

DataConnect Message Component Framework

Programmer's Reference Manual

Guide to Using the Message Component Framework

Actian Corporation
330 Potrero Avenue
Sunnyvale, CA 94085 USA

Web: <https://www.actian.com>



ACTIAN™

Refer to the license.txt file in the default installation directory for disclaimers and information about trademarks and credits.

Message Component Framework Programmer's Reference Manual

March 2007 – March 2022

Contents

1 Introduction to the Message Component Framework	1-1
Introduction.....	1-2
Overview.....	1-2
Programming Language.....	1-3
2 Getting Started.....	2-1
Physical Packaging.....	2-2
System Requirements.....	2-3
Installation	2-4
Deployment Descriptor	2-5
Error Codes	2-7
3 Component Management.....	3-1
Component Management	3-2
Managing Multiple Versions of Components.....	3-3
Viewing and Configuring Component Properties	3-6
DataConnect Version Compatibility.....	3-7
4 Component Development and Patterns.....	4-1
Component Development and Patterns	4-2
Component Types.....	4-8
Component Actions	4-9
Queue Component	4-10
Iterator Component	4-27
Aggregator Component.....	4-32
Transformer Component	4-35
Invoker Component	4-43
Validator Component	4-50
5 Component Types and Services.....	5-1
Component Types.....	5-2
Component Services	5-3
Java Language Services	5-3
Message Services for Components Written in Java	5-39
6 Component Contracts	6-1
com.pervasive.cosmos.component Contracts.....	6-2
7 Deployment Descriptor	7-1
Deployment Descriptor	7-2

Package Element.....	7-7
Description Element.....	7-8
Option Element.....	7-8
Enum Element.....	7-10
Java Element	7-11
Classpath.....	7-12
Model Element.....	7-12
Action Element	7-13
Parameter Element	7-13
XML Schema	7-14

1 Introduction to the Message Component Framework

1

Overview, Programming Languages

This chapter provides an overview of Message Component Framework, its features, and its capabilities.

- “Introduction” on page 1-2
- “Overview” on page 1-2
- “Programming Language” on page 1-3

Introduction

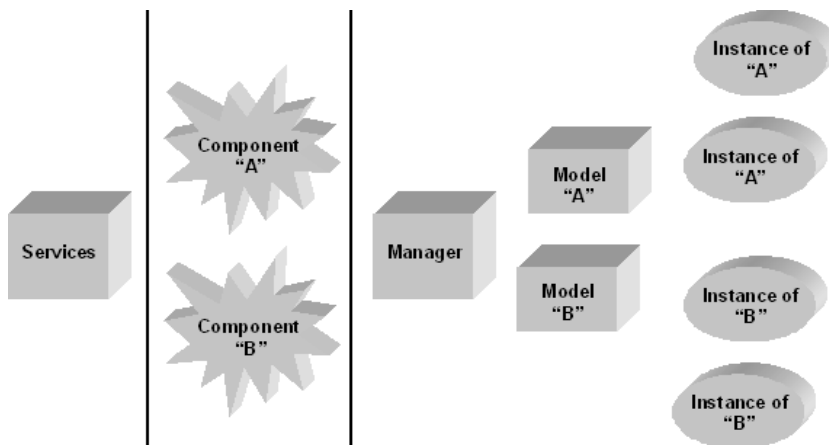
This document provides an overview of the DataConnect Message Component Framework (MCF) and provides developers with fundamental information needed to create message components.

Overview

Conceptually, the Message Component Framework consists of five parts:

- Component Services
- Component
- Component Manager
- Component Model
- Component Instance

Component Services are provided by Actian Corporation for use by Component writers. The Manager, Models, and Instances are internal to Actian Corporation. The physical implementation of actual components uses some utility classes. It also involves subclassing the Component Model to support various Component types.



Programming Language

Support is currently provided for writing Components in Java.

Java API

The API documentation for the Message Component Framework SDK can be found at the following locations:

Online

TODO (DataConnect 12.1): Online URL goes here (.../index.html)

Installation

TODO (DataConnect 12.1): Path to installed HTML file goes here (.../index.html)

The Java Application Programming Interface (API) document is an HTML file that can be viewed with your Internet browser. The API document contains the following sections:

Table 1-1 Java API Objects

API Section	Description
Overview	Provides a list of all the packages with a description for each one.
Package	Each package has a page that contains a list of its interfaces, classes, exceptions, and errors, with a summary for each one.
Class	Each class, interface, nested class, and nested interface has its own separate page. All descriptions and details for the class, constructors, methods, fields, and relationships are found on this page.
Use	This page provides a listing of packages, classes, interfaces, and methods which use the current class or interface.

API Section	Description
Tree (Class Hierarchy)	This page provides a class hierarchy for all classes. It also includes links to a hierarchy page for each individual package. Classes are organized by inheritance structure, starting with java.lang.Object . Interfaces do not inherit from java.lang.Object .
Deprecated	This page provides links to all classes, methods, interfaces, and fields that are marked as deprecated.
Index	The index contains an alphabetic list of all the classes, interfaces, constructors, methods, and fields.
Help	This page provides further description of how the API documentation is organized

2 *Getting Started*

Installing MCF Components

This chapter contains instructions for installing MCF Components. It is divided into the following topics and subtopics:

- “Physical Packaging” on page 2-2
- “System Requirements” on page 2-4
- “Installation” on page 2-5
- “Deployment Descriptor” on page 2-7

Physical Packaging

Components are packaged as archive files. The following extensions are supported:

- .zip
- .jar

The archive file contains the deployment descriptor, the implementations of the contained MCF components and any dependencies that must be delivered with the components.

Java components have quite a bit of flexibility in the way that they are packaged inside the component zip file. The Java package and classname must be locatable in the CLASSPATH based on the following set of rules:

- 1 <component archive root>
- 2 <component archive root>/classes
- 3 <component archive root>/*.jar
- 4 <component archive root>/lib
- 5 <component archive root>/lib/*.jar

This auto-generated classpath provides a variety of mechanisms for packaging components inside their component zip files. This allows packaging a Java component by building a regular jar file, adding a deployment descriptor and any dependencies of the component (.jar files of libraries required by the component, for example).

The issue connected with the Caution note below may be resolved by release date.

System Requirements

Recommended System Requirements

- DataConnect 12.x.x
- JDK 11 (DataConnect includes an embedded Java 11 runtime environment, but developers will need their own full JDK for development)
- See the DataConnect 12.x.x user documentation for hardware and operating system-specific requirements

Suggestion for Maximum Performance

Please note that these are the recommended system requirements for satisfactory operation. To ensure maximum performance, be certain that there is sufficient physical memory to meet the requirements of the local system and all applications.



Note Windows is required only for process design in the studio. Deployed processes, along with their associated components, can be operating system independent.

Installation

To install an MCF Component, drop the component archive into the Plug-Ins directory specific to your operating system and configuration.

The default for Windows: `C:\ProgramFiles\Actian\DataConnect\actian-dc-studio-version1\Plug-Ins`

The default for Linux and AIX: `/opt/Actian/di-standalone-engine-version1/runtime/di9/Plug-Ins`

However, you may choose to install into a different directory. To select a different directory, set the *[Preferences]Plugins* value in the cosmos.ini file to point to the preferred directory.

The next time a component-aware product (DataConnect Design Studio or stand-alone engine) is started, it will detect the new package. The deployment descriptor is examined to determine version information. The entire package is then extracted into the *Plug-Ins/PackageName*²/*version*² subdirectory.



Note 1 *version* represents the installed DataConnect version, such as 12.0.0-23

Note 2 *PackageName* and *version* are specified in the deployment descriptor.

Deployment Descriptor

The deployment descriptor is an XML file that describes the “package” within the archive. It provides the overall package name and version, then describes It provides the name and current version of the component, component type, component class, any installation requirements, and metadata about the component. The deployment descriptor must be named package.xml and exist in the /MC-INF directory of the archive file.

```
<Package name="Package Name" version="1.0.0" vendor="Vendor Name"
  schemaVersion="2.0">

  <!--
  Zero or more package-level options may be specified in the
  deployment descriptor. Options specified at this level
  will appear in the property sheet of each component in the
  Process Editor.
  -->
  <Option fullname="Shared Text Option" name="textOption"
    type="Text"/>

  <!--
  A deployment package may have one or more Component
  elements, each identifying a unique component within a
  single deployment package.
  -->
  <Component name="SampleComponent" version="1.0.0"
    class="Message">

    <!--
    Component is based on the Java language, and its main
    class is SampleComponent
    -->
    <Java mainclass="SampleComponent"/>

    <!--
    Component has an option named "testEnum" that has
    possible values of 1, 2, or 3.
    -->
    <Option fullname="Test of enum option" name="testEnum"
      type="Enum">
      <Description>This is an option.</Description>
      <Enum value="1" display="one"/>
      <Enum value="2" display="two"/>
      <Enum value="3" display="three"/>
    </Option>
```

```

<!--
Component is a "queue" component, and it exposes the
GetMessage, Connect, and Disconnect actions
-->
<Model type="queue">
  <Description>
    Iterates over sample messages
  </Description>
  <Action type="GetMessage">
    <Parameter type="Message"/>
    <Parameter type="Queue" usage="notused"/>

    <!--
    Zero or more action-level options may be
    specified in the deployment descriptor.
    Options specified at this level will appear in
    the property sheet of each component in the
    Process Editor.
    -->
    <Option name="actionOption"
      fullname="GetMessage Action Option"
      type="Text">
      <Description>Action Option</Description>
    </Option>
  </Action>
  <Action type="Connect"/>
  <Action type="Disconnect"/>
</Model>

  </Component>
</Package>

```

Error Codes

For a complete list of Error Codes, see the error codes documentation at [Message Component Framework Engine Error Codes](#).

3 Component Management

3

Plug-in Component Management Controls

This chapter contains the following topics:

- “Component Management” on page 3-2
- “Managing Multiple Versions of Components” on page 3-3

Component Management

DataConnect supports multiple versions of *deployed* message components. Management of multiple *deployed* component versions enables you to:

- Add and delete components as required.
- Deploy multiple, successive versions of the same component.
- Automatically use more recent versions of components in processes

Component Version Control

Management of component source code is the responsibility of the developer. It is recommended that you use a version control system, such as Git, CVS, Apache Subversion, Mercurial or Perforce to manage components and retain component revision history. If you should need to revert to a previous component version, it can then be retrieved from the version control system.


Managing Multiple Versions of Components

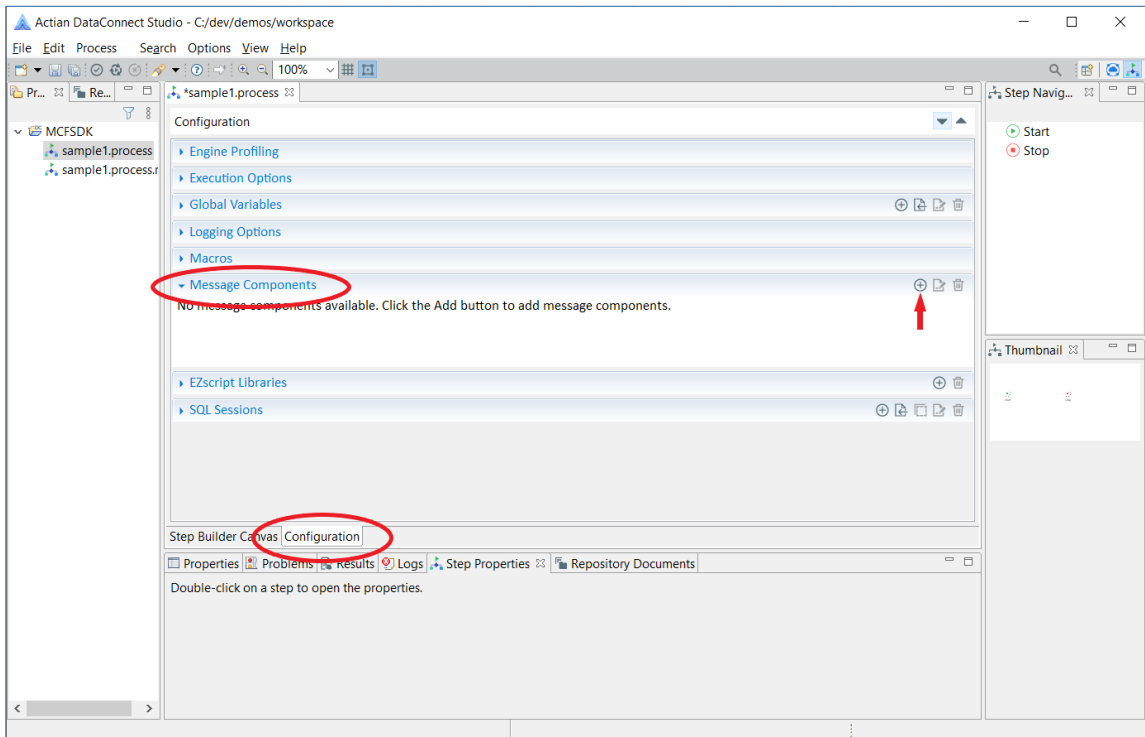
Component versions are displayed in DataConnect Studio. Component management procedures described in this section include:

- Selecting a Component in the Process Editor
- Viewing Component Properties in the Process Editor
- Version Compatibility

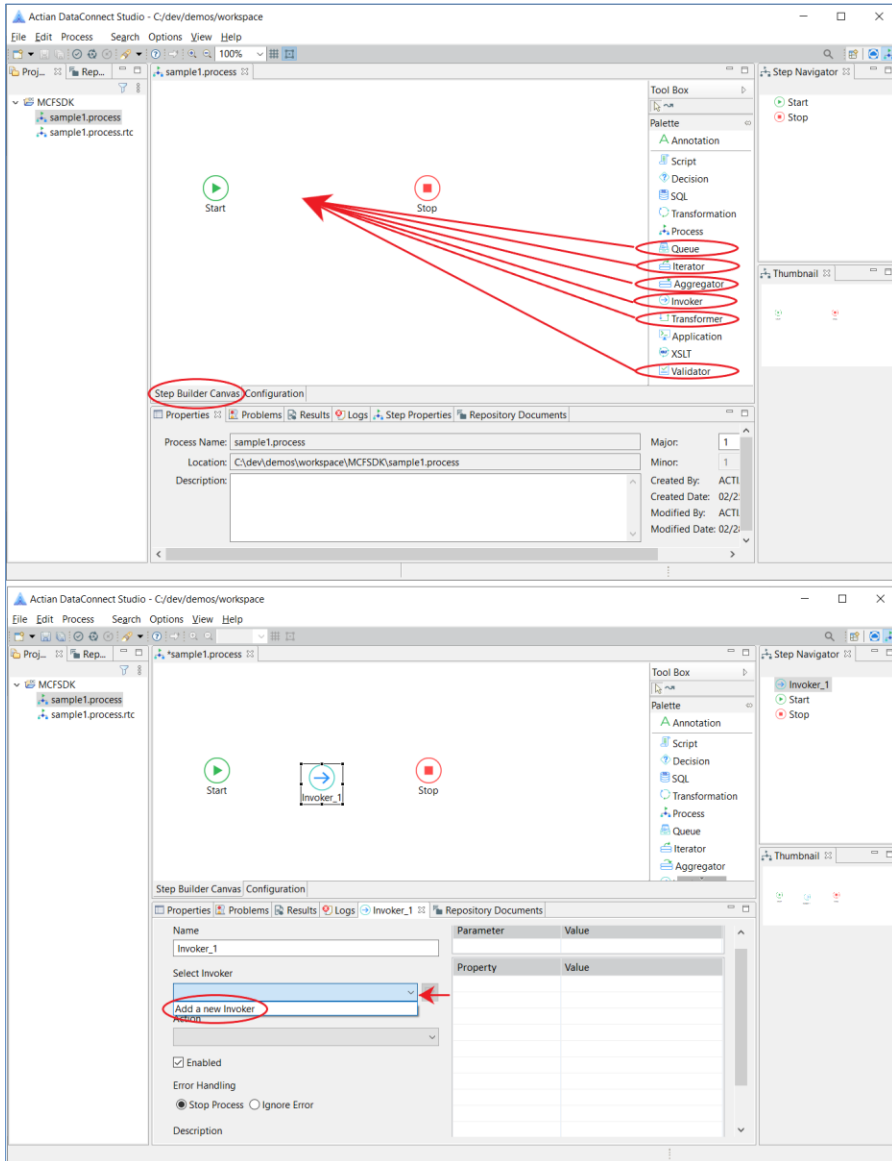
Select a Component in the Process Editor

Activate the “Add Message Component” dialog using one of the following procedures.

- Click the  button within the “Message Components section of the “Configuration” tab



- Drag a component step to the “Step Builder Canvas”. This will open a step configuration tab beneath the canvas. Under “Select component type”, open the pulldown menu then click “Add new component type”



When the “Add Message Component” dialog pops up, select the category¹, then the specific component (“Type” pulldown menu). The name and version of each available component will be displayed.

Add Message Component

Component Name
Component_1

Category
Queue

Type
JMS Queue 3.0.0

Select component type
AWS S3 Queue 1.0.0
Azure Storage Queue 1.0.0
FTP Queue 1.0.0
FTP Queue 4.1.0
File Folder Queue 1.0.0
Google Cloud Storage 1.0.0
Google Drive Queue 1.0.0
IBM MQ 1.0.0
IMAP E-mail Queue 1.3.0
JMS Queue 2.0.0
JMS Queue 3.0.0
Kafka Micro-batch Queue 1.0.0
Kafka Queue 1.0.0
MSMQ 1.0.0
POP3 E-mail Queue 1.0.2
POP3 E-mail Queue 1.3.0
RabbitMQ Component 1.0.0

☐ Global Transaction

Save Cancel

Viewing and Configuring Component Properties

Components can have properties associated with instances, individual actions, or both

Viewing and Configuring Instance-Level Component Properties

Instance-level component properties are viewed and configured in the “Add Message Component” dialog.

Add Message Component

Component Name:

Category:

Type:

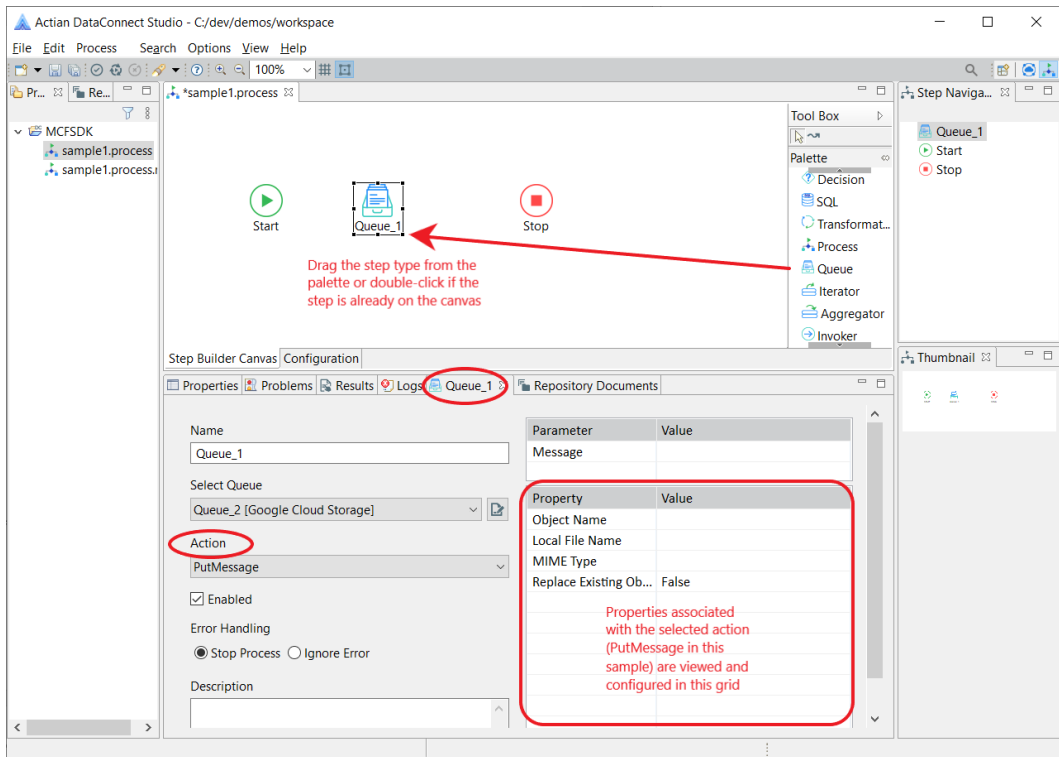
Properties (12)

Property	Value
User Name	
Password	
Messaging Styles	Point-to-Point
java.naming.factory.initial	
javax.jms.connectionfactory	
java.naming.provider.url	
PropertiesFile	
OutputMessageType	javax.jms.TextMessage
MessageSelector	
JarLocation	
Timeout	0
JMSPriority	

☐ Global Transaction

Viewing and Configuring Component Action Properties

Component action properties (if any are defined by the component) are viewed and configured in the step configuration tab. This tab is displayed below the “Step Builder Canvas” when the step is dragged onto the canvas or a step on the canvas is double-clicked.



DataConnect Version Compatibility

A component may be configured to take advantage of features introduced in a specific DataConnect release. When you attempt to run this component on a previous release, it will fail with a version error.

4 Component Development and Patterns

Component Development Detail

This chapter contains the following topics:

- "Component Development and Patterns" on page 4-2
- "Uniformity" on page 4-4
- "Component Types" on page 4-9
- "Component Actions" on page 4-10
- "Queue Component" on page 4-11
- "Iterator Component" on page 4-23
- "Transformer Component" on page 4-30
- "Invoker Component" on page 4-35
- "Validator Component" on page 4-40

Component Development and Patterns

The Message Component Framework (MCF) provides a uniform runtime environment for message components. This environment provides some basic services for components to use during execution. The environment is also responsible for the management of the component lifecycle and for the invocation of component actions. Ultimately, the framework simplifies the job of the component writer by eliminating some of the component and message object management issues.

Concurrency

The framework is responsible for managing the synchronization of component instances and access to message objects.

- Components on two or more threads can access a single message object instance.

This feature makes it possible to use multiple threads for broadcasting a message.

- Steps on two or more threads can execute actions for the same component instance.

This is important for implementing a number of integration patterns, such as the competing consumers pattern, within an integration process.

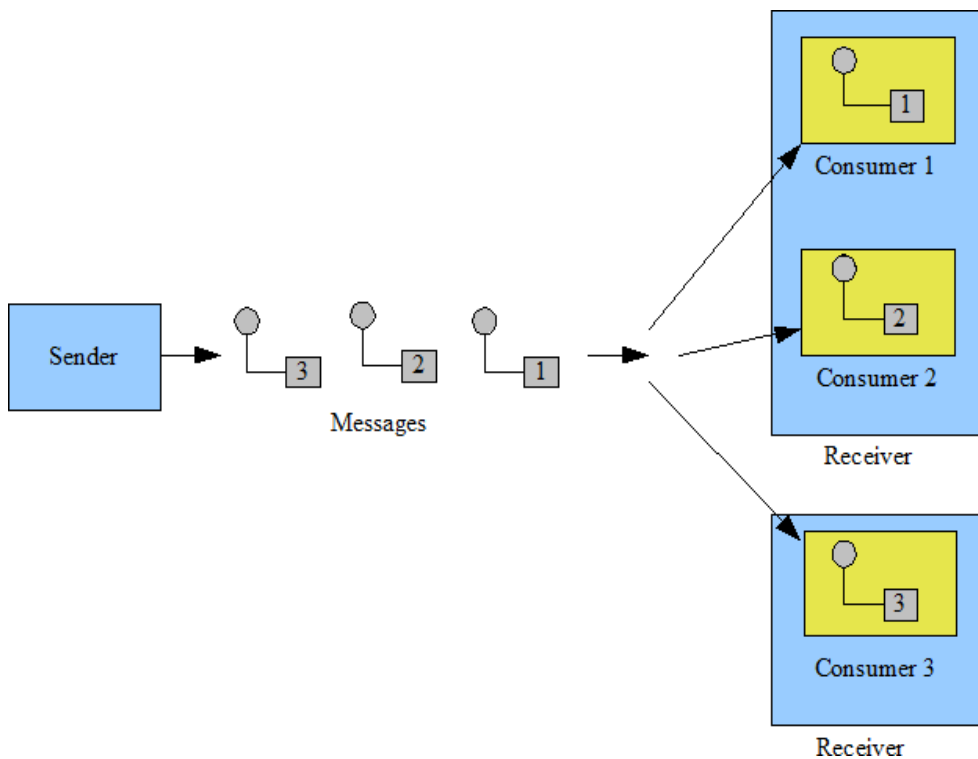


Figure 4-1 Competing Consumers

Transactions

The component framework allows transactional components to participate in a global transaction if the component supports the TransactionActions interface.



Figure 4-2 TransactionActions Interface from the MCF SDK

Uniformity

The Message Component Framework provides a consistent environment for all component types. See “Component Types” on page 4-9. The following bullets illustrate this:

Message Uniformity

The framework is responsible for providing clean message containers for the results of component actions. When a component needs to return a new message, the framework ensures that the message body and message properties have been cleared. See the Standard Message Interface graphic below.

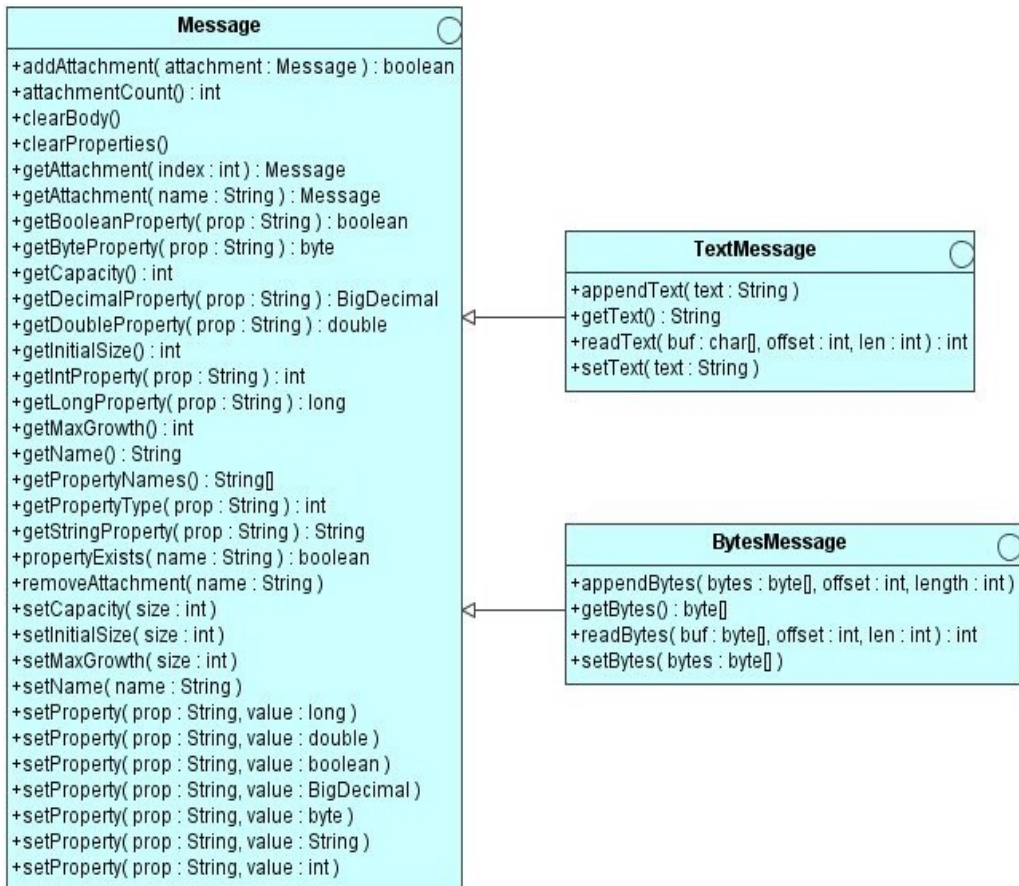


Figure 4-3 Standard Message Interfaces

Lifecycle Uniformity

Message Component Framework provides a consistent life cycle for all component types during both design time and run time. The following order is guaranteed for each component:

- A component container is created if no components from the component deployment package have yet to be used during design time or run time.

This includes the creation of a shared `com.pervasive.cosmos.component.ClassEnvironment` instance.

- A custom `java.lang.ClassLoader` is created to be shared by all components within the component deployment package.
- The selected component is loaded using the custom `java.lang.ClassLoader`
- Static initializers defined within the component implementation class will run before instantiation of a component.
- An instance of the component is created using the component public null constructor.

All components must provide a public null constructor.

- The component method `setEnvironment(com.pervasive.cosmos.component.Environment)` is called.

All components must implement the `com.pervasive.cosmos.component.Component` interface.

- The component `initialize()` method is called.

Component developers should generally consider the `initialize()` method to be the beginning of the component lifecycle. This is the point where enough of the component's surrounding environment has been initialized.

- If the component implements the `com.pervasive.cosmos.component.EnvironmentChangeListener` interface, then its `environmentChanged(com.pervasive.cosmos.component.EnvironmentChangedEvent)` method is called whenever a user makes a change to an option in the Process Editor.

It will also be called each time an option is set on the component during process execution.

- Process Execution:

See the "Process Execution Uniformity" section for more information.

- Finally, the component `destroy()` method is called when it is no longer

needed.

This generally happens at the end of a process or when the Process Editor is closed.

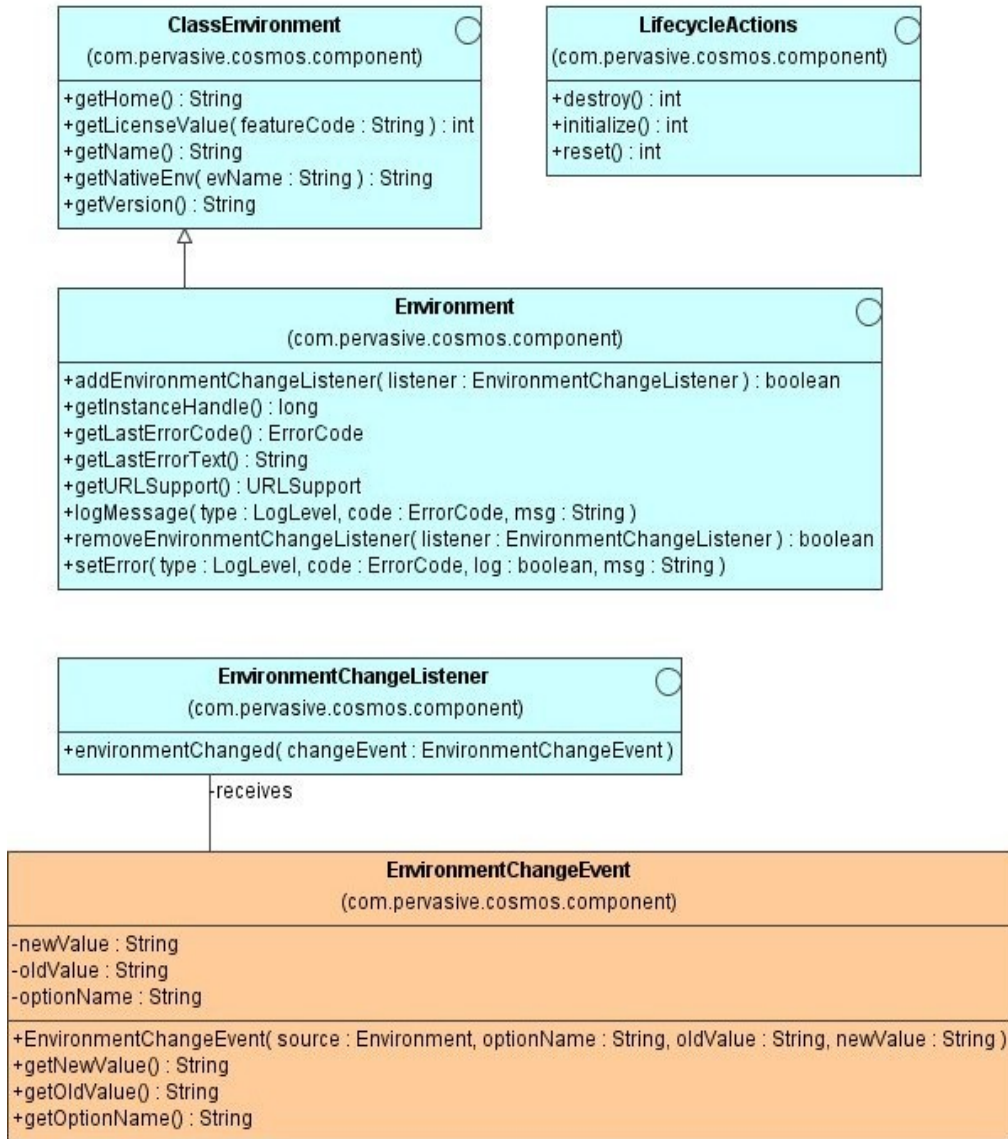


Figure 4-4 Classes and Interfaces Used in the Full Lifecycle

Process Execution Uniformity

During execution of a Process, the following actions are guaranteed to be called on a component during “Process Execution in “Lifecycle Uniformity”.

- If the component is an instance of *com.pervasive.cosmos.component.ConnectionActions* its *isConnected()* will be called before each action associated with a process step. If the *isConnected()* action returns false, the *connect* (*com.pervasive.cosmos.component.ConnectionTarget*) action will be called. The component *disconnect()* action will automatically be called at the end of the process. This behavior will not occur when components are used directly from RIFL scripts. If the developer chooses to expose the *connect()* and *disconnect()* actions in the deployment descriptor, the actions can be explicitly associated with a step. This gives the user the ability to programmatically connect and disconnect the component from its data sources and targets during process execution. The individual connection actions do not have to be defined in the deployment descriptor if the developer desires the automatic behavior described above.



Figure 4-5 Connect Actions Interface

- Step options will be set in the component environment immediately before execution of the action associated with the current step. If the component is an instance of *EnvironmentChangeListener* (see diagram), the associated *environmentChanged(EnvironmentChangeEvent)* is called for each step option immediately before execution of the component action.

Component Types

The type system in the message component framework is fairly loose. All components implement the same basic set of interfaces.

Component	Description
Aggregator	Builds a complex message object from a collection of simpler message objects.
Invoker	Provides an adaptor for external application interfaces.
Iterator	Iterate over a logical collection of messages.
Queue	Provides connectivity to a message channel.
Transformer	Transform a message object.
Validator	Validate a message object.

Component Actions

The message component framework recognizes the following types of component actions. Components will implement some subset of the complete set of actions.

Action	Description
BeginTransaction	Start a new transaction.
CommitTransaction	Commit a transaction.
Connect	Establish a connection with a resource manager.
Destroy	Called when the component is no longer needed by the process. Components can use this as a signal to perform any necessary cleanup.
Disconnect	End a connection with a resource manager.
Execute	Execute the primary function for a component.
GetMessage	Return the next message object in a collection or message channel.
Initialize	Initialize the component instance.
IsConnected	Determine if the component is connected to a source.
PrepareTransaction	Prepare a transaction for commit. Part of a two-phase commit process.
PutMessage	Add the message to a container or send the message on a message channel.
Reset	Reset component to its initial state. Provides the component with an opportunity to release resources.
RollbackTransaction	Rollback a transaction to its initial state.

Queue Component

The most common use case for MCF components is the implementation of behavior that mimics a message queue. This behavior breaks the sending or receiving of message-oriented data into enqueue and dequeue operations, respectively. The most straightforward examples of queue components are front end traditional message queue (MQ) routers, such as MSMQ, Oracle AQ, IBM MQ, or any of several JMS-based MQ routers. Actian Corporation has developed MCF components for each of the listed routers plus several others. Please see the documentation for your installation of Actian Corporation DataConnect for more information.

Queue Actions

Queue components should implement the following set of actions.

Action	Required	Description
Initialize	Yes	Initialize the component instance.
Connect	Yes	Establish a connection with a resource manager.
Disconnect	Yes	Break a connection with a resource manager.
GetMessage	Yes, if the component supports deque.	Return the next message object in a collection or message channel.
PutMessage	Yes, if the component supports enqueue.	Add a new message to the associated queue or to the data target that is being treated as a queue.
IsConnected	Yes	Return a boolean indicating the current connection state for the component.

Connect(ConnectionTarget target)

The Connect action is used to establish a connection with the queue source. The information required to make the connection comes either from property settings or from parameters to the Connect action. The ConnectionTarget parameter contains the fields defined in the following table:

Field	Description
<i>server</i>	Server name.
<i>location</i>	Resource location for the source.
<i>user</i>	User name.
<i>password</i>	Password.

GetMessage(Message message, String queue)

The GetMessage action is used to perform the actual “pop” operation on the associated queue source.

Parameter	Required	Description
<i>message</i>	Yes	When the GetMessage action is called for a queue component, the component “pops” the next message from the source and returns it in the message body of msg. The queue component may also set properties and header fields to provide additional metadata about the element.
<i>queue</i>	No	Resource location used to “pop” the next message.

The GetMessage action for a Queue component returns a result code which indicates the success or failure of the action. The typical queue component returns the following error codes:

Error Type	Required	Code	Description
ERR_OK	Yes	0	Returned if the action is successful.
(Any errors defined by DataConnect, underlying queue or data source.)	(Based on individual component requirements.)	1	Non-zero error codes returned from process steps are interpreted as an error condition. The meaning of each error code is defined by the underlying component implementation. Components should provide full documentation regarding all generated error codes.

PutMessage(Message message, String queue)

The PutMessage action is used to perform the actual “add” operation on the associated queue source.

Parameter	Required	Description
<i>message</i>	Yes	When the PutMessage action is called for a queue component, the component copies the content of the msg parameter, including properties, into a message object defined by the underlying queue implementation, and “adds” it to the remote queue (or the data target being treated as a queue).
<i>queue</i>	No	Resource location used to “add” the next message.

The **PutMessage** action for a queue component returns a result code which indicates the success or failure of the action. Typical queue components will return the following error codes:

Error Type	Required	Code	Description
ERR_OK	Yes	0	Returned if the action is successful.
(Any errors defined by DataConnect, underlying queue or data source.)	(Based on individual component requirements.	1	Non-zero error codes returned from process steps are interpreted as an error condition. The meaning of each error code is defined by the underlying component implementation. Components should provide full documentation regarding all generated error codes.

Queue Component Example

This section contains sample deployment descriptor and source code that illustrates the implementation of a basic queue component.

This sample works with files rather than an actual message queue.

Deployment Descriptor

The following deployment descriptor provides all necessary information to the MCF for locating, loading, initializing and executing actions on the queue component.

```
<Package name="Basic TestAll Samples" version="1.0.0" schemaVersion="2"
vendor="Action Corporation">
```

```
<!-- TestAll Demonstration Queue Component -->
```

```
<Component name="TestAll
    Component"version="1.0.0"
    class="Message"
    compatibleVersion="1">
```

```
<!-- Components may provide a description -->
```

```
<Description>
```

```
Demonstrates use of several of the basic MCF SDK
APIs available to queue components
```

```
</Description>
```

```
<!--
```

```
Component is implemented in Java. The top-
levelJava class to be loaded is TestAll
```

```
-->
```

```
<Java mainclass=" com.action.dc.mcfSDK.samples.TestAll"/>
```

```
<!--
```

The Model element tells the framework the type
by which this component should be classified.

It

also tells the framework which actions are supported
by the component. Some actions may also have
associated options.

This component should be classified by the framework
as a "Queue Session." It exposes the PutMessage and
GetMessage actions. The GetMessage action has an
associated option/property that can be set at the
step level.

```
-->
```

```
<Model type="queue">
  <Action type="PutMessage">
    <Parameter type="Message" usage="required"/>
    <Parameter type="Queue" usage="notused"/>
  </Action>
  <Action type="GetMessage">
    <Parameter type="Message" usage="required"/>
    <Parameter type="Queue" usage="notused"/>
    <Option type="File" name="sourceURI" fullname="Source File"/>
  </Action>
</Model>
</Component>
</Package>
```

Source Code

This source example is a simple queue component that does not use a queue directly. Instead, the `getMessage` action simply reads the first 1,000 bytes from a file using the built-in `DataConnect URLHandle` support and puts the content into the body of the `Message` object argument. Several message properties are also set by the component `getMessage` method. The `putMessage` action simply does the opposite. It prints all properties and the first 400 bytes of the message body to the process log. This component doesn't need to implement the `initialize()` method since it inherits a default implementation from `QueueComponentBase`, and it doesn't perform any special initialization procedures.

```
/*
 * TestAll.java
 *
 * Copyright (c) 2003-2022 by Actian Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.actian.dc.mcfsdk.samples;
```

```

//
// import the standard MCF SDK APIs.
//
import com.pervasive.cosmos.component.*;
import com.pervasive.cosmos.component.util.*;
import com.pervasive.cosmos.messaging.*;
import com.pervasive.cosmos.util.*;
import com.pervasive.cosmos.CosmosException;

//
// Java imports used by this component
//
import java.math.BigDecimal;
import java.net.URL;

/**
 * This class extends the convenience base class,
 * com.pervasive.cosmos.component.util.QueueComponentBase,
 * which in turn implements all the essential interfaces for a
 * queue component. Component developers are free to
 * implement the component interfaces themselves if they need
 * to extend a different base class.
 */
public class TestAll
extends QueueComponentBase
{

```

```

/**
 * components are required to provide a null constructor
 */
public TestAll()
{
}

/**
 * Populate the Message parameter with the first 1000 bytes of the file
 * referenced in the sourceURI step option. The queue parameter is not
used by this particular component.
 * @param msg the Message object to be populated by the
 *      getMessage action
 * @param queue not used by this component
 * @return error code. 0 if completed successfully.
 */
@Override
public int getMessage(Message msg, String queue)
{
    int code = ErrorCode.ERR_OK.getValue();
    Environment env = this.getEnvironment();

    //
    // Treat message as a TextMessage. Components will have to ensure
    // that this actually is a TextMessage once support is added for
    // BytesMessages.
    TextMessage message = (TextMessage) msg;

```



```

//
// get the sourceURI property that was set in the Process Editor
String sourceURI = env.getOption("sourceURI");

try
{
    // incoming message should point to some valid source url
    URLSupport urlSup = env.getURLSupport();
    if (sourceURI != null && sourceURI.length() > 0)
    {
        try
        {
            URLHandle uh = urlSup.openURL(sourceURI, "r",
                Encoding.OEM);
            if (uh == null)
            {
                message.setText("Error opening source URI " +
                    sourceURI +
                    ": " + env.getLastErrorText());
            }
        }
        else
        {
            // read until a non-blank character is found
            char ch = ' ';
            while( (ch=uh.getc()) == ' ' && ch != 0xffff );
            long pos = uh.tell();

```

```

message.appendText(
    "First Nonspace character in source is \"" + ch +
    "\" at position " + pos + ".\n");

if ( uh.seek(pos + 40, 1) > pos )
{
    message.appendText(
        "The character at position " + uh.tell() +
        " is \"" + uh.getc() + "\".\n");
}

// go back to the beginning
uh.seek(0,0);

//
// Grab first 1000 bytes of the body
char[] buf = new char[1000];
int ret = uh.read(buf, 1000);
if (ret == -1)
{
    message.setText("Error reading body file " +
        sourceURI +
        ": " + env.getLastErrorText());
}
else
{
    message.appendText(

```

```

        "The first 1000 characters of the file follows:\n" +
        new String(buf, 0, ret));
    }
    uh.close();
}
}
catch( URLHandleException urlEx )
{
    code = urlEx.getErrorCode().getValue();
    message.setText("URLHandleException occurred while " +
        "reading from source file. Message was \"" +
        urlEx.getMessage() + ".\" " +
        "Error code was " + code);
}
}
//
// Attempt to set properties of each type
message.setProperty("SourceURI", sourceURI);
message.setProperty("DecimalProperty",
    new BigDecimal(Long.MAX_VALUE));
message.setProperty("IntProperty", Integer.MIN_VALUE);
message.setProperty("DoubleProperty", Double.MAX_VALUE);
message.setProperty("BooleanProperty", true);
message.setProperty("LongProperty", Long.MAX_VALUE);
message.setProperty("ByteProperty", Byte.MAX_VALUE);

//

```

```

// set property to an environment variable
message.setProperty("SystemPath",env.getNativeEnv("PATH"));

// set classpath property
try
{
    URL[] urls = ((ComponentClassLoader)
        TestAll.class.getClassLoader()).getURLs();
    StringBuilder buffer = new StringBuilder();
    for( int j = 0; j < urls.length; j++ ) {
        buffer.append( j > 0 ? ";" : "" );
        buffer.append(urls[j].toString());
    }

    message.setProperty("ComponentLocalClassPath",
        buffer.toString());
}
catch( CosmosException exc )
{
    message.setProperty("FailedComponentLocalClassPathException",
        exc.toString());
}

}
catch( CosmosException cEx )
{

```

```

        code = ErrorCode.ERR_INVALID.getValue();
        env.setError(LogLevel.LT_WARN, ErrorCode.ERR_INVALID,
            true, cEx.getMessage());
    }
    return code;
}

/**
 * getMessage and putMessage are abstract in MessageComponentBase,
 * so they must be implemented.
 * @param msg The source DJMessage
 * @param queue queue name (not used in this component)
 * @return error code. 0 if completed successfully.
 */
@Override
public int putMessage(Message msg, String queue)
{
    // Treat message as a TextMessage. Components will have to ensure
    // that this actually is a TextMessage once support is added for
    // BytesMessages.
    TextMessage message = (TextMessage) msg;
    Environment env = this.getEnvironment();
    int code = ErrorCode.ERR_OK.getValue();
    try
    {
        String[] pnames = message.getPropertyNames();
        this.info(" Message has " + (pnames == null ? 0 : pnames.length) +

```

```

        " properties.");
String pname;
for (int i = 0; pnames != null && i < pnames.length; ++i)
{
    int ptype = msg.getPropertyType(pnames[i]);
    pname = pnames[i];
    switch (ptype)
    {
        case Message.PROPERTY_TYPE_STRING:
            this.info(" Property " + i + ":");
            this.info(" Name: " + pname);
            this.info(" Contents: " + message.getStringProperty(pname));
            break;
        case Message.PROPERTY_TYPE_INT:
            this.info(" Property " + i + ":");
            this.info(" Name: " + pname);
            this.info(" Contents: " + msg.getIntProperty(pname));
            break;
        case Message.PROPERTY_TYPE_DOUBLE:
            this.info(" Property " + i + ":");
            this.info(" Name: " + pname);
            this.info(" Contents: " + msg.getDoubleProperty(pname));
            break;
        case Message.PROPERTY_TYPE_DECIMAL:
            this.info(" Property " + i + ":");
            this.info(" Name: " + pname);
            this.info(" Contents: " + msg.getDecimalProperty(pname));

```

```

        break;
    case Message.PROPERTY_TYPE_UNKNOWN:
    default:
        this.info("Property " + i + " (" + pname +
            ") has unknown type");
        break;
    }
}
String body = message.getText();
int ilen = body.length();
if (ilen > 400)
    this.info(" Body is " + ilen + " bytes: " + body.substring(0, 400) +
        "...");
else
    this.info(" Body is " + ilen + " bytes: " + body);
}
catch( CosmosException cEx )
{
    code = ErrorCode.ERR_INVALID.getValue();
    env.setError(LogLevel.LT_WARN, ErrorCode.ERR_INVALID,
        true, cEx.getMessage() );
}
catch( Exception ioEx )
{
    code = ErrorCode.ERR_INVALID.getValue();
    env.setError( LogLevel.LT_WARN, ErrorCode.ERR_WRITERR,
        true, ioEx.getMessage() );
}

```

```

    }
    return code;
}

// log an informative message to DataConnect
private void info(String infoMsg)
{
    if ( infoMsg != null)
    {
        this.getEnvironment().logMessage(
            LogLevel.LT_INFO,
            ErrorCode.ERR_OK,
            infoMsg );
    }
}
}

```


Iterator Component

Integration solutions often need to deal with objects composed of multiple elements or to deal with large objects that need to be split into smaller objects. For example, a transaction system might box car sets of transactions into a single message. The system that processes the transactions needs to pull out each of the individual transactions from the batch.

One solution to the problem of working with composite objects is to provide a component that can decompose the object into its constituent parts.

The purpose of an Iterator component is to provide a mechanism for iterating over the elements of an object and to present each element as a separate message object.

The Iterator is an important foundation component for implementing many of the common integration patterns, such as the Splitter, Resequencer and Composed Message Processor.

An Iterator can be used to create a Splitter. The purpose of the splitter is to take something and break it up into smaller pieces.

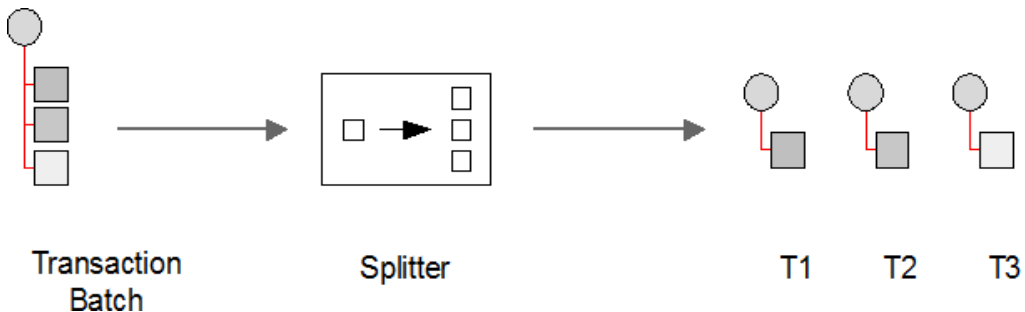


Figure 4-6 Splitter

The Iterator is used to extract the smaller pieces from the splitter source. The message object returned with each iteration can be used to create the parts that will be the outputs of the splitter.

In some cases the composition of the source object may be more complex. The elements of the object may have parent-child relationships. Similarly, each of the composite objects may be different types of objects.

Iterator Behavior

Iterator components are stateful components that know how to navigate specific kinds of collections or composed objects. The Connect action is used to connect an Iterator component instance with a data source. The data source for an Iterator is implementation specific, but most commonly they will be a file or another message object. The data source is typically specified using component property settings.

Once the Iterator source has been identified, the GetMessage action is used to advance the Iterator to the next element in the sequence and to return a copy of the element. Subsequent calls to GetMessage extract the additional elements in the sequence until the sequence is exhausted.



Note The Message Component Framework ensures that message objects used as results for the GetMessage action are cleared (properties and body) before the action is performed. This makes some of the behavior for the components more uniform and eliminates the need for each component to clear the message object before populating it with the results.

The Iterator component is like the read side of a message queue component. The main difference is that the GetMessage action for a queue changes the state of the queue by removing a message.

Iterators should not change the state of the Iterator data source. Assuming there are no changes to the data source, an Iterator will return the same results each time it is used with that source.

It is possible to have two or more Iterators connected to the same data source. The Iterators can provide the same view of the source to multiple consumers, or they can provide multiple views. It is also possible for multiple threads in a process to share the same Iterator instance. In this case each thread will see different elements of the iteration sequence.

Iterator Actions

Iterator components should implement the following set of actions:

Action	Required	Description
Initialize	Yes	Initialize the component instance.
Connect	No	Establish a connection with a source.
Disconnect	No	End a connection with a source.
GetMessage	Yes	Return the next message object in a collection or message channel.
PutMessage	No	Used to inject content via DJMessage which is to be split
IsConnected	Yes	Determine if the component is connected to a source.



Note The Connect and PutMessage actions offer alternate ways to inject source content into the iterator. One may be preferable to the other, depending on requirements.

Connect(ConnectionTarget target)

The Connect action is one of the two mechanisms available to establish a connection with the iterator source. The information required to make the connection comes either from property settings or from parameters to the

Connect action. The ConnectionTarget parameter contains the fields defined in the following table.

Field	Description
<i>server</i>	Server name.
<i>location</i>	Resource location for the server.
<i>user</i>	User name.
<i>password</i>	Password.

PutMessage(Message message, String location)

The PutMessage action offers an alternate way to inject content into the iterator via Message object.

Parameter	Required	Description
<i>message</i>	Yes	When the PutMessage action is called for an iterator, the content to be split is typically expected to be in the body of the Message argument. The iterator responds by reading the content and splitting into parts which will be retrieved via GetMessage action.
<i>location</i>	No	Typically not used in this scenario.

The PutMessage action for an Iterator component returns a result code to indicate the success or failure of the action. The Iterator component must implement each of the following error codes.

Error Type	Code	Description
ERR_OK	0	Returned if the action is successful.
(Any errors defined by DataConnect, underlying queue or data source.)	Multiple	Non-zero error codes returned from process steps are interpreted as an error condition. The meaning of each error code is defined by the underlying component implementation. Components should provide full documentation regarding all generated error codes.

GetMessage(Message message, String location)

The GetMessage action is used to perform the actual iteration over the elements in source.

Parameter	Required	Description
<i>message</i>	Yes	When the GetMessage action is called on an Iterator, the Iterator advances to the next element in the Iterator source and returns it in the message body of msg. The Iterator may also set properties and header fields to provide additional metadata about the element.
<i>location</i>	No	Resource location used to select the next message.

The GetMessage action for an Iterator component returns a result code to indicate the success or failure of the action. The Iterator component must implement each of the following error codes.

Error Type	Code	Description
ERR_OK	0	Returned if the action is successful.
ERR_EOF	1	Returned when there are no more elements left in an iteration sequence or on a queue.

Aggregator Component

Integration solutions often require filters that collect several similar or related messages in a stateful manner and produce an aggregate result (message, file, etc.) from them. For example, a transaction system might need to bill a customer only after all purchases have completed. The system that processes the transactions needs to pull information regarding each of the individual purchases and produce a single, consolidated bill.

The purpose of an Aggregator component is to provide a mechanism for collecting and storing individual messages until a complete set has been received. The Aggregator then produces a single consolidated result (message, file, etc.) as output.

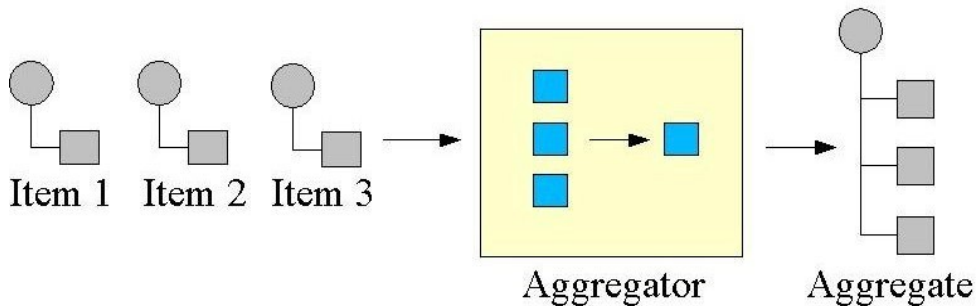


Figure 4-7 Aggregator

In some cases, the composition of source objects may be more complex. The elements of the source objects may have parent-child relationships. Similarly, the composite object may be a completely different type than the source objects.

Aggregator Behavior

Aggregators are stateful components that consume specific kinds of data or objects. These data or objects are typically messages. The data target for an Aggregator is implementation specific, but most commonly it is a file or another message object. The data target is typically specified by using the component property settings.

Once the Aggregator target is identified, the PutMessage action is used to deliver an individual part to the Aggregator. Subsequent calls to PutMessage deliver the additional elements in the sequence until the aggregation is complete. Depending on the implementation of the Aggregator, the composite result is automatically delivered to the destination (file, ftp server, etc.) or optionally can be retrieved via the

GetMessage action.

Aggregator components should not change the state of the data sources, typically the content of the messages delivered via the PutMessage action. Assuming there are no changes to the data source, an Aggregator produces the same composite result each time it is used to process the same set of source messages.

It is possible to deliver the same set of messages to two or more Aggregators that each produce different composite results. It is also possible for multiple threads in a Process to share the same Aggregator instance. In this case each thread delivers different elements to the aggregation.

Aggregator Actions

Aggregators should implement the following set of actions:

Action	Required	Description
Initialize	Yes	Initialize the component instance.
Disconnect	Yes	Break a connection with the component source.
PutMessage	Yes	Deliver the next part of the composite in a message object to the aggregator.
GetMessage	No	Return the composite result of the aggregation in a message object. This is not a required action. Most aggregators will typically send their composite results to files, remote servers, etc.
IsConnected	Yes	Return a boolean indicating the current connection state for the component.

PutMessage(Message message, String location)

The PutMessage action is used to deliver the part messages to an Aggregator component.

Action	Required	Description
<i>message</i>	Yes	When the PutMessage action is called for an aggregator, the aggregator uses the body and/or properties of the msg to add to the composite that it is creating.
<i>location</i>	No	Resource location used to retrieve the delivered message.

The PutMessage action for an Aggregator returns a result code to indicate the success or failure of the action. The Aggregator component must implement each of the following error codes:

Error Type	Code	Description
ERR_OK	0	Returned if the action is successful.
ProprietaryError Code	> ERR_TIMEEXPIRED	When possible, Aggregators should return a custom non-zero error code indicating that the aggregation is complete.

Transformer Component

Though typical transformations will be done by executing DataConnect transformation maps, sometimes the only method to accomplish a specific transformation is to write custom code. This is the case if the transformation involves connecting to an application not supported by DataConnect or the transformation is so complex that the creation of a transformation map is prohibitively difficult to accomplish. In this case, a transformer component could be written that encapsulates the logic to perform the transformation using the most efficient mechanisms available.

TransformerActions Interface

Transformer components should implement the following set of actions.

Action	Required	Description
Initialize	Yes	Initialize the component.
Execute	Yes	Transform.

Initialize()

The Initialize action is used to prepare the Transformer component for execution. Typically, it will be used to load the transformation rules and compile the transformer execution plan.

Execute(Message sourceMsg, Message targetMsg)

The Execute action is used to perform the transformation of the source identified by *sourceMsg* to a target identified by *targetMsg*.

Parameter	Required	Description
<i>sourceMsg</i>	Yes	Message object that either contains the transformation source or identifies it.
<i>targetMsg</i>	Yes	Message object that either receives the result of the transformation or identifies the location of the results.

The Execute action returns a result code to indicate the success or failure of the action. The implementation for the Execute action for a Transformer component must support the following error codes. Additional error codes can be provided by the component implementation as needed.

Error Type	Code	Description
ERR_OK	0	Returned if the action is successful.
ERR_INVALID	34	The transformation failed.

Transformer Component Example

Following is an example of a deployment descriptor and source code that implements a simple transformer component. This sample illustrates the basic instrumentation for transformation, though actual components will obviously be more complex.

Deployment Descriptor

The following deployment descriptor provides all necessary information to the MCF for locating, loading, initializing and executing actions on the transformer component.

```
<Package name="Transformer Samples" version="1.0.0" vendor="PVSW"
schemaVersion="2">

  <Component name="NullTransformer" version="1.0.0"
    class="Message" compatibleVersion="1">

    <Description>
      Sample component that performs a "null" transformation
    </Description>

    <!-- component's implementation language -->
    <Java mainclass="com.action.dc.mcfsdk.samples.NullTransformer"/>
```

```
<!-- component options (none required by this sample) -->

<!-- the component "model," which advertises the component's
      "type" (in this case, transformer) and actions, which is
      "Execute" in transformers -->
<Model type="transformer">
  <Action type="Execute">
    <Parameter type="SourceMessage"/>
    <Parameter type="TargetMessage"/>
  </Action>
</Model>
</Component>
</Package>
```

Source Code

This source example performs the simplest of all “transformations.” It simply copies the content of the source message to the target message.

```
/*
 * NullTransformer.java
 *
 * Copyright (c) 2004-2022 by Actian Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.actian.dc.mcfsdk.samples.transformersample;

import com.pervasive.cosmos.component.*;
import com.pervasive.cosmos.component.util.*;
import com.pervasive.cosmos.messaging.*;
```

```

import com.pervasive.cosmos.util.*;
import com.pervasive.cosmos.CosmosException;

import java.math.BigDecimal;
import java.io.IOException;
import java.util.logging.Logger;
import java.util.logging.Handler;
import java.util.logging.FileHandler;
import java.util.logging.SimpleFormatter;

public class NullTransformer extends TransformerComponentBase
{
    /**
     * components are required to provide a null constructor
     */
    public NullTransformer()
    {
    }

    public int execute(Message source, Message target)
    {
        int code = ErrorCode.ERR_OK.getValue();
        try
        {
            // copy properties
            String[] propNames = source.getPropertyNames();
            for( int i = 0; i < propNames.length; i++ )

```

```

{
    switch( source.getPropertyType(propNames[i]) )
    {
        case Message.PROPERTY_TYPE_STRING:
            target.setProperty(propNames[i],
source.getStringProperty(propNames[i]));
            break;
        case Message.PROPERTY_TYPE_INT:
            target.setProperty(propNames[i],
source.getIntProperty(propNames[i]));
            break;
        case Message.PROPERTY_TYPE_DOUBLE:
            target.setProperty(propNames[i],
source.getDoubleProperty(propNames[i]));
            break;
        case Message.PROPERTY_TYPE_DECIMAL:
            target.setProperty(propNames[i],
source.getDecimalProperty(propNames[i]));
            break;
        default:
            logger.warning("Source message property " +
                propNames[i] + " has unsupported type " +
                typeString(source.getPropertyType(propNames[i])));
    }
}

//
// This will need to be modified once DataConnect supports
// BytesMessage type.

```

```

        ((TextMessage)target).setText( ((TextMessage)source).getText() );
    }
    catch( CosmosException cEx )
    {
        code = ErrorCode.ERR_INVALID.getValue();
        this.getEnvironment().setError(
            LogLevel.LT_WARN,
            ErrorCode.ERR_INVALID,
            true,
            cEx.getMessage() );
    }

    return code;
}

private String typeString(int type)
{
    switch( type )
    {
        case Message.PROPERTY_TYPE_BOOLEAN:
            return "boolean";
        case Message.PROPERTY_TYPE_BYTE:
            return "byte";
        case Message.PROPERTY_TYPE_DECIMAL:
            return "decimal";
        case Message.PROPERTY_TYPE_DOUBLE:
            return "double";
    }
}

```

```
    case Message.PROPERTY_TYPE_INT:
        return "int";
    case Message.PROPERTY_TYPE_LONG:
        return "long";
    case Message.PROPERTY_TYPE_STRING:
        return "string";
    default:
        return "unknown";
    }
}

private static Logger logger;
}
```


Invoker Component

Invokers are components that adapt standard Application Programming Interfaces or remote applications. One example of this is a component that is responsible for executing business methods on a Customer Relationship Management (CRM) service. In this example, the process needs to determine which remote business method to select and the data needed to execute it. This information should be available in the source message parameter. The target message would generally contain the result of the execution of the business method.

Invoker Actions

Invoker components should implement the following set of actions:

Action	Required	Description
Execute	Yes	Invoke.

Execute(Message sourceMsg, Message targetMsg)

The Execute action is used to invoke the wrapped library or remote application using the content from the provided sourceMsg parameter as input data. The targetMsg parameter will typically contain the results of the invocation. The results could be data collected from the library or remote application or associated error messages.

Parameter	Required	Description
<i>sourceMsg</i>	Yes	Resource location used to identify the validation source.
<i>targetMsg</i>	Yes	Name of the message object used to hold the result of the Execute call.

The Execute action returns a result code to indicate the success or failure of the action. The implementation for the Execute action for an Invoker component must support the following error codes.

Error Type	Code	Description
ERR_OK	0	Returned if the action is successful.
ERR_INVALID	34	The contents of sourceMsg are not valid.

Invoker Component Example

Following is an example of a deployment descriptor and source code that implements a simple Invoker component. This sample demonstrates that some components can be useful for design-time process debugging.

Deployment Descriptor

The following deployment descriptor provides all necessary information to the MCF for locating, loading, initializing, and executing actions on the invoker component.

```
<Package name="Design Time Components" version="1.0.0" schemaVersion="2">
  <Component name="MessageBoxInvoker" version="1.0.0" class="Message">

    <Description>
      Simple Java component that pops up a message box
    </Description>

    <!-- component's implementation language -->
    <Java mainclass="com.action.dc.mcf sdk.samples.MsgBoxInvoker"/>

    <!--
      The component "model," which advertises the component's
```

"type" (in this case, transformer) and actions, which for
invokers is "Execute"

-->

```
<Model type="invoker">
  <Action type="Execute">
    <Parameter type="SourceMessage"/>
    <Parameter type="TargetMessage"/>
  </Action>
</Model>
</Component>
</Package>
```

Source Code

This source example is useful for design-time debugging. It simply displays a Swing JOptionPane with an embedded JTextArea containing the content of the source message body. The target message is not used by this component and is ignored.

```
/*
 * MsgBoxInvoker.java
 *
 * Copyright (c) 2004-2022 by Actian Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
```

- * Unless required by applicable law or agreed to in writing, software
- * distributed under the License is distributed on an "AS IS" BASIS,
- * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
- * See the License for the specific language governing permissions and
- * limitations under the License.
- */

```
package com.actian.dc.mcfsdk.samples.msgbox;
```

```
import com.pervasive.cosmos.CosmosException;
import com.pervasive.cosmos.component.util.ErrorCode;
import com.pervasive.cosmos.component.util.InvokerComponentBase;
import com.pervasive.cosmos.messaging.Message;
import com.pervasive.cosmos.messaging.TextMessage;
import com.pervasive.cosmos.util.LogLevel;
import java.awt.Dimension;
import java.math.BigDecimal;
```

```
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JDialog;
```

```
/**
```

- * Sample component that pops up a dialog with content from the source message
- * @author twaldrep
- */

```

public class MsgBoxInvoker
extends InvokerComponentBase
{

    /**
     * Creates a new instance of MsgBoxInvoker
     */
    public MsgBoxInvoker()
    {
    }

    /**
     * override the execute method
     */
    public int execute( Message source, Message target )
    {
        int rc = ErrorCode.ERR_OK.getValue();
        try
        {
            if ( source instanceof TextMessage )
            {
                //
                // This will need to be modified once DataConnect supports
                // BytesMessage type.
                String srcTxt = ((TextMessage)source).getText();
                if ( srcTxt == null )
                {

```

```

        srcTxt = "(Empty Message)";
    }

    JTextArea textArea = new JTextArea(srcTxt);
    textArea.setEditable(false);
    JScrollPane scrollPane = new JScrollPane(textArea);

    scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLB
AR_AS_NEEDED);

    scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_A
S_NEEDED);

    JOptionPane optionPane = new JOptionPane( scrollPane,
JOptionPane.PLAIN_MESSAGE );
    optionPane.setPreferredSize( new Dimension(500, 350) );
    optionPane.setOptionType( JOptionPane.OK_CANCEL_OPTION );
    JDialog dialog = optionPane.createDialog(dummyFrame, "Message
Content");
    dialog.setSize(600,400);
    dialog.show();
}
}
catch( Exception exc )
{
    this.getEnvironment().setError(
        LogLevel.LT_WARN,
        ErrorCode.ERR_INVALID,
        true,
        "Exception " + exc.toString() + " occurred while attempting " +

```

```
        "to display message content.");  
        rc = ErrorCode.ERR_INVALID.getValue();  
    }  
  
    return rc;  
}  
}
```

Validator Component

Validator components are responsible for performing custom validation on the content of an inbound message using whatever means necessary. The expected successful result is a zero-error code for the process step and appropriate messages in the target message. A nonzero result is an error condition, which will generally be proprietary to specific content. Once again, specific messages related to the error condition should be in the target message.

Validator Actions

Validator components should implement the following set of actions:

Action	Required	Description
Initialize	Yes	Initialize the Validator.
Execute	Yes	Validate.

Initialize()

The Initialize action is used to prepare the Validator component for execution.

Execute(Message sourceMsg, Message targetMsg)

The Execute action is used to perform the validation of the data specified by *sourceMsg* and return the validation results in the message object *targetMsg*.

Parameter	Required	Description
<i>sourceMsg</i>	Yes	The message containing the validation source or information about the validation source.
<i>targetMsg</i>	Yes	Name of the message object used to hold the result of the Execute call.

The Execute action returns a result code to indicate the success or failure of the action. The implementation for the Execute action for a Validator component must *minimally* support the following error codes.

Error Type	Code	Description
ERR_OK	0	Returned if the action is successful.
ERR_INVALID	34	The contents of the <i>sourceMsg</i> are not valid.

5 Component Types and Services

chapter

5

Plug-in Component Resource

This chapter contains the following topics:

- "Component Types" on page 5-2
- "Component Services" on page 5-3
- "Java Language Services" on page 5-3
- "Message Services for Components Written in Java" on page 5- 56

Component Types

Components are categorized by their general functionality. For more information on the specific functions of each component type, see Process Step documentation.

Supported component types include:

- Queue
- Iterator
- Aggregator
- Invoker
- Transformer
- Validator

Queue	Queue components behave as if they are processing message queues, i.e. GetMessage (dequeue) and PutMessage (enqueue), whether they are actually accessing queues or not.
Iterator	Iterators typically break large datasets into smaller parts which can be retrieved in order.
Aggregator	Aggregators combine multiple parts into a combined dataset which can be retrieved as a whole.
Invoker	Invokers typically execute a third party API or remote application and returns a result to the process in a single step.
Transformer	Transformer components typically perform custom conversion between two specific data formats.
Validator	Validator components perform proprietary validation on the content of an inbound message using whatever means necessary.

Component Services

Component services are functions provided by the MCF SDK to help integrate components. All components are expected to use these services to perform tasks like logging to the DataConnect engine log, set error codes and messages, get/set option/property values, etc.

Java Language Services

The Java language services are implementations of the component service contracts in the Java language. This section lists each service relative to its wrapper class.

Services provided by `com.pervasive.cosmos.component.Environment`

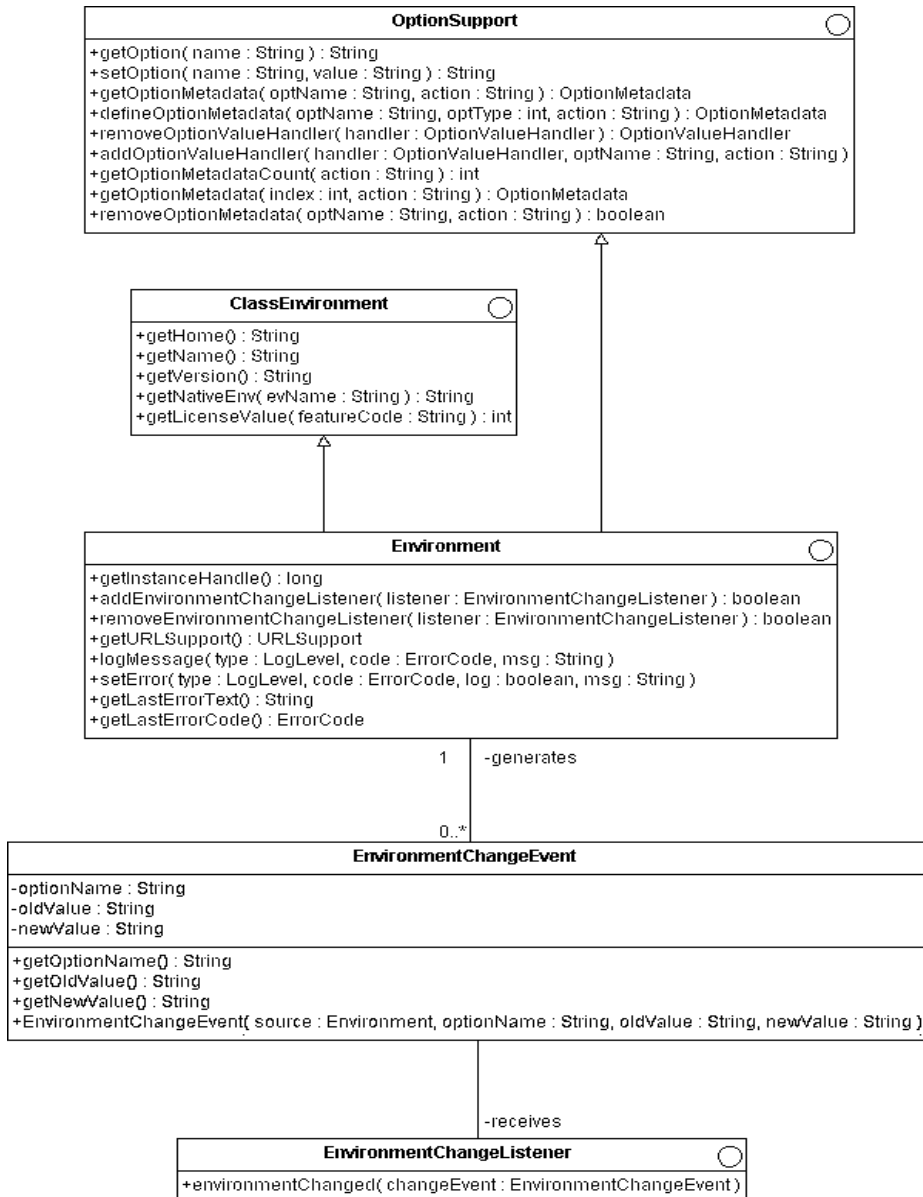
The `com.pervasive.cosmos.component.Environment` interface provides access to DataConnect engine services. The environment acts as a container for the component options/properties, similar to the behavior of an EJB (Enterprise Java Bean) context. It provides convenient access to some of the more commonly used services. For example, access to DataConnect logging, and the ability to send a signal to the Component Framework that the component has entered into an error state. The framework hands a component its Environment immediately after instantiation by calling the `setEnvironment` method. Following is a list of these services:

- `logMessage()`
- `getLicenseValue()`
- `setError()`
- `getLastErrorCode()`
- `getLastErrorText()`
- `getInstanceHandle()`
- `getHome()`
- `getName()`
- `getVersion()`
- `getURLSupport()`

Services shown below are inherited from the OptionSupport interface:

- `getOption()`
- `setOption()`
- `getOptionMetadataCount()`
- `getOptionMetadata()`
- `removeOptionMetadata()`
- `defineOptionMetadata()`
- `getOption()`

The general contract for the component Environment interface is illustrated in the class diagram presented below.



logMessage()

Writes a message to the engine log if one is open for the current thread. Return value is the error code that is passed in.

Signature: void logMessage(LogLevel type,
 ErrorCode code,
 String msg)

Arguments

Parameter	Type	Description
<i>type</i> (IN)	LogLevel Package: com.pervasive.cosmos.util	Type of message to log: LogLevel.LT_INFO = informational LogLevel.LT_WARN = warning LogLevel.LT_ERROR = error LogLevel.LT_SEVERE = severe/fatal LogLevel.LT_DEBUG = debugging
<i>code</i> (IN)	ErrorCode Package: com.pervasive.cosmos.component.util	Error code associated with this message. This can be a BATCHERR, or any other code the writer desires. This is the value that is returned.
<i>msg</i> (IN)	String	Message to be logged. This must be a fully formatted string.

getLicenseValue()

Retrieves the value for the license feature code argument.

Signature `int getLicenseValue(String featurecode)`

Arguments

Parameter	Type	Description
<i>featurecode</i> (IN)	String	A license feature code.

Remarks If there is no value for the feature code, the value of 0 is returned. This indicates that the feature is not licensed.

setError()

Sets an error code and message in the instance and logs (option) the error.

Signature `void setError(LogLevel type, ErrorCode code, boolean
log, String msg)`

Arguments

Parameter	Type	Description
<i>type</i> (IN)	LogLevelPackage: com.pervasive.cosmos.util	Type of message to log: LogLevel.LT_INFO = informational LogLevel.LT_WARN = warning LogLevel.LT_ERROR = error LogLevel.LT_SEVERE = severe/fatal

<i>code</i> (IN)	ErrorCode Package: com.pervasive.cosmos.component.util	Error code associated with this message. This can be a BATCHERR, or any other code the writer desires. This is the value that is returned.
<i>log</i> (IN)	boolean	Indicates whether this error should be logged.
<i>msg</i> (IN)	String	Error message to be saved (and possibly logged). This must be a fully formatted string.

Remarks The return value is the error code passed in.

getLastErrorCode()

Returns the last error code set on or by this instance.

Signature `ErrorCode getLastErrorCode()`

Arguments None.

Remarks Some framework services may generate errors. This along with `getLastErrorText` allows that information to be retrieved.

getLastErrorText()

Returns the last error message set on or by this instance.

Signature `String getLastErrorText()`

Arguments None.

Remarks Some framework services may generate errors. This along with `getLastErrorCode` allows that information to be retrieved.

Returns The return value may be either null or "" if no error has been set. Ownership of the string remains with the instance handle, and the pointer may become invalid the next time any framework service is called.

getInstanceHandle()

Returns the native handle to the component instance.

Signature `long getInstanceHandle()`

Arguments None.

Returns Native handle.

getHome()

Returns the component home directory as a `java.lang.String`, i.e. the location at which the component package root is extracted.

Signature `String getHome()`

Arguments None.

Returns The component home directory.

getName()

Returns the component name as specified in the deployment descriptor.

Signature `String getName()`

Arguments None.

Returns Component name.

getVersion()

Returns the component version as specified in the deployment descriptor.

Signature `String getVersion()`

Arguments None.

Returns Component version.

getURLSupport()

Returns an instance of
com.pervasive.cosmos.component.util.URLSupport.

Signature `URLSupport getURLSupport()`

Arguments None.

Returns Instance of *com.pervasive.cosmos.component.util.URLSupport*.

Remarks URLSupport instances behave as factories for
com.pervasive.cosmos.component.util.URLHandle instances.

getOption()

Returns the value of an option if it has been set.

Signature `String getOption(String optionName)`

Arguments

Parameter	Type	Description
<i>optionName</i> (IN)	String	The name of the option whose value is being retrieved.

Returns The value of a named option.

Remarks If the option value has not been set or the option isn't defined in the deployment descriptor, null is returned.

setOption()

Overrides the value of an option that has been set by the process. If the option does not exist, then a local option and value is created that is local to the instance

Signature `String setOption(String optionName, String optionValue)`

Arguments

Parameter	Type	Description
<i>optionName</i> (IN)	String	The name of the option to be set.
<i>optionValue</i> (IN)	String	The new value of the option.

Remarks This method is called by the framework to set options on the component.

Services Provided By com.pervasive.cosmos.component.OptionSupport

The following services are exposed by derivation through Environment:

- `getOptionMetadataCount()`
- `getOptionMetadata()`
- `removeOptionMetadata()`
- `defineOptionMetadata()`

getOptionMetadataCount()

Returns the number of options defined on the current component instance or relative to provided action.

Signature `int getOptionMetadataCount(String action)`

Arguments

Parameter	Type	Description
<i>action</i> (IN)	String	Specifies the action with which option count is associated. If this value is null, then returned count will be relative to component instance.

Returns Option count relative to action (if action parameter is not null), otherwise returns the count relative to component instance.

getOptionMetadata()

Returns an instance of OptionMetadata through which modifications can be made to option metadata, such as the option alias, allowable enumeration values, and more.

Signature OptionMetadata getOptionMetadata(int index, String action)

Arguments

Parameter	Type	Description
<i>index</i> (IN)	int	Index of the option within the instance or action
<i>action</i> (IN)	String	Name of the action with which the option is associated or null if the option is not associated with a specific action

Returns Matching OptionMetadata instance

Remarks Throws an IndexOutOfBoundsException if no ComponentOptionMetadata is defined at the provided index.

getOptionMetadata()

Returns an instance of OptionMetadata through which modifications can be made to option metadata, such as the option alias, allowable enumeration values, and more. Returns null if the option is not defined for the current component instance.

Signature OptionMetadata getOptionMetadata(String optionName, String action)

Arguments

Parameter	Type	Description
<i>optionName</i> (IN)	String	The name of the requested option.
<i>action</i> (IN)	String	Name of the action with which the option is associated or null if the option is not associated with a specific action

Returns Matching OptionMetadata instance or null if not found.

removeOptionMetadata()

Removes the named option and its associated metadata. If the action parameter is provided, the option is removed only if it is associated with the action.

Signature `boolean removeOptionMetadata(String optionName, String action)`

Arguments

Parameter	Type	Description
<i>optionName</i> (IN)	String	The name of the requested option.
<i>action</i> (IN)	String	Name of the action with which the option is associated or null if the option is not associated with a specific action

Returns True if found and removed; false otherwise.

defineOptionMetadata()

Defines option metadata associated with this component instance. If the option does not exist, a new option is created and returned. If the option does exist, the current instance is returned.

Signature boolean defineOptionMetadata(String optionName, intoptionType, String action)

Arguments

Parameter	Type	Description
<i>optionName</i> (IN)	String	The name of the option to retrieve or create.
<i>optionType</i> (IN)	int	The type of the option if it is to be created. Valid types are shown below: <ul style="list-style-type: none">· OptionMetadata.ENUM_TYPE<ul style="list-style-type: none">○ Option is an enumeration· OptionMetadata.NUMBER_TYPE<ul style="list-style-type: none">○ Option is a number· OptionMetadata.TEXT_TYPE<ul style="list-style-type: none">○ Option is text· OptionMetadata.FILE_TYPE<ul style="list-style-type: none">○ Option is a file· OptionMetadata.CUSTOM_TYPE<ul style="list-style-type: none">○ Option is custom. Only the component can provide its value. Custom options are described in a different section.
<i>action</i> (IN)	String	Name of the action with which the option is associated or null if the option is not associated with a specific action

Returns Newly created or existing ComponentMetadata instance.

addCustomOptionHandler()

Add a CustomOptionHandler instance with the provided option name and associated action (if any) to provide values for custom options. If the provided option name is null, then the handler will be called to handle all custom options for which no explicit handler has been registered.

Signature void addCustomOptionHandler (CustomOptionHandler handler, String optName, String action)

Arguments

Parameter	Type	Description
<i>handler</i> (IN)	CustomOptionHandler	An instance of CustomOptionHandler.
<i>optName</i> (IN)	String	Name of option for which this handler provides values.
<i>action</i> (IN)	String	Name of the action with which the option is associated or null if the option is not associated with a specific action

removeCustomOptionHandler()

Removes a previously added CustomOptionHandler. This method has no effect if the provided CustomOptionHandler has not previously been added.

Signature CustomOptionHandler
removeCustomOptionHandler(CustomOptionHandler handler)

Arguments

Parameter	Type	Description
<i>handler</i> (IN)	CustomOptionHandler	The specific instance of CustomOptionHandler to remove.

Returns The CustomOptionHandler instance that was removed or null if it has not been added.

Services Provided by **com.pervasive.cosmos.component.CustomOptionHandler**

Components that need to present a dialog or use environmental information to properly set the value for an option should implement this interface. The concept is simple. The framework will make a blocking call to the registered CustomOptionHandler to retrieve the value for the associated option. The component is free to do whatever is necessary to collect all information required to provide a return value.

getOptionValue()

Signature `String getOptionValue(String optName, String action, String oldval)`

Arguments

Parameter	Type	Description
<i>optName (IN)</i>	String	Name of the option for which the value will be provided by the component.
<i>action (IN)</i>	String	Name of the associated action if the value is being provided for a step option.
<i>oldval</i>	String	

Returns Passes a message instance and a queue name. If the queue is not needed (according to the deployment descriptor), it is passed as a null.

Remarks The component is expected to populate the body and, optionally, the properties of the provided message with retrieved content (from a message queue or other data source) or set an error condition if it fails.

Services Provided by com.pervasive.cosmos.component.OptionMetadata

Message Component Framework provides a fine-grained mechanism through which components can modify information about options that will, in turn, be presented in the Process Editor. The OptionMetadata interface serves as the contract for making these modifications. Following is a list of methods provided by the OptionMetadata interface:

- getName()
- getType()
- getAlias()
- setAlias()
- getDescription()
- setDescription()
- getDefaultValue()
- setDefaultValue()
- getMinValue()
- setMinValue()
- getMaxValue()
- setMaxValue()
- getAllowableValueCount()
- addAllowableValue()
- addAllowableValue()
- clearAllowableValues()
- setDirty()
- isDirty()

getName()

Returns the name of the associated option. This information is read- only.

Signature String getName()

Arguments None

Returns The name of the option as a string.

getType()

Returns the option type. This information is read-only.

Signature int getType()

Arguments None

Returns The type of the option as an int. The returned type will be one of the following:

- OptionMetadata.ENUM_TYPE
- OptionMetadata.NUMBER_TYPE
- OptionMetadata.TEXT_TYPE
- OptionMetadata.FILE_TYPE
- OptionMetadata.CUSTOM_TYPE

getAlias()

Returns the option alias, for example, the display name of the option presented by the Process Editor. If no alias is set, the return value for this method is the option name. This is consistent with the Process Editor, which displays the option name if the alias is not set.

Signature String getAlias()

Arguments None

Returns The alias as a string or the name as a string if the alias is not set.

setAlias()

Sets the option alias. The alias is the long name for the option displayed by the Process Editor. Setting the option alias has no effect on the option name.

Signature void setAlias(String alias)

Arguments

Parameter	Type	Description
<i>alias</i> (IN)	String	String value of the new alias.

Returns None.

getDescription()

Returns the option description.

Signature String getDescription()

Arguments None

Returns The description as a string or null if no description has been set.

setDescription()

Sets the option description. This method has no effect if the provided description is null.

Signature void setDescription(String description)

Arguments

Parameter	Type	Description
<i>description</i> (IN)	String	String value of the new description.

Returns None.

getDefaultValue()

Returns the option default value if previously set.

Signature String getDefaultValue()

Arguments None

Returns The default value as a string (regardless of type) or null if no default value has been set.

setDefaultValue()

Sets the option default value. This method has no effect if the provided default value is null.

Signature void setDefaultValue(String defaultValue)

Arguments

Parameter	Type	Description
<i>defaultValue</i> (IN)	String	String value of the new default value.

Returns None.

getMinValue()

Returns the option minimum allowable value. If set, the Process Editor will not allow the user to enter a smaller value.

Signature String getMinValue()

Arguments None

Returns The minimum value as a string or null if no minimum value has been set.

setMinValue()

Sets the option minimum value. This method has no effect if the provided minimum value is null.

Signature void setMinValue(String minValue)

Arguments

Parameter	Type	Description
<i>minValue</i> (IN)	String	String value of the minimum value.

Returns None.

getMaxValue()

Returns the option maximum allowable value. If set, the Process Editor will not allow the user to enter a larger value.

Signature String getMaxValue()

Arguments None

Returns The maximum value as a string or null if no maximum value has been set.

setMaxValue()

Sets the option maximum value. This method has no effect if the provided maximum value is null.

Signature void setMinValue(String maxVal)

Arguments

Parameter	Type	Description
<i>maxValue</i> (IN)	String	String value of the maximum value.

Returns None.

getAllowableValueCount()

Gets a count of allowable values for this option. This method is only meaningful for options of type Enum.

Signature `int getAllowableValueCount()`

Arguments None

Returns Allowable value count as an int or -1 if not applicable.

getAllowableValue()

Gets the AllowableValue at the specified index. This method is typically used in conjunction with the `getAllowableValueCount()` method. It is only meaningful to use this method with options of type Enum.

Signature `AllowableValue getAllowableValue(int index)`

Arguments `int index`

Index of the requested allowable value.

Returns AllowableValue instance at the specified index.

Remarks An `IndexOutOfBoundsException` is thrown if the provided index is out of range.

addAllowableValue()

Add a new allowable value for this option. This method is only meaningful with respect to Enum-type options.

Signature void addAllowableValue(String value, String display)

Arguments

Parameter	Type	Description
<i>value</i> (IN)	String	The new allowable value.
<i>display</i> (IN)	String	The value to be displayed in the Process Editor for the allowable value.

Returns None.

clearAllowableValues()

Clears all allowable values for this option. This method is typically used if the component needs to define a new set of allowable values for this option. As with the other AllowableValue methods, this method is only meaningful with Enum-type options.

Signature void clearAllowableValues()

Arguments None.

Returns None.

setDirty()

Toggles this option as dirty or not dirty. Generally, any change to the metadata of an option such as adding or removing allowable values, changing the default value, or changing the alias will mark the option as dirty. A dirty option is a signal to the Process Editor to clear old values for the option set by the user and revert back to a "clean" state. The developer may not wish for this to happen under some circumstances, or may want to mark the option as dirty without making any changes to option metadata. For both of these reasons, `setDirty()` is provided to the component developer.

Signature `void setDirty(boolean dirty)`

Arguments

Parameter	Type	Description
<i>dirty</i> (IN)	boolean	Dirty if true, not dirty if false.

Returns None.

isDirty()

Returns true if this option is currently marked as dirty. See the `setDirty()` method for a definition of "dirty."

Signature `boolean isDirty()`

Arguments None.

Returns True if the option is dirty, false otherwise.

Services Provided by `com.pervasive.cosmos.component.util.URLSupport`

URLSupport provides convenience methods that facilitate using the DataConnect engine built-in File IO support to open a provided URL. A `com.pervasive.cosmos.component.util.URLSupport.openURL()` provides this service. Any URL mechanism that is supported by DataConnect is valid.

openURL()

Opens a URL for reading or writing.

Signature `URLHandle openURL(String url, String mode, Encoding encoding)`

Arguments

Parameter	Type	Description
<code>url</code> (IN)	String	The URL to open.
<code>mode</code> (IN)	String	The mode which will be used to open the file, i.e. "r", "w" or "a" with optional "+" and/or "b"
<code>encoding</code> (IN)	Encoding Package: <code>com.pervasive.cosmos.util.Encoding</code>	The character encoding of the content of the URL's target

Returns Returns null if on error, in which case more information may be retrieved via the `getLastErrorXXX` calls. The return value is a `URLHandle` which should be used with the other `xxxURL` calls.

Remarks URLs may be regular URLs, or may simply be a filename. The supported protocol schemes are:

file	HTTPS	xml:ref
FTP	gzip	djmessage
HTTP	xml:db	djstream

Characters are always read as Unicode characters (wchar_t). The encoding parameter specifies what encoding the original source or target is in, so that the conversion may be performed. The special value PLENC OEM indicates that the default encoding of the system should be used. The ISO-8859-1 encoding is handled specially; it simply reads the bytes and widens them by setting the upper byte(s) of the Unicode character to 0.

Services Provided By `com.pervasive.cosmos.component.util.URLHandle`

URLHandle provides DataConnect File IO services related to a URL opened by URLSupport. Following is a list of these services:

- `URLHandle.write()`
- `URLHandle.close()`
- `URLHandle.read()`
- `URLHandle.getc()`
- `URLHandle.putc()`
- `URLHandle.seek()`
- `URLHandle. tell()`

URLHandle.close()

Closes a handle returned by openURL.

Signature `int close()`

Arguments None.

Returns Returns zero if successful, non-zero if error.

Remarks Indicates that the component instance has completed its use of a particular file. After calling closeURL the handle is no longer valid.

URLHandle.read()

Reads characters from the URL into the specified array.

Signature `int read(char[] buf, int len)`

Arguments

Parameter	Type	Description
<i>buf</i> (OUT)	char	Character array to be populated by the read operation.
<i>len</i> (IN)	int	Maximum number of characters to read.

Returns The return value is the number of characters actually read, or -1 if an error occurs.

URLHandle.write()

Writes characters to the URL.

Signature `int write(char[] buf, int len)`

Arguments

Parameter	Type	Description
<i>buf</i> (IN)	char[]	Buffer of characters to write.
<i>len</i> (IN)	int	Number of characters to write from buffer.

Returns Return value is the number of characters written.

Remarks This method returns LEN. A return value of -1 indicates an error occurred.

URLHandle.getc()

Returns the next character available from the source, or the special value 0xffff if at EOF.

Signature `char getc()`

Arguments None.

URLHandle.putc()

Writes the character to the URL.

Signature `char putc()`

Arguments None.

Returns A return of `-1` indicates an error occurred. Any other value indicates success.

URLHandle.seek()

Seeks to the specified position in the file.

Signature `int seek(long where, int how)`

Arguments

Parameter	Type	Description
<i>where</i> (IN)	long	Position to seek to.
<i>how</i> (IN)	int	Modifier to indicate whether to position absolute (SEEK_SET, 0), relative to current position (SEEK_CUR, 1), or backwards from the end (SEEK_END, 2).

Returns The new file position is returned, or `-1` to indicate an error occurred.

Remarks There *where* and *how* Arguments are the same as the standard C seek function.

URLHandle.tell()

Returns the current file position.

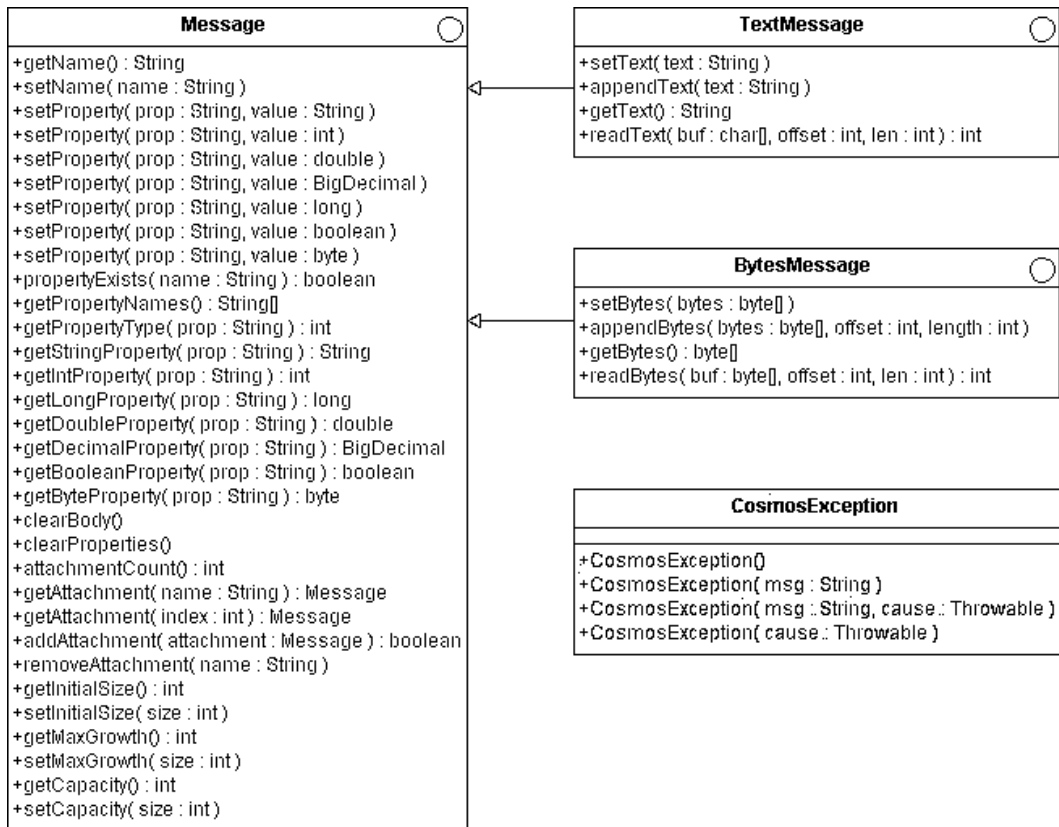
Signature `long tell()`

Arguments None.

Message Services for Components Written in Java

Java components use a standard Message interface shared with other DataConnect Java SDKs. Currently, the framework supports passing instances of *com.pervasive.cosmos.messaging.TextMessage* to and from components. With this said, the actual interface passed into SDK methods, such as *getMessage()*, *putMessage()*, and *peekMessage()*, is *com.pervasive.cosmos.messaging.Message*.

The reason for this abstraction is to provide future support for *com.pervasive.cosmos.messaging.BytesMessages*. The following class diagram shows the contract for DataConnect Message installation:



6 *Component Contracts*

6

Plug-in Component Developer Resource

In this chapter, you will find advanced level information. This chapter is divided into the following topics:

- "Component Interface" on page 6-2
- "Component Constructor" on page 6-20

The framework contains adaptor layers for Java or C components. This section describes the requirements for components written in Java.

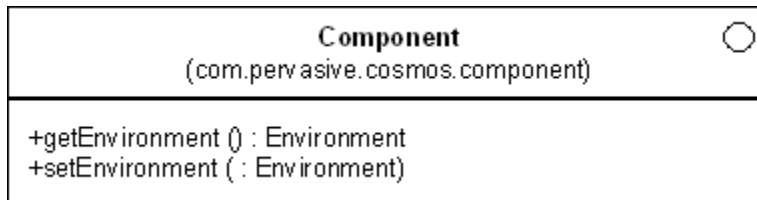
Java The Message Component Framework provides a series of interfaces. A component developer should first understand these two important interfaces:

- `com.pervasive.cosmos.component.Component` Interface
- `com.pervasive.cosmos.component.ComponentActions` Interface

com.pervasive.cosmos.component Contracts

Component Interface

The Component interface establishes the basic contract for components without requiring any component actions. The UML class diagram for the Component interface follows:



This section describes the following selected actions that may be supported by components.

- `Component.setEnvironment()`
- `ComponentActions` Interface
- `PutMessageActions.putMessage()`
- `ExecuteActions.execute()`
- `LifecycleActions.initialize()`
- `LifecycleActions.reset()`
- `LifecycleActions.destroy()`
- `ConnectionActions.connect()`
- `ConnectionActions.disconnect()`
- `ConnectionActions.isConnected()`
- `TransactionActions.beginTransaction()`
- `TransactionActions.prepareTransaction()`
- `TransactionActions.commitTransaction()`
- `TransactionActions.rollbackTransaction()`

- GetMessageActions.getMessage()
- Other Techniques or Contracts
- Component Constructor

Component.setEnvironment()

This method is called by the component framework to set a component's Environment instance. It is called immediately after component instantiation.

Signature `void setEnvironment(Environment environment)`

Arguments

Parameter	Type	Description
<i>environment</i> (IN)	Environment	Set component Environment instance.

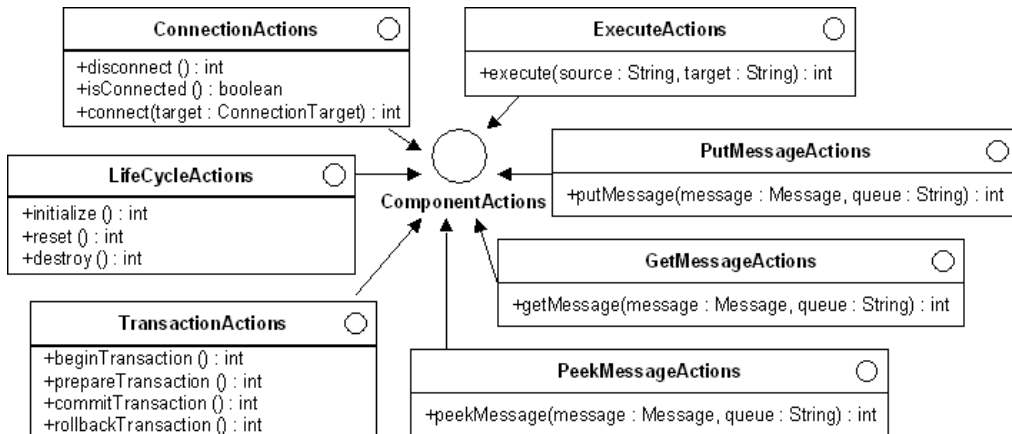
ComponentActions Interface

The ComponentActions interface is a marker, but extensions of it define specific actions that are recognized by the framework. Java components must implement the *com.pervasive.cosmos.component.Component* interface as well as support one or more of the extensions of the *com.pervasive.cosmos.component.ComponentActions* interface.



Note Components that do not implement Component are not recognized as valid. Components that do not implement at least one *ComponentActions* interface are not deemed to support any actions.

The UML diagram for the ComponentActions interface and its currently defined extensions follows.



PutMessageActions.putMessage()

The PutMessage action is used to inject the content (either direct or described) of the provided message into the target system.

Signature `int putMessage(Message message, String queue)`

Arguments

Parameter	Type	Description
<i>message</i> (INOUT)	Message	The message delivered to the component.
<i>queue</i> (IN)	String	The name of the associated queue, or null if not applicable.

Returns Passes a message instance and a queue name. If the queue is not needed (according to the deployment descriptor), it is passed as a null.

Remarks The component is expected to use the content of the message as necessary (forward to an external queue, etc.) or set an error condition if it fails.

ExecuteActions.execute()

The execute action is implemented by components that perform some type of data movement, conversion, or some other action from a provided source to a provided target.

Signature `int execute(Message inMessage, Message outMessage)`

Arguments

Parameter	Type	Description
<i>inMessage</i> (INOUT)	Message	The message delivered to the component
<i>outMessage</i> (INOUT)	Message	The result message of the action to be used by subsequent steps

Returns The component is expected to set an error condition if a failure occurs while performing its task.

LifecycleActions.initialize()

The Component Framework calls this method immediately after component instantiation. License checks (if required) are typically performed in this action.

Signature `int initialize()`

Arguments None.

Returns The component is expected to set an error condition if it fails during initialization.

Remarks Called by the framework to request that the component initialize itself.



Note Component developers should consider the initialize method to be the beginning of the component lifecycle.

LifecycleActions.reset()

Called explicitly as an action to request that the component reset itself.

Signature `int reset()`

Arguments None.

Returns The component is expected to set an error condition if it fails to reset.

LifecycleActions.destroy()

Called by the framework just before the component and immediately before disposal.

Signature `int destroy()`

Arguments None.

Returns The component is expected to set an error condition if it fails during final cleanup.

Remarks Called by the framework immediately prior to disposal. This gives the component an opportunity to perform final cleanup.

ConnectionActions.connect()

Establishes a connection to the message source/destination.

Signature `int connect(ConnectionTarget target)`

Arguments

Parameter	Type	Description
<i>target</i> (IN)	ConnectionTarget	The target to which the component should connect.

Returns The component is expected to set an error condition if it fails to connect.

Remarks This action may involve connecting to a messaging service on a specified host, opening a directory on a local filesystem, or some other scenario. The connection information is passed within a ConnectionTarget wrapper. The deployment descriptor will specify which fields within the ConnectionTarget are actually used.



Note Unused arguments may be passed as either empty strings or null.

ConnectionActions.disconnect()

Called to instruct the component to disconnect from the message source/destination, if a connection is currently established.

Signature `int disconnect()`

Arguments None.

Returns The component is expected to set an error condition if it fails to disconnect properly.

ConnectionActions.isConnected()

Indicates that the Message Component is currently connected.

Signature `boolean isConnected()`

Arguments None.

Returns None.

TransactionActions.beginTransaction()

Signal to the component to begin a transaction with its data source or target.

Signature `int beginTransaction()`

Arguments None.

Return Value The component is expected to set an error condition if it fails to begin the transaction.

TransactionActions.prepareTransaction()

Gives a transactional, two-phase component the opportunity to invoke *prepare()*.

Signature `int prepareTransaction()`

Arguments None.

Returns The component is expected to set an error condition if the request to “prepare” fails.

Remarks If the component session with its data source or target is transactional and supports two-phase commit, this call invokes *prepare()*.

TransactionActions.commitTransaction()

Requests that the component commit its transaction with its source/target.

Signature `int commitTransaction()`

Arguments None.

Returns The component is expected to set an error condition if its attempt to commit the transaction fails.

TransactionActions.rollbackTransaction()

Requests that the component rollback or undo the uncommitted transaction with its data source/target.

Signature `int rollbackTransaction()`

Arguments None.

Returns The component is expected to set an error condition if the rollback fails.

GetMessageActions.getMessage()

Retrieves a message from a source, such as a message queue.

Signature `int getMessage(Message message, String queue)`

Arguments

Parameter	Type	Description
<i>message (INOUT)</i>	Message	Message to be populated by the component.
<i>queue (IN)</i>	String	Name of the associated queue if required by the component. Can be null.

Returns Passes a message instance and a queue name. If the queue is not needed (according to the deployment descriptor), it is passed as a null.

Remarks The component is expected to populate the body and optionally the properties of the provided message with retrieved content (from a message queue or other data source) or set an error condition if it fails.

Other Techniques or Contracts

Other Java techniques or Contracts not listed in the diagrams are expressed as follows:

Signature `static { statements; ... }`
 `[ClassName].class.getClassLoader()`
 `static { java statements; ... }`

Arguments None.

Returns The first path element of the `ClassLoader` returned by `[ClassName].class.getClassLoader()` is the home location of the component.

Remarks Java components may optionally get the equivalent of the *onLoad(path)* method by creating a static block within within their primary component class.

Component Constructor

Components must provide a null or no argument constructor so that the framework is able to create instances of it.

Signature `public ClassName()`

Arguments None.

Returns None.

7 Deployment Descriptor

Plug-in Component Developer's Resource

In this chapter, you will find advanced level information. This chapter is divided into the following topics:

- "Deployment Descriptor" on page 7-2
- "Package Element" on page 7-6
- "Description Element" on page 7-7
- "Option Element" on page 7-8
- "Enum Element" on page 7-10
- "Java Element" on page 7-12
- "Classpath" on page 7-13
- "Model Element" on page 7-14
- "Action Element" on page 7-15
- "Parameter Element" on page 7-16
- "XML Schema" on page 7-17

Deployment Descriptor

The deployment descriptor provides metadata to Message Component Framework Component Manager that is needed to interface with components at runtime and during design in the graphical user interface. This metadata includes the following for the package:

- Name of the package (required attribute)
- Version of the package (required attribute)
- Vendor of the package (optional attribute)
- Version of the deployment descriptor schema to which this deployment descriptor conforms (optional attribute)
- Description of the package (optional element)
- Shared option definitions (optional elements)
- Component definitions (one or more required elements)

Each Component in the package encapsulates the following information:

- Name of the Component (required attribute)
- Version of the Component (required attribute)
- Type of the Component (required attribute with value of "Message")
- Programming language of Component (required element, currently Java is publicly supported)
- Locally defined options (zero or more elements)
- Model type (required element that defines the component category and encapsulates actions)
- Actions (one or more required elements within the component's Model)

The deployment descriptor is an XML document that consists of a single Package element that describes the package name, package version, and vendor. The example deployment descriptor below is referenced in the sections that follow:

```
<Package vendor="Vendor Software, Inc." name="Optional name of package"
    version="Optional package version" schemaVersion="2">

    <!-- Optional package description -->
```

<Description>

This is an example deployment descriptor

</Description>

<!--

Packages can have option definitions that are shared across all components

-->

<Option type="Number" name="SharedOpt" fullname="Common set option"
default="0"/>

<!--

Each component within a package is represented by a Component element.

-->

<Component name="Component 1" version="1.0.0" class="Message"
compatibleVersion="1">

<!-- Optional component description -->

<Description>

First component in the package

</Description>

<!--

The component is implemented in Java. Its' main component class is
com.vendor.component.Component1.

-->

<Java mainclass="com.vendor.component.Component1"/>

```
<!--
```

Option definitions scoped to a single component

```
-->
```

```
<Option type="Text" name="compOpt1" fullname="Component Option 1"
      default=""/>
```

```
<Option type="Text" name="compOpt2" fullname="Option 2" default=""/>
```

```
<!--
```

The model represents a component's type and all of its supported actions. This component represents a "queue session" that supports the "GetMessage" action.

```
-->
```

```
<Model type="queue">
```

```
  <Action type="GetMessage">
```

```
    <!--
```

Two arguments are required by this action.

```
-->
```

```
    <Parameter type="Message" usage="required"/>
```

```
    <Parameter type="Queue" usage="required"/>
```

```
  </Action>
```

```
</Model>
```

```
</Component>
```

```
<!--
```

Packages can define multiple components.

```

-->
<Component name="Component 2" version="1.1.0" class="Message"
    compatibleVersion="1">

    <!-- Optional component description -->
    <Description>
        Second component in the package
    </Description>

    <!--
    The component is implemented in Java. Its' main component class is
    com.vendor.component.Component2.
    -->
    <Java mainclass="com.vendor.component.Component2"/>

    <!--
    The model represents a component's type and all of its supported
    actions. This component is an "aggregator" that supports the
    "PutMessage" action.
    -->
    <Model type="aggregator">
        <Action type="PutMessage">
            <!--
            PutMessage takes two arguments. This component requires the
            "Message" parameter, but it doesn't use the "Queue" parameter.
            -->
            <Parameter type="Message" usage="required"/>

```

```
<Parameter type="Queue" usage="notused"/>
</Action>
</Model>
</Component>
</Package>
```

This example contains two components. One is a queue component, the other is an aggregator. The following statements describe the first component within the package:

The first component is version 1.0.0 of a component named “Component 1.” It is implemented in Java, and its primary component class is `com.vendor.component.Component1`. It is a queue component that supports the following action:

- GetMessage - Responsible for retrieving the next message from an associated queue. This method accepts the following two arguments:
 - Message – The message object that the component is expected to populate from the associated queue.
 - Queue – The queue from which to populate the Message parameter.

The following three options are explicitly supported by this component:

- SharedOpt - Number type option inherited from the Package level.
- compOpt - Locally-defined Text option.
- compOpt2 - Locally-defined Text option.

Package Element

The package element serves as the root element of the deployment descriptor. It defines the package name, version, vendor, schema version, description, shared options, and component definitions.

name Attribute

Provides the name of the package. This serves as the root of the deployed namespace of the component.

version Attribute

Defines the package version. This serves as the second level of the deployed namespace of the component. The name and version combined serve as the deployed home directory of the component relative to the locally-defined Plugins directory.

vendor Attribute

Defines the vendor of the package. This attribute is optional and is not used within DataConnect. It is in place for future enhancements to metadata displayed for each component.

schemaVersion Attribute

Specifies the version of the Message Component Framework deployment descriptor schema to which this deployment descriptor conforms. The maximum version of the deployment descriptor schema currently supported by DataConnect is “2”.

Description Element

This element appears as a child of several elements. It provides descriptive text for the containing element/object, used by the graphical user interface. All text content is part of the description. Leading and trailing whitespace is stripped.

Option Element

Each Option element describes one property/option that may be set for the component. These options can have different types:

- Text – This option type allows anything to be entered. A list of enum values is allowed. If present, they suggest a dropdown list that may be displayed. However, the option value is not required to be one of these values.
- Enum – This option type restricts values to one of the enum values provided.
- Number – This option type is a number. Either integer or decimal values are allowed. A list of enum values is allowed, as in the Text type. In addition, a maximum and/or minimum value may be specified.
- Boolean – Allows a true/false value only.
- File – Intended use of this option is to represent a file or universal resource indicator. Within the component no special handling is used, however it indicates to a graphical user interface that a file selection dialog should be used when setting this option.

name Attribute

Name of the option. This is used both in display lists for graphical user interface access and for programmatically setting options such as command-line overrides, connect strings, and more.

type Attribute

Specifies the type of the option. Must be one of text, enum, number, or boolean.

min Attribute

If the type of the option is number, this attribute sets an optional minimum value for the number. This value is inclusive, that is, for min=0, the value 0 is allowed.

max Attribute

If the type of the option is number, this attribute sets an optional maximum value for the number. This value is inclusive, that is, for `max= 1`, the value 1 is allowed.

default Attribute

Allows specifying the default value of the property. This should match the setting to be found if you set a `getOption` for this property on a newly created component.

static Attribute

Specifies that the definition of this option is shared as a single instance across all component instances.

hidden Attribute

Specifies that the property is hidden in the component visual design dialog. Useful when the property is essential to implementation, but not necessary or desirable to reveal to the user.

read-only Attribute

Specifies that the property is visible as read-only in the component visual design dialog. Useful when it is preferred to display the property to the user but prohibit modification.



Note If the option is defined at the Package level, a single instance of the option definition is shared across all instances of all components within the package that are used in a single process. If the option is defined at the Component or Action levels, a single instance of the option metadata is shared across all instances of the defining component. If the static attribute is set to false or not provided, then each affected component instance will have a separate instance of this option definition.

Enum Element

This element appears as a child of the Option element. It provides one of the enumerated values for the option. For text and numeric options, the enum values are available as an aid to the graphical user interface.

value Attribute

The actual value used to set the option and the expected value when the current value is retrieved.

display Attribute

Provides a more user-friendly version of the value for display purposes. If this attribute is not present, the value attribute is used.

Component Element

The component element encapsulates all information needed by the Message Component Framework to classify, locate, load, instantiate and execute actions on a single component.

name Attribute

Provides the name of the component. This is used to display and select the component in various places.

version Attribute

Defines the version of the component. The version should be numeric, in the fixed format x.y or x.y.z. Any other version will cause version compatibility to be not supported.

class Attribute

Defines the class of the component, what kind of component it is. Currently, the only supported value is Message.

compatibleVersion Attribute

Describes which prior versions of the component that this version is compatible with. The value 1 implies that this version can be used in place of any other requested 1.x version. If the value is 1.2, this version, 1.3, could be used in place of a request for 1.2, but not 1.1.



Note The compatibleVersion attribute may be set to x.y or x, to imply that in all cases, the x.y values must match, or only the x value must match. If this attribute is missing, then all versions are assumed to be compatible.

Java Element

The presence of this element indicates the component is written in Java. Only one instance of this element may appear.



Note If the Java Element is used, no other language definition element is allowed.

mainclass Attribute

Defines the main class for the component. This class is expected to implement the Component interface and supported ComponentAction interfaces.

Classpath

For consistency in classpath management across all components, a classpath is automatically generated for all Java components based on the contents of the each deployment package. The order of the classpath follows:

- System classpath built by the DataConnect Engine, which contains the jar files distributed within the runtime/di9/jars directory
- <Plugin directory>/<Package name>/<Package Version>/
- <Package home>/classes
- <Component home>/*.jar
- <Component home>/lib
- <Component home>/lib/*.jar

Given this set of classpath formation rules, components should have ample opportunity to include necessary resources, such as custom configuration, dependent jar files, and more.

Model Element

This element provides information specific to a particular component. Within the broad component class set by the class attribute in the Component element, this element sets a subtype and describes the actions supported by this component. The iterator component is shown in this example.

type Attribute

Declares this component to be a given subtype of the class defined in the Component element. Whether this attribute is required and what values it can have, depend on the component class. For the Message class, the currently recognized subtypes are the following:

- Queue
- Iterator
- Aggregator
- Invoker
- Transformer

- Validator

Action Element

For each action supported by the component, there must be an Action element in the deployment descriptor. The information is used at design time to allow display of appropriate actions and their arguments, and at runtime. For each Model type, there is a predefined set of valid actions. Each action has a predefined set of arguments. For example, GetMessage(message, queue) takes a message instance and a queue name. In some cases, the queue name may not be needed, or it may be desirable to display the queue name to the user with a different name and prompt.

The Action element defines the arguments used by this component and allows setting the graphical user interface prompt and description information. If the Action element contains no Parameter elements, the default set of arguments, including usage, prompt and description, are assumed. If the Action element contains any Parameter element, the action is assumed to use all and only the parameters listed.

type Attribute

One of the predefined action types supported by the current model.

Parameter Element

Defines one of the parameters used by the containing Action.

type Attribute

One of the predefined (for this action) parameters.

usage Attribute

Defines whether the parameter is required. If the value is required, then a value for the parameter must be provided. If the usage attribute is optional, a predefined default is used. If the usage attribute is not used, this parameter is ignored. This value is provided for completeness.

XML Schema

Following is the XML schema for package.xml deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <!--
  Package is the new root element of the deployment descriptor.
  Its purpose is primarily to provide an outer wrapper for a
  set of component sets.  Secondly, it identifies the vendor,
  overall version of the package, and top-level description
  -->
  <xs:element name="Package">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Description" type="xs:string" minOccurs="0"/>
        <xs:element name="Option" type="OptionType" minOccurs="0"
maxOccurs="unbounded"/>
        <xs:element ref="Component" maxOccurs="unbounded" minOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name" use="required"/>
      <xs:attribute name="version" use="required"/>
      <xs:attribute name="schemaVersion" use="optional"/>
      <xs:attribute name="vendor" use="optional"/>
    </xs:complexType>
  </xs:element>
```

<!--

Component is the root element of the component deployment descriptor that was used in Cosmos versions 8.4 - 8.6. It is carried forward with little change for sake of backward compatibility.

-->

<xs:element name="Component">

<xs:complexType>

<xs:sequence>

<xs:element name="Description" type="xs:string" minOccurs="0"/>

<xs:choice>

<xs:element name="Java" type="JavaType"/>

<xs:element name="C" type="CType"/>

<xs:element name="Builtin" type="BuiltinType"/>

</xs:choice>

<xs:element name="Option" minOccurs="0" maxOccurs="unbounded" type="OptionType"/>

<xs:element name="Model" type="ModelType"/>

</xs:sequence>

<xs:attribute name="class" use="required" type="xs:NCName"/>

<xs:attribute name="compatibleVersion" use="optional" type="xs:NMTOKEN"/>

<xs:attribute name="name" use="required"/>

<xs:attribute name="version" use="required" type="xs:NMTOKEN"/>

</xs:complexType>

</xs:element>

<!--

The component is written in the Java programming language

-->

```
<xs:complexType name="JavaType">
```

```
  <xs:attribute name="mainclass" use="required" type="xs:NCName"/>
```

```
</xs:complexType>
```

<!--

The component is written in the C programming language

-->

```
<xs:complexType name="CType">
```

```
  <xs:attribute name="library" use="required"/>
```

<!--

The "platform" attributes are carried over from the 8.6 deployment descriptor for backward compatibility. They are used as library name modifiers for different platforms. For example, on windows if the attribute "library" equals "mycomp" and the "win" attribute is set to "win32", then the string used by the engine for finding the associated component library is "mycompwin32.dll". No known C component deployment descriptors currently use these attributes.

Of the elements and attributes carried forward from the MCF deployment descriptor for Cosmos 8.6, these are the most likely candidates for change or complete removal.

-->

```
<xs:attribute name="win" type="xs:string" use="optional"/>
```

```

<xs:attribute name="linux" type="xs:string" use="optional"/>
<xs:attribute name="hpux" type="xs:string" use="optional"/>
<xs:attribute name="aix" type="xs:string" use="optional"/>
<xs:attribute name="solaris" type="xs:string" use="optional"/>
</xs:complexType>

```

<!--

The component is built into the DataConnect engine

-->

```

<xs:complexType name="BuiltinType">
  <xs:attribute name="type" type="xs:nonNegativeInteger"/>
</xs:complexType>

```

<!--

The Model identifies the "type" of component, i.e. queue, transformer, iterator, etc. It also identifies the actions supported by this component.

-->

```

<xs:complexType name="ModelType">
  <xs:sequence>
    <xs:element name="Action" type="ActionType" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="type" use="required" type="ModelEnum"/>
</xs:complexType>

```

<!--

Identifies an individual action supported by the component


```
-->
<xs:complexType name="ActionType">
  <xs:sequence>
    <xs:element name="Parameter" maxOccurs="unbounded" minOccurs="0"
type="ParameterType"/>
    <xs:element name="Option" maxOccurs="unbounded" minOccurs="0"
type="OptionType"/>
  </xs:sequence>
  <xs:attribute name="type" use="required" type="ActionEnum"/>
</xs:complexType>
```

<!--

Identifies an individual parameter of an action

-->

```
<xs:complexType name="ParameterType">
  <xs:sequence>
    <xs:element name="Prompt" type="xs:string" minOccurs="0"/>
    <xs:element name="Description" type="xs:string" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="name" use="optional" type="xs:NCName"/>
  <xs:attribute name="type" use="required" type="ParameterEnum"/>
  <xs:attribute name="usage" use="optional" type="UsageEnum"/>
  <xs:attribute name="seq" use="optional" type="xs:NCName"/>
</xs:complexType>
```

<!--

Identifies a configuration option supported by the component. This value will typically be set at design time within the Process Editor

TODO:: Add support for conditional options. This will be necessary
for step option support.

-->

```
<xs:complexType name="OptionType">
  <xs:sequence>
    <xs:element name="Description" type="xs:string" minOccurs="0"/>
```

<!--

TODO: The validation for the Methods element should be handled in
a better way. Though Methods is not required (or even used)
by Java components, it is **required** by C components. This
level of document validation is more appropriate for
something like schematron rather than xml schema.

-->

```
<xs:element name="Methods" minOccurs="0">
  <xs:complexType>
    <xs:attribute name="get" use="required" type="xs:NCName"/>
    <xs:attribute name="set" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:element name="Enum" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <!-- here for future dependent option support -->
      <xs:element name="Option" type="OptionType"
        minOccurs="0" maxOccurs="unbounded"/>
```

```

    </xs:sequence>
    <xs:attribute name="value" use="required" type="xs:string"/>
    <xs:attribute name="display" use="required" type="xs:string"/>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="default" use="optional"/>
<xs:attribute name="fullname" use="required"/>
<xs:attribute name="name" use="required" type="xs:NCName"/>
<xs:attribute name="type" use="required" type="OptionTypeEnum"/>
<xs:attribute name="min" use="optional" type="xs:decimal"/>
<xs:attribute name="max" use="optional" type="xs:decimal"/>
<xs:attribute name="static" use="optional" type="StaticTypeEnum"/>
<!--
The direction attribute is here only for backward compatibility
with Cosmos 8.6. It will be removed in a subsequent release
of the product.
-->
<xs:attribute name="direction" use="optional" type="DirectionEnum"/>
</xs:complexType>

<!--
Enumeration of currently supported component types
-->
<xs:simpleType name="ModelEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="iterator"/>

```

```

<xs:enumeration value="aggregator"/>
<xs:enumeration value="queue"/>
<xs:enumeration value="invoker"/>
<xs:enumeration value="transformer"/>
<xs:enumeration value="router"/>
<xs:enumeration value="validator"/>
</xs:restriction>
</xs:simpleType>

```

```

<!--

```

Enumeration of currently supported actions

```

-->

```

```

<xs:simpleType name="ActionEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Connect"/>
    <xs:enumeration value="Disconnect"/>
    <xs:enumeration value="IsConnected"/>
    <!-- isConnected added for backward compatibility with existing
      deployment descriptors -->
    <xs:enumeration value="isConnected"/>
    <xs:enumeration value="BeginTransaction"/>
    <xs:enumeration value="PrepareTransaction"/>
    <xs:enumeration value="RollbackTransaction"/>
    <xs:enumeration value="CommitTransaction"/>
    <xs:enumeration value="GetMessage"/>
    <xs:enumeration value="PutMessage"/>
    <xs:enumeration value="Execute"/>
  </xs:restriction>
</xs:simpleType>

```

```

    <xs:enumeration value="Initialize"/>
    <xs:enumeration value="Reset"/>
    <xs:enumeration value="Destroy"/>
  </xs:restriction>
</xs:simpleType>

<!--
Enumeration of currently supported parameter types
-->
<xs:simpleType name="ParameterEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Message"/>
    <xs:enumeration value="Queue"/>
    <xs:enumeration value="Server"/>
    <xs:enumeration value="User"/>
    <xs:enumeration value="Password"/>
  </xs:restriction>
</xs:simpleType>

<!--
Enumeration of supported option types
-->
<xs:simpleType name="OptionTypeEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Text"/>
    <xs:enumeration value="File"/>
    <xs:enumeration value="Dir"/>

```

```

    <xs:enumeration value="Number"/>
    <xs:enumeration value="Enum"/>
    <xs:enumeration value="Boolean"/>
    <xs:enumeration value="Folder"/>
    <xs:enumeration value="Custom"/>
    <xs:enumeration value="Password"/>
  </xs:restriction>
</xs:simpleType>

<!--
Enumeration of acceptable values for the static attribute
-->
<xs:simpleType name="StaticTypeEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="true"/>
    <xs:enumeration value="yes"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>

<!--
Enumeration of parameter usage options
-->
<xs:simpleType name="UsageEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="required"/>

```

```

        <xs:enumeration value="optional"/>
        <xs:enumeration value="notused"/>
    </xs:restriction>
</xs:simpleType>

<!--
Enumeration of possible values for direction
-->
<xs:simpleType name="DirectionEnum">
    <xs:restriction base="xs:string">
        <xs:enumeration value="input"/>
        <xs:enumeration value="output"/>
        <xs:enumeration value="both"/>
    </xs:restriction>
</xs:simpleType>
</xs:schema>

```