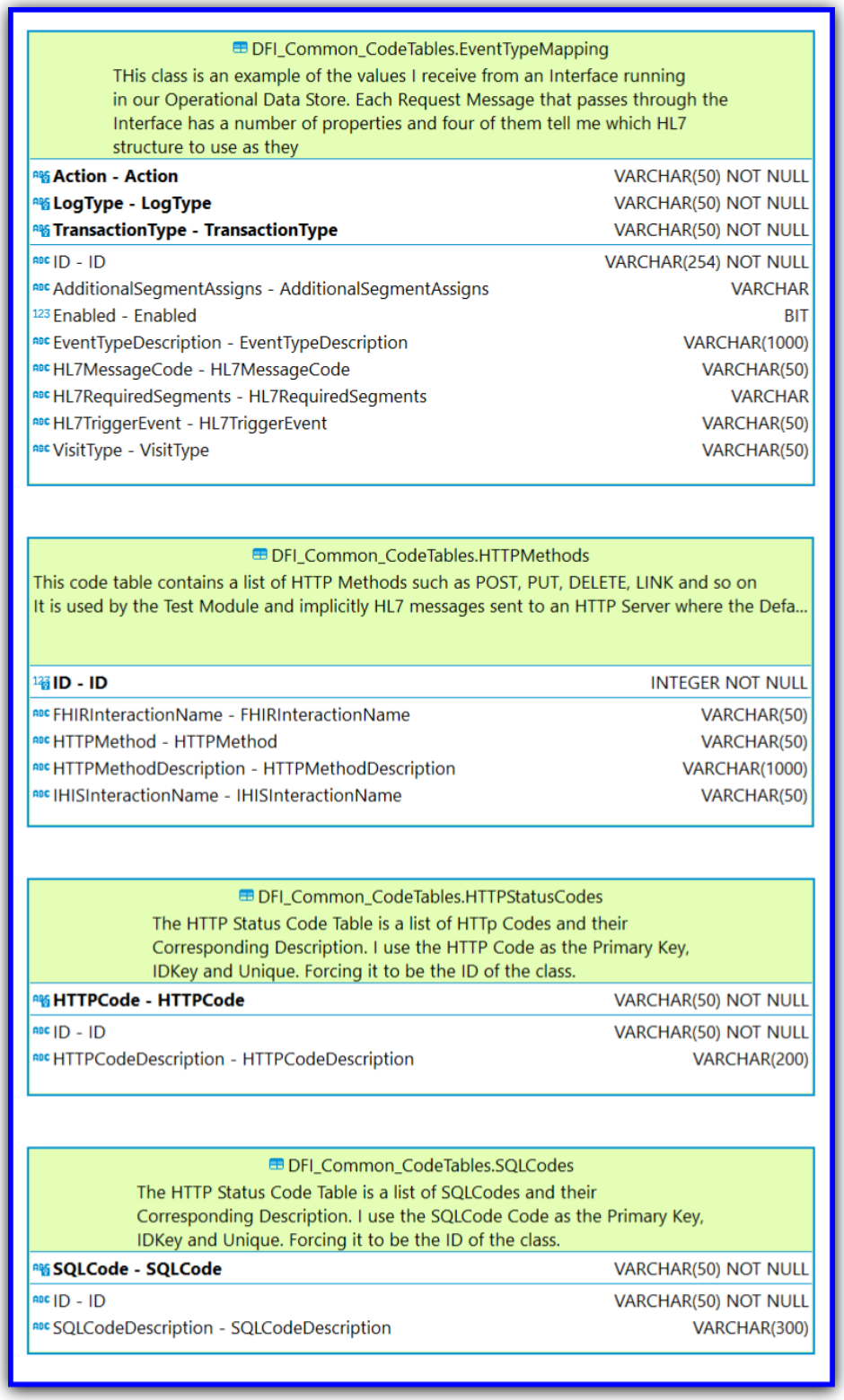
There in a nutshell, you have the core functionality of the Data Flow Interface Model.

I am now going to discuss some sundry functionality, and then I will discuss the deployment Model.

Sundry Functionality

Code Tables

Code tables and Code Table Mappings are useful for handling fields that have a set distinct set of values. They are used in your Interfaces for code fields such as HTTP Codes. I will not spend a lot of time on this as each customer will have different requirements. To summarise, here are some of the Code Tables I maintain.



DFI_Common_CodeTables.EventTypeMapping

THis class is an example of the values I receive from an Interface running in our Operational Data Store. Each Request Message that passes through the Interface has a number of properties and four of them tell me which HL7 structure to use as they

Action - Action	VARCHAR(50) NOT NULL
LogType - LogType	VARCHAR(50) NOT NULL
TransactionType - TransactionType	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(254) NOT NULL
AdditionalSegmentAssigns - AdditionalSegmentAssigns	VARCHAR
Enabled - Enabled	BIT
EventTypeDescription - EventTypeDescription	VARCHAR(1000)
HL7MessageCode - HL7MessageCode	VARCHAR(50)
HL7RequiredSegments - HL7RequiredSegments	VARCHAR
HL7TriggerEvent - HL7TriggerEvent	VARCHAR(50)
VisitType - VisitType	VARCHAR(50)

DFI_Common_CodeTables.HTTPMethods

This code table contains a list of HTTP Methods such as POST, PUT, DELETE, LINK and so on
It is used by the Test Module and implicitly HL7 messages sent to an HTTP Server where the Defa...

ID - ID	INTEGER NOT NULL
FHIRInteractionName - FHIRInteractionName	VARCHAR(50)
HTTPMethod - HTTPMethod	VARCHAR(50)
HTTPMethodDescription - HTTPMethodDescription	VARCHAR(1000)
IHISInteractionName - IHISInteractionName	VARCHAR(50)

DFI_Common_CodeTables.HTTPStatusCodes

The HTTP Status Code Table is a list of HTTp Codes and their Corresponding Description. I use the HTTP Code as the Primary Key, IDKey and Unique. Forcing it to be the ID of the class.

HTTPCode - HTTPCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
HTTPCodeDescription - HTTPCodeDescription	VARCHAR(200)

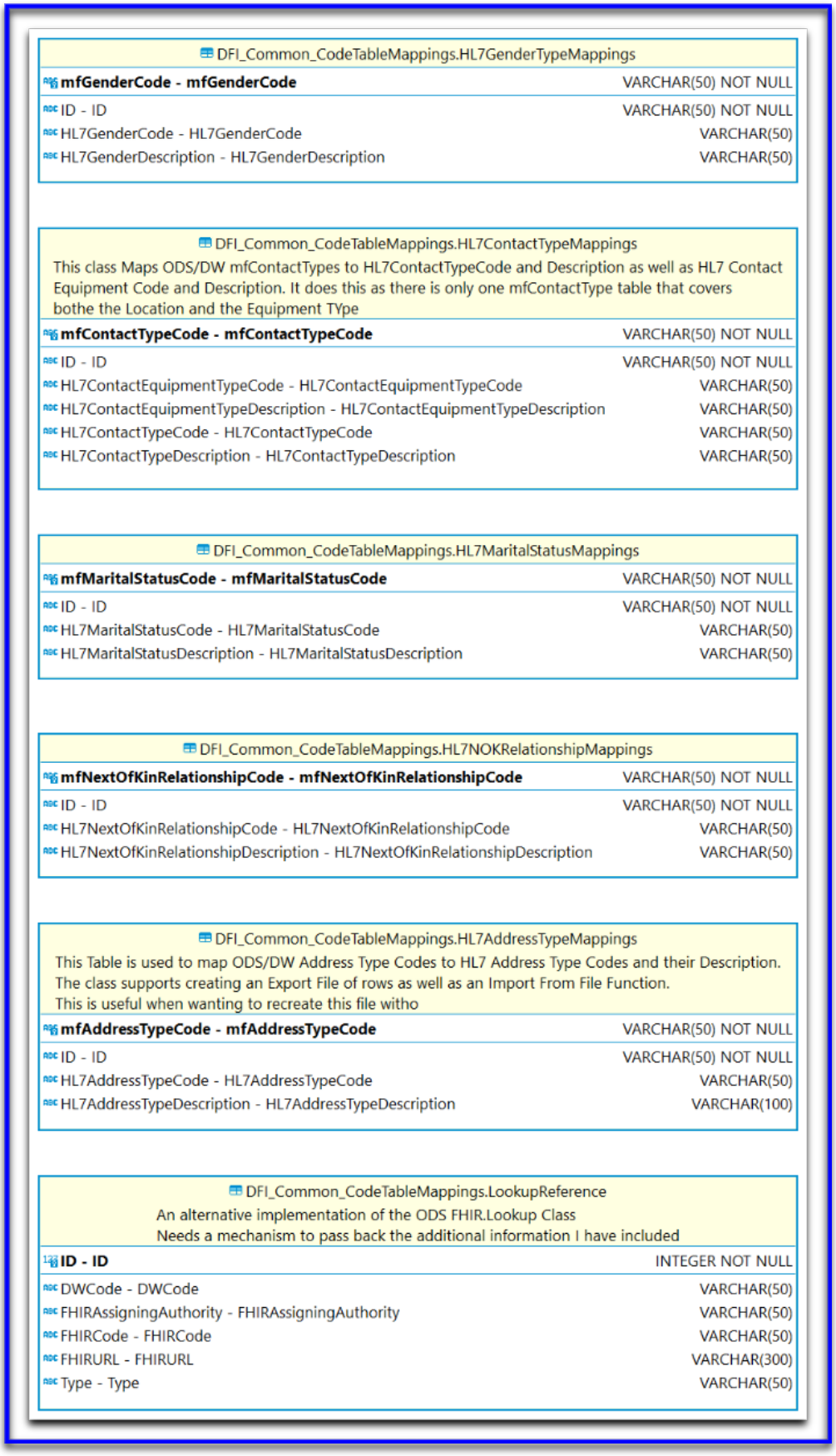
DFI_Common_CodeTables.SQLCodes

The HTTP Status Code Table is a list of SQLCodes and their Corresponding Description. I use the SQLCode Code as the Primary Key, IDKey and Unique. Forcing it to be the ID of the class.

SQLCode - SQLCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
SQLCodeDescription - SQLCodeDescription	VARCHAR(300)

Code Table Mappings

Code Table Mappings are used to transform internal values into HL7 or FHIR equivalents. Here are some that I use.



DFI_Common_CodeTableMappings.HL7GenderTypeMappings

mfGenderCode - mfGenderCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
HL7GenderCode - HL7GenderCode	VARCHAR(50)
HL7GenderDescription - HL7GenderDescription	VARCHAR(50)

DFI_Common_CodeTableMappings.HL7ContactTypeMappings

This class Maps ODS/DW mfContactTypes to HL7ContactTypeCode and Description as well as HL7 Contact Equipment Code and Description. It does this as there is only one mfContactType table that covers bothe the Location and the Equipment TYpe

mfContactTypeCode - mfContactTypeCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
HL7ContactEquipmentTypeCode - HL7ContactEquipmentTypeCode	VARCHAR(50)
HL7ContactEquipmentTypeDescription - HL7ContactEquipmentTypeDescription	VARCHAR(50)
HL7ContactTypeCode - HL7ContactTypeCode	VARCHAR(50)
HL7ContactTypeDescription - HL7ContactTypeDescription	VARCHAR(50)

DFI_Common_CodeTableMappings.HL7MaritalStatusMappings

mfMaritalStatusCode - mfMaritalStatusCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
HL7MaritalStatusCode - HL7MaritalStatusCode	VARCHAR(50)
HL7MaritalStatusDescription - HL7MaritalStatusDescription	VARCHAR(50)

DFI_Common_CodeTableMappings.HL7NOKRelationshipMappings

mfNextOfKinRelationshipCode - mfNextOfKinRelationshipCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
HL7NextOfKinRelationshipCode - HL7NextOfKinRelationshipCode	VARCHAR(50)
HL7NextOfKinRelationshipDescription - HL7NextOfKinRelationshipDescription	VARCHAR(50)

DFI_Common_CodeTableMappings.HL7AddressTypeMappings

This Table is used to map ODS/DW Address Type Codes to HL7 Address Type Codes and their Description. The class supports creating an Export File of rows as well as an Import From File Function. This is useful when wanting to recreate this file witho

mfAddressTypeCode - mfAddressTypeCode	VARCHAR(50) NOT NULL
ID - ID	VARCHAR(50) NOT NULL
HL7AddressTypeCode - HL7AddressTypeCode	VARCHAR(50)
HL7AddressTypeDescription - HL7AddressTypeDescription	VARCHAR(100)

DFI_Common_CodeTableMappings.LookupReference

An alternative implementation of the ODS FHIR.Lookup Class
Needs a mechanism to pass back the additional information I have included

ID - ID	INTEGER NOT NULL
DWCode - DWCode	VARCHAR(50)
FHIRAssigningAuthority - FHIRAssigningAuthority	VARCHAR(50)
FHIRCode - FHIRCode	VARCHAR(50)
FHIRURL - FHIRURL	VARCHAR(300)
Type - Type	VARCHAR(50)

The Debugger

I created a Debug Logger Class that can be inserted into your code. It is similar to $$$TRACE in that in PRODUCTION you can set Debugging to FALSE and no Debug Logs will be generated.

The Schema of the Debug Logger is as follows:

persistent class **DFI.Common.Debug.Status** extends %Persistent

This class is basically a configuration settings class for the Debug Logging Component. In DEV and QC namespaces where functionality is being tested then we typically want debugging turned on but in Production we don't want that overhead and so we set the property to 0 (false) and when the Debug Logger tries to create a new debug log record this condition is tested and if if the value is 0 no record will be created Much like $$$TRACE tests the Ensemble Production Setting "Log Trace Events"

Inventory

Parameters	Properties	Methods	Queries	Indices	ForeignKeys	Triggers
1	2	2		1		

Summary

Properties	
DebuggingIsOn	RowId

Parameters

• parameter **GlobalName** = "^DFI.Common.Debug.Status";

For the purposes of Global Mappings the Global Names in the Storage Definition have been modified to be more readable than the Ensemble generated Global Names. This Parameter informs the developer that should they delete the Storage Definition then they should replace the Glabal Names with the Value in the Parameter where the * = D, I or S

Properties

• property **DebuggingIsOn** as %Boolean [InitialExpression = 1];

This class has this one property and its purpose is to indicate if Debugging is On or Not. I could have just tested to see if ^DFI.Common.Debug.Logging=1 but I am trying to avoid using raw global references in favour of proper classes.

• property **RowId** as %Integer [Private,InitialExpression = 1];

The class has only one record with an ID of 1

Methods

• classmethod **GetDebugOnOff(ByRef tSC As %Status)** as %Boolean [SQLProc =]

This method finds the first record in this table and returns the value of the property DebuggingIsOn. The value of the property is set using a method in the EMCI.Debug.Logging class.

• classmethod **SetDebugOnOff(pOnOff As %Boolean = 1)** as %Status [SQLProc =]

This method sets the DebuggInOn Flag to the parameter pOnOff. pOnOff is a Boolean so pOnOff must be 0 or 1 If the Status Object does not exist it will be created with DebuggingIsOn=1

Indices

• index **(PK** on **RowId)** [IdKey,PrimaryKey,Unique];

persistent class **DFI.Common.Debug.Logging** extends %Persistent

This is the standard Debug Logging class for the DFI Codebase

All of the DFI.* classes have an Include statement in them

Include DFIInclude

The DFIInclude.inc file has the following #define in it

#define DebugLog(%s1,%s2,%s3,%s34) do ##class(DFI.Common.Debug.Logging).CreateDebugLog($classname(),%s1,%s2,%s3,%s4)

In Class Method Code you add calls to the Debug Logger as follows:

$$$DebugLog($username,"MyKey","This is my Debug Message",.dSC)

To enable Debug Logging execute the following code in the namespace where your production is running

do ##class(DFI.Common.Debug.Logging).SetDebugingOnOff(1)

To find out if Debugging is On or Off:

set pDebugging=##class(DFI.Common.Debug.Logging).GetDebugOnOff(.tSC)

pDebugging will be set to 1 (ON) or 0 (OFF)
The CreateDebugLog() method checks to see if Debugging is Turned On or OFF. Debug Logs are only created if Debug Logging is turned ON.

It is advisable to turn Debug Logging OFF in the Live Interface Production

Inventory

Parameters	Properties	Methods	Queries	Indices	ForeignKeys	Triggers
1	5	6		4		

Summary

Properties			
ClassName	CreateTS	Key	Message

Parameters

• parameter **GlobalName** = "^DFI.Common.Debug.Logging*";
For the purposes of Global Mappings the Global Names in the Storage Definition have been modified to be more readable than the Ensemble generated Global Names. This Parameter informs the developer that should they delete the Storage Definition then they should replace the Glabal Names with the Value in the Parameter where the * = D, I or S

Properties

• property **ClassName** as %String(MAXLEN=150);
The Class Name that calls the Debug Log
• property **CreateTS** as %TimeStamp [InitialExpression = $zdt($h,3)];
The CreateTS is the TimeStamp when the Debug Log Record is created
• property **Key** as %String(MAXLEN=100) [Required];
The Key should be a meaningful reference to the area of code where the Debug Log record is being created from.
• property **Message** as %String(MAXLEN=3641144,TRUNCATE=1);
The Message is a Text string specified by the developer.
• property **Username** as %String [Required];
The Username is $username by default but can be specified

Methods

• classmethod **CreateDebugLog(pClassName As %String = "", pUsername As %String = $username, pKey As %String(MAXLEN=100)={$classmethod($classname(), "GetKey")}, pMessage As %String(MAXLEN=3641144, TRUNCATE=1)="", ByRef pStatus As %Status = $$$OK)** [SQLProc =]
The CreateDebugLog() method creates a new entry in the Debug Log Table.
It defaults ClassName to the current class from which it was invoked.

If pUsername is not supplied it will default to $username
If you port the code to Production you can leave the debug calls in your code but turn off debugging. The method GetDebugOnOff() returns the value in the Status Record and if the property DebugingIsOn is False no record is created

• classmethod **GetDebugOnOff(ByRef tSC As %Status)** as %Boolean [SQLProc =]

This method returns the value of the Debug Logging Flag. If the DFI.Common.Debug.Status record does not exist it will be created with a default value of 1 (ON).

• classmethod **GetKey(pClassname As %String = $classname())** as %String

If no key is specified in CreateDebugLog() then the method will generate a default key in the format DebugKey_{N} where {N} is a sequential integer derived from the globsl ^DFI.Common.Debug.NextKey

• classmethod **PurgeDebugLog(pNumberOfDays As %Integer = "", ByRef pRowCount As %Integer)** as %Status [SQLProc =]

This method will purge all Debug logs older that Current Date - pNumberOfDays. It will passback the Number of Debug Log Rcords it has deleted. If a log record cannot be deleted an error will be displayed and the method will continue execution

• classmethod **SetDebugOnOff(pOnOff As %Boolean)** as %Status [SQLProc =]

This method Sets a flag to indicate whether debugging is turned ON of OFF.
The CreateDebugLog() method checks whether the value of this flag is ON or OFF. If Debugging is OFF No Debug Log will be created. The SetDebugingOnOff() code can be overridden to get a value for the flag from an application configuration class.

• classmethod **Test(pUser As %String = "Nigel", pKey As %String = "Key1", pMessage As %String = "Message 1", ByRef pStatus As %Status)** [SQLProc =]

Indices

• index **(CDT** on **CreateTS)**;

• index **(CN** on **ClassName)**;

• index **(Key** on **Key)**;

• index **(UN** on **Username)**;

Additional Business Services

There are two additional Business Services

The House Keeping Service

All to often I come across Interfaces that do no house keeping at all and as a result any developer created Queue, Reports, Log Files and any other transient data that is generated by the Interface just builds up over time and the Interface Database grows larger as time wears on.

The Data Flow Interface has a House Keeping Service that gets the Number of Days to retain things from the Configuration Settings. The House Keeping Service runs once a day and removes any data or files older than the Number of days as specified in the configuration settings.

The Alert Notification System

The Data Flow Interface has an Alert Notification System that allows the developer to create Alerts from with their code, assign a priority too the error or warning situation. They may also specify a Grace period such that if the error occurs within that grace period no notification will be sent. They can also specify how many times an error must occur before an alert is sent to the different email lists depending on the type of situation and the relevant people that need to be notified of the situation.

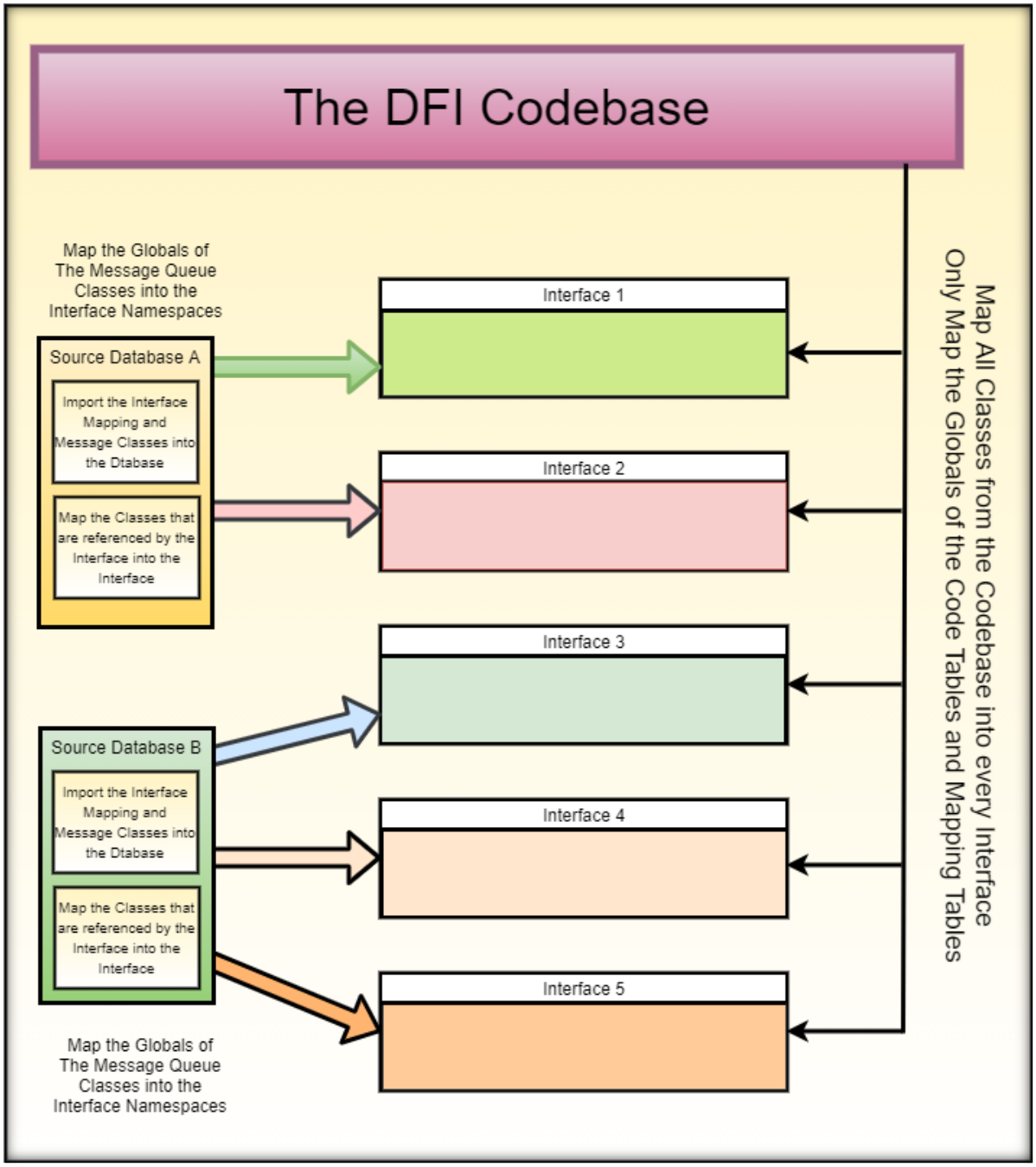
This is a topic for another discussion.

The Package and Global Mapping Model

The whole purpose of designing this Data Flow Interface was to create one code base that could be reused many times. I have achieved that as follows.

1) All classes in the Code Base are mapped into every new Interface Namespace
2) Only the Code Table and Code Table Mapping Globals are mapped into every Namespace
3) The Interface Mapping Class is loaded into the Source Database
4) The Interface Specific Message Queue Classes are loaded into the Source Database
5) The Message Queue Globals are mapped into the Interface Namespace they were created for
6) And that's it.
7) Piece of Cake

Here is a diagram that demonstrates this principle.



Conclusion

There are some aspects of the Data Flow Interface that I have not discussed in this article, I will cover those in a separate article in the near future. I trust that the information that I have supplied gives you enough information for you to be able to understand the overall methodology that I have used in the design and development of this Interface Model

Nigel Timothy Salm