Custom Data Access with MapObjects Java Edition

Next Generation Command and Control System (NGCCS)
Tactical Operations Center (TOC) 3-D

Concurrent Technologies Corporation
Derek Sedlmyer
James Taylor
05/24/2005

Abstract: *Integrating data has always been a challenge with every IT application, especially GIS. GIS applications have many different data formats that are not ubiquitous among all GIS applications. Couple this with the DoD, which also has many data formats for data messaging and storage, and accessing data in a GIS quickly turns into a daunting task. MapObjects Java Edition was designed to allow developers to integrate custom data via their Content Provider Framework. ESRI developed several Content Providers to access external data such as ArcIMS servers, ArcSDE, shapefiles, CAD files, and various raster and vector product files. This presentation will provide an in-depth tutorial on how to develop a Content Provider, show the Content Provider in action, and share lessons learned in implementation.*

Introduction

A Content Provider makes it easy to adapt MapObjects Java Edition to a custom data source. The Content Provider can talk directly to a custom data source, query data from a database, or listen for data being pushed to it. The Content Provider interfaces within MapObjects Java Edition give you the flexibility to ingest data in any manner that is desirable.

The Content Provider interfaces also separate the data access and conversion logic from the application logic. This allows the data access code to operate independently from the application code, since the application code and data access code are no longer tightly integrated together. With this separation, complex business logic for data access and conversion can be contained completely within the Content Provider. The Content Provider can then be tested and validated separately from the application itself.

Once a Content Provider is developed, it can be used in any MapObjects Java Edition application. This application can be a desktop application, a web application, or an applet. This promotes reuse amount many different applications, and also different types of applications.

This paper will act as a guide for a developer that wishes for MapObjects Java Edition to ingest custom data. This custom data can be in any format, provided that APIs exist to read it. This will first provide a background on the MapObjects Java Edition Content

Provider framework, and then will go on to describe a sample implementation of a Content Provider to read an arbitrary XML file that contains geospatial data.

Background

MapObjects Java Edition is a pure Java Mapping API. MapObjects Java Edition can be used in desktop application, Java applets, as well as server-side environments.

MapObjects Java Edition has a framework for accessing external data sources called the Content Provider framework. The framework abstracts away data access so that MapObjects Java Edition can access potentially any GIS data source. Figure 1 contains the class diagram of the Content Provider framework in MapObjects Java Edition. This class diagram was provided by ESRI and found in the API documentation for MapObjects Java Edition.

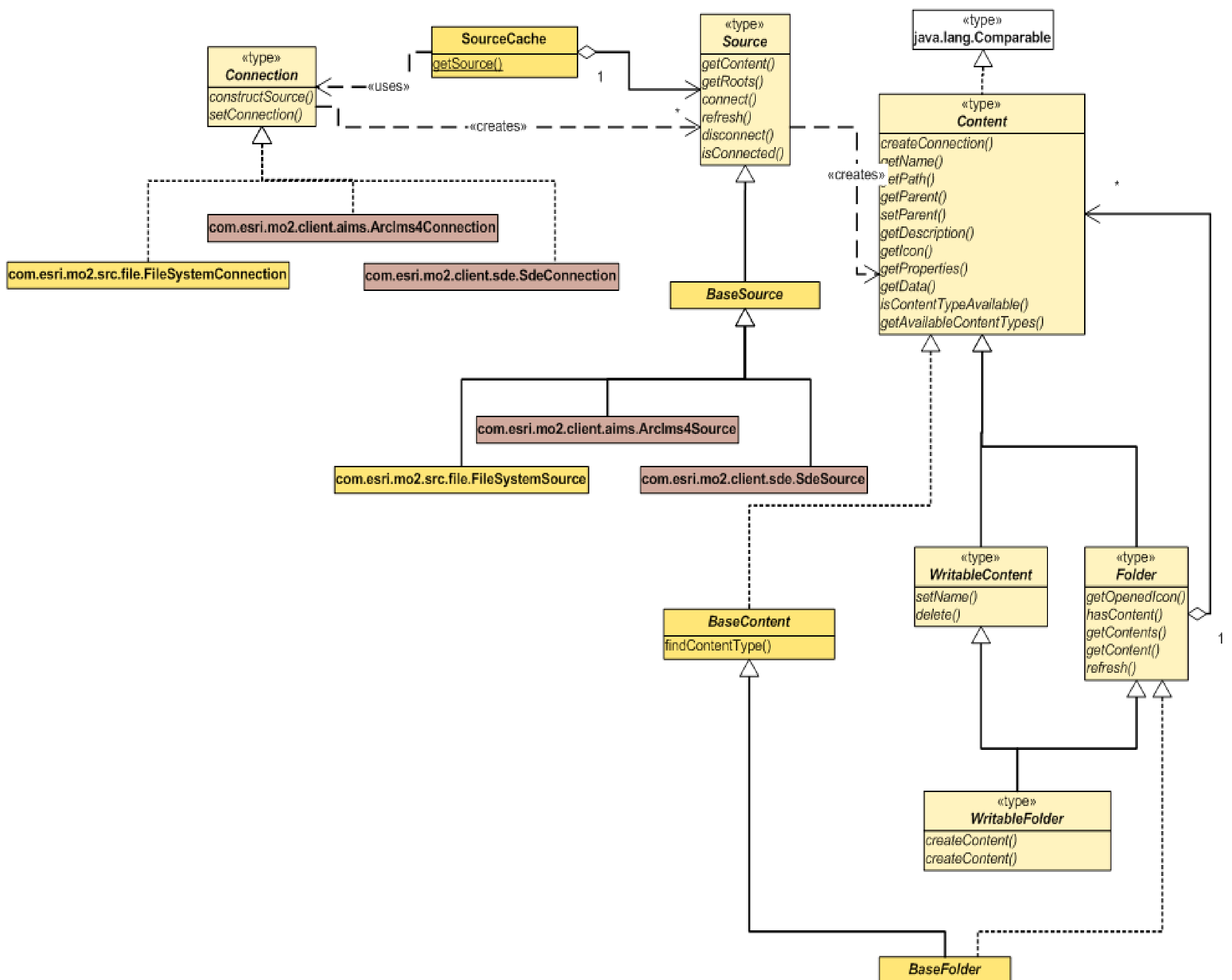

Figure 1 Content Provider Framework Class Diagram

The Content Provider framework typically produces Layer objects so that they can be added to a Map object for display. The Layer objects are exposed from classes that implement the Content interface. Figure 2 shows a class diagram of how the Content Provider, Layer Sources, and Layers are interfaced together. This class diagram was provided by ESRI and found in the API documentation for MapObjects Java Edition.

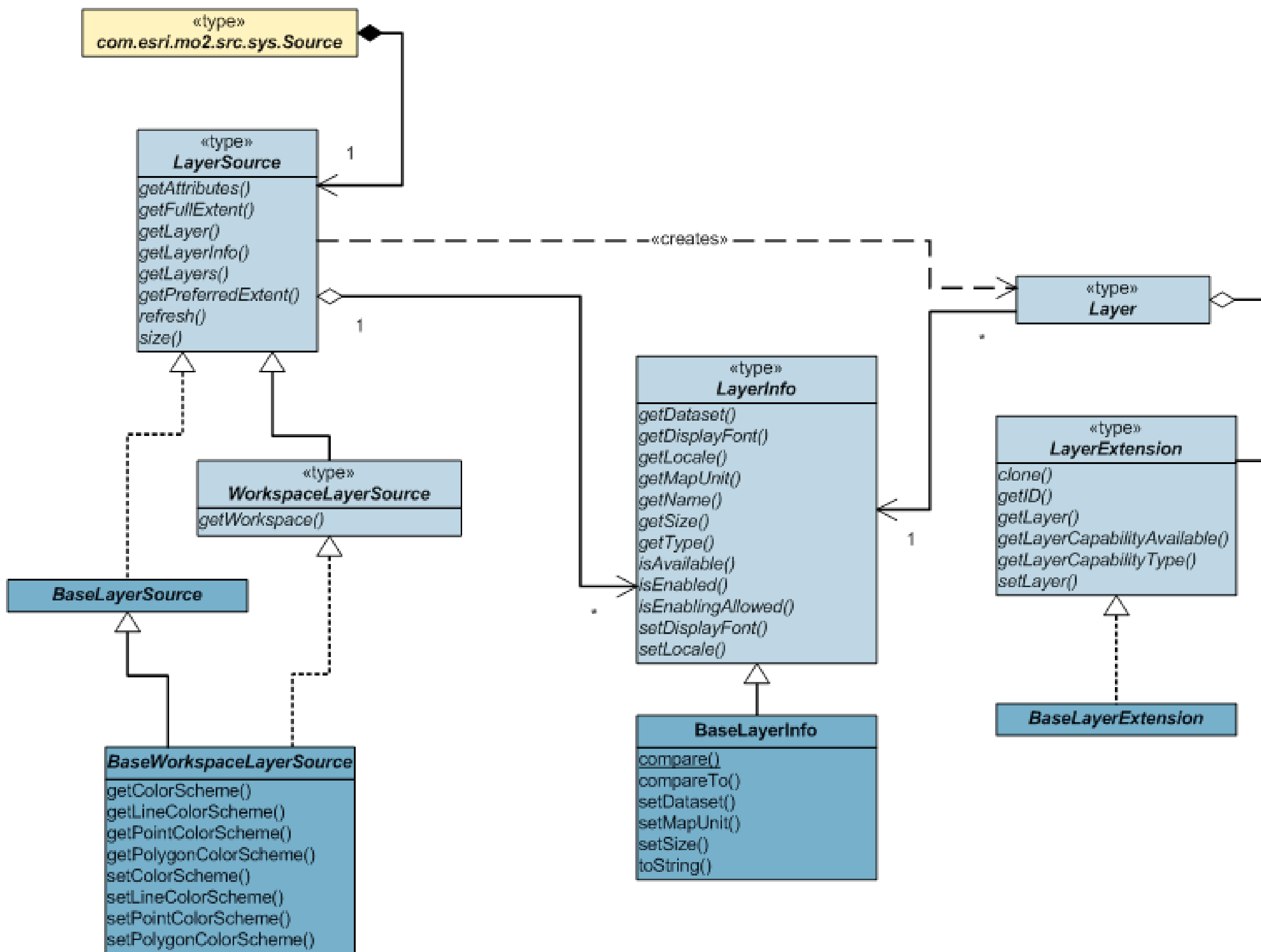

Figure 2 Content Provider/Layer Source/Layer Interactions

MapObjects Java Edition also has various Layer implementations to hold and access a variety of different data. For instance, a FeatureLayer is intended to access vector data; an ImageLayer accesses raster data; and an ImageServerLayer accesses an image service on an ArcIMS server. Figure 3 shows a class diagram of the different Layer implementations. This class diagram was provided by ESRI and found in the API documentation for MapObjects Java Edition.

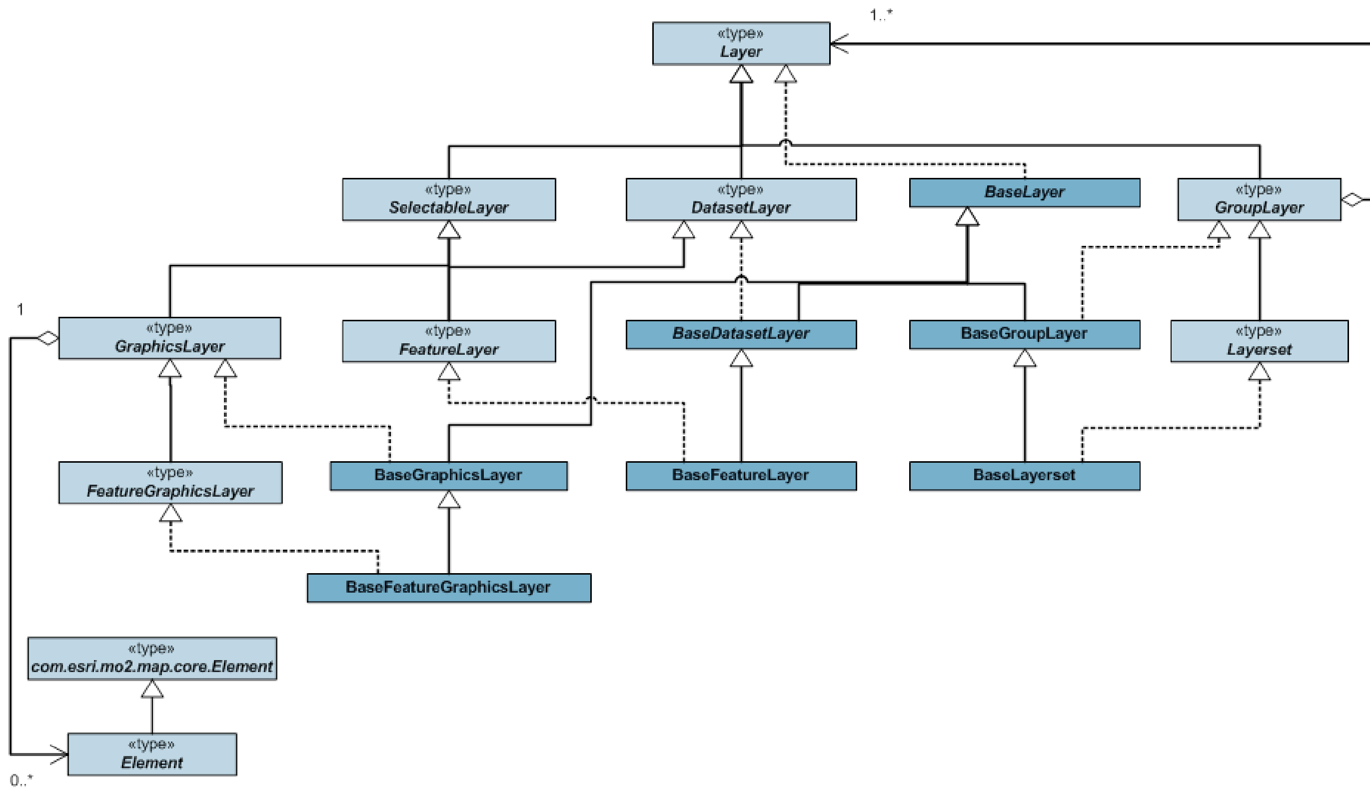

Figure 3 Layer Implementations Class Diagram

Content Provider Components

This section discusses the various components that make up a Content Provider and describe the purpose of each.

Layer Source

A Layer Source is responsible for managing a connection to an external data source and the datasets that make up the external data source. A Layer Source object will create and return the necessary Layer objects in order to view the data on a Map object.

ESRI marks the `com.esri.mo2.map.dpy.LayerSource` class as deprecated. It states that the developer should use the Content API to retrieve layers. A Layer Source still needs to be developed in order to support serialization and de-serialization from an extended ArcXML (AXL) file.

Within MapObjects Java Edition, there are many different default Layer Sources already developed. The most common is the `BaseWorkspaceLayerSource` class. It is responsible for creating Feature Layer objects from external sources that contain vector data. It uses a `Workspace` object that contains Feature Classes and Feature Datasets to create Layer objects. The second most common

is the `BaseImageLayerSource` class. It is responsible for creating Image Layer objects from external sources whose data is an image format.

Layer Source Factory

A `LayerSourceFactory` is responsible for constructing a specific `LayerSource` for attributes that are specified. Each Layer Source implementation should also have a `LayerSourceFactory` implementation.

An instance of the `LayerSourceFactory` needs to be added to the `LayerSourceRegistry` so that it can be found later. The `LayerSourceFactory` instance is indexed by a specific Type string value within the `LayerSourceRegistry`. The Type string value can be used to retrieve the `LayerSourceFactory` that should be used to construct a particular `LayerSource`.

Map Dataset

A Map Dataset is responsible for querying the external data source, via the Layer Source, and producing the data for the Map to draw. A Map Dataset will be associated with a Layer object, so that the Layer object can call upon it to retrieve data. The Layer will pass along query criteria such as attribute filters and spatial filters to the Map Dataset in order to retrieve only the necessary data.

There are two main Map Dataset interfaces within MapObjects Java Edition – `FeatureClass` and `ImageClass`. The `FeatureClass` interface should be implemented in order to access vector data and associated attribute values. `FeatureClass` implementations will manage the attribute fields and their types, and also will be responsible for querying the external data source for the data, and returning it. The `ImageClass` interface should be implemented in order to access raster data. `ImageClass` implementations will be responsible for querying the external data source for the image, converting the image to the appropriate type for drawing, and returning it.

Cursor

A `Cursor` object is responsible for iterating over a set of data results and is used to provide custom filtering on the data or data manipulation. `Cursor` objects will be returned from the Map Dataset object's search methods. A `Cursor` is synonymous with a `java.util.Iterator` object.

MapObjects Java Edition provides several `Cursor` implementations to perform a variety of tasks such as geometry transformations, selection set filtering, and query filtering.

Workspace

A `Workspace` object contains a set of Map Datasets, such as `FeatureClass` objects. It is used in conjunction with a `WorkspaceLayerSource` so that it can easily produce Layer objects.

ContentProvider

A `ContentProvider` object is the main entry point for the Content Provider. It provides methods to access the root `Content` objects. There are two base types for Content Providers. A `ConnectionContentProvider` should be used for Connection-based content providers. A `FileSystemContentProvider` should be used for File-based content providers.

Source

A `Source` object is a representation of an external source of data. Its purpose is to produce `Folder` object that contains the `Content`. Source objects can be cached through the `SourceCache` object. `Source` objects are constructed from the `Connection` object. `Source` objects typically contain Dataset objects that are used to query the external data source.

`Content` can be retrieved from the `Source` object via a path-like string. Each `Content` within the `Source` will have a unique name. The `Source` object can be considered the root folder for a specific Content Provider.

Content

MapObjects Java Edition's documentation states that a `Content` object is a lightweight proxy representation of an object that exists within some external database of filesystem. The `Content` object can expose Feature Layers, Feature Classes, Feature Datasets, Image Layers, Image Classes, or other types of datasets or layers.

Folder

A `Folder` object is a specialized version of a `Content` object that contains multiple `Content` objects. A `Folder` is used to arrange and organize multiple `Content` objects.

XML Content Provider

This section will step through how to create a Content Provider using various code examples. This assumes that the reader has some level of experience in developing applications using MapObjects Java Edition. It is beneficial to review the MapObjects Java Edition Developer's Guide for more information on working with MapObjects Java Edition, developing custom Feature Classes, and developing Content Providers.

This simple Content Provider will access an XML file located on the file system. The XML file is in a custom format developed by *CTC*. Refer to Appendix A for the schema of the XML file. Refer to Appendix B to view the actual XML file.

Create LayerSource Implementation

This represents the Layer Source that will be used to create `com.esri.mo2.map.dpy.FeatureLayer` objects that know how to draw data from the XML File.

A `BaseWorkspaceLayerSource` was chosen for simplicity. All that is needed is to create a custom `Workspace` implementation that produces Feature Dataset objects, and the `BaseWorkspaceLayerSource` will take care of creating the Layer object. Creation of the `Workspace` implementation is covered in Section 4.2.2.

getLayerSource Method

This method is used to get an instance of the XML File Layer Source. Retrieving a handle to the instance is controlled through this method to ensure that only one exists in the JVM at any one time. This method will determine if an instance of the XML File Layer Source is in the cache. If one is found, then it is returned, otherwise, a new one is created, added to the cache, and then returned. Subsequent requests for an instance will return the cached instance.

getAttributes Method

This method returns the attributes of the XML File Layer Source as a `HashMap`. These attributes will be used when serializing to an AXL Configuration File.

The XML File Layer Source has two attributes – Type and File. Type is required and is unique among all the different Layer Sources defined in the system. The Type attribute in the XML File Layer Source is always set to "XMLFileLayerSource". File is the location of XML File.

When the configuration of the XML File Layer Source is serialized to an AXL file, it will appear like the following in the AXL file (within the <FOLDERS> element):

```
<FOLDER name="ws-0" type="XMLFileLayerSource">
<ATTRIBUTE name="File" value="C:\data\sample_data.xml"/>
</FOLDER>
```

getWorkspace Method

The getWorkspace method returns a reference to the Workspace object that should be used by the super class to acquire workspace information. The Workspace object is contained as a private data member within the XML File Layer Source class.

Create Workspace Implementation

Refer to the CXMLFileWorkspace.java file contained in the source repository. The `CXMLFileWorkspace` class is contained within this file. This class is a `Workspace` implementation in order to retrieve `com.esri.mo2.data.feat.Dataset` objects and the names of the datasets from the external source. In this case, the external source is the XML file.

getDatasetNames Method

This method will parse the XML file, in this case using XMLBeans, and iterate over all of the Dataset elements within the file. The names of each dataset will be placed into an array, and the array will be returned.

getDataset Method

This method will create a `CXMLFileFeatureClass` object for the specified dataset name, and return it. More details on the `CXMLFileFeatureClass` class are contained in Section 4.2.3.

getDatasets Method

This method simply retrieves the names of all the datasets by calling the `getDatasetNames` method. Then, for each name, `getDataset` is called to create a dataset for that name. Each dataset is then added to the array and returned. Each element of the array will be of type `CXMLFileFeatureClass` that will be detailed in Section 4.2.3.

Create a Feature Class Implementation

A Feature Class needs to be implemented in order to access the data within the XML file when the Layer needs data to draw. `CXMLFileFeatureClass` implements the `FeatureClass` interface, and access the XML file for the appropriate data.

When the `CXMLFileFeatureClass` is constructed it must initialize the various properties of the Feature Class such as fields, coordinate system, and feature type (Point, Line, or Polygon). The XML file will be accessed in order to get that information.

searchEnvelope Method

This method is implemented in order to access the data from the XML file. In this method, the XML file is parsed, and the correct dataset element for the Feature Class is found. That dataset element is handed off to a `Cursor` implementation so that `Feature` objects can be created. The spatial and where clause filtering is handled at a higher level through MapObjects Java Edition via a `BaseQueryFilterCursor` object. The `Cursor` implementation is covered in Section 4.2.4.

Creating a Cursor Implementation

A Cursor is an object that sometimes does not need implemented. There are several default Cursor implementations available with MapObjects Java Edition. In this Content Provider, a `Cursor` is implemented just to show how to implement one.

The Cursor implementation for this Content Provider is contained within the `CXMLFileCursor` class. The main part of the `Cursor` code is contained within the peek method. In this method a Feature element is pulled from the XML file, and the data is placed into a MapObjects Java Edition Feature object. The Feature object is then set in the `_next` data member within the Cursor. When the next method is called on the cursor, the value in the `_next` data member will be returned.

As of right now, there have been enough classes created to create a Layer which could then be added to a Map, perform a search and retrieve results from the XML file, display the data on a Map, and serialize the configuration to an AXL configuration file. Next, classes that implement some of the `Content` API interfaces (contained in com.esri.mo2.src.sys) will be implemented so that the Layers can be accessed via the Content API, and through the Catalog Model within JoViewPlus, and other MapObjects Java Edition applications.

ContentProvider, Source, and Connection Implementations

Since our external data source is file-based, we do not have to implement these interfaces for this Content Provider. They are already implemented for us in the `com.esri.mo2.src.file` package.

Create a FileHandler Implementation

As mentioned before, we do not have to implement `ContentProvider`, Source, or `Connection` interfaces, but we do have to implement the `FileHandler` interface. The `FileHandler` interface is responsible for identifying which files have relevant data for the Content Provider. A

FileHandler implementation only needs to be done for Content Providers whose data resides in a file.

The CXMLFileHandler class implements the FileHandler interface. The primary method that needs implemented is the filter method. The filter method takes in an array of File objects. The File objects each need to be checked to see if this Content Provider can read them. To do this, the file must have an .xml extension, and must be able to be parsed by the XMLBeans that were generated from the schema. If those conditions hold true, then a new CXMLFileFolder object is added to the LinkedList argument passed into the filter method. The CXMLFileFolder class is covered in Section 4.2.8.

Within the CXMLFileHandler class, there is an implementation of a LayerSourceFactory. A LayerSourceFactory implementation is created in order to aid in the de-serialization process from AXL. The responsibility of a LayerSourceFactory is to create a specific LayerSource. The parameters necessary to create the Layer Source are contained within a HashMap argument in the constructLayerSource method in the LayerSourceFactory. The LayerSourceFactory interface is implemented by the CXMLFileLayerSourceFactory class within the CXMLFileHandler class.

An instance of CXMLFileLayerSourceFactory needs to be added to the LayerSourceRegistry object. The LayerSourceRegistry associates a string Type value with an instance of a LayerSourceFactory. This association takes place within a static initializer in the CXMLFileHandler class. An instance of CXMLFileLayerSourceFactory is created, and added to the LayerSourceRegistry with the Type value equal to "XMLFileLayerSource". When de-serialization from AXL occurs, MapObjects Java Edition will use the "Type" attribute from the Folder to create a LayerSource. The LayerSource is then used to create the necessary Layer objects to be added to the Map.

Allowing MapObjects Java Edition to discover the FileHandler

The custom FileHandler implementation, CXMLFileHandler, needs to be "published" so that it can be discovered by MapObjects Java Edition. To allow MapObjects Java Edition to find CXMLFileHandler a file named com.esri.mo2.file.FileHandler was added to the folder META-INF/services. The com.esri.mo2.file.FileHandler file contains the fully qualified name of the CXMLFileHandler class on the first line of the file. The contents of com.esri.mo2.file.FileHandler would be:

mil.joint.webcop.ext.moje.xmlfile.CXMLFileHandler

The META-INF/services folder will also need to appear on the classpath or included in the JAR file for the Content Provider so that MapObjects Java Edition can find it.

Creating a Folder Implementation

A `Folder` is an extended version of the `Content` class. A `Folder` can contain multiple `Content` objects. Creating a `Folder` implementation is a mechanism of organizing `Content` objects. Arranging `Content` objects in `Folder` objects is optional

A custom Folder implementation was created with the `CXMLFileFolder`. Its responsibility is to read the Dataset elements from the XML, create Content objects from them, and return them from the `getContents` method. The Content objects created are of type `CXMLFileContent`, which is discussed in Section 4.2.9.

Creating a Content implementation

The `Content` interface is implemented when an object needs to be exposed. Most `Content` implementations expose some sort of MapDataset or Layer object. The object is exposed through the `getData` method within the Content implementation.

MapObjects Java Edition needs to know what types of objects are exposed with the custom Content implementation, so a method called `getAvailableContentTypes` needs to be implemented. It returns a String array of the fully qualified class names of the objects that are exposed by the `Content`.

Other methods within the Content interface need to be implemented. The `getName` and `getIcon` methods need to be implemented to produce a name and a visual icon for the specific `Content`.

Deploying the XML Content Provider

The XML Content Provider should be deployed in the JAR format. The JAR file must contain the com.esri.mo2.file.FileHandler file within the META-INF/services folder.

The JAR file can be placed anywhere on the file system, but must be included on the classpath in order for MapObjects Java Edition to be able to use it. To use the Content Provider within JoViewPlus, perform the following steps:

Be sure MapObjects Java Edition is installed.
Open <MOJE_HOME>/Scripts/moj_run.bat in a text editor.
Append the path to the Content Provider JAR file, and all third party JARS to the end of the CPATH variable.

Save moj_run.bat, and Exit.

Using the XML Content Provider

JoViewPlus is the quickest way to test out the Content Provider. An XML file with the data should be present on an accessible file system. Follow these steps to use the Content Provider:

Open JoViewPlus by going to Start | Programs | ESRI | MapObjects Java Edition 2.0 | JoViewPlus
Click on the Add Data button on the toolbar, the Catalog window will appear.
Browse the file system, to the XML file that contains the data, and double-click on it.
A list of datasets within the XML file will appear, choose the dataset you wish to view.
You should see the data symbolized in the map view.

Implementation Notes

The XMLBeans package was used to parse the XML file. The XMLBeans package was used to read the XML Schema and create JavaBean classes to hold the data from the XML file. This makes it easier to navigate through the data, and converting between data types. The XML Schema that was used is detailed in Appendix A.

Appendix A: XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://webcop.joint.mil/ext/moje/xmlfile/beans"
elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:simple="http://webcop.joint.mil/ext/moje/xmlfile/beans"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="Datasets" type="simple:DatasetsType">
		<xs:annotation>
			<xs:documentation>Comment describing your root
element</xs:documentation>
		</xs:annotation>
	</xs:element>
	<xs:complexType name="DatasetsType">
		<xs:sequence>
			<xs:element ref="simple:Dataset"
maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="DatasetType">
		<xs:sequence>
			<xs:element ref="simple:DatasetConfiguration"/>
			<xs:sequence>
				<xs:element ref="simple:Feature"
maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:sequence>
		<xs:attribute name="datasetName" type="xs:string"
use="required"/>
		<xs:attribute name="geometryType" use="required">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:enumeration value="POINT"/>
					<xs:enumeration value="LINE"/>
					<xs:enumeration value="POLYGON"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:attribute>
	</xs:complexType>
	<xs:element name="Dataset" type="simple:DatasetType"/>
	<xs:complexType name="FeatureType">
		<xs:sequence>
			<xs:sequence>
				<xs:element name="Field"
type="simple:FieldType" maxOccurs="unbounded"/>
			</xs:sequence>
			<xs:choice>
				<xs:element ref="simple:Point"/>
				<xs:element ref="simple:Polyline"/>
				<xs:element ref="simple:Polygon"/>
			</xs:choice>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="PointType">
		<xs:attribute name="x" type="xs:double"/>
```

```
            <xs:attribute name="y" type="xs:double"/>
      </xs:complexType>
      <xs:complexType name="PathType">
            <xs:choice>
                  <xs:element name="Point" type="simple:PointType"
maxOccurs="unbounded"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="HoleType">
            <xs:choice maxOccurs="unbounded">
                  <xs:element name="Point" type="simple:PointType"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="RingType">
            <xs:choice maxOccurs="unbounded">
                  <xs:element name="Point" type="simple:PointType"/>
                  <xs:element name="Hole" type="simple:HoleType"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="PolygonType">
            <xs:choice>
                  <xs:element name="Ring" type="simple:RingType"
minOccurs="0" maxOccurs="unbounded"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="PolylineType">
            <xs:sequence>
                  <xs:element name="Path" type="simple:PathType"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="FieldType">
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="value" type="xs:string"
use="optional"/>
      </xs:complexType>
      <xs:element name="Ring" type="simple:RingType"/>
      <xs:element name="Hole" type="simple:HoleType"/>
      <xs:element name="Polygon" type="simple:PolygonType"/>
      <xs:element name="Polyline" type="simple:PolylineType"/>
      <xs:element name="Point" type="simple:PointType"/>
      <xs:element name="Feature" type="simple:FeatureType"/>
      <xs:complexType name="DatasetConfigurationType">
            <xs:sequence>
                  <xs:element ref="simple:FieldConfiguration"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="FieldConfigurationType">
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:element name="FieldConfiguration"
type="simple:FieldConfigurationType"/>
      <xs:element name="DatasetConfiguration"
type="simple:DatasetConfigurationType"/>
</xs:schema>
```

Appendix B: Sample XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<Datasets xmlns="http://webcop.joint.mil/ext/moje/xmlfile/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://webcop.joint.mil/ext/moje/xmlfile/beans
schema.xsd">
    <Dataset datasetName="XMLFileTest" geometryType="POINT">
        <DatasetConfiguration>
            <FieldConfiguration name="ID"/>
        </DatasetConfiguration>
        <Feature>
            <Field name="ID" value="200"/>
            <Point x="0.0" y="0.0"/>
        </Feature>
    </Dataset>
</Datasets>
```

Author Contact Information

Derek Sedlmyer, Software Engineer, *CTC*, sedlmyed@ctc.com, 814-269-6532

James Taylor, Software Engineer, *CTC*, taylorj@ctc.com, 814-269-6863