Building Applications with FME Objects

SAFE SOFTWARE
E-mail: info@safe.com • Web: www.safe.com

Safe Software Inc. makes no warranty either expressed or implied, including, but not limited to, any implied warranties of merchantability or fitness for a particular purpose regarding these materials, and makes such materials available solely on an “as-is” basis.

In no event shall Safe Software Inc. be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of purchase or use of these materials. The sole and exclusive liability of Safe Software Inc., regardless of the form or action, shall not exceed the purchase price of the materials described herein.

This manual describes the functionality and use of the software at the time of publication. The software described herein and the descriptions themselves, are subject to change without notice.

Copyright

© 2000–2005 Safe Software Inc. All rights are reserved.

Revisions

Every effort has been made to ensure the accuracy of this document. Safe Software Inc. regrets any errors and omissions that may occur and would appreciate being informed of any errors found. Safe Software Inc. will correct any such errors and omissions in a subsequent version, as feasible. Please contact us at:

Safe Software Inc.
Fax: 604-501-9965
docs@safe.com
www.safe.com

Safe Software Inc. assumes no responsibility for any errors in this document or their consequences, and reserves the right to make improvements and changes to this document without notice.

Trademarks

FME is a registered trademark of Safe Software Inc. All brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Document Information

Document Name: Building Applications with FME Objects
Version: February 2005

Contents

Intended Audience vii
Conventions Used in this Manual vii
Reading the Code Samples vii

About This Manual vii

For More Information ix
How to Contact Us ix

Chapter 1 About FME Objects 1

What FME Objects Provides 1
Which Product Do I Need? 2
Overview of FME Objects 4
FMEOSession 4
FMEOFeature 4
FMEOReader 4
FMEOWriter 4
FMEODialog 4
FMEOPipeline 5
FMEOCoordSysManager 5
FMEOSpatialIndex 5
FMEOLogFile 5
Support for Open Standards 6

Chapter 2 Working with Sessions 7

Session Settings 9
Handling Errors 9
The Log File Object 10
Function Configuration 11
Destroying a Session 11

Chapter 3 Working with Features 13

Manipulating Feature Attributes 17
Interpreting Simple Geometries 20
Interpreting Arc and Ellipsoid Geometries 20
Interpreting Donut Geometries 23
Interpreting Aggregate Geometries 25
Working with OpenGIS Geometries 26
Using Schema Features 26

Chapter 4 Reading Features from a Dataset 31

Creating a Reader 32
Opening a Reader 34
Getting Reader Settings from the User 34
Getting Information about Available Readers 36
Reading Schema Features 38
Reading Data Features 39
Using Constraints 39
Closing a Reader 41

Chapter 5 Writing Features to a Dataset 43

Creating a Writer 44
Opening a Writer 44
Getting Writer Settings from the User 45
Getting Information About Available Writers 47
Writing Features 47
Closing a Writer 49

Chapter 6 Translating Features 51

Chapter 7 Working with Coordinate Systems . . . 55

Examining a Feature's Coordinate System 57
Tagging Features on Input 59
Reprojecting Features on Output 59
Tagging Individual Features 60
Creating Temporary Coordinate Systems 60
Creating Persistent Coordinate Systems 62

Chapter 8 Working with Spatial Indexes 63

Creating and Opening a Spatial Index 64
Indexing Features 66
Performing Spatial Queries 66
Closing a Spatial Index 67

Chapter 9 Advanced Feature Processing 69

Creating Area Topology 69
Dissolving Polygons 71
Buffering Features 72
Generating a Point Inside a Polygon 73
Performing an Arbitrary Function on a Feature . . . 74
Offsetting, Rotating, and Scaling Features 75
Manipulating Aggregate Features 75
Manipulating Donut Features 76
Applying a Factory Pipeline 77
  Creating and Defining a Pipeline 78
  Inserting Features into a Pipeline 79

Retrieving Features from a Pipeline 79
Putting it all Together 80

Chapter 10 Working with Collections 83

Using the Vector Collections 84
Using the String Array Collection 85

Chapter 11 Troubleshooting Tips 89

Problems Creating and Initializing a Session 89
Run-Time Exceptions 90

Contents

List of Diagrams

Read Feature Sequence Diagram 5
Write Feature Sequence Diagram 6
Methods and Properties of the FMEOSession Object 8
Methods of the FMEOLogFile Object 10
FME Feature Conceptual Data Model 13
Methods and Properties of the FMEOFeature Object 16
Sample Arc Geometry 22
Sample Donut Geometry 24
Methods and Properties of the FMEOReader Object 31
Methods and Properties of the FMEOWriter Object 43
FME Coordinate System Conceptual Data Model 55
Methods and Properties of the FMEOCoordSysManager Object 56
Methods and Properties of the FMEOSpatialIndex Object 64
Area Representation: (a) Spaghetti and (b) Topological Models 70
Dissolving Polygons: (a) Original and (b) Dissolved 72
Methods and Properties of the FMEOPipeline Object 77
Methods and Properties of the FMEORectangleVector Object 84
Methods and Properties of the FMEOFeatureVectorOnDisk Object 85
Methods and Properties of the FMEOStringArray Object 85

About This Manual

This manual helps you learn about the FME Objects API through both conceptual discussions of the problems FME Objects solves as well as practical examples of such solutions.

This guide illustrates the functionality of FME Objects in the context of Visual Basic. However, you are not limited to using Visual Basic. FME Objects can also be used from C, C++, C#, Java, and Delphi and transitioning the examples to one of these languages is not difficult. Example applications in Visual Basic, C, C++, C#, and Java are available for download at:

www.safe.com/products/fmeobjects/fmeobjects_sample_applications.htm

Intended Audience

This guide is intended for a technical audience: anyone wanting to embed powerful spatial data access and processing capabilities into applications using FME Objects. The advanced capabilities of FME Objects will be of particular interest to software professionals in the fields of geographic information systems (GIS) and spatial data management.

This guide assumes that the reader has working knowledge of Visual Basic.

This guide will provide the most benefit to developers that have completed either the FME Tutorial and Solutions Guide (contained on the FME CD), or the FME training course.

Conventions Used in this Manual

The following special fonts and conventions are used throughout this manual.

`Monospaced type`	This font is used to offset object names, file and directory names, and similar text.
Notes contain important points or tips. For example:	**Note** A reader cannot be reopened once it is closed. A new reader must be created.
Warnings highlight situations that could cause errors or conflicts. For example:	**WARNING** There can only be one session active within a process – attempting to create a second one will result in an error.

Reading the Code Samples

Sample code is provided throughout this document to illustrate concepts. The following global variables are used to simplify the code:

```
Public m_fmeSession As FMEOSession
Public m_fmeLogfile As FMEOLogFile
Public m_fmeFeatureVector As FMEOFeatureVector
Public m_fmeSchemaVector As FMEOFeatureVector
Public m_fmeMessages As FMEOStringArray
Public m_fmeReader As FMEOReader
Public m_fmeWriter As FMEOWriter
Public m_fmeSpatialIndex As FMEOSpatialIndex
```

The following constants are used to enhance the readability of sample code:

```
Private Enum GeometryType
      foUndefined = 0
      foPoint = 1
      foLine = 2
      foPolygon = 4
      foDonut = 8
      foPip = 256
      foAggregate = 512
End Enum

' FME geometry constants
Private Const kFME_Geometry = "fme_geometry"
Private Const kFME_Geom_Point = "fme_point"
Private Const kFME_Geom_Line = "fme_line"
Private Const kFME_Geom_Polygon = "fme_polygon"
Private Const kFME_Geom_Donut = "fme_donut"
Private Const kFME_Geom_Aggregate = "fme_aggregate"
Private Const kFME_Geom_Undefined = "fme_undefined"

' FME type constants
Private Const kFME_Type = "fme_type"
Private Const kFME_Type_Point = "fme_point"
Private Const kFME_Type_Arc = "fme_arc"
Private Const kFME_Type_Ellipse = "fme_ellipse"
Private Const kFME_Type_Text = "fme_text"
Private Const kFME_Type_Line = "fme_line"
Private Const kFME_Type_Area = "fme_area"
Private Const kFME_Type_No_Geom = "fme_no_geom"

'   Point, Arc, Ellipse, and Text attributes
Private Const kFME_Rotation = "fme_rotation"
Private Const kFME_Primary_Axis = "fme_primary_axis"
Private Const kFME_Secondary_Axis = "fme_secondary_axis"
Private Const kFME_Start_Angle = "fme_start_angle"
Private Const kFME_Sweep_Angle = "fme_sweep_angle"
Private Const kFME_End_Angle = "fme_end_angle"
```

For More Information

For more information about FME Objects, or for supporting information, refer to the following documents available from Safe Software's website:

- *FME Getting Started*
- *FME Functions, Factories and Transformers*
- *FME Readers and Writers*
- *FME Suite on-line help files*

How to Contact Us

Safe Software's website contains the latest news, tips, samples, and bug fixes; follow the link to the FME Objects product:

`www.safe.com`

Join Safe Software's **FME Objects Yahoo Group**:

`www.safe.com/support/online_communities.htm`

You can also e-mail Safe Software at:

`support@fmeobjects.com`

CHAPTER 1

About FME Objects

FME Objects provides a set of software components for working with spatial data. Software developers use FME Objects to build spatial data access and processing capabilities into new or existing applications, and deliver those applications to end users.

To create sophisticated spatial applications with FME Objects, you will first need to learn the underlying FME. FME comes with a comprehensive set of reference documentation to help you get up to speed. At a minimum, you should acquaint yourself with FME concepts, as described in the *FME Foundation* Manual, before undertaking an FME Objects development project.

What FME Objects Provides

FME Objects provides several key capabilities to your application, including the ability to:

- Read data from any FME-supported format.
- Write data to any FME-supported format.
- Spatially index data and perform complex spatial queries.
- Re-project data from one coordinate system to another.
- Form polygonal features from input linear features.
- Merge point features and containing polygonal features.
- Remove duplicate data.
- Generate interior points for polygons.
- Generalize linear and polygonal data.
- Create a buffer around spatial data.

This is only a sampling of what FME Objects can do for your application. Refer to the *FME Functions, Factories, and Transformers* manual for a complete list of the processing functions available.

Which Product Do I Need?

The choice between FME, FME Plug-in SDK, and FME Objects is not always obvious. Here are some guidelines to help you choose the right one for your needs.

> **WARNING** Choosing correctly between FME and FME Objects can make a significant difference in your development effort and schedule. If you are still not confident about which product is best for your needs, please contact Safe Software's technical support and provide a detailed description of your particular situation. We will do our best to help you get started on the right track.

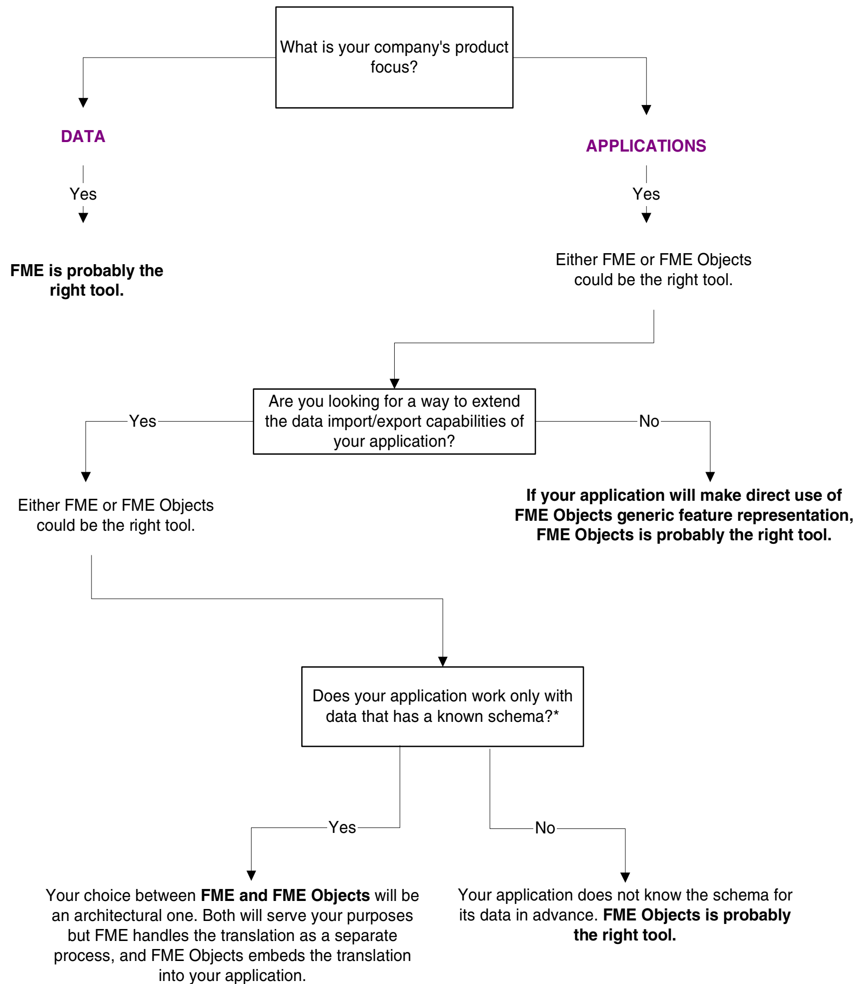
FME Plug-in SDK is required to develop a new reader plug-in, writer plug-in, or reader and writer plug-in for a format that is not currently supported by FME. The Plug-in SDK is also useful for creating custom FME factories and functions.

FME is used when you require a solution for reading, writing and transforming spatial data. If your focus is on data generation, data compilation, data translation, etc., then FME is probably the right product to use. When the schema of the data is known a priori, an application can extend it's data import and export capabilities by creating and controlling an external FME process.

FME Objects is used to build spatial data access and processing capabilities directly into new or existing applications. FME Objects is ideally suited for situations where the schema of the data is not known a priori.

The following road map will assist you in choosing between FME and FME Objects.

.



WARNING Choosing correctly between FME and FME Objects can make a significant difference in your development effort and schedule. If you are still not confident about which product is best for your needs, please contact Safe Software's technical support and provide a detailed description of your particular situation. We will do our best to help you get started on the right track.

Whether you choose the FME or FME Objects, if FME does not already support the format or formats you need to work with, the FME Plug-in SDK enables you

add support for a new format. The FME Plug-in SDK can also help you develop custom factories and functions.

Overview of FME Objects

This section describes the major objects that comprise FME Objects and provides illustrations of common interactions between them.

FMEOSession

The `FMEOSession` object is the root for an FME Objects session. All other objects are created by the `FMEOSession` object.

FMEOFeature

The `FMEOFeature` object provides a generic spatial data representation consisting of a set of attributes and an optional geometry with an associated coordinate system. The `FMEOFeature` object can also be used to process non-spatial data.

FMEOReader

The `FMEOReader` object presents a generic view of a source dataset. `FMEOReader` has a built-in cache to improve performance when a data source is accessed multiple times. Applications can read data from multiple source datasets in parallel by opening multiple `FMEOReader` objects. The `FMEOReader` object also provides access to schema information.

FMEOWriter

The `FMEOWriter` object presents a generic view of a destination dataset. Applications can write data to multiple datasets in parallel by opening multiple `FMEOWriter` objects. The `FMEOWriter` object allows applications to specify the schema of the data to be written.

FMEODialog

The FMEODialog object provides access to the standard FME dialog boxes. `FMEODialog` works closely with `FMEOReader` and `FMEOWriter` to encapsulate all the format specific information that readers and writers require during initialization.

FMEOPipeline

The `FMEOPipeline` object allows applications to access all the processing capabilities of the FME (for example, calculating intersections, forming polygons, creating buffers, etc.)

FMEOCoordSysManager

The `FMEOCoordSysManager` object provides a means for applications to retrieve and define coordinate system information: projections, datums, ellipsoids, and units.

FMEOSpatialIndex

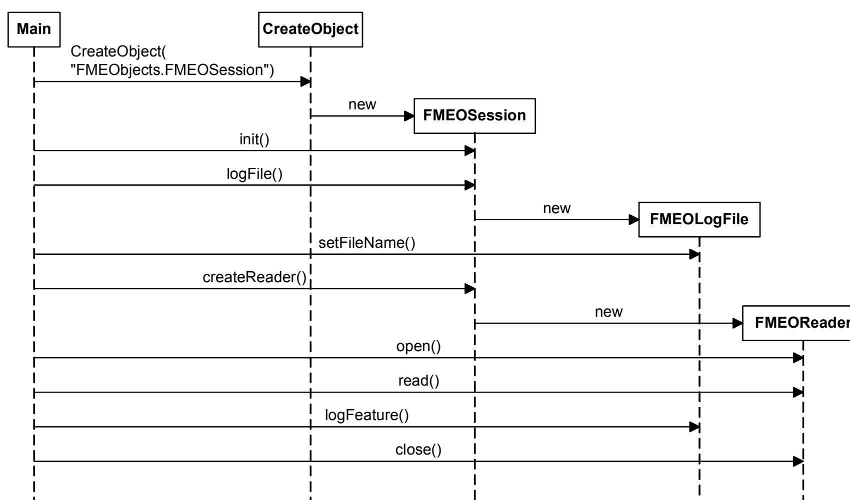
The `FMEOSpatialIndex` object lets applications store features on disk so that they may work on large datasets in an efficient manner. `FMEOSpatialIndex` allows applications to keep their memory footprint to a minimum.

FMEOLogFile

The `FMEOLogFile` object allows applications to log messages and features to a text file. This is useful for testing and debugging applications, both during development and after they've been deployed.

The following sequence diagram illustrates how the FMEOSession, FMEOLogFile, and FMEOReader objects collaborate to read a feature from a dataset and log it to a file.

Sequence Diagrams



Read Feature Sequence Diagram

The sequence diagram below illustrates how the FMEOSession, and FMEOWriter objects collaborate to write schema feature and a data feature to a dataset.

Main
CreateObject
CreateObject(
"FMEObjects.FMEOSession")
new
FMEOSession
init()
createWriter()
new
FMEOWriter
open()
addSchema()
write()
close()

Write Feature Sequence Diagram

Support for Open Standards

FME Objects provides support for the Open GIS Consortium's Well-Known Text (WKT) format. The `FMEOFeature` object can import and export geometry from WKT. The `FMEOCoordSysManager` object can define a new FME coordinate system from a WKT representation and it can also export an existing FME coordinate system as WKT.

Note The C++, C, and Java FME Objects APIs also support the Open GIS Well-Known Binary (WKB) format.

FME Objects allows applications to read and write XML-based formats.

CHAPTER

2

Working with Sessions

The starting point for using FME Objects is to create an FME Objects session object, *FMEOSession*. Unless your application is intended to operate in environments with limited virtual memory, the simplest approach is to create the `FMEOSession` object when your application is initializing and have it persist until the application shuts down. You are then left with the architectural choice of either making your FME Objects session globally accessible or passing it as a parameter to any object or method that requires it.

> **WARNING** There can only be one session active within a process – attempting to create a second one will result in an error.

In Visual Basic, the `CreateObject` method creates and returns a reference to an ActiveX object. FME Objects provides one ActiveX object that can be created using this method, namely `FMEOSession`. The following line of code shows how to assign the `FMEOSession` object returned by `CreateObject` method to the `m_fmeSession` global object variable.

```
Set m_fmeSession = CreateObject("FMEObjects.FMEOSession")
```

Once created, an `FMEOSession` object must be initialized before being used. The following code fragment demonstrates how to initialize an FME Objects session.

```
Dim fmeSessionSettings As FMEOStringArray
Set fmeSessionSettings = m_fmeSession.createStringArray
Call m_fmeSession.init(fmeSessionSettings)
```

The methods and properties of the `FMEOSession` object are shown in the figure below.

Methods and Properties of the FMEOSession Object

All other FME Objects ActiveX objects are created by the session object. For example, the following line creates a string array object and assigns it to an object variable.

```
Set fmeSessionSettings = fmeSession.createStringArray
```

The `fmeHome`, `fmeVersion`, and `fmeBuildNumber` properties can be used to obtain information about the FME installation being used by the application.

In the rest of this chapter, you'll learn about:

- initializing session settings
- changing session settings
- handling errors
- creating a log file
- configuring functions and factories

Session Settings

FME Objects accepts the same configuration settings as FME. See the *FME Configuration* chapter of the *FME Foundation* manual for details.

Session settings are specified using the `init` method. For example, to set the location of the temporary directory used by FME Objects, you would use the following code:

```
Dim fmeSessionSettings As FMEOStringArray
Set fmeSessionSettings = m_fmeSession.createStringArray
fmeSessionSettings.append("FME_TEMP")
fmeSessionSettings.append("d:\temp")
m_fmeSession.init(fmeSessionSettings)
```

The `updateSettings` method can be used to change the session settings after the session has been initialized.

Handling Errors

An important quality of any well-written application is its ability to handle run-time errors. In the best case, an application's error handling will anticipate errors and recover from them transparently to the user. However, it is difficult for any developer to anticipate every error that can possibly occur. When something unexpected occurs, a well-designed error handler should gracefully terminate the application and record information about the error to an error log or display it to the user. A basic FME Objects application error handler consists of the following lines:

```
Sub MyProcedure()

  On Error GoTo ERROR_HANDLER

  ‘ Your code goes here
  ‘...
  Exit Sub

ERROR_HANDLER:
  If Err.Number < 0 Then
     Err.Number = Err.Number Xor vbObjectError
  End If
  Err.Raise Err.Number, Err.Source, Err.Description

End Sub
```

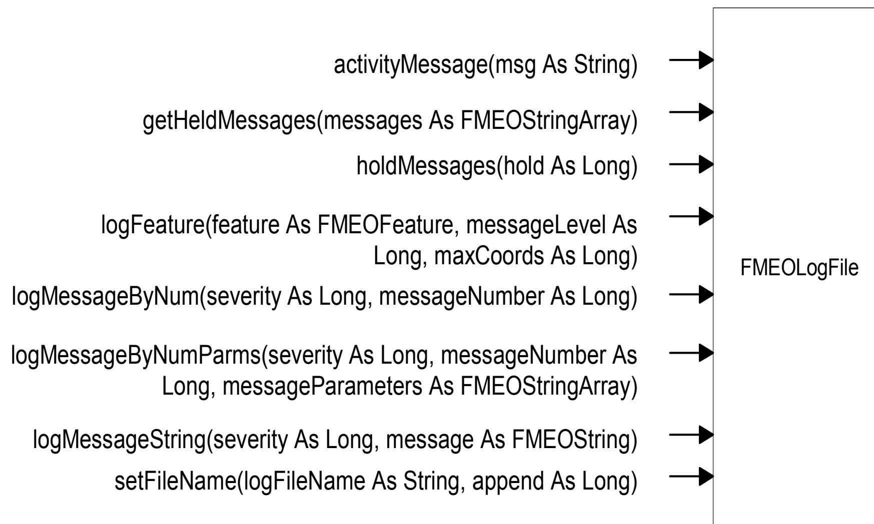
WARNING If an error occurs in FME Objects, the `FMEOSession` object is invalidated and cannot be used for the remainder of the application process’s lifetime. The application must create a new `FMEOSession` object if it wishes to continue using the services of FME Objects.

To assist you with further analysis of run-time errors, the `FMEOSession` object has two properties that contain valuable diagnostic information: `lastErrorNum`, and `lastErrorStackTrace`. If your application maintains its own log file, these properties contain valuable information that can be recorded to facilitate troubleshooting.

Note For the sake of brevity, error handlers have been omitted from sample procedures provided in this document. Nonetheless, we strongly encourage you to use them in your FME Objects code.

The Log File Object

The `logFile` property of the `FMEOSession` object allows your application to access the `FMEOLogFile` object for the session. Using the `FMEOLogFile` object, a log file can be named, thereby instructing FME Objects to log its configuration, progress, performance, statistics, warnings, and error messages to this file as it executes.



Methods of the FMEOLogFile Object

The following example names the log file using the application path and title and logs a test message to it.

```
Public m_fmeLogfile As FMEOLogFile
Set m_fmeLogfile = m_fmeSession.logFile
sLogFileName = App.Path & "\" & App.Title & ".log"
Call m_fmeLogfile.setFileName(sLogFileName, False)
Call m_fmeLogfile.logMessageString(0, "FME Objects test.")
```

The FMEOLogFile object can be instructed to capture a copy of each message written out to the log file. This is useful if your application needs to review messages previously generated. Calling the holdMessages method with a value of True causes FMEOLogFile to begin copying messages into its internal buffer. Calling the holdMessages method with a value of False stops the buffering. For example, the following code fragment would result in the fmeMessages string array being loaded with two entries: “First Message” and “Second Message”.

```
Set m_fmeMessages = m_fmeSession.createStringArray
Call m_fmeLogfile.HoldMessages(True)
Call m_fmeLogfile.logMessageString(0, "First Message")
Call m_fmeLogfile.logMessageString(0, "Second Message")
Call m_fmeLogfile.HoldMessages(False)
Call m_fmeLogfile.logMessageString(0, "Third Message")
Call m_fmeLogfile.getHeldMessages(m_fmeMessages)
```

Calls to the holdMessages method have no effect on the contents of the log file itself; all messages are still logged as usual. In the previous example, all three messages would appear in the log file.

Function Configuration

FME functions are exposed toFME Objects applications through the pipeline object, FMEOPipeline, and the feature object, FMEOFeature. Some functions require configuration lines as described in the *FME Functions, Factories, and Transformers* manual. The FMEOSession configure method allows you to specify one function configuration row at a time. The following code fragment illustrates how to specify configuration information for the @SQL function.

```
Dim fmConfigRow as FMEOStringArray
Set fmeConfigRow = m_fmeSession.createStringArray
fmeConfigRow.append("SQL")
fmeConfigRow.append("sampleConn")
fmeConfigRow.append("SERVER_TYPE")
fmeConfigRow.append("ORACLE")
fmeConfigRow.append("SERVER_NAME")
fmeConfigRow.append("sea_world")
m_fmeSession.configure(fmeConfigRow)
```

Destroying a Session

Once you have finished working with the FME Objects session object, you can optionally ask Visual Basic to destroy the object reference as follows:

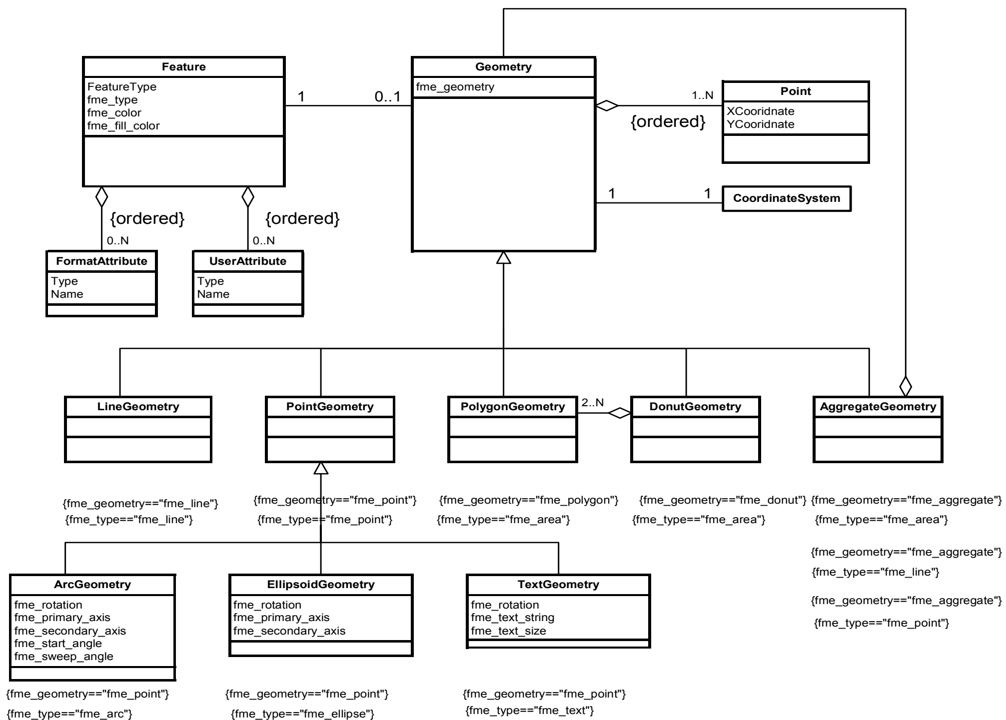
```
Set m_fmeSession = Nothing
```

This has the direct effect of releasing all the resources associated with the session.

CHAPTER 3

Working with Features

The feature object, FMEOFeature, is the fundamental data unit of an FME Objects application. At a coarse level, a feature consists of a set of attributes and a geometry with an associated coordinate system. When an application reads from a dataset, it receives the data one feature at a time. When an application writes to a dataset, it sends the data one feature at a time. Features can be manipulated using all of the standard FME functions and factories. The conceptual data model for features is shown below.



FME Feature Conceptual Data Model

This is a conceptual model of the data encapsulated by the `FMEOFeature` object. In other words, FME Objects does not provide a one-to-one mapping between these classes and the objects exposed by the API: access to all of this data is provided through one object, FMEOFeature.

The paragraphs that follow describe the conceptual classes, their attributes, and the relationships between them. The notation `Class::Attribute` is used to refer to an attribute of a class.

Every format supported by FME identifies features by feature type. The feature type is a well-defined data classification scheme. For example, a simple classification scheme is to identify features according to the file name or database table where they reside, in which case the feature's type is the file base name or table name, respectively. For level-based formats, features are typically identified by level: the feature's type is the level in which it resides. Another common classification is to identify features according to the thematic layer to which they belong (for example, roads, railways, rivers, etc.). Although these are some of the more common classification schemes, the list of possible schemes is quite broad because the classification is chosen to best suit the structure of the data. This primary classification is known as the feature's type and is captured in the `Feature::FeatureType` attribute.

Feature attributes are categorized into one of the following three groups:

- format-specific attributes
- user attributes
- generic FME attributes

A Feature is associated with zero or more `FormatAttributes`. A `FormatAttribute` represents an attribute that is specific to a format. Some examples are: `autocad_block_name` and `sde30_justification`.

A Feature is also associated with zero or more `UserAttributes`, which represent custom attributes that hold domain information about a feature such as: `parcel_identifier`, `owner_name`, `date_surveyed`, etc.

Generic FME attribute names are prefixed by `fme_`. The benefit of using generic attributes over format-specific attributes is that they have the same meaning in all the readers and writers that support them. For example, the `fme_color` and `fme_fill_color` feature attributes represent the red, green, and blue intensities of a feature (see the *FME Architecture* chapter of the *FME Foundation* manual for details). If a format has generic color support, the reader will add both the generic FME color attributes and the format-specific color attributes to features. Writers that support generic color will give precedence to the format-specific color attributes if they are present in addition to the generic attributes.

The most important generic attributes are `Feature::fme_type` and `Geometry::fme_geometry`. Both of these relate to the geometry of a feature. The Geometry class represents a feature's positional information. Feature geometry may consist of points, lines, or areas. Features that contain multiple geometric parts are said to have an aggregate geometry. Features with no geometry are also supported.

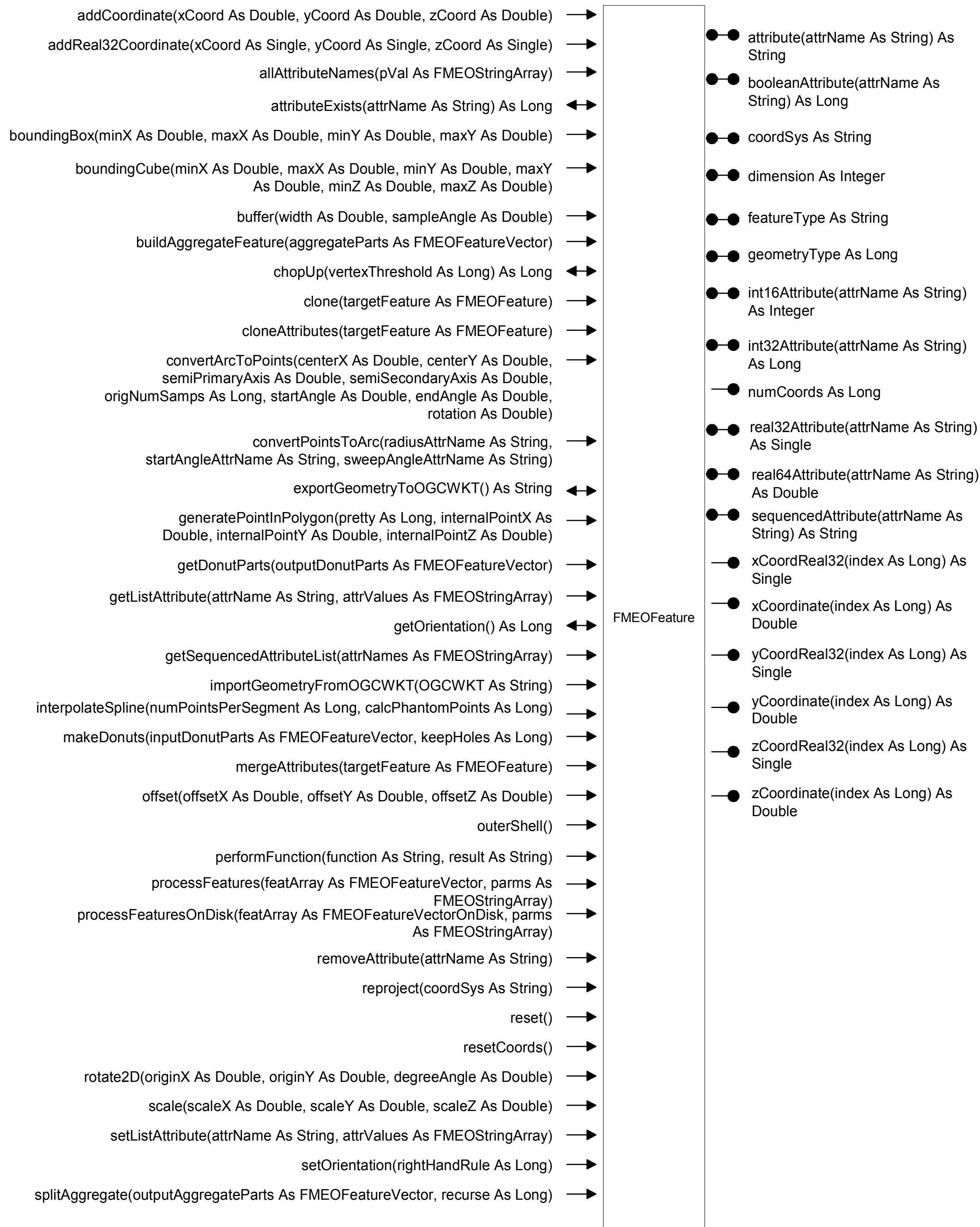
The distinction between `fme_type` and `fme_geometry` is an important one: `fme_geometry` indicates the geometry type of the actual coordinates, whereas `fme_type` specifies how that geometry is to be interpreted. For example, a point geometry type can be interpreted as one of the following FME types: point, arc, ellipse, or text. The one-to-one relationship between Feature and Geometry is restricted by the legal combinations of `fme_type` and `fme_geometry`. The valid combinations of `fme_type` and `fme_geometry` are shown in the table below:

fme_type \ **fme_geometry**	fme_point	fme_line	fme_polygon	fme_donut	fme_aggregate	fme_no_geom
fme_point	X				X	
fme_arc	X					
fme_ellipse	X					
fme_text	X					
fme_line		X			X	
fme_area			X	X	X	
fme_undefined						X

If you are building an application that needs to be format-neutral, then you must use only generic FME attributes to process features. If you avoid using format-specific attributes, it will be much simpler to adapt your application to work with new formats as they are introduced. If your application only needs to work with a set of predefined formats that is unlikely to grow, then you can use format attributes with the knowledge that adding support for new formats will be more involved.

Each feature geometry is associated with only one coordinate system. For a detailed discussion of the coordinate system-related operations that can be performed on a feature, refer to *Working with Coordinate Systems* on page 55.

The methods and properties of `FMEOFeature` follow and are discussed in more detail in the remainder of this chapter.



Methods and Properties of the FMEOFeature Object

In the rest of this chapter, you will learn how to:

- manipulate feature attributes
- interpret feature geometries
- use the schema for a feature

Issues regarding the relationship between features and coordinate systems are covered in Chapter 7, *Working with Coordinate Systems*.

Chapter 9, *Advanced Feature Processing*, covers the use of feature processing methods such as `performFunction` and `processFeatures` and discusses how to apply factory pipelines to features.

Note All code samples deal only with two-dimensional geometry for simplicity. If your application handles three-dimensional geometry, it is straightforward to extend the sample code.

Manipulating Feature Attributes

Format attributes, user attributes, and generic FME attributes are all manipulated using the set of properties and methods summarized by the table below:

Property or Method	Description
`allAttributeNames`	This method returns a list of all the format, user, and generic attributes associated with a feature.
`attribute`	This property is used to get or set an attribute as a string value.
`attributeExists`	This method is used to test if an attribute exists; returns `FME_TRUE` if the attribute exists, `FME_FALSE` otherwise.
`booleanAttribute`	This property is used to get or set an attribute as a boolean value.
`int16Attribute`	This property is used to get or set an attribute as a 2-byte integer value.
`int32Attribute`	This property is used to get or set an attribute as a 4-byte integer value.
`real32Attribute`	This property is used to get or set an attribute as a 4-byte floating point value.
`real64ttribute`	This property is used to get or set an 8-byte floating point valued attribute.
`cloneAttributes`	This method is used to copy the feature type and all the attributes from the source feature to a target feature. The original feature type and attributes on the target feature are lost. **Note:** This method does not copy the source feature's geometry to the target feature. If you also want to copy geometry, use the `clone` method.
`clone`	This method is used to copy the feature type, all the attributes, and the geometry from the source feature to a target feature. The original feature type and attributes on the target feature are lost.

The `GetFeatureAttributes` procedure below uses the `getAllAttributeNames` method of `FMEOFeature` to iterate through all the attributes belonging to the input feature. The results are placed in two collections: one for the attribute names and one for the attribute values.

```
Sub GetFeatureAttributes(ByVal fmeFeature As FMEOFeature, _
                         ByVal colNameList As Collection, _
                         ByVal colValueList As Collection)
    Dim fmeAttributeNames As FMEOStringArray
    Dim sName As String
    Dim sValue As String
    Dim lEntries As Integer
    Dim i As Integer
    Dim d As Double
    Dim b As Boolean

    Set fmeAttributeNames = m_fmeSession.createStringArray
    Call fmeFeature.allAttributeNames(fmeAttributeNames)

    lEntries = fmeAttributeNames.entries
    For i = 0 To lEntries - 1
        sName = fmeAttributeNames.element(i)
        sValue = fmeFeature.attribute(sName)
        Call colNameList.Add(sName)
        Call colValueList.Add(sValue)
    Next i
End Sub
```

In addition to retrieving the value of an attribute, the `attribute` property of `FMEOFeature` can also be used to set an attribute to a string value. If the attribute being set already exists, its value is overwritten; otherwise a new attribute is created. The following code fragment checks if an attribute called `my_attribute` exists. If it does exist, its value is replaced by `"changed_value"`. If it does not exist, it is created and its value is set to `"new_value"`:

```
If fmeFeature.attributeExists("my_attribute") Then
    fmeFeature.attribute("my_attribute") = "changed_value"
Else
    fmeFeature.attribute("my_attribute") = "new_value"
EndIf
```

The value of a feature attribute can be accessed using any of the attribute properties of the FMEOFeature object because FME objects will automatically perform a type coercion when requried. Automatic type coercion must be used carefully to avoid undefined results. The table below illustrates the type

coercions that produce valid results. An entry in the table with no value indicates that the result of that coercion is undefined.

property \ attribute value	"dog"	False	32767	2147483647	9.183	9.183E-41	1.797E+307
attribute	"dog"	"No"	"32767"	"2147483647"	"9.183"	"9.183E-41"	"1.797E+307"
booleanAttribute		False					
int16Attribute		0	32767		9		
int32Attribute		0	32767	2147483647	9		
real32Attribute		0	32767		9.183	9.183E-41	
real64Attribute		0	32767	2147483647	9.183	9.183E-41	1.797E+307

The `cloneAttributes` method of `FMEOFeature` can be used to copy the attribution from a source feature to a target feature. The following code fragment shows how to do this:

```
Dim fmeTargetFeature as FMEOFeature
Set fmeTargetFeature = m_fmeSession.createFeature
fmeSourceFeature.cloneAttributes(fmeTargetFeature)
```

WARNING The value of any attribute on the target feature that has the same name as an attribute on the source feature will be overwritten.

Interpreting Simple Geometries

Simple geometries are geometries that are not composed of other geometries, and do not have associated generic FME attributes that affect their positional representation. More concretely, features with simple geometries are those with the following combinations of `fme_geometry` and `fme_type`: `{fme_line, fme_line}, {fme_point, fme_point}, {fme_polygon, fme_area}.`

The `GetSimpleCoords` procedure below shows how to extract the coordinates from a feature with simple geometry and adds the resulting list of points to the `colDisplayList` collection.

```
Private Sub GetSimpleCoords(ByVal fmeFeature As FMEOFeature,
                            ByVal colDisplayList As Collection)
    Dim objPoint As clsPointObject
    Dim lCoordCount As Integer
    Dim i As Integer
    Dim colPointList As New Collection

    lCoordCount = fmeFeature.numCoords
    For i = 0 To lCoordCount - 1
        Set objPoint = New clsPointObject
        objPoint.X = fmeFeature.xCoordinate(i)
        objPoint.Y = fmeFeature.yCoordinate(i)
        Call colPointList.Add(objPoint)
    Next i
    Call colDisplayList.Add(colPointList)
End Sub
```

The `GetSimpleCoords` procedure uses a simple point class, `clsPoint`, which is defined as follows.

```
Dim dXCoord As Double
Dim dYCoord As Double
Public Property Get X() As Double
    X = dXCoord
End Property
Public Property Let X(ByVal dX As Double)
    dXCoord = dX
End Property
Public Property Get Y() As Double
    Y = dYCoord
End Property
Public Property Let Y(ByVal dY As Double)
    dYCoord = dY
End Property
```

Interpreting Arc and Ellipsoid Geometries

The `EllipsoidGeometry` class (see the diagram *FME Feature Conceptual Data Model* on page 13) describes a circle or an ellipse. Features with ellipsoid

geometry are those with fme_geometry equal to fme_point and fme_type equal to fme_ellipse.

The ArcGeometry class describes a circular or elliptical arc. Features with arc geometry are those with an fme_geometry value equal to fme_point and an fme_type value equal to fme_arc. The ArcGeometry class contains the following attributes: fme_rotation, fme_primary_axis, fme_secondary_axis, fme_start_angle, and fme_sweep_angle, which are described as follows:

Attribute	Description
fme_rotation	The rotation angle of the primary axis for the ellipse that defines the arc. The rotation angle is specified in degrees from the horizontal axis in a counter-clockwise direction.
fme_primary_axis	Length of the primary axis for the ellipse that defines the arc.
fme_secondary_axis	Length of the secondary axis for the ellipse that defines the arc.
fme_start_angle	The start angle measured counterclockwise from the primary axis.
fme_sweep_angle	The angle on the ellipse that define the arc, measured counterclockwise from the start angle.

The figure below illustrates a sample arc geometry for the following values

Let a = primary readius; b = secondary radius; tS = start angle; tE = end angle tE-tS = sweep angle; thetaS = start angle in degrees = 45; thetaE = end angle in degrees = 180; thetaE - thetaS = sweep angle in degrees = 135

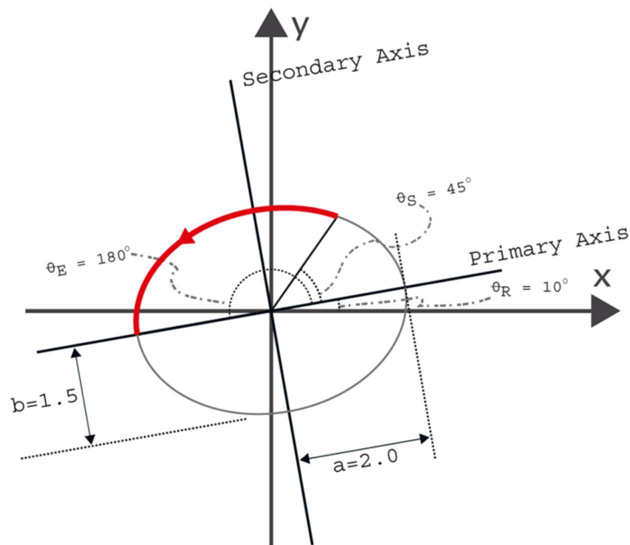
fme_primary_axis = a = 2.0; fme_secondary_axis = b = 1.5; fme_rotation = 10

fme_start_angle = tS = arctan((a/b)*tan(thetaS))

fme_sweep_anle = tE - tS = arctan((a/b)*tan(thetaE))

arc tan stands for the inverse tangent of an angle.

Attribute	Value
fme_rotation	10
fme_primary_axis	2.0
fme_secondary_axis	1.5
fme_start_angle	53.13
fme_sweep_angle	126.87



Sample Arc Geometry

An ellipse geometry is a special case of an arc geometry, where the sweep angle is 360 degrees.

The `GetArcCoords` procedure uses `FMEOFeature's` `convertArcToPoints` method to get a vectorized representation of a feature with arc or ellipse type. The resulting list of points is added to the `colDisplayList` collection.

```
Private Sub GetArcCoords(ByVal fmeFeature As FMEOFeature, _
                         ByVal sFmeType As String, _
                         ByVal colDisplayList As Collection)
    Dim dCenterX As Double
    Dim dCenterY As Double
    Dim dRotation As Double
    Dim dPrimaryAxis As Double
    Dim dSecondaryAxis As Double
    Dim dStartAngle As Double
    Dim dSweepAngle As Double
    Dim dEndAngle As Double
    Dim fmeTempFeature As FMEOFeature

    Set fmeTempFeature = m_fmeSession.createFeature()
    Call fmeFeature.Clone(fmeTempFeature)
    dCenterX = fmeTempFeature.xCoordinate(0)
    dCenterY = fmeTempFeature.yCoordinate(0)
    dPrimaryAxis = fmeTempFeature.attribute( _
                                         kFME_Primary_Axis)
    dSecondaryAxis = fmeTempFeature.attribute( _
```

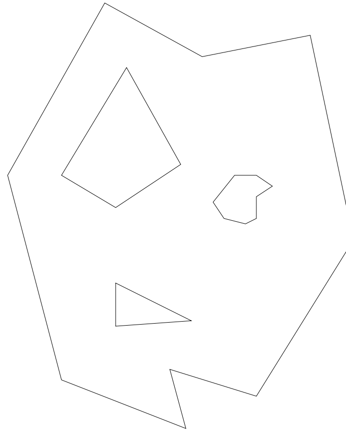
```
                                            kFME_Secondary_Axis)
    dRotation = fmeTempFeature.attribute(kFME_Rotation)

    If sFmeType = kFME_Type_Arc Then
        dStartAngle = fmeTempFeature.attribute( _
                                             kFME_Start_Angle)
        dSweepAngle = fmeTempFeature.attribute( _
                                             kFME_Sweep_Angle)
        dEndAngle = dStartAngle + dSweepAngle
    ElseIf sFmeType = kFME_Type_Ellipse Then
        dStartAngle = 0
        dEndAngle = 360
    Else
        Debug.Print "Invalid fme_type: " & sFmeType
        Exit Sub
    End If
    Call fmeTempFeature.convertArcToPoints(dCenterX, _
                dCenterY, dPrimaryAxis, dSecondaryAxis, _
                0, dStartAngle, dEndAngle, dRotation)
    Call GetSimpleCoords(fmeTempFeature, colDisplayList)
End Sub
```

`FMEOFeature's` `convertArcToPoints` method, which is used to convert the arc or ellipse into a series of line segments, has the effect of modifying the geometry of the feature upon which it operates. To ensure that the passed-in feature remains unchanged, a copy of the original feature is created using the `FMEOFeature's` `clone` method and this copy is modified instead.

The `GetSimpleCoords` procedure, introduced in the previous section, is used to copy the feature's points into the display list collection.

Interpreting Donut Geometries

The DonutGeometry class is used to represent area features with holes, such a lakes with islands. A proper `DonutGeometry` class (see the diagram *FME Feature Conceptual Data Model* on page 13) contains only polygons that do not overlap each other or share common edges; all of the inner polygons are disjoint and fully contained within the outer polygon. However, FME Objects does not guarantee that all features with donut geometry follow these rules. For example, when data is read from a data source that supports donut geometries but do not enforce non-intersecting and non-overlapping donuts, FME Objects will respect the original geometry. The figure below shows an example donut geometry.



Sample Donut Geometry

Features with donut geometry are those with an `fme_geometry` equal to `fme_donut` and an `fme_type` value equal to `fme_area` . As illustrated in the Feature Conceptual Data Model diagram, the `DonutGeometry` class is composed of two or more `PolygonGeometry` classes; one outer polygon and one or more inner polygons.

The `GetDonutCoords` procedure extracts the coordinates from a feature with donut geometry and adds the resulting list of points to the `colDisplayList` collection. The `getDonutParts` method of `FMEOFeature` is used to get the constituent polygons. The `GetSimpleCoords` procedure is then used to extract the coordinates of the constituent polygons and copy them into the display list collection.

```
Private Sub GetDonutCoords(ByVal fmeFeature As FMEOFeature, _
                           ByVal colDisplayList As Collection)
    Dim fmeFeatureVector As FMEOFeatureVector
    Dim fmePolygonFeature As FMEOFeature
    Dim lEntries As Integer
    Dim i As Integer

    Set fmeFeatureVector = m_fmeSession.createFeatureVector
    Call fmeFeature.getDonutParts(fmeFeatureVector)
    lEntries = fmeFeatureVector.entries
    For i = 0 To lEntries - 1
        Set fmePolygonFeature = fmeFeatureVector.element(i)
        Call GetSimpleCoords(fmePolygonFeature, _
                                       colDisplayList)
    Next i
End Sub
```

Note The getDonutParts method of FMEOFeature does not clear the input feature vector, it only appends to it.

The GetSimpleCoords procedure is used to copy the feature's points into the display list collection.

Interpreting Aggregate Geometries

A feature with aggregate geometry has an fme_geometry value equal to fme_aggregate and an fme_type value equal to either fme_point, fme_line, or fme_area. FME Objects uses aggregates to represent features with multi-part geometries: geometries that are composed of several disjoint pieces. In most situations, the components of an aggregate are homogeneous. That is, if fme_type is fme_point, then the aggregate contains point geometries; if fme_type is fme_line then the aggregate contains line geometries; and so on. However, your application should be designed to handle non-homogeneous aggregates gracefully since it is possible that some data sources may contain such features. Non-homogeneous aggregates have no value for fme_type.

The GetAggregateCoords procedure extracts the coordinates from a feature with aggregate geometry and adds the resulting lists of points to the colDisplayList collection. The splitAggregate method of FMEOFeature is used to split the aggregate into its constituent parts. The second parameter to the splitAggregate method is a Boolean value that instructs the feature how to deal with aggregates of aggregates. If this parameter set to True, splitAggregate recusively splits nested aggregates; if it is set to False, only the first level of aggregation is split.

```
Private Sub GetAggregateCoords( _
                        ByVal fmeFeature As FMEOFeature, _
                        ByVal colDisplayList As Collection)
    Dim fmeFeatureVector As FMEOFeatureVector
    Dim fmeComponentFeature As FMEOFeature
    Dim lEntries As Integer
    Dim i As Integer

    Set fmeFeatureVector = m_fmeSession.createFeatureVector
    Call fmeFeature.splitAggregate(fmeFeatureVector, True)
    lEntries = fmeFeatureVector.entries
    For i = 0 To lEntries - 1
        Set fmeComponentFeature = fmeFeatureVector.element(i)
        Call GetFeatureCoords(fmeComponentFeature, _
                                              colDisplayList)
    Next i
End Sub
```

The GetFeatureCoords procedure is used to extract the coordinates of each constituent geometry.

```
Private Sub GetFeatureCoords( _
                            ByVal fmeFeature As FMEOFeature, _
                           ByVal colDisplayList As Collection)
    Dim sFmeType As String
    Dim sFmeGeometry As String
    Dim lFmeGeometry As Long
    Dim colPointList As Collection

    lFmeGeometry = fmeFeature.GeometryType
    sFmeGeometry = fmeFeature.attribute(kFME_Geometry)
    sFmeType = fmeFeature.attribute(kFME_Type)
    Select Case lFmeGeometry
         Case Is = foPoint
            Select Case sFmeType
                Case Is = kFME_Type_Point, kFME_Type_Text
                    Call GetSimpleCoords(fmeFeature, _
                                               colDisplayList)
                Case Is = kFME_Type_Arc, kFME_Type_Ellipse
                    Call GetArcCoords(fmeFeature, sFmeType, _
                                               colDisplayList)
            End Select
         Case Is = foLine, foPolygon
            Call GetSimpleCoords(fmeFeature, colDisplayList)
         Case Is = foDonut
            Call GetDonutCoords(fmeFeature, colDisplayList)
         Case Is = foAggregate
           Call GetAggregateCoords(fmeFeature, colDisplayList)
    End Select
End Sub
```

Note The `GetSimpleCoords`, `GetArcCoords`, and `GetDonutCoords` procedures have all been previously introduced in this chapter.

Working with OpenGIS Geometries

The OpenGIS Consortium defines a Well-Known Text (WKT) representation for feature geometries. The `FMEOFeature` object allows your application to import a geometry from WKT and export a geometry to WKT using the `importGeometryFromOGCWKT` and `exportGeometryFromOGCWKT` methods respectively.

Using Schema Features

Schema features are useful when you need to understand the structure of input data or when you want to define a new structure or change the existing structure of data being written out.

FME Objects data features are identified according to a well-defined classification scheme such as: Theme, Layer, Level, Table, Class, File, etc. A schema feature provides the data model for the features in one category of this

classification. For a given category, the schema feature data model describes attribute names and types and allowable geometries for all the data features belonging to that category.

> **Note** A feature's type should not be confused with its `fme_type`. In the Feature Conceptual Data Model Diagram, these are represented by `Feature::FeatureType` and `Feature::fme_type` respectively.

Chapter 4, *Reading Features from a Dataset*, describes how to access schema features using the readSchema method of the FMEOFeature object. The schema feature corresponding to a data feature can be identified using the getFeatureType method of the FMEOFeature object. Data feature attributes have two explicit properties: name and value. A data feature attribute's type is implicitly provided by its corresponding schema feature.

For example, the following schema feature:

Feature Type: `Road`

Attribute Name	Value
name	fme_char(64)
numberOfLanes	fme_int32
averageSlope	fme_real64
fme_geometry	{fme_line}

describes the following data feature:

Feature Type: `Road`

Attribute Name	Value
fme_type	fme_line
fme_geometry	fme_line
name	Scott Road
numberOfLanes	4
averageSlope	0.03

The `GetSchemaInfo` procedure shows how to get the type for each of the user attributes on a data feature, given its schema feature:

```
Public Sub GetSchemaInfo( _
                     ByRef fmeDataFeature As FMEOFeature, _
                     ByRef fmeSchemaFeature As FMEOFeature)
    Dim i As Integer
    Dim lCount As Integer
```

```
        Dim sName As String
        Dim sType As String
        Dim sMsg As String
        Dim fmeAttributeNames As FMEOStringArray

        Set fmeAttributeNames = m_fmeSession.createStringArray
        Call fmeDataFeature.allAttributeNames(fmeAttributeNames)
        lCount = fmeAttributeNames.entries
        For i = 0 To lCount - 1
            sName = fmeAttributeNames.element(i)
            sType = fmeSchemaFeature.attribute(sName)
            If sType <> "" Then
                sMsg = sMsg & sName & ": "
                sMsg = sMsg & sType & vbCrLf
            End If
        Next
        MsgBox sMsg, vbOKOnly, "GetSchemaInfo"
        Exit Sub
    End Sub
```

The following table lists the valid types for user attributes:

Attribute Type	Description
`fme_char(width)`	This type indicates that the attribute is a string. The `width` parameter is the maximum number of characters in the string.
`fme_date`	This type indicates that the attribute is a date. Date attributes are represented as formatted strings.
`fme_decimal(width, decimal)`	This type indicates that the attribute is a decimal number. The `width` parameter is the maximum number of digits used to represent the number, including the decimal point. The `decimal` parameter controls the precision by specifying the number of digits to the right of the decimal point.
`fme_real32`	This type indicates that the attribute is a single precision floating point number.
`fme_real64`	This type indicates that the attribute is a double precision floating point number.
`fme_int16`	This type indicates that the attribute is a 16-bit signed integer.
`fme_int32`	This type indicates that the attribute is a 32-bit signed integer.
`fme_boolean`	This type indicates that the attribute is a boolean value.

If your application creates a new user attribute on a feature with the intent of writing the feature to a destination dataset, it must add an attribute to the corresponding schema feature (that is, the one with the same feature type) to indicate the attribute’s type. The attribute added to the schema feature must

have the same name as the attribute added to the data feature, and its value must be one of the attribute types listed in the table above.

Note Schema features do not contain type information for generic FME attributes (for example, `fme_rotation`).

Schema feature attributes with names that begin with an asterisk (`*`) are format-specific and should not be required by your application.

In addition to describing attribute types, schema features contain a list of the possible geometries for the feature type they describe. The `GetGeomInfo` procedure below illustrates how to retrieve list of allowable geometry types for a schema feature:

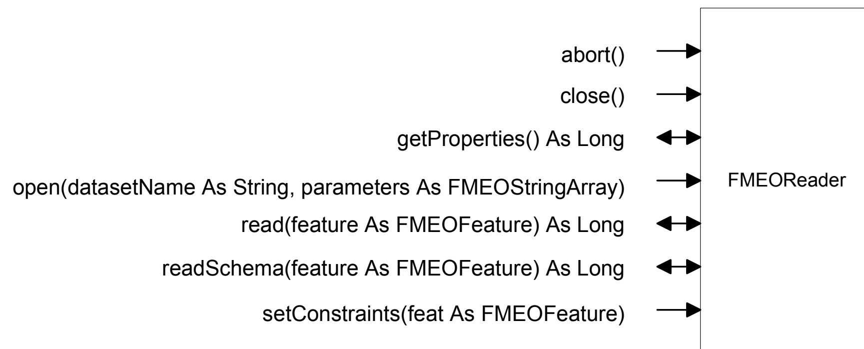
```
Public Sub GetGeomInfo(ByRef fmeSchemaFeature As FMEOFeature)
    Dim i As Integer
    Dim sName As String
    Dim sGeom As String
    Dim sMsg As String
    Dim bFinished As Boolean

    i = 0
    bFinished = False
    Do While Not bFinished
        sName = "fme_geometry{" & i & "}"
        sGeom = fmeSchemaFeature.attribute(sName)
        If sGeom <> "" Then
            sMsg = sMsg & sName & ": "
            sMsg = sMsg & sGeom & vbCrLf
        Else
            bFinished = True
        End If
        i = i + 1
    Loop
    MsgBox sMsg, vbOKOnly, "GetGeomInfo"
End Sub
```

CHAPTER

4

Reading Features from a Dataset

The FME universal reader object, *FMEOReader*, provides access to data stored in any of the supported formats. The methods and properties of FMEOReader provide a simple and configurable interface.



Methods and Properties of the FMEOReader Object

The FMEOReader object returns two types of features: schema features and data features. Schema features describe the model of a dataset. Every format supported by FME Objects identifies data features according to a well-defined data classification scheme. This primary classification is known as the feature's type and it serves as the main handle to a data feature. A schema feature is a data model for all the features of one feature type. A schema feature describes the attributes, coordinate system, and allowed geometries that features of that type share. Schema features are described in detail in the chapter *Working with Features* on page 13.

In this chapter, you will learn how to:

- create a reader
- open a reader

- get reader settings from a user
- get information about available readers
- retrieve schema features from a reader
- retrieve data features from a reader
- close a reader

Issues regarding the relationship between readers and coordinate systems are discussed in Chapter 7, *Working with Coordinate Systems*.

Creating a Reader

To create a new reader you must call the `createReader` method of the `FMEOSession` object. The following code fragment shows how to do this:

```
Dim fmeDirectives As FMEOStringArray
Set fmeDirectives = m_fmeSession.createStringArray
fmeDirectives.append ("OUTPUT_STATS")
fmeDirectives.append ("NO")
Set m_fmeReader = m_fmeSession.createReader("SHAPE", _
                                 False, fmeDirectives)
```

The format type identifier, an uppercase alphanumeric string that uniquely identifies the reader, controls which reader is created. It is the first parameter passed to the `createReader` method, and in the above example, `SHAPE` is used to request an ESRI Shapefile reader. The *FME Readers and Writers* manual provides a Quick Facts table for each format. The first row in this table contains the format type identifier for the reader. The table below shows the Quick Facts table created for ESRI Shapefile:

Format Type Identifier	SHAPE
Reader/Writer	Both
Dataset Type	Directory
Feature Type	File base name
Typical File Extensions	`.shp (.shx, .dbf)`
Automated Translation Support	Yes
User-Defined Attributes	Yes
Coordinate System Support	No
Generic Color Support	No
Spatial Index	Optional
Schema Required	Yes
Transaction Support	No

Geometry Type Attribute	SHAPE_GEOMETRY

Geometry Support

Geometry	Supported	Geometry	Supported
aggregate	yes	polygon	yes
circles	no	donut polygon	yes
circular arc	no	line	yes
elliptical arc	no	point	yes
ellipses	no	text	no
none	yes	3D	yes

Note FME Objects can read a format only if the value of the “Reader/Writer” row of the Quick Facts table is either “Reader” or “Both”.

The second parameter to the `createReader` method is a Boolean value that instructs the reader whether or not to create an on-disk feature cache. If your application needs to access datasets more than once, it should use caching to improve performance. Caching reduces the amount of file I/O required when accessing a dataset multiple times. When caching is specified, FME Objects builds a disk-based spatial index that provides fast random access to features and allows advanced spatial and attribute queries. This functionality is covered in more detail in the *Using Constraints* section.

The third and final parameter to the `createReader` method is a string array containing a list of directives that control the reader creation. Reader creation directives are specified as a set of name and value pairs. The following directives are supported:

Directive Name	Description
COORDSYS	The name of the coordinate system with which all features should be tagged (see *Tagging Features on Input* section of Chapter 7, *Working with Coordinate Systems*).
USER_PIPELINE	The name of a file containing a factory pipeline to be applied to all features read. If the name is not a full path name, then the pipeline file must be in the pipeline directory under the FME installation. Specifying this directive results in all features being passed through the user pipeline before being given to the application.
OUTPUT_STATS	If NO, then no reader statistics are output to the log file. The default is YES.

WARNING A string array must be created and passed in to the `createReader` method even if there are no directives being supplied to it.

Opening a Reader

Once you've created a reader, you can open it on a specific dataset. The following code fragment opens the SHAPE reader created in the previous section on a dataset called `c:\shape`.

```
Dim fmekeywords As FMEOStringArray
Set fmeKeywords = m_fmeSession.createStringArray
fmeKeywords.append("MEASURES_AS_Z")
fmeKeywords.append("YES")
fmeKeywords.append("IDs")
fmeKeywords.append("roads.shp")
fmeKeywords.append("IDs")
fmeKeywords.append("rivers.shp")
Call m_fmeReader.open("c:\shape", fmeKeywords)
```

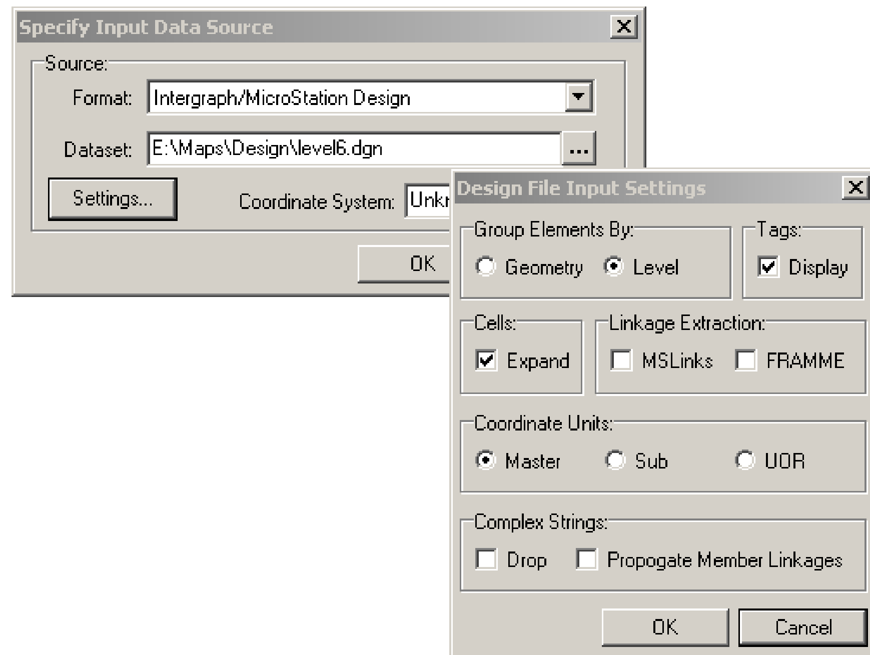
The first parameter to `FMEOReader's open` method is the dataset name. In the case of the ESRI Shapefile reader, a directory dataset is required. Other readers may require a file dataset, a database dataset, or a URL dataset as stated by the Dataset Type entry of their Quick Facts table.

The second parameter to the `open` method is a string array containing a list of keywords that control the operation of the reader, specified as a set of name and value pairs. The *Reader Keywords* section of each chapter in the *FME Readers and Writers* manual lists the keywords processed by individual readers. In this example, a value of `YES` for the `MEASURES_AS_Z` keyword directs the Shapefile reader to treat measures data as elevations. The `IDs` keyword directs the reader to open only the `roads.shp` and `rivers.shp` files.

Getting Reader Settings from the User

The approach for creating and opening a reader described up to this point in the chapter gives you full control over the directives used to create the reader, the keywords passed as parameters to the reader, and the dataset opened by the reader. If your application gives the end user the ability to specify all this information interactively, you can take advantage of the services provided by the `FMEODialog` object to simplify the process of creating and opening a new reader. `FMEODialog's sourcePrompt` method displays a dialog window and handles all the end user interaction to determine the format, dataset, directives, and keywords.

For example, the "Specify Input Data Source" and the "Design File Input Settings" dialogs are shown below:



WARNING The `FMEODialog` object is only available on Windows platforms – it is not available on other operating systems.

The `SourcePrompt` method returns the information specified by the user in a form that can be passed directly to `FMEOSession's createReader` and `FMEOReader's open` methods. The `SourcePrompt` procedure demonstrates this approach:

```
Public Sub SourcePrompt(sSourceDir As String)
    Dim fmeDialog As FMEODialog
    Dim fmeDirectives As FMEOStringArray
    Dim fmeKeywords As FMEOStringArray
    Dim sDataset As String
    Dim sFormat As String
    Dim bCompleted As Boolean

    Set fmeDialog = m_fmeSession.createDialog
    Set fmeDirectives = m_fmeSession.createStringArray
    Call fmeDirectives.append("LIMIT_FORMATS")
    Call fmeDirectives.append("MIF")
    bCompleted = fmeDialog.SourcePrompt("MIF", sSourceDir, _
                            sFormat, sDataset, fmeDirectives)
    If bCompleted = True Then
        Set m_fmeReader = m_fmeSession.createReader(sFormat, _
```

```
                          True, fmeDirectives)
        Set fmeKeywords = m_fmeSession.createStringArray
        Call m_fmeReader.open(sDataset, fmeKeywords)
        m_bReaderOpened = True
    End If
End Sub
```

The following input directives are supported by the `sourcePrompt` method:

Directive Name	Description
TITLE	Specifies the title to be used for the dialog box. The default is “Specify Input Data Source”.
LIMIT_FORMATS	Limits the formats accessible through the “Format:” drop-down list box as list of format identifiers separated by | characters. By default, all available formats are accessible.
SETTINGS_ONLY	If YES, then only the settings box for the passed in source format and dataset is shown. The default is NO.
SPATIAL_SETTINGS	If YES, then any settings box that allows for specifying spatial constraints will have that aspect remain enabled. If NO, then the spatial area specification aspect of the settings boxes is disabled. The default is YES. This directive is useful for disabling spatial settings when a source dialog is used for gathering only schema information.

Getting Information about Available Readers

Using the Specify Input Data Source dialog, the end user can choose a source format from the list of all the formats available to your application. If your application needs to access the list of available formats programmatically, it can do so using the `getAvailableFormats` method of the `FMEODialog` object. Furthermore, your application can obtain detailed information about any of the available formats using the `getFormatInfoEx` method. The `GetReaderInfo` procedure displays a dialog box with all the information for a specified reader.

```
Public Sub GetReaderInfo(ByVal sFormatName As String)
    Dim lPos As Long
    Dim sMsg As String
    Dim sDirection As String
    Dim bSpatialIndex As Boolean
    Dim fmeDialog As FMEODialog
    Dim fmeFormats As FMEOStringArray
    Dim fmeFormatInfo As FMEOStringArray

    Set fmeDialog = m_fmeSession.createDialog
    Set fmeFormats = m_fmeSession.createStringArray
    Set fmeFormatInfo = m_fmeSession.createStringArray
    Call fmeDialog.getAvailableFormats(fmeFormats)
    lPos = GetIndex(fmeFormats, sFormatName)
    If lPos = -1 Then
        sMsg = "Format not available: "
```

```
            sMsg = sMsg & sFormatName & vbCrLf
            MsgBox sMsg, vbOKOnly, "GetReaderInfo"
            Exit Sub
        End If
        Call fmeDialog.getFormatInfoEx(sFormatName, fmeFormatInfo)
        sMsg = "Format: " & sFormatName
        lPos = GetIndex(fmeFormatInfo, "FORMAT_LONG_NAME")
        sMsg = sMsg & vbCrLf & "Long Name: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "DATASET_TYPE")
        sMsg = sMsg & vbCrLf & "Dataset Type: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "INPUT_OUTPUT")
        sMsg = sMsg & vbCrLf & "Direction: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "FILE_EXTENSIONS")
        sMsg = sMsg & vbCrLf & "File Filter: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "COORD_SYSTEM_AWARE")
        sMsg = sMsg & vbCrLf & "Coordinate System Aware: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "SOURCE_SETTINGS")
        sMsg = sMsg & vbCrLf & "Has Source Settings: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "DESTINATION_SETTINGS")
        sMsg = sMsg & vbCrLf & "Has Destination Settings: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        lPos = GetIndex(fmeFormatInfo, "AUTOMATED_TRANSLATION")
        sMsg = sMsg & vbCrLf & "Supports Automated Translation: "
        sMsg = sMsg & fmeFormatInfo.element(lPos + 1)
        MsgBox sMsg, vbOKOnly, "GetReaderInfo"
    End Sub
```

Note See Chapter 10 for the implementation of the GetIndex procedure.

The `getFormatInfoEx` method of `FMEODialog` returns the information about a format as a set of name and value pairs. The following table describes each of the fields returned:

Field Name	Description
FORMAT_LONG_NAME	The format name.
DATASET_TYPE	The type of dataset used by this format. Range: DIR: Directory or set of files DIRONLY: Directory only FILE: Single file only FILEDIR: FILE for input, DIR for output DIRFILE: DIR for input, FILE for output DATABASE: Database URL: Uniform Resource Locator (For more details, see the Quick Facts section in *About This Manual* in the *FME Readers and Writers* manual).
INPUT_OUTPUT	Specifies whether reading only, writing only, or both reading and writing is available for this format. Range: INPUT, OUTPUT or BOTH.
FILE_EXTENSIONS	Specifies the file extensions usually associated with this format separated by	characters.
COORD_SYSTEM_AWARE	Specifies whether datasets of this format can store coordinate system information and, if so, whether the format reader can extract it. Range: YES or NO.
SOURCE_SETTINGS	If reading is available for this format, this field specifies whether the settings box is available from the Specify Input Data Source dialog. Range: YES or NO.
DESTINATION_SETTINGS	If writing is available to this format, this field specifies whether the settings box is available from the Specify Output Data Source dialog. Range: YES or NO.
AUTOMATED_TRANSLATION	Specifies whether this format supports automated reading, writing, or both. Range: INPUT, OUTPUT or BOTH.

Additional information about an `FMEOReader` object can be obtained using `getProperties` method. For example, the following code can be used to determine whether a reader has a spatial index:

```
Set fmeProperties = m_fmeSession.createStringArray
Call m_fmeReader.getProperties("fme_prop_spatial_index", _
fmeProperties)
```

Reading Schema Features

Your application is not forced to read the schema features from a dataset before accessing the data features; it can go straight to reading the data features.

The readSchema method returns a single schema feature each time it is called. When there are no more features to read, the readSchema method returns True; otherwise, it returns False. The LogSchemaFeatures procedure logs all the schema features for a dataset.

```
Public Sub LogSchemaFeatures()
    Dim bLastSchema As Boolean
    Dim fmeSchemaFeature As FMEOFeature

    bLastSchema = False
    Set fmeSchemaFeature = m_fmeSession.createFeature
    Do While bLastSchema = False
        bLastSchema = m_fmeReader.readSchema(fmeSchemaFeature)
        If bLastSchema = False Then
          Call m_fmeLogfile.logFeature(fmeSchemaFeature, 1, 1)
       End If
     Loop
End Sub
```

Reading Data Features

Once your application has created and opened a reader (and optionally read the schema features) it's ready to start reading the data features. This task is accomplished using the read method. The read method returns one data feature at a time. When there are no more features to read, the read method returns True; otherwise, it returns False. The following code fragment illustrates how to read all the features from a reader into a feature vector.

```
bEnd = False
Do While bEnd = False
    Set fmeDataFeature = m_fmeSession.createFeature
    bEnd = m_fmeReader.read(fmeDataFeature)
    If bEnd = False Then
        Call fmeFeatureVector.append(fmeDataFeature)
    End If
Loop
```

Note An important aspect of this code fragment is that a new FMEOFeature object is created before every call to the read method, otherwise the every new feature read would overwrite the previous feature read.

Using Constraints

Contraints give your application the ability to ask FME Objects to limit the features read to only those that match a set of criteria by performing simple spatial queries and attribute queries. Also, after the last feature has been read,

your application can start a new pass through the features in the input dataset using the `setConstraints` method.

For convenience, the mechanism for specifying a constraint uses an `FMEOFeature` object, the constraint feature, where the constraints to be applied are described by the attributes on this feature. The constraint feature is passed to `setConstraints` as an input parameter. The name of the attribute used to specify the type of filtering to be used is `fme_search_type`. The following table describes the different values supported for `fme_search_type`:

fme_search_type value	Description

`fme_all_features`	This search type is used to read the subset of features from the dataset that meet a certain test criteria. The value of the `fme_test` attribute on the constraint features specifies the test criteria. The format of the test criteria is:
	`<value> <operator> <value>`
	Where `<value>` can be a literal constant or an attribute value function and `<operator>` must be one of `<, >, =, !=, >=, or <=`. Example test criteria are:
	`@Length() > 100`
	`@Value(Highway) = "I 90"`
	`@FeatureType() = "Road"`
	If the constraint feature does not have an `fme_test` attribute, all features are read.

`fme_envelope_intersects`	This search type is used to read the subset of features from the dataset whose bounding box intersects a rectangular selection area. The bounding box of the constraint feature is used as the selection area. The `fme_envelope_intersects` search type can be combined with an attribute test criteria by supplying the `fme_test` attribute on the constraint feature (see `fme_all_features`).
`fme_closest`	This search type is used to retrieve the closest feature to a selection point. If the constraint feature has only one point, that point is used as the search point; otherwise the centroid of the constraint feature's bounding box is used. When the `fme_closest` search type is used, an additional attribute, `fme_max_distance`, must be used to specify the maximum allowable distance from the search point. The `fme_closest` search type can be combined with an attribute test criteria by supplying the `fme_test` attribute on the constraint feature (see `fme_all_features`). The semantics of this operation is to select the closest feature to the search point if it meets the attribute test criteria; otherwise no feature is selected.

If your application performs multiple passes through the input dataset, it is highly recommended that you enable caching when the reader is created. As a general rule, use caching if you will be reading the input dataset more than two times (to compensate for the overhead of creating the cache).

When caching is enabled, all the features in the input dataset will be loaded into the cache as they are read for the first time. The `setConstraints` method, called

in any of the configurations described previously, will start a new read from the cache instead of the original input dataset.

Caching works differently when you are working with a source dataset that has a spatial index of its own (for example, SDE). In this case, the cache is reloaded when a constraint query results in an extent that is not wholly contained inside the cached extent. Your application can determine if a format has a spatial index using the `getProperties` method. When reading from such formats, an FME Objects reader cache is not essential, but may still provide a performance benefits.

Closing a Reader

A reader is closed using the `close` method as demonstrated by the following code fragment:

```
Call m_fmeReader.Close
```

After a reader is closed, you can optionally ask Visual Basic to destroy the object reference as follows:

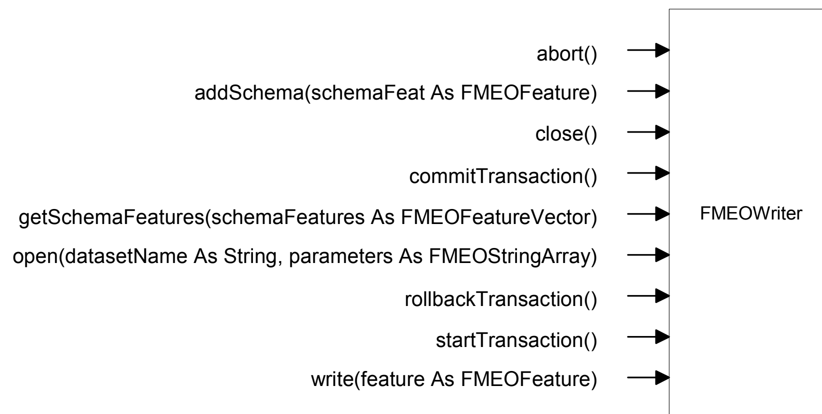
```
Set m_fmeReader = Nothing
```

This has the direct effect of releasing all the resources associated with the reader.

CHAPTER 5

Writing Features to a Dataset

The FME universal writer object, FMEOWriter, allows your application to create new datasets in any supported format. The methods and properties of `FMEOWriter` provide a simple and configurable interface.



Methods and Properties of the FMEOWriter Object

In this chapter, you will learn how to:

- create a writer
- open a writer
- get writer settings from the user
- get information about available writers
- create new features
- close a writer

Issues regarding the relationship between writers and coordinate systems are covered in detail in Chapter 7, *Working with Coordinate Systems*.

Creating a Writer

To create a new writer you must call the `createWriter` method of the FMEOSession object. This following code fragment shows how to do this:

```
Dim fmeDirectives As FMEOStringArray
Set fmeDirectives = m_fmeSession.createStringArray
fmeDirectives.append ("OUTPUT_STATS")
fmeDirectives.append ("NO")
Set m_fmeWriter = m_fmeSession.createWriter("SHAPE", _
                                           fmeDirectives)
```

The first parameter passed to the `createWriter` method is the format type identifier, an uppercase alphanumeric string that specifies the writer being created. In the above example, "SHAPE" is used to request an ESRI Shapefile writer. The *FME Readers and Writers* manual provides a Quick Facts table for each format. The first row in this table contains the format type identifier for the writer.

Note FME Objects can write to a format only if the value of the "Reader/Writer" row of the Quick Facts table is either "Writer" or "Both".

The second parameter to the `createWriter` method is a string array containing a list of directives that control the writer creation. Writer creation directives are specified as a set of name and value pairs. The following directives are supported:

Directive Name	Description
`COORDSYS`	The name of the coordinate system that all features should be reprojected to (see *Reprojecting Features on Output* section of the chapter *Working with Coordinate Systems*).
`USER_PIPELINE`	The name of a file containing a factory pipeline to be applied to all features written. If the name is not a full path name, then the pipeline file must be in the pipeline directory under the FME installation. Specifying this directive results in all features being passed through the user pipeline before being given to the destination writer.
`OUTPUT_STATS`	If `NO`, then no writer statistics are output to the log file. The default is `YES`.

WARNING A string array must be created and passed in to the createWriter method even if there are no directives being supplied to it.

Opening a Writer

Once you've created a writer, you can open it with a specific dataset. The following code fragment opens the SHAPE writer created in the previous section with a dataset called `c:\shape`.

```
Set fmeKeywords = m_fmeSession.createStringArray
fmeKeywords.append("MEASURES_AS_Z")
fmeKeywords.append("YES")
Call m_fmeWriter.open("c:\shape", fmeKeywords)
```

The first parameter to `FMEOWriter's open` method is the dataset name. In the case of the Shapefile writer, a directory is required. Other writers may require a file, a database, or a URL dataset as stated by the Dataset Type entry of their Quick Facts table.

The second parameter to the `open` method is a string array containing a list of keywords that control the operation of the writer, specified as a set of name and value pairs. The *Writer Keywords* section of each chapter in the *FME Readers and Writers* manual lists the keywords processed by individual writers. In this example, a value of `YES` for the `MEASURES_AS_Z` keyword directs the Shapefile writer to treat measures data as elevations.

> **WARNING** All writers support the DEF keyword. FME Objects treats the `DEF` keyword differently than other keywords. The recommended approach for specifying the information typically associated with a `DEF` keyword is to use a schema feature. See the *Writing Features* section for details.

Getting Writer Settings from the User

The approach for creating and opening a writer described up to this point allows your application to control: the directives used to create the writer, the keywords passed as parameters to the writer, and the dataset opened by the writer. If the end user of your application is to specify this information interactively, you can use the services provided by the `FMEODialog` object. `FMEODialog's destPrompt` method displays a dialog window that allows the user to specify the format, dataset, directives, and keywords.

For example, the "Specify Output Data Destination" and the "Design File Output Settings" dialogs are shown below:

WARNING The FMEODialog object is only available on Windows platforms.

The `DestPrompt` method returns information specified by the user in a form that can be passed directly to `FMEOSession's createWriter` and `FMEOWriter's open` methods. The following procedure demonstrates this approach:

```
Public Sub DestPrompt(sDestDir As String)
    Dim fmeDialog As FMEODialog
    Dim fmeDirectives As FMEOStringArray
    Dim fmeKeywords As FMEOStringArray
    Dim sDataset As String
    Dim sFormat As String
    Dim bCompleted As Boolean

    Set fmeDialog = m_fmeSession.createDialog
    Set fmeDirectives = m_fmeSession.createStringArray
    Call fmeDirectives.append("LIMIT_FORMATS")
    Call fmeDirectives.append("MIF")
```

```
    bCompleted = fmeDialog.DestPrompt("MIF", sDestDir, _
                             sFormat, sDataset, fmeDirectives)
    If bCompleted = True Then
        Set m_fmeWriter = m_fmeSession.createWriter( _
                                                sFormat, _
                                                fmeDirectives)
        Set fmeKeywords = m_fmeSession.createStringArray
        Call m_fmeWriter.open(sDataset, fmeKeywords)
    End If
End Sub
```

The following input directives are supported by the `destPrompt` method:

Directive Name	Description
TITLE	Specifies the title to be used for the dialog box. The default is "Specify Output Data Destination".
LIMIT_FORMATS	Limits the formats accessible though the "Format:" drop-down list box as list of format identifiers separated by | characters. By default all available formats are accessible.
SETTINGS_ONLY	If YES, then only the settings box for the passed in destination format and dataset is shown. The default is NO.

Getting Information About Available Writers

If your application needs to access the list of available formats programmatically, it can do so using the `getAvailableFormats` method of the `FMEODialog` object. Furthermore, your application can obtain detailed information about any of the available formats using the `getFormatInfoEx` method. The details of using these methods are covered in the section *Getting Information about Available Readers* in Chapter 4, *Reading Features from a Dataset*.

Writing Features

Some applications need to create new features from scratch and then write them out to a destination dataset. An example is an application that allows users to annotate a map and save the annotations.

Writing features is a two-step process. First, the schema for each feature type is supplied to the writer using the `addSchema` method. Schema features are only essential if the "Schema Required" row of the Quick Facts table is "Yes". If the value is "No", the writer ignores schema features. Once all the possible schemas have been specified, the second step is to write out the data features using the `write` method.

To create a blank schema feature your application calls the `createFeature` method of the `FMEOSession` object. A valid schema feature is then built by specifying the following information:

- Feature type: The feature type described by the schema feature is set using the `featureType` property. The feature type corresponds to the theme, layer, level, file or database table name depending on the output format; check the Feature Type entry in the format's Quick Facts table.
- List of user attributes: If the format supports user-defined attributes (that is, the "User-Defined Attributes" row of the Quick Facts table is "Yes"), each user attribute that describes a data features of this type is added using the `sequencedAttribute` property.
- List of valid FME types: The list of one or more `fme_type` values that are valid for data features of this feature type is specified in a list attribute named `fme_geometry` using the `setListAttribute` method.

Here's an example of how to create a schema feature:

```
Dim fmeSchemaFeature As FMEOFeature
Dim fmeGeomList As FMEOStringArray
Set fmeSchemaFeature = m_fmeSession.createFeature
fmeSchemaFeature.featureType = "tracts"
fmeSchemaFeature.sequencedAttribute("CNTY_FIPS") = _
                                    "fme_decimal(15,3)"
fmeSchemaFeature.sequencedAttribute("PERIMETER") = _
                                    "fme_decimal(15,3)"
fmeSchemaFeature.sequencedAttribute("SQ_MILES") = _
                                    "fme_decimal(15,3)"
fmeSchemaFeature.sequencedAttribute("STATE") = _
                                    "fme_char(20)"
Set fmeGeomList = m_fmeSession.createStringArray
fmeGeomList.append ("fme_area")
Call fmeSchemaFeature.setListAttribute("fme_geometry", _
                                             fmeGeomList)
```

Note Geometry attributes do not need to be specified on a schema feature, they are known implicitly.

Once a schema feature is created and passed to the writer, your application can write data features of that schema to the dataset as shown by the following code fragment:

```
Call m_fmeWriter.addSchema(fmeSchemaFeature)
lFeatureCount = m_fmeFeatureVector.entries
For i = 0 To lFeatureCount - 1
  Set fmeFeature = m_fmeFeatureVector.element(i)
  Call m_fmeWriter.write(fmeFeature)
Next i
```

This code assumes that all features contained in `fmeDataFeatureVector` have the same feature type as `fmeSchemaFeaure`.

> **WARNING** Both the `addSchema` and `write` methods of the FMEOWriter object clear the contents of the feature that is passed to them. If your application needs to use the information in a feature object after writing it out, you can use the `clone` method of the FMEOSession class to make a copy and then pass the copy to the writer instead of the original.

Closing a Writer

A writer must be closed after the last feature is written to a dataset. If a writer is not closed, the dataset may be invalid. A writer is closed using the `close` method as demonstrated by the following code fragment:

```
Call m_fmeWriter.Close
```

After a writer is closed, you can optionally ask Visual Basic to destroy the object reference as follows:

```
Set m_fmeWriter = Nothing
```

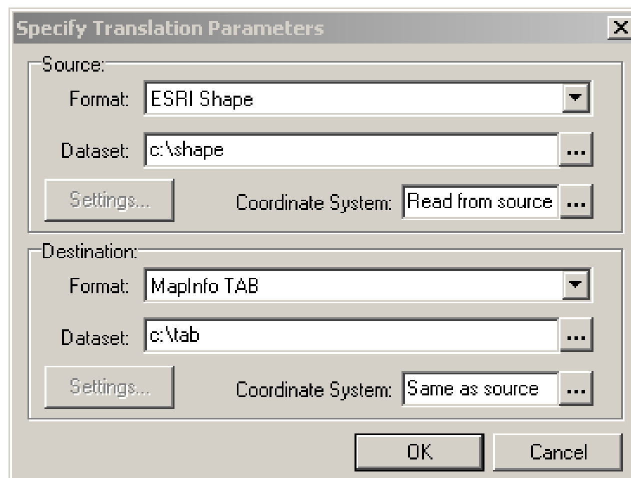
This has the direct effect of releasing all the resources associated with the writer.

CHAPTER

6

Translating Features

The design of the FME Objects API makes it very simple to translate features from one format to another. The first step is for your application to create and open the desired reader and writer. You can accomplish this by using the techniques described in Chapter 4, *Reading Features from a Dataset*, and Chapter 5, *Writing Features to a Dataset*. If the end user will specify all the translation parameters, you can take advantage of the services provided by the `FMEODialog` object to simplify the process of creating and opening a reader and writer. `FMEODialog's xlatePrompt` method displays a dialog window that allows the end user to specify the format, dataset, and keywords for both a reader and a writer.



WARNING The `FMEODialog` object is only available on Windows platforms.

The `xlatePrompt` method returns the information specified by the user in a form that can be passed directly to `FMEOSession's createReader` and

createWriter, FMEOReader's open, and FMEOWriter's open methods. The following procedure demonstrates this approach:

```
Public Sub TranslationPrompt(sSourceDir As String, _
                                sDestDir As String)
    Dim sDestDataset As String
    Dim sDestFormat As String
    Dim sSrcDataset As String
    Dim sSrcFormat As String
    Dim bCompleted As Boolean
    Dim bEnd As Boolean
    Dim fmeFeature As FMEOFeature
    Dim fmeDialog As FMEODialog
    Dim fmeSrcDirectives As FMEOStringArray
    Dim fmeDestDirectives As FMEOStringArray
    Dim fmeKeywords As FMEOStringArray

    Set fmeKeywords = m_fmeSession.createStringArray
    Set fmeDialog = m_fmeSession.createDialog
    Set fmeSrcDirectives = m_fmeSession.createStringArray
    Set fmeDestDirectives = m_fmeSession.createStringArray
    bCompleted = fmeDialog.xlatePrompt("MIF", sSourceDir, _
                 "MIF", sDestDir, _
                 sSrcFormat, sSrcDataset, fmeSrcDirectives, _
                 sDestFormat, sDestDataset, fmeDestDirectives)
    If bCompleted = True Then
        Set m_fmeReader = m_fmeSession.createReader( _
                            sSrcFormat, _
                            True, fmeSrcDirectives)
        Call m_fmeReader.open(sSrcDataset, fmeKeywords)
        Set m_fmeWriter = m_fmeSession.createWriter( _
                                    sDestFormat, _
                                    fmeDestDirectives)
        Call m_fmeWriter.open(sDestDataset, fmeKeywords)
End Sub
```

The following input directives are supported by the xlatePrompt method:

Directive Name	Description
TITLE	The title to be used for the dialog box.
LIMIT_FORMATS	The short names of the only formats that will be allowed, separated by | characters.
SPATIAL_SETTINGS	If YES, then any settings box that allows for specifying spatial constraints will have that aspect remain enabled. If NO, then the spatial area specification aspect of the settings boxes is disabled. The default is YES. This directive is useful for disabling spatial settings when a source dialog is used for gathering only schema information.

If the TITLE directive is specified as part of both the source and destination directives, the value provided in the destination directive will take precedence.

Once a reader and writer have been opened, the process of translating features is as simple as passing the schema features and then the data features from the reader to the writer. The code fragment below shows how to do this.

```
bEnd = False
Do While bEnd = False
    bEnd = m_fmeReader.readSchema(fmeFeature)
    If bEnd = False Then
        Call m_fmeWriter.addSchema(fmeFeature)
    End If
Loop
bEnd = False
Do While bEnd = False
    bEnd = m_fmeReader.read(fmeFeature)
    If bEnd = False Then
        Call m_fmeWriter.write(fmeFeature)
    End If
Loop
```

WARNING Both the `addSchema` and `write` methods of the FMEOWriter object clear the contents of the feature that is passed to them. If your application needs to use the information in a feature object after writing it out, you must remember to clone it first and pass the clone to the writer instead of the original.

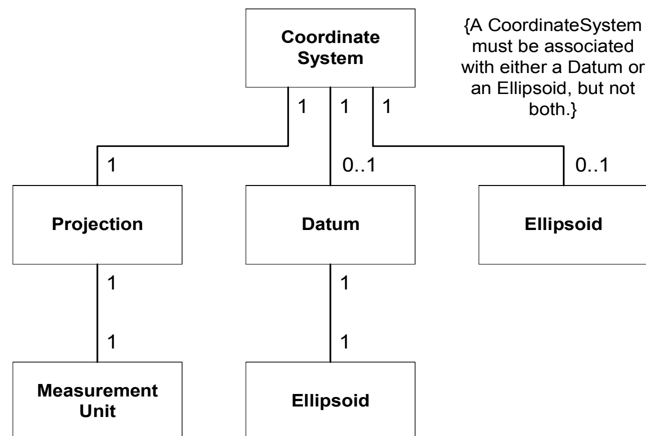
Once all schema features and data features have been transferred from the source dataset to the destination dataset, the reader and writer objects must be closed as follows:

```
Call m_fmeReader.Close
Call m_fmeWriter.Close
```

CHAPTER 7

Working with Coordinate Systems

In the context of GIS, a two-dimensional coordinate system is a reference grid used to measure linear distances on a plane. A three-dimensional coordinate system measures distances in a three-dimensional space. A coordinate system is typically defined by a map projection, a spheroid of reference (or ellipsoid), a datum, one or more standard parallels, a central meridian, and possible shifts in the horizontal and vertical directions. Coordinate systems used for cartography are typically cartesian: the axes are orthogonal to each other and they use the same measurement unit. The FME Objects data model for coordinate systems is shown below.



FME Coordinate System Conceptual Data Model

This is a conceptual model of the data encapsulated by the `FMEOCoordSysManager` object: FME Objects does not provide a one-to-one mapping between the classes illustrated and the objects exposed by the API. Instead there are methods and properties on the `FMEOCoordSysManager` object

that provide access to all of these classes. These methods and properties are illustrated below and discussed in more detail the subsections that follow.

.

Methods and Properties of the FMEOCoordSysManager Object

The majority of FME Objects application developers do not need to be concerned with coordinate systems because FME Objects' generic behavior for handling translations of data with coordinate systems is sufficient for most applications. However, for those applications with special requirements in this area, the `FMEOCoordSysManager` object offers a powerful and flexible way to deal with coordinate systems.

If your application requires custom coordinate system functionality, you should start by reading the *Coordinate System Support* chapter in the *FME Foundation* manual. This reference provides fundamental information about how FME Objects supports coordinate systems. Another important source of information is the *FME Readers and Writers* manual: each chapter begins with a Quick Facts table that contains a Coordinate System Support row. A value of "Yes" in this row indicates that the format can store coordinate system information, the reader can read this information if it is present in the source dataset, and the writer will save the coordinate system information to the destination dataset. A value of "No" indicates that features coming from the reader will always have an unknown coordinate system and features with a known coordinate system passed to the writer will be saved without the coordinate system information.

All FME Objects feature geometries are associated with a coordinate system, and at the most general level, there are two different coordinate system-related operations that can be performed on a feature. The first is to change its coordinate system to a new coordinate system without actually converting the feature itself to the new coordinate system. This operation is known as *tagging*. The second is to change the feature's coordinate system to a new coordinate system and at the same time convert the feature itself to the new coordinate system. This operation is know as *reprojecting*.

When FME Objects reads from a format that does not have coordinate system support, the resulting feature geometries are associated with the special "Unknown" coordinate system. Readers with coordinate system support may also return features associated with the "Unknown" coordinate system or with a "non-earth" coordinate system, which, in the context of reprojecting, is equivalent to the "Unknown" coordinate system. When dealing with such features, the reprojecting operation degenerates to a tagging operation because there is insufficient information to perform a coordinate conversion. Following the same reasoning, reprojecting a feature to the "Unknown" coordinate system is equivalent to tagging it with "Unknown".

In the rest of this chapter, you'll learn how to:

- examine a feature geometry's coordinate system
- tag all features on input
- reproject all features on output
- tag individual features
- reproject individual features
- create a temporary coordinate system for use during a session
- add a persistent coordinate system to FME Objects

Examining a Feature's Coordinate System

If you are reading features from a format that has coordinate system support, your application may need to determine the details of the geometry's coordinate system to decide how to process the features. The `GetCoordSys` procedure demonstrates how to use the `getCoordSysParms` method, in combination with the `ellipsoid`, `datum`, and `unit` properties, to obtain all the information for the coordinate system associated with `fmeFeature`. The coordinate system information is returned in four string arrays: `fmeCoordSysParams`, `fmeUnitParams`, `fmeDatumParams`, and `fmeEllipsoidParams`.

```
Sub GetCoordSys(ByRef fmeFeature As FMEOFeature, _
            ByRef fmeCoordSysParams As FMEOStringArray, _
            ByRef fmeUnitParams As FMEOStringArray, _
            ByRef fmeDatumParams As FMEOStringArray, _
            ByRef fmeEllipsoidParams As FMEOStringArray)

    Dim sCoordSysName As String
    Dim sUnitName As String
    Dim sEllipsoidName As String
    Dim sDatumName As String
    Dim fmeCoordSysMan As FMEOCoordSysManager
    Dim lPosition As Long

    sCoordSysName = fmeFeature.coordSys
    Set fmeCoordSysMan = m_fmeSession.coordSysManager
    Set fmeCoordSysParams = fmeCoordSysMan.getCoordSysParms _
```

```
                                            (sCoordSysName)

    lPosition = GetIndex(fmeCoordSysParams, "UNIT")
    If lPosition >= 0 Then
        sUnitName = fmeCoordSysParams.element(lPosition + 1)
        Set fmeUnitParams = fmeCoordSysMan.unit(sUnitName)
    End If

    lPosition = GetIndex(fmeCoordSysParams, "DT_NAME")
    If lPosition >= 0 Then
        sDatumName = fmeCoordSysParams.element(lPosition + 1)
        Set fmeDatumParams = fmeCoordSysMan.datum(sDatumName)
        lPosition = GetIndex(fmeDatumParams, "ELLIPSOID")
        If lPosition >= 0 Then
            sEllipsoidName = fmeDatumParams.element( _
                                            lPosition + 1)
           Set fmeEllipsoidParams = fmeCoordSysMan.ellipsoid _
                                           (sEllipsoidName)
        End If
    Else
        lPosition = GetIndex(fmeCoordSysParams, "EL_NAME")
        If lPosition >= 0 Then
            sEllipsoidName = fmeCoordSysParams.element( _
                                            lPosition + 1)
           Set fmeEllipsoidParams = fmeCoordSysMan.ellipsoid _
                                           (sEllipsoidName)
        End If
    End If
End Sub
```

The `GetIndex` function is used to get the index of a string array element. If the specified element does not exist, `GetIndex` returns -1. The code for the `GetIndex` function is provided in the *Working with Collections* chapter.

WARNING When your application reads from a format with coordinate system support and directory dataset type, it is possible that not all the files in a source directory have the same coordinate system. In this situation, FME Objects will automatically reproject all the features to the first coordinate system read.

If your application needs to obtain the definition of a coordinate system in the OpenGIS Well-Known-Text (WKT) format, use the `getCoordSysAsOGCDef` method instead of `getCoordSysParms. The getCoordSysAsOGCDef` method supports several dialects of WKT (see API reference for details).

Tagging Features on Input

A common requirement is to tag a source dataset with a new coordinate system. The need for this operation arises in the following situations:

- The source dataset does not have an associated coordinate system and you would like to tag it with one.
- The source dataset has the wrong coordinate system and you would like to tag it with the correct one.

The following code fragment instructs the reader to tag all input features with the 10TM115-27 coordinate system:

```
fmeDirectives.append ("COORDSYS")
fmeDirectives.append ("10TM115-27")
Set m_fmeReader = m_fmeSession.createReader( _
                        strSourceFormat, True, fmeDirectives)
```

If you are using `FMEODialog's` `sourcePrompt` method to obtain source dataset information from the user, the user is automatically given the option to select a coordinate system for tagging. For example, if the source dataset prompt is displayed with the following command:

```
bCompleted = fmeDialog.sourcePrompt("", "", _
strSourceFormat, strSourceDataset, fmeUserDirectives)
```

and the user selects 10TM115-83 from the Coordinate System list box, then the `fmeUserDirectives` string array will contain the `COORDSYS,10TM115-83` pair when the method returns.

> **WARNING** If multiple sets of `COORDSYS, value` pairs are passed to `FMEOSession's` `createReader` method, the first pair will be used.

Reprojecting Features on Output

The following code fragment instructs the writer to reproject all features to the 10TM115-27 coordinate system:

```
fmeDirectives.append ("COORDSYS")
fmeDirectives.append ("10TM115-27")
Set m_fmeWriter = m_fmeSession.createWriter( _
                                  strDestFormat, fmeDirectives)
```

The reprojection will take place even if the destination format does not support coordinate system information.

If you are using `FMEODialog's` `destPrompt` method to obtain destination dataset information interactively, the user is automatically given the option to select a coordinate system to reproject to. For example, say the destination dataset prompt is displayed with the following command:

```
bCompleted = fmeDialog.destPrompt("", "", strDestFormat, _
strDestDataset, fmeUserDirectives)
```

If the user selects 10TM115-83 from the Coordinate System listbox, then the `fmeUserDirectives` string array will contain the `COORDSYS,10TM115-83` pair when the method returns.

> **WARNING** If multiple sets of `COORDSYS, value` pairs are passed to `FMEOSession's createWriter` method, the first pair will be used.

Tagging Individual Features

The `coordSys` property of the `FMEOFeature` object can be used to tag a feature with a new coordinate system. For example, the following code fragment iterates through a feature vector and tags each feature with a new coordinate system.

```
lFeatureCount = m_fmeFeatureVector.entries
For i = 0 To lFeatureCount - 1
    m_fmeFeatureVector.element(i).coordSys = "10TM115-27"
Next i
```

Creating Temporary Coordinate Systems

If needed, your application can create a new custom coordinate system. This is accomplished using the `unit`, `datum`, and `ellipsoid` properties and the `defineCoordSys` method.

> **WARNING** Units, datums, ellipsoids, and coordinate systems created using this approach have a lifetime equal to the session. The next section provides instructions for creating persistent coordinate systems.

For example, the following code fragment uses the `ellipsoid` property of the `FMEOCoordSysManager` object to define a new ellipsoid called *`MyEllipsoid`*. This new ellipsoid can be referenced during the remainder of the session lifetime.

```
fmeParameters.append ("E_RAD")
fmeParameters.append ("6376950")
fmeParameters.append ("P_RAD")
fmeParameters.append ("6356352.616")
fmeCoordSysMan.ellipsoid("MyEllipsoid") = fmeParameters
```

Similarly, the following code fragment uses the `datum` property to create a new datum called *`MyDatum`*, which uses the `MyEllipsoid` ellipsoid.

```
fmeParameters.Clear
fmeParameters.append ("ELLIPSOID")
fmeParameters.append ("MyEllipsoid")
fmeParameters.append ("USE")
fmeParameters.append ("3PARAMETER")
```

```
fmeParameters.append ("DELTA_X")
fmeParameters.append ("582.0")
fmeParameters.append ("DELTA_Y")
fmeParameters.append ("105.0")
fmeParameters.append ("DELTA_Z")
fmeParameters.append ("414.0")
fmeCoordSysMan.datum("MyDatum") = fmeParameters
```

Finally, the following code fragment uses `defineCoordSys` to create a new temporary coordinate system that references `MyDatum`.

```
fmeParameters.Clear
fmeParameters.append ("PROJ")
fmeParameters.append ("TM")
fmeParameters.append ("UNIT")
fmeParameters.append ("METER")
fmeParameters.append ("DT_NAME")
fmeParameters.append ("MyDatum")
fmeParameters.append ("PARM1")
fmeParameters.append ("9")
fmeParameters.append ("ORG_LAT")
fmeParameters.append ("0")
fmeParameters.append ("X_OFF")
fmeParameters.append ("3500000")
fmeParameters.append ("Y_OFF")
fmeParameters.append ("0")
fmeParameters.append ("MAP_SCL")
fmeParameters.append ("1")
fmeParameters.append ("SCL_RED")
fmeParameters.append ("1")
Call fmeCoordSysMan.defineCoordSys(fmeParameters, _
                                  sCoordSysName)
```

If your application needs to define a temporary coordinate system based on an OpenGIS Well-Known-Text (WKT) formatted string, use the `defineCoordSysFromOGCDef` method instead of `defineCoordSysParms`. The `defineCoordSysFromOGCDef` method supports several dialects of WKT (see API reference for details).

Creating Persistent Coordinate Systems

A file of local coordinate system definitions is automatically loaded and made available to each FME Objects session. This file is found in the `Reproject` subdirectory under the FME Objects installation directory and is called `LocalCoordSysDefs.fme`. It contains a series of `COORDINATE_SYSTEM_DEF`, `DATUM_DEF`, `ELLIPSOID_DEF`, and `UNIT_DEF` lines that define additional, application-specific coordinate systems.

Users may edit these files to add their own definitions. This process is described in detail in the *Coordinate Systems* section of the FME on-line help files.

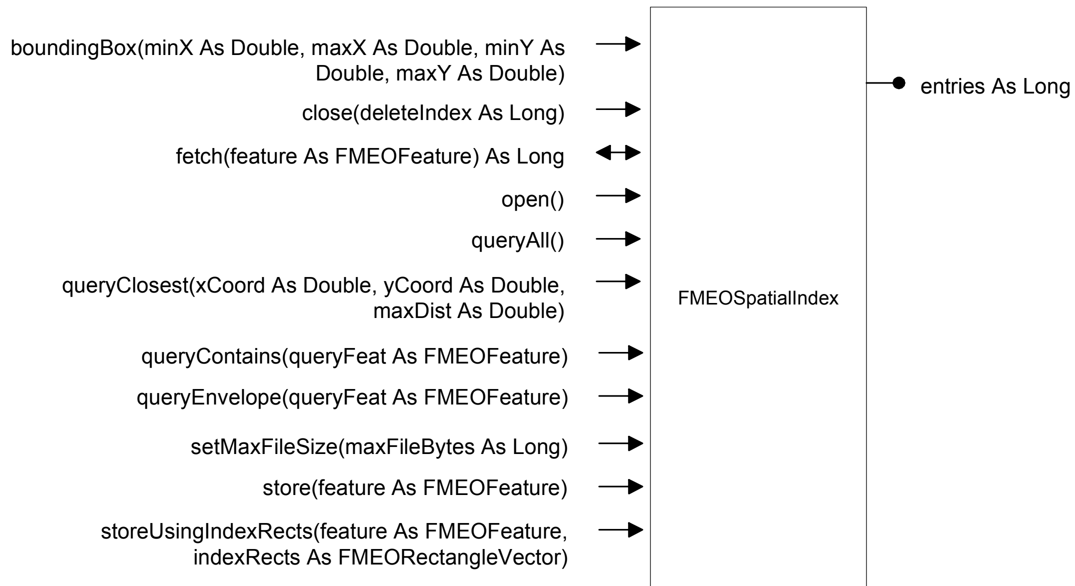
CHAPTER

8

Working with Spatial Indexes

Processing a spatial query involves the execution of complex and costly geometric operations. For example, imagine trying to find the single feature, out of tens of thousands of features, that is closest to a given point. A brute force solution consisting of a sequential scan of the entire domain requires a large number of disk accesses and an equally large number of evaluations of geometric predicates. The performance of a spatial query such as this can be greatly improved by creating an index on the collection of features to be searched. An index on spatial data is referred to as a spatial index. When using a spatial index, the collection of features for which the index is built is said to be indexed.

While the `FMEOReader` object offers some spatial indexing functionality in the form of the `setConstraints` method (see *Using Constraints* section of *Reading Features From a Dataset* chapter), the FME spatial index object, `FMEOSpatialIndex`, provides a sophisticated, easy to use spatial index based on the R+ Tree data structure. The methods and properties of `FMEOSpatialIndex` are illustrated below.



Methods and Properties of the FMEOSpatialIndex Object

If your application is reading from a format that provides a native spatial index (you can check using the `getProperties` method of `FMEOReader`), the simplest spatial query mechanism is the `setConstraints` method of `FMEOReader` to access the native index. However, even in situations where a native spatial index is available, your application may still benefit from using an FME Objects spatial index. For example, if there is a high network latency, a local spatial index can improve performance. Also, an FME Objects spatial index is a useful tool for creating a small working set of features from a large source dataset.

In this chapter, you will learn how to:

- create and open a spatial index
- index a set of features
- perform spatial queries on indexed features
- close a spatial index

Creating and Opening a Spatial Index

To create a new spatial index you must call the `createSpatialIndex` method of the `FMEOSession` object. Once created, a call to the `open` method prepares it for use. This following code fragment shows how to do this:

```
Dim fmeDirectives As FMEOStringArray
Dim sFileName As String
```

```
sFileName = sSpatialDir & "\SpatialIndex.ffs"
Set fmeDirectives = m_fmeSession.createStringArray
fmeDirectives.append ("PASSPHRASE")
fmeDirectives.append ("test")
fmeDirectives.append ("BYTE_ORDER")
fmeDirectives.append ("BIG_ENDIAN")
Set m_fmeSpatialIndex = m_fmeSession.createSpatialIndex( _
                      sFileName, "WRITE", fmeDirectives)
m_fmeSpatialIndex.open
```

The first parameter to `createSpatialIndex` is the name of the file used to store the indexed features, including the path. By convention, an `.ffs` (FME feature store) extension is used to name the feature data file. An additional file with the same base name as the data file but with an `.fsi` (FME spatial index) extension will be created by FME Objects to store the spatial index itself.

The second parameter is a string value that specifies the access mode, which can be either `WRITE` or `READ`. Use the `WRITE` access mode when you are creating a new index; use `READ` to access an existing index. When a spatial index is created for reading, the data file specified and the associated index file must exist, otherwise an exception will occur. In read mode, an attempt to add a feature to the index causes an exception to be raised. When a spatial index is created for writing, the data file does not need to exist. If the named file does exist, it is overwritten. In write mode, attempts to query the index return no features. It is not possible to change the access mode while a spatial index is open; to switch between read mode and write mode, an index must be closed and reopened.

The third and final parameter to the `createSpatialIndex` method is a string array containing a list of directives that control the spatial index creation. Spatial index creation directives are specified as a set of name and value pairs. The following directives are supported:

Directive Name	Description
`PASSPHRASE`	If a pass phrase directive is supplied when a spatial index is created, the same pass phrase must be supplied in future attempts to open the spatial index; otherwise the attempt will fail. If no pass phrase is supplied, access to the spatial index is unrestricted.
`BYTE_ORDER`	This directive is only relevant when a spatial index is created. It specifies the byte ordering to use for storing multi-byte data types in the index files. The value can be one of: `BIG_ENDIAN`, `LITTLE_ENDIAN`, or `NATIVE`. If `BIG_ENDIAN` is specified, the leftmost bytes are most significant. If `LITTLE_ENDIAN` is specified, the rightmost bytes are most significant. If a byte order is not specified or is specified as `NATIVE`, the default byte order of the host computer's architecture is used. **Note:** Both types of spatial index files can be read back on any machine, but there is a significant performance penalty in reading/writing spatial index files that do not match the byte order of the machine on which the process is running.

Indexing Features

The store method of the FMEOSpatialIndex object is used to add a single feature at a time to a spatial index that has been opened in write mode. This code sample adds each of the features in the m_fmeFeatureVector, one by one, into a spatial index.

```
Dim lEntries As Integer
Dim i As Integer

lEntries = m_fmeFeatureVector.entries
For i = 0 To lEntries - 1
Call m_fmeSpatialIndex.store(m_fmeFeatureVector.element(i))
Next i
```

The result is that the features in the m_fmeFeatureVector are indexed. Because the store method creates a deep copy of each feature indexed in the .ffs file, changes made to the original features in m_fmeFeatureVector have no effect on the spatial index.

Performing Spatial Queries

A spatial query is performed by calling one of the query methods provided by the FMEOSpatialIndex object. Methods that require a geometric primitive for comparison take an FMEOFeature object as a parameter. The query geometry is described by the geometry on this feature, which is called the query feature.

Method	Description
queryAll	Selects all the features that have been indexed.
queryClosest	Selects the closest feature to the specified point. If there are no features within the maximum distance, nothing is selected.
queryContains	Selects all the features that contain the query feature.
queryEnvelope	Selects all the features whose bounding box intersects the bounding box of the query feature.

To perform an envelope query on the features in a spatial index, you would use code like this:

```
Dim fmeQueryFeature As FMEOFeature
Set fmeQueryFeature = m_fmeSession.createFeature
Call fmeQueryFeature.addCoordinate(0, 0, 0)
Call fmeQueryFeature.addCoordinate(1, 1, 0)
Call m_fmeSpatialIndex.queryEnvelope(fmeQueryFeature)
```

FMEOSpatialIndex maintains an internal result set containing the features selected by a spatial query. This code fragment shows how to use the fetch method to iterate through the result set.

```
Dim fmeDataFeature As FMEOFeature
Dim bEnd As Boolean
Set fmeDataFeature = m_fmeSession.createFeature
bEnd = False
Do While bEnd = False
    bEnd = m_fmeSpatialIndex.fetch(fmeDataFeature)
Loop
```

When the end of the result set is reached, the next call to `fetch` will return the first feature again (assuming the result set is not empty).

Closing a Spatial Index

A spatial index is closed using the `close` method as demonstrated by the following code fragment:

```
Call m_fmeSpatialIndex.Close(True)
```

CHAPTER 9

Advanced Feature Processing

Chapter 3 introduces the FME feature object, `FMEOFeature`, and describes each of its components: attribution, geometry, and coordinate system. The basic techniques for manipulating feature attributes described in Chapter 3 are prerequisite reading for this chapter, which covers how to perform more sophisticated and powerful processing on features.

When discussing feature processing, it is useful to distinguish between feature-based processing and collection-based processing. Feature-based processing refers to the manipulation of individual features, independently of any other features. Collection-based processing refers to the manipulation of sets of interrelated features.

In this chapter, you will learn how to:

- create area topology (collection-based)
- dissolve polygons (collection based)
- buffer features (feature-based)
- generate a point inside a polygon (feature-based)
- perform an arbitrary function on a feature (feature-based)
- offset, rotate, and scale feature geometry (feature-based)
- manipulate aggregate features (feature and collection-based)
- manipulate donut features (feature and collection-based)
- apply a factory pipeline (feature and collection-based)

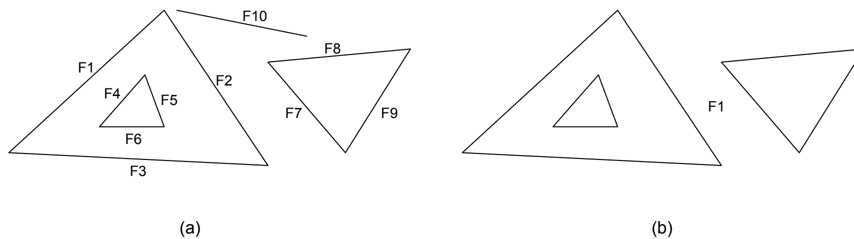
Creating Area Topology

Some GIS products store features using what is commonly referred to as a *spaghetti data model*. In a spaghetti model, the geometry of any feature is described independently of other features in a collection.

A common spatial data processing problem is to convert area features from a spaghetti model to a topological model. FME Objects can solve this problem if the lines are properly noded: the beginning and ending locations (or nodes) of

lines are the same for all lines that meet at that node. Creating area topology is an example of collection-based feature processing.

The figure below shows an initial set of ten line features (a), in a spaghetti model, and the resulting aggregate feature (b), in a topological model. Note that feature F10 is not used in the topological representation because it does not belong to an area feature.



Area Representation: (a) Spaghetti and (b) Topological Models

The `processFeatures` method of `FMEOFeature` provides a simple way to create area topology for properly noded lines. Given a set of line features that describe an area feature, `processFeatures` attempts to form one area feature that has either a polygon geometry, a donut geometry, an aggregate geometry of polygons, an aggregate geometry of donuts, or an aggregate geometry of polygons and donuts. The resulting area geometry is assigned to the feature on which the call is made.

As an example, assume `m_fmeFeatureVector` contains a set of line features of the same feature type with the same attribute values. The following code fragment shows how to use `processFeatures` to create an area geometry based on the set of line features contained in `m_fmeFeatureVector`. Since all the features have the same attribute values, the attributes for `fmeFeature` are cloned from the first feature in the list. The resulting geometry is assigned to `fmeFeature`.

```
Set fmeParams = m_fmeSession.createStringArray
Call fmeParams.append("fme_convert_to_area")
Call fmeParams.append("fme_drop_line")
Set fmeFeature = m_fmeSession.createFeature
Call m_fmeFeatureVector.element(0).cloneAttributes _
                                         (fmeFeature)
Call fmeFeature.processFeatures(m_fmeFeatureVector, _
                                             fmeParams)
```

WARNING When `processFeatures` is called, the application relinquishes all control over the features in the feature vector. The passed in features are deleted. If your application requires the original features for further processing, use `FMEOFeature's` clone method to make a deep copy of each feature before calling `processFeatures`.

By default, unused line features (such as F10 in the diagram) are returned in the feature array `m_fmeFeatureVector`. If there are no unused line features then the feature array is returned empty. If you do not want unused lines to be returned, append `fme_drop_line` to `fmeParams`.

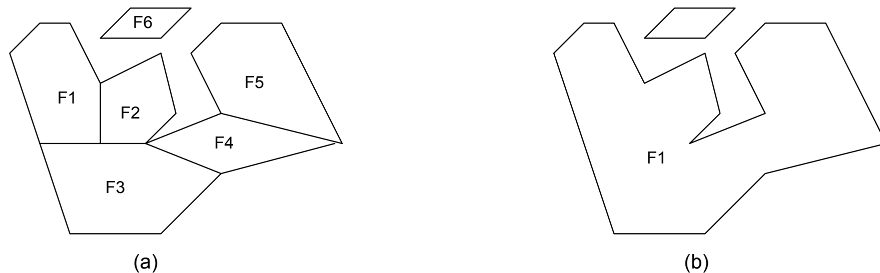
Note For applications designed to handle very large datasets, the `processFeaturesOnDisk` has the same semantics as `processFeatures` but works with a disk-based collection object, `FMEOFeatureVectorOnDisk` (see Chapter 10 for a discussion of collection objects).

Dissolving Polygons

The process of dissolving polygons involves the removal of shared edges between adjacent polygons. This powerful operation is used to simplify datasets by joining all adjacent polygons with a common attribute value into a single polygon. Dissolving polygons has many practical applications. For example, a developer working with the lot descriptions for a municipality dissolves all of those that are zoned for commercial use to determine where the development opportunities are.

The `processFeatures` method of FMEOFeature provides a simple way to dissolve polygons. Given a set of properly noded polygon features, it attempts to form one area feature, with either a polygon geometry, a donut geometry, or an aggregate of polygons or donuts. Dissolving polygons is another example of collection-based feature processing.

As an example, the figure below shows an initial set of six polygon features (a) and the resulting dissolved feature (b).



Dissolving Polygons: (a) Original and (b) Dissolved

As an example, assume m_fmeFeatureVector contains a set of polygon features of the same feature type with the same attribute values. The following code fragment shows how to use processFeatures to dissolve the features contained in m_fmeFeatureVector. Since all the features have the same attribute values, the attributes for fmeFeature are cloned from the first feature in the list. The resulting geometry is assigned to fmeFeature.

```
Set fmeParams = m_fmeSession.createStringArray
Call fmeParams.append("fme_polygon_dissolve")
Call fmeParams.append("fme_drop_line")
Set fmeFeature = m_fmeSession.createFeature
Call m_fmeFeatureVector.element(0).cloneAttributes _
                                        (fmeFeature)
Call fmeFeature.processFeatures(m_fmeFeatureVector, _
                                            fmeParams)
```

WARNING When processFeatures is called, the application relinquishes all control over the features in the feature vector. The passed in features are deleted. If your application requires the original features for further processing, use FMEOFeature's clone method to make a deep copy of each feature before calling processFeatures.

By default, interior line features are returned in the feature array, m_fmeFeatureVector. If you do not want interior lines to be returned, append fme_drop_line to the fmeParams string array.

Buffering Features

A buffer zone, or simply a buffer, is a polygon enclosing an area within a specified distance from a point, line or polygon feature. Buffers are useful for proximity analysis, for example, to find all stream segments within 300 feet of a proposed logging area. Buffering is an example of feature-based geometric processing.

The buffer method of FMEOFeature provides a simple way to create a buffer around a feature. It replaces the geometry of the feature with a new area geometry representing the buffer. The following code fragment creates a buffer feature for each feature in the m_fmeFeatureVector collection, and appends the new feature to the collection. The result is that m_fmeFeatureVector contains both the original features and the buffer features. Each buffer feature is 0.1 ground units from the original feature, sampled at every 5 degrees. The attributes for a buffer feature are cloned from the original feature.

```
Dim fmeFeature As FMEOFeature
Dim i As Integer
Dim lCount As Integer

lCount = m_fmeFeatureVector.entries
For i = 0 To lCount - 1
```

```
    Set fmeFeature = m_fmeSession.createFeature
    Call m_fmeFeatureVector.element(i).Clone(fmeFeature)
    Call fmeFeature.buffer(0.1, 5)
    fmeFeature.attribute("fme_type") = "fme_area"
    Call m_fmeFeatureVector.append(fmeFeature)
Next i
```

The buffer method of FMEOFeature uses the `@Buffer` FME function; see the *Functions, Factories, and Transformers* manual for more details.

Generating a Point Inside a Polygon

Given a polygon feature, your application may need to generate a point that lies inside the polygon. The `generatePointInPolygon` feature-based geometric processing method of `FMEOFeature` does exactly that.

The following code fragment creates a point feature for each polygon feature in the `m_fmeFeatureVector` collection and appends the new feature to the collection. The result is that `m_fmeFeatureVector` contains both the original polygon features and the newpoint features. The attributes for a point feature are cloned from the polygon feature.

```
Dim fmeFeature As FMEOFeature
Dim sFeatureType As String
Dim i As Integer
Dim lCount As Integer
Dim X As Double
Dim Y As Double
Dim z As Double

sFeatureType = m_fmeFeatureVector.element(0).featureType
lCount = m_fmeFeatureVector.entries
For i = 0 To lCount - 1
    Set fmeFeature = m_fmeSession.createFeature
    Call m_fmeFeatureVector.element(i). _
            generatePointInPolygon(False, X, Y, z)
    Call fmeFeature.addCoordinate(X, Y, z)
    fmeFeature.featureType = sFeatureType
    fmeFeature.attribute("fme_type") = "fme_point"
    fmeFeature.GeometryType = foPoint
    Call m_fmeFeatureVector.append(fmeFeature)
Next i
```

The first parameter to `generatePointInPolygon` is the pretty flag. If the pretty flag is set to True, `generatePointInPolygon` will attempt to find a central position for the generated inside point, which may take longer to compute. If the feature is three-dimensional, the `z` value is calculated to be the average of all points on the feature.

Performing an Arbitrary Function on a Feature

FME functions provide a flexible means of applying specific algorithmic operations on features. There are two kinds of functions: attribute functions that are used to compute the value for an attribute; and feature functions that operate on a complete feature. Feature functions modify the feature's attributes, the feature's geometry, or both. If your application requires use of FME functions, you should start by reading the *FME Functions* chapter in the *FME Foundation* manual, which provides fundamental information about how FME functions work. The *FME Functions, Factories and Transformers* manual is also a useful reference.

The `performFunction` method of the `FMEOFeature` object is a powerful feature-based processing mechanism that can be used to invoke any FME function. For example, the following line of code invokes the `@Log` feature function to log a feature:

```
Call fmeFeature.performFunction("@Log(Data Feature:,-1)", _
                                                 sResult)
```

> **WARNING** `performFunction` reports an error if there is white space after any of the commas that separate the parameters of the FME function call.

When `performFunction` is used to invoke a feature function the result parameter returns an empty string. On the other hand, when an attribute function is invoked, the result parameter returns the function output. For example, the following code fragment adds a `lot_area` attribute to a feature.

```
Call fmeFeature.performFunction("@Area(1000)", sResult)
fmeFeature.real32Attribute("lot_area") = sResult
```

Offsetting, Rotating, and Scaling Features

A feature is offset to a new position by translating each of its coordinates to a new location. The offset is specified by the translations applied in the `x, y,` and optionally `z` dimensions. The `offset` method of the `FMEOFeature` object is used to offset a feature's geometry.

Another useful geometric transformation is the rotation of a feature around a specified pivot point. The `rotate2D` method of the `FMEOFeature` object is used to rotate a feature's geometry counterclockwise around a pivot point.

In general, you can change the size of a feature's geometry by multiplying the distances between its points by a factor. This operation is called scaling. The `scale` method of the `FMEOFeature` object is used to scale a feature's geometry.

Offsetting, rotating, and scaling are all examples of feature-based geometric processing. This following code fragment offsets, rotates, and scales a feature.

```
Call m_fmeFeatureVector.element(i).offset(dXOffset, _
                                    dYOffset, dZOffset)
Call m_fmeFeatureVector.element(i).rotate2D(dXCenter, _
                                    dYCenter, dAngle)
Call m_fmeFeatureVector.element(i).scale(dXFactor, _
                                    dYFactor, dZFactor)
```

Manipulating Aggregate Features

The *Interpreting Aggregate Geometries* section in Chapter 3 introduces the concept of a feature with an aggregate, or multi-part, geometry. The sample code in that section illustrates the use of `FMEOFeature's splitAggregate` method to obtain a list of the non-aggregate geometries that comprise an aggregate.

Note The `splitAggregate` method of `FMEOFeature` does not clear the input feature vector, it only appends to it.

If your application needs to go the other way, that is, to create an aggregate instead of dismantling one, there are two methods available: `chopUp`, and `buildAggregateFeature`.

The `chopUp` feature-based processing method is used to turn an existing non-aggregate area or line feature into an aggregate. If the number of vertices in the feature's geometry is greater than a specified threshold, the feature is chopped up into an aggregate where each member of the aggregate has fewer than the threshold number of vertices. The minimum vertex threshold is ten. For an area feature, `chopUp` subdivides the area so that no area piece has more than the number of vertices. A line feature is broken into segments that meet the vertex threshold. The `chopUp` method is invoked like this:

```
Call m_fmeFeatureVector.element(i).chopUp(20)
```

The `buildAggregateFeature` collection-based processing method uses the geometries from a set of input features to create an aggregate geometry. The `buildAggregateFeature` method is invoked like this:

```
Call fmeFeature.BuildAggregateFeature(m_fmeFeatureVector)
```

Manipulating Donut Features

The *Interpreting Donut Geometries* section in Chapter 3 introduces the concept of a feature with a donut geometry. The sample code in that section illustrates

the use of FMEOFeature's `getDonutParts` method to obtain a list of the polygon geometries that comprise a donut.

> **Note** The `getDonutParts` method of `FMEOFeature` does not clear the input feature vector, it only appends to it.

If your application needs to go the other way, that is, to create a donut instead of dismantling one, it can use the `makeDonuts` method.

The `makeDonuts` method constructs one or more donut features from a set of polygonal features. If `makeDonuts` is able to construct more than one donut, the resulting geometry is an aggregate. A single donut geometry contains only polygons that do not overlap each other or share common edges; all of the inner polygons are disjoint and fully contained within the outer shell polygon. The `makeDonuts` method is invoked like this:

```
Call fmeFeature.makeDonuts(m_fmeFeatureVector, False)
```

The second parameter to `makeDonuts` is the `keepHoles` flag. If it is set to `True`, `makeDonuts` will return any polygons that were used as holes in the feature vector.

The `outerShell` method is a convenience method that replaces the geometry of a donut feature with a polygon feature that describes the donut's outer shell. If `outerShell` is called on a non-donut feature, it will have no effect on the feature's geometry. The `outerShell` method is invoked like this:

```
Call fmeFeature.outerShell
```

If you are working with a format that has specific orientation requirements (that is, left-hand rule or right-hand rule) for the coordinates of donut parts, your application can make use of the `getOrientation` and `setOrientation` convenience methods to control the order in which coordinates are stored.

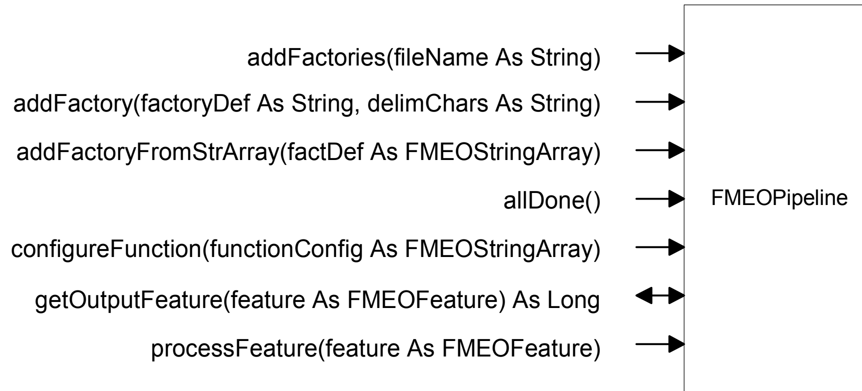
Applying a Factory Pipeline

FME feature factories can be combined with feature and attribute value functions, and chained together into factory pipelines. Factory pipelines enable FME Objects to perform sophisticated feature-based and collection-based processing tasks. In fact, all the feature processing applications covered so far in this chapter, which all use built-in `FMEOFeature` methods, can also be performed using feature factories instead. Using factory pipelines instead of the built-in methods requires more effort to set up, but is more powerful and more easily modified and extended as requirements evolve.

If you are planning to make use of factory pipelines in your application, it is recommended that you start by reading the *FME Factories* chapter in the *FME Foundation* manual, which provides fundamental information about how FME

factories work. The *FME Functions, Factories and Transformers* manual is also a useful reference.

The `FMEOPipeline` object provides an interface for constructing and using a factory pipeline. The methods and properties of `FMEOPipeline` are illustrated below.



Methods and Properties of the FMEOPipeline Object

Using a factory pipeline involves the following steps:

- creating and defining the pipeline
- inserting features into the pipeline
- retrieving features from the pipeline

The rest of this section covers each of these steps in detail. The last section shows how to put it all together into a configurable framework for executing arbitrary factory pipelines.

Creating and Defining a Pipeline

To create a new factory pipeline, you must call the `createFactoryPipeline` method of the `FMEOSession` object. This following code fragment shows how to do this:

```
Set fmePipeline = m_fmeSession.createFactoryPipeline( _
                                          "Test", fmeDirectives)
```

The first parameter passed to the `createFactoryPipeline` method is the pipeline name. The second parameter allows you to specify configuration lines

for the factories and function in the pipeline, as a set of name value pairs. The name of the configuration keyword must be preceded by "--".

> **Note** The configureFunction method of `FMEOPipeline` can also be used to specify configuration lines for functions such as `@Lookup`, `@Relate`, and `@Sql`, which require configuration information to operate.

Once a pipeline has been created, there are three ways of specifying the feature factories that it contains:

- Put all the factories in a text file and call the `addFactories` method.
- Put each individual factory into a string and call the `addFactory` method for each.
- Put each individual factory into an FME string array and call the `addFactoryFromStringArray` for each.

You should always begin your pipeline development using the `addFactories` method – this is the best approach for several reasons. First, having the entire factory pipeline together in one text file is less error-prone to develop and debug. Pipeline files follow standard FME mapping file syntax (including continuation characters). Also, you can test a factory pipeline file using the FME Universal Viewer to apply the pipeline to an existing dataset (see Apply Pipeline... context menu on a View).

> **Note** When you are using the FME Universal Viewer for testing, be aware that if a View consists of more than one Dataset, each individual Dataset will be processed by a separate pipeline instance.

Once the functionality of your pipeline has been designed, implemented and tested, the `addFactory` and `addFactoryFromStringArray` methods are useful if you want the pipeline code to be compiled with the rest of your application code so it is directly visible to the end user. This approach has the benefit of protecting your intellectual property, and may also help to simplify the installation script for your application. Migrating from a file-based pipeline to either the `addFactory` or the `addFactoryFromStringArray` method is straight forward. The following code fragment shows how to use the `addFactory` method.

```
sFactory = "FACTORY_DEF * DonutHoleFactory "
sFactory = sFactory & "INPUT FEATURE_TYPE * "
sFactory = sFactory & "fme_geometry fme_donut "
sFactory = sFactory & "OUTPUT OUTERSHELL FEATURE_TYPE * "
sFactory = sFactory & "OUTPUT HOLE FEATURE_TYPE *"
Call fmePipeline.addFactory(sFactory, " ")
```

WARNING When the space character is used as delimiter for addFactory (such as above), function calls must not contain any spaces in the parameter list. If a factory includes references to attribute values containing spaces, you will need to choose a character other than space as delimiter (for example, "|" or ";").

The last section in this chapter illustrates the use of the addFactories method.

Inserting Features into a Pipeline

Once a factory pipeline has been created and defined, it is ready to start accepting features. To add a feature to a factory pipeline, your application must call FMEOPipeline's processFeature method like this:

```
Call fmePipeline.processFeature( _
       m_fmeFeatureVector.element(i))
```

The processFeature method has the side effect of resetting the passed-in feature, which is returned void of all attributes and geometry. If your application needs to retain a copy of original feature object, you can use FMEOFeature's clone method to make a copy before calling processFeature.

Retrieving Features from a Pipeline

Your application can retrieve and process a feature once the feature has passed through the pipeline. Features that are not processed by any factories in the pipeline, pass untouched through the pipeline.

If the pipeline contains only feature-based processing factories, any resulting features are available immediately after processFeature returns control to the application. To retrieve a feature, use getOutputFeature like this:

```
Call fmePipeline.processFeature( _
       m_fmeFeatureVector.element(i))
bEnd = fmePipeline.getOutputFeature(fmeFeature)
```

When the last feature has been read, getOutputFeature returns True.

When a pipeline contains one or more collection-based processing factories, such as TopologyFactory, PolygonFactory, or SortingFactory, your application is responsible for notifying the pipeline once the last feature has been inserted using FMEOPipeline's allDone method. Any features that were blocked in the pipeline will be available for retrieval immediately after allDone returns control to the application. If getOutputFeature is called before allDone, and there are no features ready for output, getOutputFeature will return True even though there may still be features in the pipeline.

Note A pipeline cannot be re-used after the allDone method is called. In other words, it is not valid to call addFeature on a pipeline after allDone.

Putting it all Together

`ApplyPipeline` illustrates how to combine everything covered so far in this section to create a procedure that processes the set of features in `m_fmeFeatureVector` using a file-based factory pipeline identified by `sPipelineFile`.

```
Sub ApplyPipeline(sPipelineFile As String)

    Dim i As Integer
    Dim lCount As Integer
    Dim bEnd As Boolean
    Dim fmePipeline As FMEOPipeline
    Dim fmeFeature As FMEOFeature
    Dim fmeDirectives As FMEOStringArray

    Set fmePipeline = m_fmeSession.createFactoryPipeline( _
                                      "Test", fmeDirectives)
    Call fmePipeline.addFactories(sPipelineFile)

    lCount = m_fmeFeatureVector.entries
    For i = 0 To lCount - 1
        Call fmePipeline.processFeature( _
               m_fmeFeatureVector.element(i))
    Next i
    Call fmePipeline.allDone
    m_fmeFeatureVector.Clear
    bEnd = False
    Do While bEnd = False
        Set fmeFeature = m_fmeSession.createFeature
        bEnd = fmePipeline.getOutputFeature(fmeFeature)
        If bEnd = False Then
            Call m_fmeFeatureVector.append(fmeFeature)
        End If
    Loop
End Sub
```

If you wanted to create area topology using this approach, it would suffice to call `ApplyPipeline` with the following pipeline file:

```
FACTORY_DEF * PolygonFactory                              \
  FACTORY_NAME POLYGONBUILDER                             \
  INPUT FEATURE_TYPE *                                    \
     fme_geometry fme_line                                \
     fType @FeatureType()                                 \
  GROUP_BY fType                                          \
  VERTEX_NODED                                            \
  OUTPUT POLYGON FEATURE_TYPE *                           \
     @FeatureType(&fType)                                 \
     @RemoveAttributes(fType)                             \
     fme_type fme_area
```

To dissolve polygons, the following pipeline file can be used:

```
FACTORY_DEF * PolygonDissolveFactory                        \
  FACTORY_NAME POLYGONDISSOLVER                             \
  INPUT FEATURE_TYPE *                                      \
     fme_geometry fme_polygon                               \
     fType @FeatureType()                                   \
  GROUP_BY fType                                            \
  OUTPUT POLYGON FEATURE_TYPE *                             \
     @FeatureType(&fType)                                   \
     @RemoveAttributes(fType)                               \
     fme_type fme_area
```

The pipeline files for buffering features and generating points in polygons are left to the reader as an exercise.

CHAPTER 10

Working with Collections

The FME Objects API contains the following collection objects: `FMEOStringArray, FMEOFeatureVector` and `FMEORectangleVector`. These objects are primarily intended for moving data in and out of the FME Objects API, either as parameters or as return values. They are not designed to replace general-purpose collection objects such as Visual Basic Collections.

If you are integrating FME Objects into an application that is already using different collection objects for processing, you can write procedures to convert from your existing collections to FME Objects collections as needed. For example, the `FeatureVectorToCollection` procedure copies an `FMEOFeatureVector` object to a VBA Collection object.

```
Sub FeatureVectorToCollection(ByRef colFeatures As Collection)

    Dim i As Integer
    Dim lCount As Integer

    lCount = m_fmeFeatureVector.entries
    For i = 0 To lCount - 1
        If i = 0 Then
           Call colFeatures.Add(m_fmeFeatureVector.element(i))
        Else
            Call colFeatures.Add( _
                        m_fmeFeatureVector.element(i), , , i)
        End If
    Next i
End Sub
```

WARNING The `If` statement inside the `For` loop is required to preserve the order of the features in the feature vector. If order is not important, you can improve performance by removing it.

This example brings up an important difference between FME Objects collection objects and Visual Basic collection objects: FME Objects uses 0-

based indexing while Visual Basic uses 1-based indexing. You should keep this in mind when you are working with FME Objects collections.

In the rest of this chapter, you'll learn how to:

- use the rectangle and feature vector collections
- use the string array collection

Using the Vector Collections

The `FMEOFeatureVector` collection and `FMEORectangleVector` collection have basically the same semantics, the only difference is that the first stores `FMEOFeature` objects while the second stores `FMEORectangle` objects. The methods and properties of `FMEORectangleVector` are illustrated below.

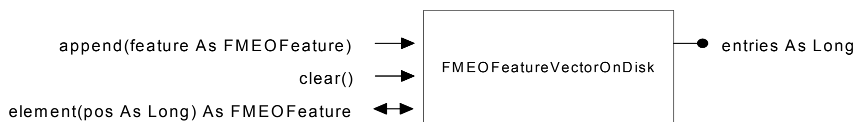
append(rectangle As FMEORectangle)
clear()
contains(rectangle As FMEORectangle) As Long
insertAt(index As Long, rectangle As FMEORectangle)
removeLast() As FMEORectangle
retrieve(index As Long) As FMEORectangle
FMEORectangleVector
entries As Long

Methods and Properties of the FMEORectangleVector Object

The following table describes each of these methods and properties:

Method	Description
`append`	Adds a rectangle to the end of the vector.
`clear`	Removes all rectangles from the vector.
`contains`	If the vector contains the specified rectangle, this method returns `True,` otherwise it returns `False.`
`entries`	The total number of rectangles in the vector; zero if the vector is empty.
`insertAt`	Adds a rectangle to the vector at the position specified by index. The index must be between zero and the length of the vector (use the entries property to verify that it is within range). The rectangle previously at index is moved to index+1, and so on. If an invalid index is specified, an exception is raised.
`removeLast`	Removes the last rectangle in the vector and returns it. If the vector is empty, an exception is raised.
`retrieve`	Returns the rectangle at position specified by index. The index must be between zero and the length of the vector, less one (use the entries property method to verify that it is within range). If an invalid index is specified, an exception is raised.

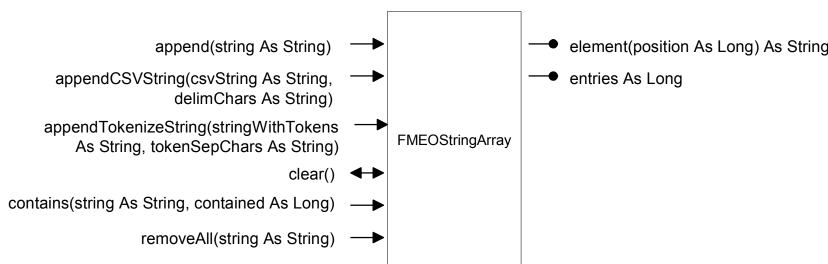
If your application is designed to work with very large datasets, you should consider using the `FMEOFeatureVectorOnDisk` collection instead of `FMEFeatureVector`. By storing features on disk instead of in memory, your application can handle large data volumes without being limited by the amount of virtual memory on the computers on which it operates. The methods and properties of `FMEOFeatureVectorOnDisk` are illustrated below.



Methods and Properties of the FMEOFeatureVectorOnDisk Object

Using the String Array Collection

The `FMEOStringArray` collection object stores String objects. Many FME Objects methods and properties take string arrays as parameters, both input and output. The methods and properties of `FMEOStringArray` are illustrated below.



Methods and Properties of the FMEOStringArray Object

The following table describes each of these methods and properties:

Method	Description
`append`	Adds a string to the end of the array.
`appendCSVString`	Splits the specified string into elements based on the delimiter character, and appends the entries to the end of the array. If the specified delimiter string contains more than one character, the first character is used as the delimiter.
`appendTokenizeString`	Splits the specified string into tokens, where each token is separated by one or more occurrences of the token string. This method is similar to appendCSVString, however in this version no null entries are added to the array (see the code fragment that follows for details).

Method	Description
clear	Removes all strings from the array.
contains	If the vector contains the specified string, this method returns True, otherwise it returns False.
element	The string element at the specified index. The index must be between zero and the length of the array, less one (use the entries property to verify that it is within range). If an invalid index is specified, an exception is raised.
entries	The total number of strings in the array (zero if the array is empty).
removeAll	Removes all strings from the array.

The following code fragment would result in an array containing nine elements: string1, "string 2", aaa, bbb, ccc, <NULL>, eee, xxx, and yyyy.

```
Set fmeArray = m_fmeSession.createStringArray
sDelimiter = ";;"
sToken = "   "

s1 = "string1"
s2 = """string 2"""
s3 = "aaa;bbb;ccc;;eee"
s4 = "xxx      yyyy      "
Call fmeArray.append(s1)
Call fmeArray.append(s2)
Call fmeArray.appendCSVString(s3, sDelimiter)
Call fmeArray.appendTokenizeString(s4, sToken)
```

A common operation on a string array is to determine the index of an element. The `GetIndex` procedure implements this functionality.

```
Function GetIndex(fmeStringArray As FMEOStringArray, _
                                    element As String) As Long

    Dim i As Integer
    Dim lCount As Integer

    GetIndex = -1
    lCount = fmeStringArray.entries
    For i = 0 To lCount - 1
        If fmeStringArray.element(i) = element Then
            GetIndex = i
            Exit For
        End If
    Next i
End Function
```

If there are duplicate entries, `GetIndex` returns the index of the first one found.

CHAPTER

11

Troubleshooting Tips

Chapter 2 discusses how to build error handling into your application and how to use the FMEOLogFile object to log information to a text file. Error dialogs and log files are simple and informative diagnostic tools for troubleshooting your application, however sometimes they do not provide sufficient information to locate a problem. This chapter provides additional information to help you identify and recover from difficult errors.

Problems Creating and Initializing a Session

The first thing to check if your application cannot successfully create and initialize a session is your FME Objects licensing. An easy way to ensure you have a valid license is to run the FME Universal Viewer. If the FME Viewer reports a licensing problem, contact Safe Software to obtain a valid license.

If you are running a version older than FME 2002 SR1, then an upgrade to the latest may fix the problem.

The next thing to check is whether there is a problem in the FME Objects installation. For the purpose of this discussion, let's assume the folder where the FME is installed is called InstallDirectory. If any of the following steps result in changes to your system configuration, reinstall FME Objects and restart your application.

1 Check the system's PATH by selecting Start > Run > "cmd", and then typing path. Make sure InstallDirectory is in the path. If there are entries in the path for old versions of the FME, delete them.

2 Check FME_HOME in the system Registry by Start > Run > "RegEdit", and navigate to HKEY_LOCAL_MACHINE\SOFTWARE\Safe Software Inc.\Feature Manipulation Engine. Make sure FME_HOME is set to InstallDirectory. IF there are FME_HOME entries for old versions of the FME, delete them.

3 Check the registration of FME Objects COM components by starting Visual Basic, opening a sample project and selecting Project > References. Select FMEObjects 1.0 Type Library and make sure that the location of

FMEObjects.dll is the same as InstallDirectory. If this location is correct, go back to RegEdit and search FMEOSession. If it is found, click on the InprocServer32 entry to ensure that the "(Default)" entry has a value of "InstallDirectory\FMEObjects.dll".

Run-Time Exceptions

When you encounter a run-time exception while developing your application, it's usually easy track down the cause using one of the standard debugging tools at your disposal: error handlers, log files, and code debuggers. On the other hand, run-time errors encountered by a deployed application are much more difficult to track down. FME Objects has a built-in run-time diagnostic facility to assist you in this situation.

If your deployed application is generating an unhandled run-time exception, the following environment variables may help you to track down the source of the problem.

Variable Name	Description
FME_LOG_FILENAME	Specifies the name of the file that FME Objects will use for logging.
FME_LOG_APPEND	If the log file already exists, this flag specifies whether FME Objects should append to it or overwrite it. Valid choices are YES and NO. The default is YES.
FME_LOG_MAX_FEATURES	Specifies the maximum number of features that will be logged. Use this environment variable to prevent the accidental creation of extremely large log files. The default is unlimited.
FME_DEBUG	Specifies additional levels of diagnostic information to log. Any of the following values are valid: DUMP_CONFIG - Log the configuration object when a reader or writer is created. DETAILED_DEBUG -Log detailed processing information. BADNEWS - Log the location and message of any internal software exception. FME_STACK_TRACE - Log execution trace information. To specify a combination of one or more of these values, separate them with a comma.

You will need to restart your application after setting or changing the value of any of these environment variables.