The Common Warehouse Metamodel as a Foundation for Active Object Models in the Data Warehouse Environment

John D. Poole

Principal Software Engineer,
Hyperion Solutions Corporation

Member, OMG CWM Working Group

900 Long Ridge Rd
Stamford, CT 06902-1135
USA

john_poole@hyperion.com
+1 203 703 4359

10 April, 2000

Abstract

The Common Warehouse Metamodel (CWM) is a recently adopted OMG metadata standard for the data warehouse domain. Although designed primarily for metadata interchange between tools and repositories, CWM has sufficient semantics to serve as the basis for the construction of Active Object Models that are capable of driving data warehouse tools, providing a platform for completely dynamic data warehouse architectures. This position paper demonstrates these semantics by showing that CWM readily aligns with a number of well-known software patterns that are fundamental to Active Object Models.

Introduction

The Common Warehouse Metamodel (CWM) is a recently adopted standard of the Object Management Group (OMG) for metadata interchange in the data warehouse environment [CWM00].

CWM supports a model-driven approach to metadata interchange, in which object models representing tool-specific metadata are constructed according to the syntactic and semantic specifications of a common metamodel. This means that tools agree on the fundamental concepts of the domain and are capable of understanding a wide range of models representing particular metadata instances. CWM resolves a number of very significant metadata and tool integration problems facing the data warehousing community [Chan00].

It is likely that initial implementations of CWM will use models in a manner that could be characterized as both *static* and *externalized*, with respect to the tools interchanging them. Models will be static in the sense that, although fully capable of modeling behavior, most early CWM models will probably not directly influence tool execution. Models will be externalized in the sense that most tools will map these shared models to their private, internal implementation models, and not use them directly as definitions of their private metadata. This overall approach, however, is completely reasonable when CWM is used to drive tools having pre-defined, hard-wired architectures.

On the other hand, the semantics and interfaces of CWM make it an extremely powerful model for the construction of a new generation of data warehouse tools that are both dynamically configurable and highly adaptive to different environments. Such tools would possess *dynamic architectures* driven by *Active Object Models* [FY98, JO98, John98], with CWM serving as the foundational metamodel guiding the creation of these Active Object Models.

This paper demonstrates that CWM is an ideal foundation for building Active Object Models for the data warehousing environment, by showing that CWM readily aligns with several established, fundamental patterns for Active Object Models.

Common Warehouse Metamodel Overview

CWM Architecture

The Common Warehouse Metamodel is structured as a collection of related metamodels (or sub-metamodels), each metamodel occupying its own package, and with a minimal number of inter-package dependencies. This is illustrated in Fig. 1.

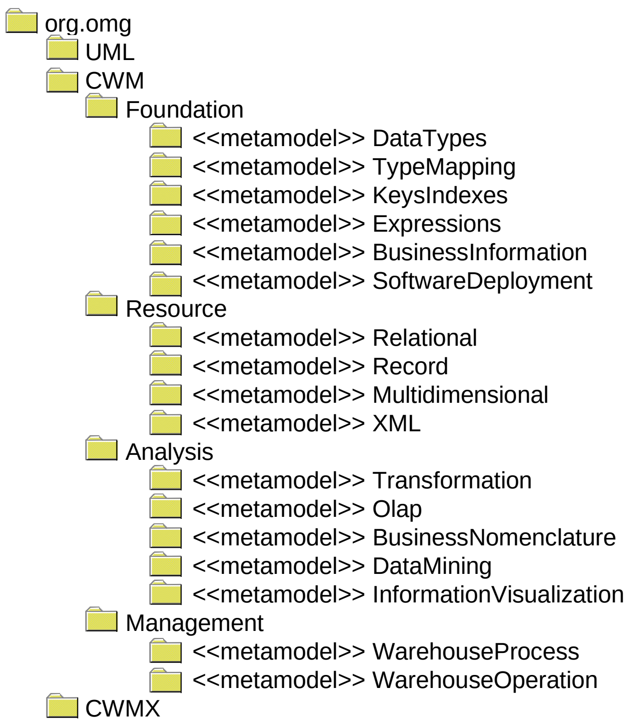
org.omg
UML
CWM
Foundation
<<metamodel>> DataTypes
<<metamodel>> TypeMapping
<<metamodel>> KeysIndexes
<<metamodel>> Expressions
<<metamodel>> BusinessInformation
<<metamodel>> SoftwareDeployment
Resource
<<metamodel>> Relational
<<metamodel>> Record
<<metamodel>> Multidimensional
<<metamodel>> XML
Analysis
<<metamodel>> Transformation
<<metamodel>> Olap
<<metamodel>> BusinessNomenclature
<<metamodel>> DataMining
<<metamodel>> InformationVisualization
Management
<<metamodel>> WarehouseProcess
<<metamodel>> WarehouseOperation
CWMX

Figure 1: CWM Architecture

All CWM packages are dependent upon the core packages of the OMG's Unified Modeling Language (UML), which provides a syntactic foundation for CWM. The CWM metamodels, and any models based on them, are defined in UML.

CWM extends the UML language, in the sense that every CWM metaclass derives either directly or indirectly from metaclasses of UML. For example, CWM's Relational Table metaclass inherits directly from UML Class, while CWM's Relational Column inherits directly from UML Attribute. Thus, CWM can be viewed as a *domain-specific language* for specifying data warehouse models. An implementation of CWM provides an *object-oriented framework* for constructing data warehouse models.

The CWM *Foundation* layer is comprised of metamodels that support the modeling of basic structural elements (e.g., abstract keys and indexes, expressions, data types and type mapping systems, business information, and software components).

Metamodels of the *Resource* layer are used to define operational data resources (as either sources or targets of warehouse activities).

The *Analysis* layer provides a means for modeling information analysis services that are commonly used in data warehouses. Perhaps the most important of these services are Transformations, which model general data movement and lineage. CWM Transformations are used to map models of Analysis services to models of physical Resources. The Transformation metamodel also serves as a general-purpose, model construction tool by providing structural mappings between arbitrary CWM/UML model elements, including those residing across semantic boundaries (e.g., conceptual-logical-physical, and abstraction-refinement).

Finally, the *Management* layer provides metamodels representing warehouse processes and operations. These allow for the modeling of scheduled events (e.g., daily extracts and loads), as well as the tracking of activity status and completion, and the logging of changes made to warehouse elements.

OMG Metamodeling Architecture

CWM not only extends the UML syntax, but, in a much broader sense, extends the OMG *Metamodeling Architecture*, which is comprised of:

- **UML,** as a <u>formal language</u> for defining the structure and semantics of metadata (i.e., for defining metamodels and models).

- **XMI** (**X**ML **M**etadata **I**nterchange) as an <u>interchange mechanism</u> for metamodels and models defined in UML, using XML.

- **MOF** (**M**eta **O**bject **F**acility), which defines common interfaces and semantics for interoperable metamodels, including reflective capabilities. This also includes the MOF-to-IDL (Interface Definition Language) mapping, which defines an <u>interface specification</u> for the discovery and management of models, through programmatic APIs.

In addition to defining common semantics for metamodels, the MOF also serves as the model for UML (i.e., MOF ultimately defines the language in which a UML metamodel is expressed). Since CWM derives from UML, the MOF is also the model for CWM. All CWM models are expressed in UML and implement MOF semantics.

These relationships are summarized in Table 1 below, which illustrates the classic, four-level modeling hierarchy. Since the MOF is capable of describing itself, the modeling stack need not go beyond four levels.

Meta-Level	**Modeling Level**	**Examples**
M3	Meta-Metamodel/Meta-meta-metadata	MOF Class, Attribute, Association, Package, Operation
M2	Metamodel/Meta-metadata	UML Class, Attribute CWM Table, Column
M1	Model/Metadata	Product : <u>Table</u> ProductType : <u>Column</u>

M0	Data/Object	"Toaster" "Television" "Stereo"

CWM Implementation

Each CWM metamodel has a representation as an XML DTD (according to the XMI rules), as well as an IDL definition. Both of these representations are provided by the CWM specification. The DTDs are relevant when CWM models are serialized and exchanged between tools as XMI documents. Any tool exporting metadata via XMI must construct an XMI rendering of its metadata in a manner that is legal with respect to the DTDs. Any tool importing metadata via an XMI document must validate the model against the DTDs (or otherwise have assurance that the model is valid).

The IDL is relevant when building CWM object models in memory or storing them in repositories, since it defines the necessary interfaces, method signatures and package structure that the model must support. For example, a metaobject representing a CWM OLAP Dimension must implement an accessor/mutator pair for each of its **isTime** and **isMeasure** Boolean attributes. This interface is defined in IDL as

```
interface Dimension : DimensionClass, Foundation::Core::Class
   {
      boolean is_time() raises (Reflective::MofError);
      void set_is_time(in boolean new_value)
         raises (Reflective::MofError);
      boolean is_measure ()
         raises (Reflective::MofError);
      void set_is_measure (in boolean new_value)
         raises (Reflective::MofError);
   };
```

and has the equivalent Java mapping:

```
public interface Dimension extends org.omg.UML.Foundation.Core.Class
{
      public boolean isIsTime();
      public void setIsTime(boolean value);
      public boolean isIsMeasure();
      public void setIsMeasure(boolean value);
}
```

Most data warehouses using CWM for model-driven metadata interchange are likely to use a *shared store* (i.e., a *metadata repository* or *server*) for persisting and publishing their CWM models. Publishing shared models via a shared store means that the number of "semantic connections" that tools must understand and implement is reduced considerably [Tolb00]. A given tool need not understand the proprietary metadata format of each tool it inter-operates with. It only needs to understand the standard models and interfaces of the shared store. This considerably reduces the costs of developing and deploying data warehouse tools.

CWM as a Foundation for Active Object Models

In this section, we show that CWM can be used as a basis for constructing *Active Object Models*, resulting in *metadata-driven, dynamic architectures*, for which behavior is not bound by hard-coded logic, but is readily modified or extended by changing the Active Object Model at run time. We do this by showing that the CWM metamodel readily aligns with several, fundamental patterns of Active Object Models (or, equivalently, that the CWM metamodel is capable of generating models consistent with these patterns).

Justification

There are a number of significant reasons why one might want to construct data warehouse tools based on Active Object Models, including the overall adaptability of tools to new environments, and providing users with the ability to modify their system's behavior simply by altering its models. There are disadvantages, as well, including efficiency issues (since Active Object Models are interpreted) and the complexities of building model-based tools. However, these issues apply to all dynamic architectures, not just data warehouses, so we will not go into them here. For more information, see both [FY98] and [JO98].

Assumptions

The one key assumption that our investigation is predicated on involves establishing the correct meta-level for the Active Object Models being considered. Much of the pattern literature regarding Active Object Models generally centers on software implementation and programming; in other words, M1-level descriptors and M0-level instances (according to Table 1). When dealing with metamodels and models, however, we are focused on M2-level descriptors and M1-level instances (what are often referred to as "metaobjects"; i.e., objects representing metadata). So, for example, when we talk about dynamically modifying an instance at run time, we mean modifying an M1 metaobject at run time, and not (at least, not directly) modifying any M0 data object that the metaobject might describe. Management of M0 objects is the responsibility of the tool, rather than the model driving the tool. This assumption seems like a reasonable one to base our pattern alignment investigation on.

Pattern Alignment

This section describes several fundamental Active Object Model patterns and shows how the CWM metamodel aligns with them. We first cover the general Dynamic Object Model pattern of Johnson [John98], and then the PROPERTY, SMART VARIABLE, SCHEMA and ACTIVE OBJECT-MODEL patterns of Foote and Yoder [FY98].

1. Dynamic Object Model Pattern

The Dynamic Object Model pattern, defined by Johnson [John98], defines associations (rather than compositions) between Entities, Properties (i.e., attributes), Types and Rules (i.e., behaviors). This pattern is shown below in Fig. 2. (Note that this pattern can be regarded as a form of metamodel.)

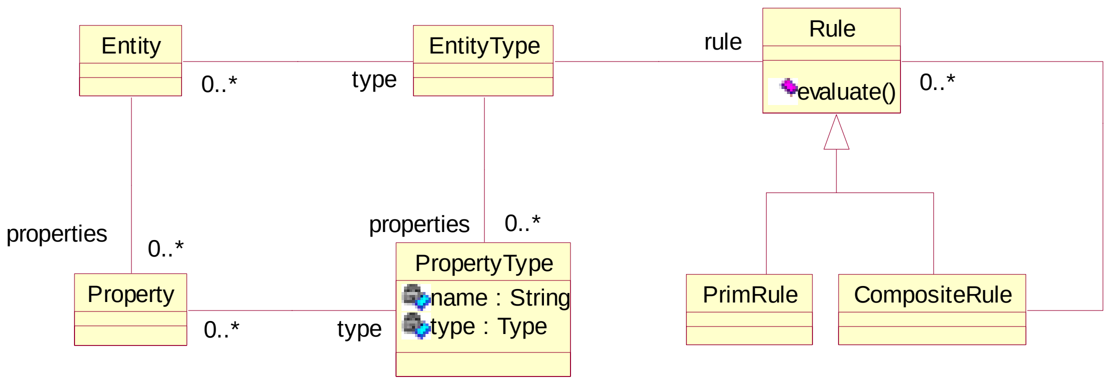

Figure 2: Johnson's Dynamic Object Model

A system based on the Dynamic Object Model has the following characteristics:

- An Entity's type can be assigned or changed at run time.
- An Entity's Properties are not static and can be defined or modified dynamically.
- The Type of a Property can be defined or modified at run time.
- All Property Types supported by an Entity of a particular Type can be determined at run time.
- The behavior specification (e.g., algorithm, method, or operation) of an Entity can be assigned or changed at run time (i.e., object behavior can be modified dynamically).

The Dynamic Object Model maps to the CWM metamodel in the following manner:

- Entity maps to UML Class or UML/CWM descendants of UML Class (e.g., an instance of CWM Relational Table is an instance of Entity).
- Property maps to UML Attribute or UML/CWM decendants of UML Attribute (e.g., an instance of CWM Relational Column is an instance of Property).

- Alternatively, Property may also be mapped to UML TaggedValue. In UML, TaggedValues can be attached to any model element to create "property lists" (simple name/value string mappings).

- EntityType maps to UML Classifier or UML/CWM descendants of UML Classifier.

- PropertyType may be mapped either to UML Classifier or any of its UML subclasses (e.g., UML DataType, UML Class and UML Interface), and any CWM descendants of the UML subclasses of UML Classifier.

- Rule may be mapped to UML BehavioralFeature or any its UML descendants. The CWM metamodel does not explicitly subclass UML BehavioralFeature, but CWM model elements can always be associated with subclasses of UML Behavioral Feature, via the UML owner/feature association relating Classifier and Feature.

In a live CWM object model, an instance of Attribute can always be added to an instance of Class (or removed or modified) via the Feature-specific methods of the MOF Classifier interface. (Note that in all of the interface descriptions below, we've elided the exception handling declarations and most methods for the sake of brevity). Specifically:

```
interface Classifier : ClassifierClass, Core::Namespace,
GeneralizableElement
      {
         FeatureUList feature();
         void set_feature (in FeatureUList new_value);
         void add_feature (in Core::Feature new_element);
         void add_feature_before (
            in Core::Feature new_element,
            in Core::Feature before_element);
         void modify_feature (
            in Core::Feature old_element,
            in Core::Feature new_element);
         void remove_feature (in Core::Feature old_element);
         . . .
      };
```

For example, an instance of Relational Table ("Product") can always have a new Column ("ProductDescription") added to it dynamically as a new Feature.

If TaggedValues are used to build dynamic property lists, then interface support is provided by the UML TaggedValue itself:

```
interface TaggedValue : TaggedValueClass
      {
         . . .
         Core::ModelElement model_element ();
         void set_model_element (in Core::ModelElement new_value);
         void unset_model_element ();
         . . .

      };
```

Interface support is also available in CWM for dynamically assigning types, where a type is an instance of a descendant or subclass of UML Classifier, such as UML Class, UML DataType, or UML Interface. For a UML Attribute, type assignment is performed via methods of the Attribute's inherited StructuralFeature interface:

```
interface StructuralFeature : StructuralFeatureClass, Feature
      {
         . . .
         Classifier type ();
         void set_type (in Classifier new_value);
      };
```

Dynamically assigning types to subclasses of UML Class in CWM is a little less direct. A metaobject can always define an Attribute (either first class or dynamic) whose value represents a reference to a type object. Defining such a reference Attribute with a multiplicity greater than one enables the metaobject to support multiple types. For example, an array of instances of UML Interface could be used to represent "marker" interfaces of the metaobject [Gran98].

Finally, any instances of descendants or subclasses of UML BehavioralFeature (e.g., UML Method or UML Operation) can be dynamically attached to subclasses of UML Classifier (or removed from or modified) using the Feature-specific methods of the MOF Classifier interface described earlier (since UML BehaviorElement derives from UML Feature).

Thus, we have shown that CWM supports all of the characteristics of Johnson's Dynamic Object Model, strictly by virtue of its use of UML syntax and adherence to MOF semantics. The UML/MOF model elements, associations and interfaces described above are the minimal structures that provide compliance with the Dynamic Object Model pattern. They are fundamental elements of the UML/MOF class hierarchy, and are therefore available to all of CWM and its derived metamodels and instances (in fact, they are available to any modeling system based on OMG's Metamodeling Architecture).

2. Domain-Specific Language Patterns

Foote and Yoder [FY98] have observed that the following patterns often arise in emerging domain-specific languages or frameworks: PROPERTY, SMART VARIABLE, SCHEMA and ACTIVE OBJECT-MODEL. They note that the SMART VARIABLE and SCHEMA patterns are often treated as variations on PROPERTY, and that these three patterns generally form a foundation for ACTIVE OBJECT-MODEL.

It was stated earlier that CWM is a domain-specific language for data warehouse metadata, and in the following, we show that CWM does indeed support these patterns.

2.1 PROPERTY

The stated goal of the PROPERTY pattern is to provide runtime mechanisms for adding, changing or deleting properties or attributes of instances at run time. Note that this is essentially the same pattern described in the previous section. Foote and Yoder [FY98] describe both simple property lists (name/value pairs) as well as properties with type descriptors. In CWM, these concepts map to the UML syntax and MOF semantics of TaggedValues and Attributes, respectively, as described in the previous section.

2.2 SMART VARIABLE

The objective of the SMART VARIABLE pattern is to provide a means for intercepting and taking action on references to variables. Foote and Yoder [FY98] state that a minimal, idiomatic solution consists of having all references to variables (both public and private) go through accessor functions. This is certainly supported by CWM/MOF, since the MOF-to-IDL mapping prescribes a regular accessor/mutator-style for referencing variables.

2.3 SCHEMA

The SCHEMA pattern enables an object to expose its data structures at run time. In the case of CWM, this requirement is generally satisfied by MOF reflection. On the other hand, higher level structural mappings in CWM models implemented via the Foundation and Transformation packages can be discovered or exposed using the interfaces of those packages, plus MOF reflection on individual metaobjects.

2.4 ACTIVE OBJECT-MODEL

The ACTIVE OBJECT-MODEL pattern is essentially the same pattern as described in the previous section. The objective of the ACTIVE OBJECT-MODEL is to define objects, their states, and the events and conditions under which objects change state. This section and the previous section together have demonstrated that the UML/MOF architecture generally satisfies the ACTIVE OBJECT-MODEL pattern. CWM's extension of UML/MOF into the data warehousing domain provides a semantic foundation for constructing active warehouse models that can be used to dynamically modify the behaviors of a wide variety of data warehousing tools.

Conclusion

Foote and Yoder [FY98] describe an Active Object Model as "an object model that provides 'meta' information about itself so it can be changed at run time" and that this pattern tends to "arise as domain-specific frameworks evolve to address an ever widening range of domain-specific needs". They further state that "A system with an ACTIVE OBJECT-MODEL has an explicit object model that it interprets at run time. If you change the object model, the system changes its behavior" [FY98].

This paper has demonstrated that this description readily applies to models described by the OMG Metamodeling Architecture in general, and CWM in particular. Although CWM was originally intended as a metamodel for describing "external" models of shared metadata, it clearly possesses all of the necessary semantics, interfaces and structure required to formulate Active Object Models for driving data warehouse tools. What is required, of course, is a new generation of tools with architectures based on Active Object Models. This class of tools would represent a tremendous leap forward in the progression from hard-wired architectures to systems that are completely model-driven.

Acknowledgements

I would like to thank both Dan Chang, of IBM's Database Technology Institute, and Doug Tolbert, of Unisys Corporation, for a number of discussions on these issues which have helped shape many of the ideas presented in this paper.

References

[Chan00] Daniel T. Chang, "Common Warehouse Metamodel (CWM), UML and XML", presentation to the Metadata Conference/DAMA Symposium, Arlington, VA, March 22, 2000. http://www.dama.org/

[CWM00] Common Warehouse Metamodel Specification, Volumes 1 & 2. http://www.omg.org/. Also see http://www.cwmforum.org/

[FY98] Brian Foote and Joseph Yoder, "Metadata and Active Object-Models", PloP'98, Allerton Park, August, 1998. http://www-cat.ncsa.uiuc.edu/~yoder/papers/patterns/Metadata/metadata.pdf

[Gran98] Mark Grand, "Patterns in Java, Volume 1", John Wiley & Sons, 1998. http://www.wiley.com/compbooks/

[Tolb00] Doug Tolbert, "CWM: A Model-based Architecture For Data Warehouse Interchange", submitted to the Workshop on Evaluating Software Architectural Solutions 2000, University of California, Irvine, May, 2000. http://www.isr.uci.edu/events/wesas2000/

[JO98] Ralph E. Johnson and Jeff Oakes, "The User-Defined Product Framework", 1998. http://st.cs.uiuc.edu/pub/papers/frameworks/udp

[John98] Ralph E. Johnson, "The Dynamic Object Model Architecture", 1998. http://st-www.cs.uiuc.edu/users/johnson/papers/dom/