JavaÔ Metadata Interface and Data Warehousing

A JMI white paper by

John D. Poole

November 2002

Abstract. This paper describes a model-driven approach to data warehouse administration by presenting a detailed scenario illustrating how JMI-enabled tools might be used in the realization of a data warehouse schema. Java code fragments illustrating the use of JMI interfaces are provided throughout. We also show how a JMI resource might integrate with the J2EE™ Connector Architecture.

Introduction

This paper describes the management of a data warehouse using the Java Metadata Interface (JMI) by walking through a detailed scenario in which a data warehouse is modeled and automatically constructed using meta data-driven tools. JMI is the pervasive access mechanism for managed meta data, and CWM is the metamodel defining that meta data. This scenario consists of five distinct components, and we illustrate how JMI supports each component:

1. **Meta data service initialization**. A centralized meta data service supporting a common metamodel is generated and brought online.

2. **Model construction**. The meta data service is used to create shared meta data based on the common metamodel. This meta data is then published to the rest of the environment.

3. **Tool initialization**. Each tool constructs its own internal information structures and links to other tools, based on shared meta data published via the central meta data service.

4. **Metamodel interoperability**. A new tool that supports a different metamodel is introduced to the data warehouse. JMI Reflection is used to reconcile differences between the tool-specific metamodel and the common metamodel used by the rest of the data warehouse.

5. **Data warehouse information flow**. The overall flow of data and control through the data warehouse is illustrated. These flows are meta data-driven and facilitated by JMI programmatic interfaces (both metamodel-specific and reflective) and by the XMI bulk import/export mechanism supported by JMI.

The following subsections address each of the five components of the JMI-enabled data warehouse management scenario.

Meta Data Service Initialization

In the first step of the scenario, a data warehouse administrator initializes a *JMI-enabled meta data service* as the central meta data store for the data warehouse. By *meta data service*, we mean a general-purpose tool that is capable of providing a complete implementation of the JMI specification. An implementation of the JMI specification might, for example, consist of a server process that realizes the JMI interfaces corresponding to some MOF-compliant metamodel, the JMI reflective interfaces, and the XMI reader and writer interfaces. Precisely *how* a JMI-enabled meta data service is implemented is not prescribed by the JMI specification itself.

As illustrated in Figure 1, the administrator's warehouse management tool connects to the JMI service via the J2EE™ Connector Architecture's Custom Client Interface (CCI). In this case, we assume that the Connection interface has several custom methods that facilitate access to the JMI service. These methods are implementation-specific (i.e., not prescribed by JMI).

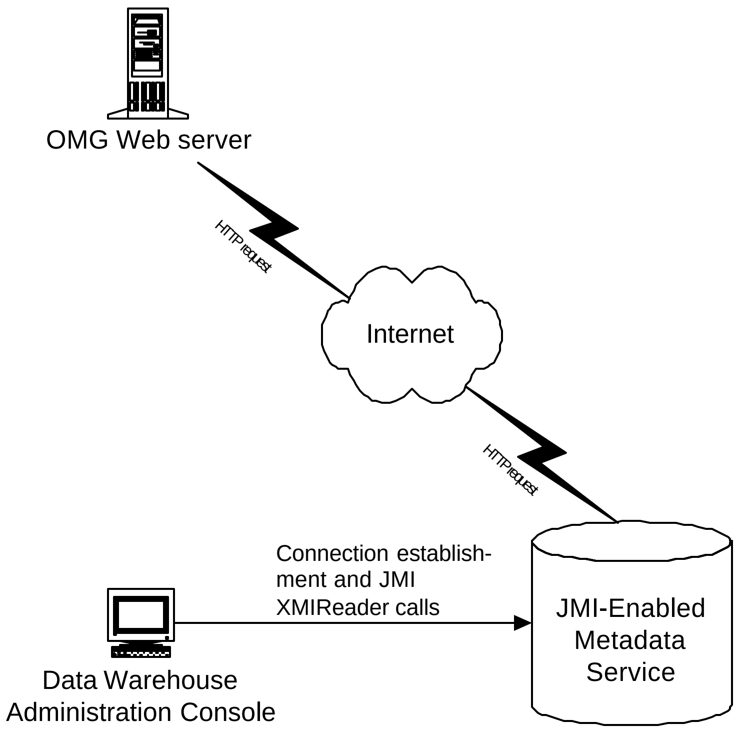
OMG Web server
HTTP request
Internet
HTTP request
Connection establish-ment and JMI XMIReader calls
JMI-Enabled Metadata Service
Data Warehouse Administration Console

Figure 1: Meta Data Service Initialization

The custom Connection interface is defined as follows:

```
public interface com.mdservice.Connection {

      public void close() throws com.mdservice.ResourceException;

      public javax.resource.cci.ConnectionMetaData getMetaData()
            throws com.mdservice.ResourceException;

      public javax.jmi.reflect.RefPackage getTopLevelPackage()
            throws com.mdservice.ResourceException;

      public javax.jmi.xmi.XmiReader getXmiReader()
            throws com.mdservice.ResourceException;

      public javax.jmi.xmi.XmiWriter getXmiWriter()
            throws com.mdservice.ResourceException;
}
```

The `getTopLevelPackage()` method returns a single instance of RefPackage, and the semantics of this method specify that this RefPackage object represents the *outermost package extent* of the MOF Model instances supported by the JMI-enabled service. In other words, in this particular implementation strategy, we have established, purely as a convention, that there is always a single, outermost package provided that contains all of the metamodel-specific package extents[1].

The Connection interface also provides the custom factory methods `getXmiReader()` and `getXmiWriter()`. These methods are used for instantiating the XmiReader and XmiWriter interfaces, respectively, within the context of the connected meta data service.

A data warehouse administration client first connects to the JMI service and obtains the (empty) outermost package extent of the meta data service by calling the `getTopLevelPackage()` method. Next, the client creates an instance of XmiReader and invokes the read() method on the XmiReader interface, supplying both the RefPackage returned by the previous call to `getTopLevelPackage()`, and a URL pointing to an XMI document containing the MOF metamodel to be loaded. In this case, the XMI document contains the definition of the Object Management Group's Common Warehouse Metamodel (CWM), rendered as MOF Model instance.

The semantics of this particular implementation of the XmiReader::read() operation are such that the imported CWM XMI element is interpreted as an instance of the MOF Model and is subsequently used in generating the internal structure of the outermost package extent. In other words, the JMI service is "hard-wired" to understand MOF. If it imports an instance of the MOF Model (i.e., a MOF-compliant metamodel), it subsequently builds the package structure

[1] Note that one might use JNDI to select a particular ConnectionFactory from a local JNDI tree. In this case, it happens that the ConnectionFactory is bound to precisely one outermost RefPackage instance, since each connection has a getTopLevelPackage() method that returns a single instance. Other implementation strategies might provide some alternative means of locating any one of several top level packages.

prescribed by the MOF Model instance. If, on the other hand, it imports an instance of a metamodel (i.e., meta data) corresponding to some MOF-compliant metamodel whose structure had already been built within the server, then it will treat the XMI element content as meta data and use it populate the meta data server.

Once the CWM metamodel is loaded, the JMI service generates a complete set of metamodel-specific (tailored) Java interfaces, according to the JMI mapping templates, as well as a corresponding library of Java implementation classes. These implementation classes constitute a *realization* of a CWM server. Note, once again, that this particular implementation strategy of the JMI service is not defined by the JMI specification. Figure 1 illustrates the JMI meta data service initialization event. Figure 2 illustrates the CWM metamodel.

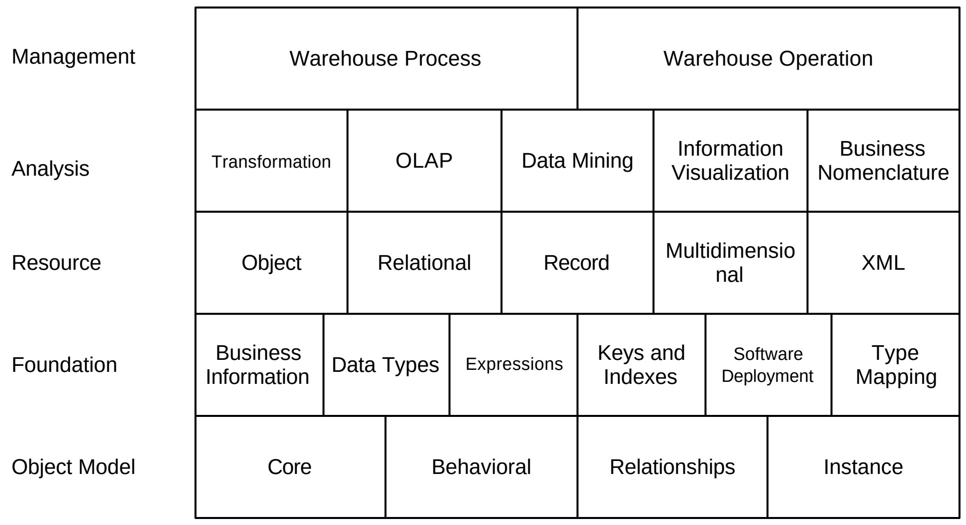

Figure 2: CWM Metamodel

The sequence of connection and JMI calls used to initialize the JMI service might consist of the following:

```
ConnectionFactory cxf = new ConnectionFactory();
Connection cx = cxf.getConnection( properties );
RefPackage msTlp = Connection.getTopLevelPackage();
XmiReader xmiReader = Connection.getXmiReader();
xmiReader.read( "http://cgi.omg.org/docs/ad/01-02-03.txt", msTlp );
```

Model Construction

Now that the JMI service has been initialized with the CWM metamodel, we can begin to construct a model of our data warehouse based on CWM. This is illustrated in Figure 3 below:

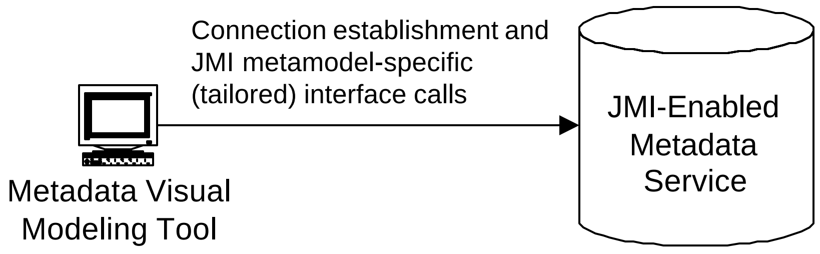
Connection establishment and
JMI metamodel-specific
(tailored) interface calls
JMI-Enabled Metadata Service
Metadata Visual Modeling Tool

Figure 3: Model Construction

Here, a CWM-aware visual modeling tool is used to construct model instances. The modeling tool connects to the JMI service and then acquires the outermost RefPackage extent from the connection. From this point forward, the modeling tool directs its activities against local (client-side) JMI interfaces that are specific to the CWM metamodel. The user might, for example, drill-down on the outermost package to discover the metamodel-specific packages clustered in this package. These would consist of packages such as Warehouse Process, Warehouse Operation, Transformation, OLAP, etc. Each is a RefPackage instance corresponding to one of CWM metamodel packages shown in the block diagram of Figure 2.

The end user constructs a CWM model by selecting various classes from each of the CWM packages and creating instances of them. For example, the user might create an OLAP Dimension by selecting a Dimension class icon from a palette representing OLAP objects, and then dragging it onto a display area. Internally, the modeling tool performs local calls against the CWM metamodel-specific JMI interfaces. The following code fragment illustrates several JMI calls that would be performed in the construction of a CWM OLAP Dimension instance:

```
// Get the DimensionClass proxy
org.omg.java.cwm.analysis.olap.DimensionClass dc
    = olapPkg.getDimension();

// Create a Time Dimension

org.omg.java.cwm.analysis.olap.Dimension timeDim
    = dc.createDimension();
timeDim.setName( "Time" );
timeDim.setTime( true );
```

Now, let's assume that the data warehouse model, once completed, consists of instances of modeling elements taken from each of the following CWM packages:

- Record package: Used to model the raw data feeds supplying information to the data warehouse.

- Relational package: Used to construct models of both the operational data store (ODS) and the dimensional star-schema analysis database of the data warehouse.

- Transformation package: Used to model the data transformations going from the raw data source to the ODS, as well as from the ODS to the dimensional star-schema database. These models serve as the primary descriptions of any extract, transform, and load (ETL) process that the data warehouse might implement.

- OLAP package: Used to model the OLAP (Online Analytical Processing) abstractions exposed by the data warehouse for analysis and reporting. This model includes a mapping to the relational star-schema model, as a source of OLAP data.

- Warehouse Process and Warehouse Operation packages: Used to model the data warehouse control processes that manage and track any ETL activities that might be performed, based on the Transformation models.

The complete data warehouse model is stored in a single RefPackage that is an instance of the CWM metamodel, and is now available to the rest of the data warehousing environment via the JMI interfaces of the central JMI service.

Tool Initialization

Now that the data warehouse schema has been completely defined in terms of a centrally stored CWM model, the data warehouse can be physically generated. This is done through the meta data initialization of a number of JMI-enabled data warehousing tools. This is illustrated in Figure 4.

A data warehouse administration tool (possibly a backend to the modeling tool described earlier) initializes each of the installed data warehousing tools. The administration tool opens a connection on each of the data warehouse tools, and subsequently launches an XmiReader on each Connection. The administration tool then launches an XmiWriter on its own Connection to the JMI service, supplying it with a temporary output stream. The administration tool then exports the entire CWM model to the output XMI stream. It invokes each XmiReader::read() method on each tool-wise Connection, supplying the temporary stream as a parameter, as well as each tool's outermost RefPackage extent. The precise behavior of the XmiReader::read() method is determined by each tool implementation, but in general, tools will consume only those parts of the CWM model that they require or understand. For example, the OLAP server's implementation of XmiReader::read() might scan the entire XMI stream, looking specifically for instances of CWM OLAP, Relational, and Transformation packages, and use these to populate the content of its outermost RefPackage.

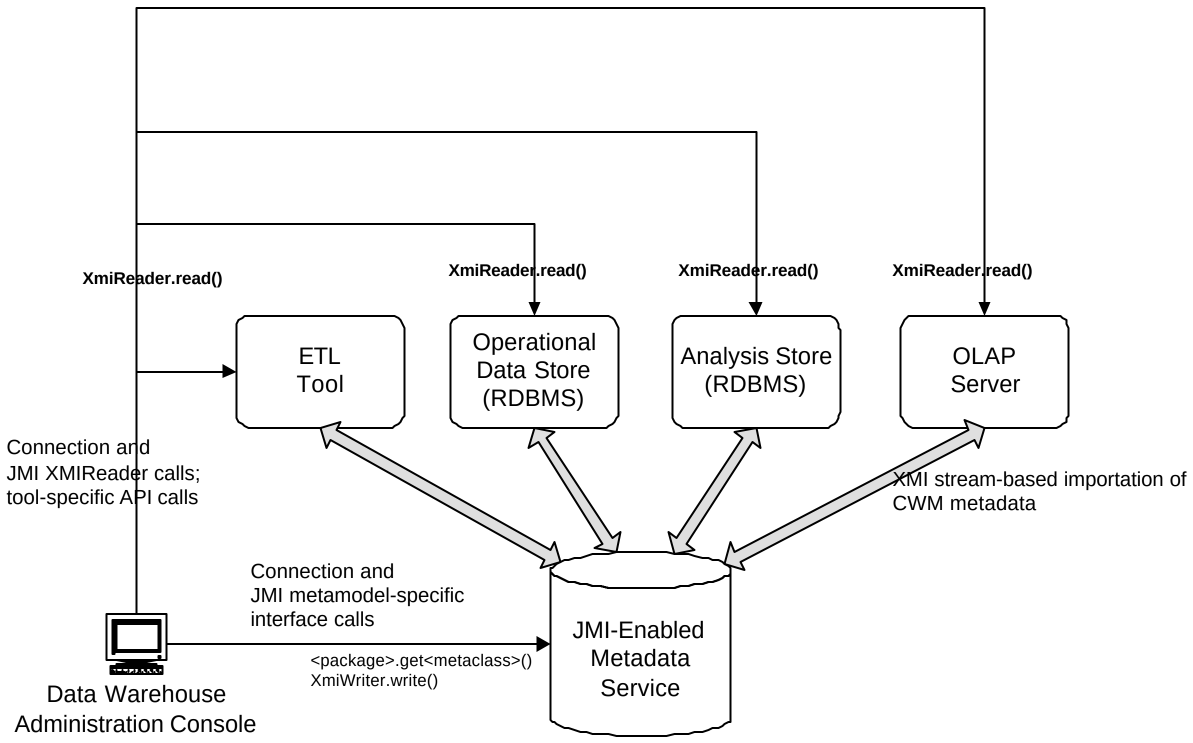
XmiReader.read()
XmiReader.read()
XmiReader.read()
XmiReader.read()
ETL Tool
Operational Data Store (RDBMS)
Analysis Store (RDBMS)
OLAP Server
Connection and JMI XMIReader calls; tool-specific API calls
XMI stream-based importation of CWM metadata
Connection and JMI metamodel-specific interface calls
<package>.get<metaclass>()
XmiWriter.write()
JMI-Enabled Metadata Service
Data Warehouse Administration Console

Figure 4: Tool Initialization

An alternative approach would be for the administration tool to simply notify each of the data warehouse tools that a new CWM model has been created and is now available. Each tool would connect separately to the JMI service and could peruse the content of the outermost RefPackage extent, navigating the model structure through JMI interface calls. Each tool would then subsequently load the particular meta data that it requires (i.e., once each tool has discovered an appropriate RefPackage instance within the CWM model), via either the JMI programmatic interfaces or the XmiReader and XmiWriter interfaces.

Metamodel Interoperability

This portion of the data warehouse scenario describes advanced functionality that can be realized through MOF/JMI reflection. In this case, users of the OLAP server have acquired an advanced, multidimensional visualization/reporting software package. Like other tools comprising the data warehouse, the new visualization tool is also JMI-enabled. However, it uses a different MOF metamodel than CWM; specifically, some MOF-compliant metamodel that represents very specific aspects of advanced, multidimensional, visual analysis. The new tool needs to be integrated with the rest of the data warehouse, and, in particular, since the tool supports meta data-driven data integration with OLAP servers, it must be integrated at the meta data level with the CWM OLAP model used by the OLAP server. This is illustrated in Figure 5 below:

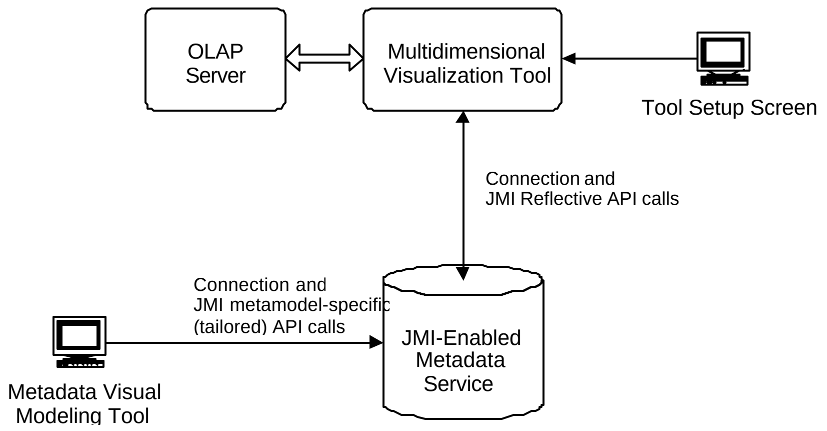
OLAP Server
Multidimensional Visualization Tool
Tool Setup Screen
Connection and JMI Reflective API calls
Connection and JMI metamodel-specific (tailored) API calls
JMI-Enabled Metadata Service
Metadata Visual Modeling Tool

Figure 5: Metamodel Interoperability

To effectively integrate the new visualization tool with the rest of the environment, the data warehouse modeler first connects to the JMI service and then creates an instance of the CWM Visualization metamodel to represent the visualization concepts, as they relate to the OLAP model. That is, visualization meta data is constructed and linked to the various OLAP model objects that are to be visualized (e.g., Cube, Dimension).

Now, the new visualization tool needs to be integrated at the meta data-level with the CWM Visualization model just created. Although the visualization tool is JMI/MOF-compliant, it does not understand CWM. It is driven by its own MOF metamodel. The data warehouse modeler uses the visualization tool's set-up screen to first build the tool's meta data. The modeler then reconciles both types of meta data by constructing a programmatic script (written in Java) that connects to the JMI service and uses JMI Reflective programming to acquire equivalent objects from the CWM model. In this manner, the visualization tool is driven by its own meta data, but can dynamically map to an equivalent CWM Visualization model, and therefore indirectly to the OLAP model elements that are to be rendered. This level of indirection provided by JMI Reflection enables the advanced visualization/reporting tool to perform meta data-directed processing of OLAP data, even though its metamodel is different from CWM.

Data Warehouse Information Flow

The final diagram in Figure 6 shows the overall flow of information through the integrated data warehouse. The clear arrows represent the general flow or progression of data through the data warehouse, all the way from the ODS to the advanced visualization/reporting software. This data flow is inherently meta data-driven, and meta data in this environment has a single and centralized representation in terms of JMI. Shared meta data is defined by a MOF-compliant metamodel (i.e., CWM), but the JMI-enabled meta data service is not tied to any particular metamodel, and is capable of loading the CWM metamodel and dynamically generating an internal implementation of CWM.

Communication of shared meta data is achieved through JMI interfaces. XmiReader and XmiWriter interfaces are used to transfer complete models or specific packages of models in a bulk format for loading into tools. On the other hand, metamodel-specific (CWM, in this case) JMI interfaces are used by client tools for browsing and possibly creating or modifying existing meta data structures. Finally, JMI Reflection is used to facilitate meta data integration between tools whose metamodels differ, but are otherwise MOF-compliant.

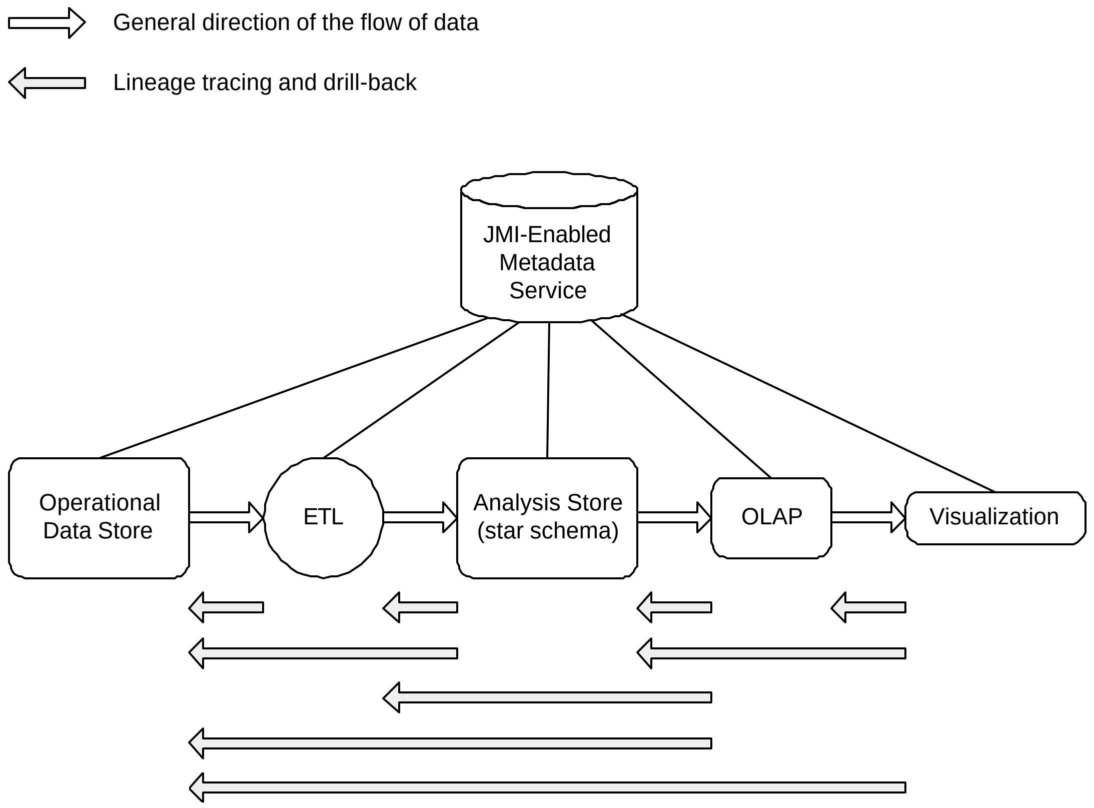
General direction of the flow of data
Lineage tracing and drill-back
JMI-Enabled Metadata Service
Operational Data Store
ETL
Analysis Store (star schema)
OLAP
Visualization

Figure 6: Data Warehouse Information Flow

Summary

This paper has provided an extensive example of model-driven data warehouse management, in which JMI provides a common access mechanism for meta data, and CWM provides the common metamodel. We've demonstrated that a complete data warehouse can be defined, initialized, and then placed into operation relatively easily through the use of centralized meta data and common meta data access mechanisms. The model-driven approach to integration greatly enhances the return-on-investment (ROI) of any data warehouse or supply chain, since it reduces the costs of integrating best of breed implementation tools considerably.