To appear in Proc. OOPSLA & GPCE Workshop 2004: Best Practices for Model Driven Software Development.

Managed Evolution of a Model Driven Development Approach to Software-based Solutions

Dan Matheson, Robert France, James Bieman, Roger Alexander, James DeWitt, Nathan McEachen
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523
dan.matheson@comcast.net
{france, bieman, rta, dewittj, mceachen}@cs.colostate.edu

OOPSLA & GPCE Workshop 2004 : Best Practices for Model Driven Software Development

Abstract

The growing complexity of software systems has led to interest in model-based development techniques that raise the level of abstraction at which systems are conceived and implemented. The *Model Driven Architecture* (MDA) initiative from the Object Management Group (OMG) is a well-known example. This approach results in a series of related models. In order to effectively manage the evolution of the abstract models from the early design phases to the more concrete models that can be used to generate code an approach for recording the intermediate models and the transformations is needed. The capture of the models at different stages in the design evolution also provides a place to associate the rationale behind the design decisions. This paper develops a set of requirements and proposes an architecture and design for capturing the evolution snap-shots of the models in Model Driven Development (MDD).

1 Introduction

Modern software systems are increasingly pervasive and open-ended, and are expected to deliver critical services in a dependable manner. Software development approaches that raise the level of abstraction at which software systems are conceived, implemented and evolved can be used to better manage the complexity of developing modern software systems. Work on model-based development approaches that support the use of models as the primary artifacts of software development is based on this premise.

In industry, work on model-based approaches is exemplified by software development tools and techniques that claim to support the Object Management Group's (OMG) vision of model-based development as framed by the *Model Driven Architecture* (MDA) [MDA2001]. The MDA advocates an approach in which models of software that abstract over technology-specific details are systematically transformed to deployable technology-specific implementations.

The transformations (e.g., refinement and refactoring transformations) are embodiments or collections of design decisions or technology specific bindings. While the models have specific relationships between them, the transformation steps are often a discontinuous jump forward rather than a specifiable mechanical transformation. The discontinuous jump is most noticeable in the early stages of design refinement and solution exploration where fundamental decisions are made by the software engineer in order to satisfy solution requirements.

One of the main strengths of any model-based approach is the ability to investigate the possible solution space of a problem with a low investment of time and effort. One can see similar processes at work when an architect designs a building. Rough sketches are used at first to gather and communicate back to the owner the fundamental building requirements. The sketches are refined and supplemented with other models to reaffirm the fundamental requirements, as well as to create more detailed specifications. Eventually a set of blueprints, materials lists, and a construction timeline is produced that allows the craftsmen to build the structure. As a change occurs a specific model or specification is affected and to a greater or lesser extent corresponding changes are propagated to other models and specifications. The building architects and general contractors have well defined mechanisms and processes for tracking changes and their impacts.

Just as the building architect must keep track of all the models and decisions, productive use of MDA and other model-based approaches require support for tracking and managing changes to models. The tracking mechanism needs to be independent of the models since it records the context of the evolution. The evolution context is the set of transformations, decisions, tool instructions and process steps that move one or more source models to target models. The tracking mechanism is effectively implemented via a repository containing the model artifacts, the evolution context and the relationships.

One of the characteristics of a good engineer is to learn from the experiences of his fellow engineer. In this spirit we look to the experiences of mechanical and electrical engineers in their product development exercises. These discrete manufacturing (DM) industries have faced similar problems related to evolving both single part designs and assembly designs. The solution developed for this industry and automated from established manual activities is a repository approach called Product Data Management (PDM) [PDM2000, STEP1999]. If the models of a MDD approach are looked upon as assemblies of solution elements then many of the principles and ideas of PDM can be applied to the MDD model evolution problem.

Section 2 gives a more detailed description of some of the problems and situations a MDD approach generates and extracts some of the important the requirements that need to be addressed. Section 3 proposes a repository design that addresses the requirements. Section 4 concludes with an overview of how we plan to extend and implement the ideas presented in this paper.

2 Model Driven Development

This section describes several different MDD approaches and illustrates the many common problems and requirements they have. Experiences from actual manual application of both MDA and Aspect-Oriented Modeling (AOM) approaches are also cited. At the end of this section related work is presented.

2.1 Model Driven Architecture

Systematic support for model transformations is considered to be key to the success of MDA. A transformation defines changes that are applied to a source model to produce a target model [Judson2003]. The changes defined by a transformation can be classified as being vertical or horizontal [FranceBieman1992]. Design refinement and detailing of model elements are examples of vertical model transformations. In a vertical transformation the source model is either more abstract or more specific than the target model. Changing a model to correct a design error or to enhance design quality provide examples of horizontal model transformations. In a horizontal transformation the source and target models are at the same level of abstraction. The MDA emphasizes vertical model transformations in which models describing designs in platform independent terms are systematically transformed to platform specific designs. Each refinement step is a binding or decision to utilize a particular construction technique or technology in the solution.

The refinement of the models takes place over a period of time. If the entire solution or product lifecycle is considered, the time frame can be decades, involve several technology revolutions, major functional extensions and people turnover. A product that one of the authors started designing in 1982 is still in production today. The ability to retain abstract models of the design for education of new software engineers or the ability to show when, where and why a particular design decision was made or a certain technology was applied for maintenance purposes is highly valuable.

Experience Applying MDA

A recent attempt (started in 2002 and still ongoing) to apply MDA ideas to a product design problem revealed a number of shortcomings in tools, process and communication of understanding between the software engineers. The problem was a re-design of common data models across a suite of cooperating, but independent system and network management products. A major complaint of many customers was the need to re-enter essentially the same data when an additional product was installed. The goal was to align and normalize the common data across the product line with minimum disruption to the release schedules and version compatibility.

The UML tools used for creating design models, primarily UML class models, were effective for producing a single model at one point in time. Trying to compare a refined model with its more abstract predecessor required the use of two computers and two projectors and a text based narrative of the design change rationale. This was possible in the face-to-face meetings, but as travel became more restrictive alternate techniques were necessary,such as the creation of larger documents. In addition the shared source code tool was inadequate for handling the binary output of some of the tools.

The individual processes and the team processes required extra effort. In part the extra effort was because of tool shortcomings listed earlier. A larger part of the extra process effort was because several disjoint tools needed to be used to document both the model and the changes proposed. The processes, mostly ad hoc, included education tasks, many of which were discovered only as mis-communication was uncovered. The education tasks covered UML basics, UML documentation and diagram conventions, MDA concepts and domain knowledge exchange. Often these tasks needed repeating as team membership fluctuated.

Uneven experience and skill with UML modeling, design intent communication, and domain knowledge were the major sources of the communication problems. Since the UML models were distributed via email over a space of nine time zones, the comments and improvement suggestions became uncoordinated. An additional factor was the focus of different R&D teams on the parts of the model that most concerned them. The result was that teams and engineers were often reviewing out-of-date models or comparing models with different sets of changes.

Requirements from the MDA Experience

There are several actions and tools that would have prevented or reduced the problems encountered during the attempt to use MDA as described above.

To help with coordinating the many models that needed to be developed a central repository that could be accessed across all the time zones is needed. A repository would eliminate the email distribution, provide tracking of access and allow development of the different models at different speeds. There are several requirements on such a repository (including, but not limited to):

- Store and retrieve several different types of information, from UML models to text and code.
- Store and retrieve several different formats of information, including binary.
- Relate the different types of information.
- Control access to subsets of the information.
- Record who made a change.
- Record why a change was made.
- Notify that a change was made.

- Propagate a change.
- Use different tools on the same information.
- Maintain several versions during solution exploration.

2.2 Aspect-Oriented Modeling (AOM)

Aspect-Oriented Modeling (AOM)[Clark1999][France2004] is a design approach that allows developers to focus on addressing design concerns separately. AOM can be seen as one variety of a MDD approach. These solutions are composed to create a comprehensive design. In the AOM approach that we developed, an aspect-oriented design consists of a primary model that describes the primary functional structure of a design, and a set of aspects that each describes a crosscutting solution that addresses a design concern [France2004]. UML diagrams are used to describe solution views specified by primary models and aspects. Composing aspects and a primary model together result in an integrated view of the design.

Aspects and primary models are developed separately and thus composition involves mapping concepts described by aspects to concepts in the primary model's application domain [France2004]. The mappings are not properties of any of the models being composed. It can be viewed as part of the composition context.

After the models are composed, the result is tested and analyzed. There are often emergent properties that are discovered as a result of the model composition. Some of these properties are acceptable and some require a redesign of either the aspect or the primary model. In some cases the composition directives need reworking. After making the changes the models are composed and again tested and analyzed.

AOM can be viewed as an MDA approach that adds support for separation of concerns that are non-orthogonal to the concerns used to determine the functional structure of a design. These non-orthogonal concerns are said to crosscut the primary design structure.

Requirements from an AOM Approach

The ability to manage several models along with composition directives and keep them properly synchronized is a primary requirement for a successful AOM approach. There is a need to be able to version each of the components separately while still maintaining the synchronization.

The management of the related models and the composition mapping is effectively implemented via a repository containing the models and the composition context. While it was not described above it is fair to assume that

multiple people are involved in a design approach using AOM. The requirements supporting an AOM approach include:

- Storing and retrieve several models and composition information.
- Relate several different information types.
- Control access to subsets of the information.
- Record who made a change.
- Record why a change was made.
- Notify that a change was made.
- Propagate a change.
- Maintain multiple versions of the models and composition directives.

2.3 Related Work

A small number of researchers are currently developing AOM approaches. In the approach proposed by Clarke et al. [Clarke1999] a design, called a Theme, is created for each system requirement. A comprehensive design is a composition of subjects. Subjects are expressed as UML model views, and composition merges the views provided by the subjects. As part of the Early Aspects initiative, Moreira et al. have targeted multi-dimensional separation throughout the software cycle [Rashid2003]. This work supports modularization of broadly scoped properties at the requirements level to establish early trade-offs, provide decision support and promote traceability to artifacts at later development stages.

The Query View Transformation (QVT) work in OMG is aimed at defining a specification for the transformation of one model into another model [OMGAD040401]. The QVT RFP seeks a standard solution for model manipulation. Queries take as input a model, and select specific elements from that model. Views are models that are derived from other models. Transformations take as input a model and update it or create a new model. It is important to note that queries, views and transformations can be split into two distinct groups. Queries and transformations take models and perform actions upon them, resulting in a new or changed model in the case of a transformation. In contrast, views themselves are models and are inherently related to the model from which they are created. Queries and transformations may possibly create views, but views themselves are passive.

The OVT work will need storage capabilities that maintain the relationships and data. Our repository work should be able to support a QVT environment.

3 A Repository Centric Solution

It is clear that the control and manipulation of multiple models in a MDD or AOM approach requires a structure outside the models in which to store the relationships and constraints between the models. In this section we give an

overview of the repository approach. There are many common requirements between the MDA example and the AOM work described above. There are also a few differences. The repository approach should be flexible enough to cover all the requirements as well as future requirements.

3.1 Deficiencies of Current Approaches

The current IDE (Integrated Development Environment) like approaches to model development are centered on a single closed model and have evolved from a code development cycle (edit, compile, debug). The code development cycle is deeply embedded into the IDE interaction model and architecture. This approach was often seen in early Mechanical Computer Aided Design (MCAD) tools, but changed over time as the restrictions of processes embedded in the tool became obvious and a hindrance.

Several current products that claim to support MDA have a repository component. However, the repository is tightly coupled to the tool and unavailable to other independent tools. Several of the products have interfaces for extension of the repository, but only in the fixed context of that product.

A major strength of our repository approach is the ability to record and manage the different roles a model and model components play in a tool neutral format. The model information is recorded in fine granularity in the repository greatly enhancing reuse. The repository also supports extensibility as a basic capability so that new relationships and information can be added independently from the current model definitions.

3.2 The Repository Design

The repository design starts with a responsibility architecture that describes the fundamental components and their relationships to each other. Each component is described in terms of responsibilities, constraints and abilities. The repository responsibility architecture is shown in Illustration 1.

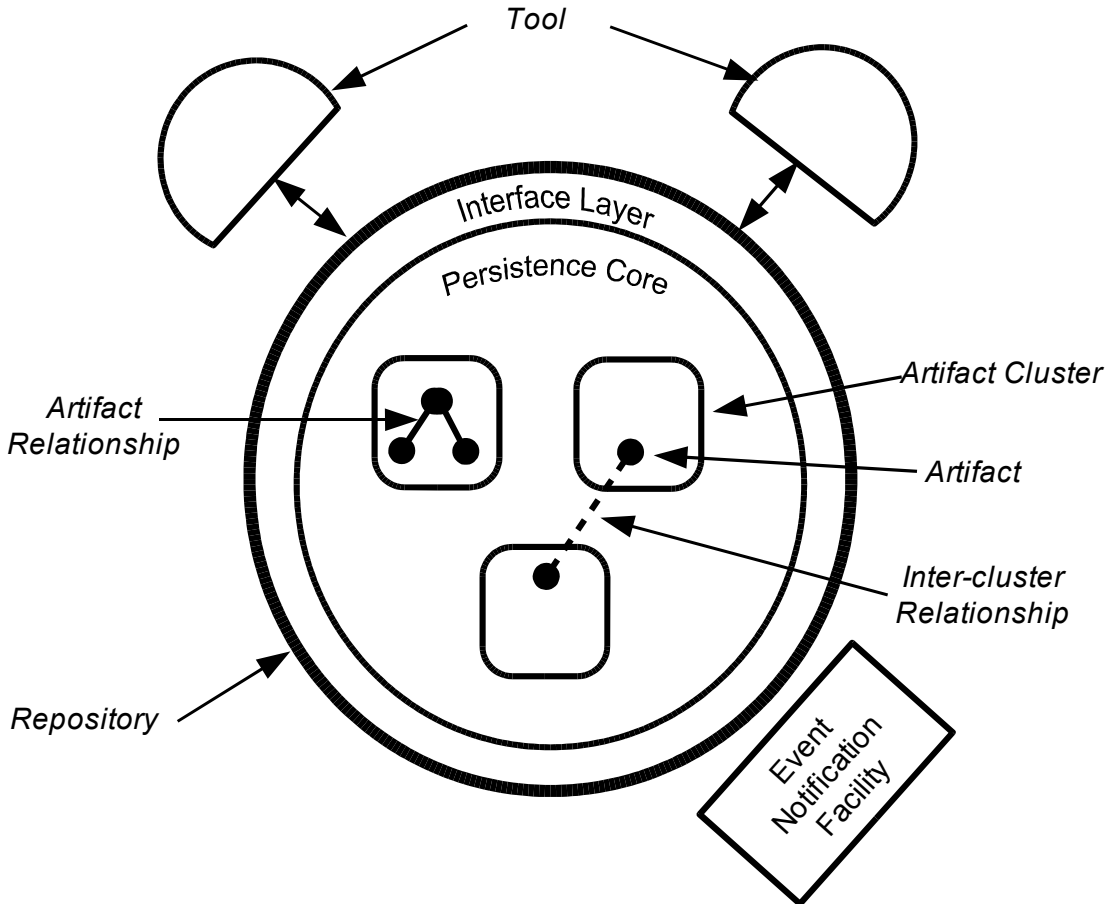

Illustration 1 Repository Responsibility Architecture

The major components of the responsibility architecture are the *Repository*, the *Tools*, the A*rtifact Clusters*, the *Artifacts*, the artifact *Relationships* and the *Event Notification Facility*. Each of these is explained in detail in the following sections.

The Repository

The primary responsibility of the repository is to provide the *basic persistence* and *integrity* of the stored artifacts. The ACID (atomicity, consistency, integrity, durability) properties of the CRUD (create, retrieve, update, delete) actions on the artifacts must be supported.

As there is a requirement to communicate with several different tools, both current and future, a data exchange mechanism is needed. Experience from IGES (Initial Graphics Exchange Specification) [IGES1995] and STEP (Standard for the Exchange of Product Model Data)[STEP] standardization efforts has shown that a standard neutral format, covering syntax and semantics, is very effective for a multi-product open tool environment. The current suggestion for this is XML Metadata Interchange (XMI)[XMI2003] for UML model artifacts.

However, XMI does not cover all the data to be stored in the repository. Using XML and XML Schema as a standard along with a mechanism for transformations will fill the requirement. As the other data to be stored is defined the need to support additional standards will appear. An alternative is extension of the XMI standard for some of this data or the development of new more specific data exchange standards.

As the purpose is to support model evolution the repository needs support at a fundamental level for versioning of artifacts. There is a need for maintaining several active versions of an artifact. The PDM Enablers specification [PDM2000] specifies a model for achieving this requirement and will be explained in more detail later in this section.

The relationships between artifacts must be maintained with the same integrity as the artifacts themselves. Versioning of relationships and the existence of several types of relationships between artifacts in parallel must be maintained.

It is a goal of this approach to work with as broad a spectrum of tools as possible. In order to achieve this the repository will need to support several different communication protocols to the tool applications.

Both for effective initial incremental development and to keep the repository viable for the future it is necessary that some basic extension mechanisms be built into the repository solution. A partial list of extensions that will need to be supported are:

- New artifacts
- New artifact clusters
- Changing a cluster or artifact
- Changing inter-cluster relationships
- Adding and modifying data transformations
- Adding and modifying data integrity rules

In order to meet the requirements on the repository in a manner that provides flexibility for development and maintenance the repository is split into two major sets of responsibilities, a Persistence Core and an Interface Layer. The persistence core encapsulates the capabilities related to persisting the artifacts and is easily seen as a type of database. The interface layer contains capabilities for communicating via a variety of protocols with the tool applications.

Artifact

The artifact is the atomic building block of the repository. The artifact represents a set of data or attributes so tightly related that it makes no sense to further decompose it. This does not mean that all values that could be present in the

artifact are set. The repository needs to support the ability to have incomplete, but stable artifacts so that interruptibility of work is possible. The interruption of work comes from extended development tasks, as well as from goals that require the use of several tools and decisions to complete. A partial list of artifacts includes:

- UML class
- System requirement
- Use case
- Java source code
- MDA PIM to PSM mapping
- AOM composition rules

Relationships need to be built between artifacts. For effective extensibility any kind of relationship can be built. However, the data integrity rules and access control rules might prevent when and how the relationship between two artifacts is created, updated or deleted.

There is a complex artifact. A complex artifact is a higher level artifact representing a number of simple artifacts in a specific relationship. For example a UML model is a complex artifact with simple artifacts for the classes, associations between class and the repository links bonding the simple artifacts together. A partial list of complex artifacts includes:

- UML class model
- UML sequence model
- Java source code and code analysis

Artifact Cluster

The artifact cluster collects a set of artifacts and relationships between the artifacts that satisfies a logical relationship requirement. The artifact cluster is a collecting or organizing entity that allows higher level abstractions and groupings of artifacts to be created. It is one of the fundamental extension mechanisms of the persistence core.

At first glance there is similarity between the artifact cluster and a complex artifact. The intent is different. The artifact cluster is intended to be used for a human view of some collection or organization of artifacts, while the complex artifact is intended for real composite structures. For example an auto would be a complex artifact, while the auto dealership would be a artifact cluster.

To maintain the integrity of a artifact cluster or a set of artifact clusters a set of data integrity rules are needed. These rules go far beyond the low level ACID types of integrity checking, in that they can involve data other than that being changed and complex calculations.

A partial list of artifact clusters includes:

- Project representation
- Tests and test results
- A person's work
- A solution design experiment

Relationships

The relationships between artifacts whether inter-complex-artifact, intra-cluster or inter-cluster are handled with the same data integrity goals as the artifacts. There are manual, dependency and structural relationships. The structural relationships are formed when creating or storing an entity with structure into the repository such as a UML class model.

The manual relationships are the ones that have been explicitly created by a user or via a tool application. The manual relationship could be the consequence of a higher level action that reflects a dependency relationship such as MDA refinement or relating an analysis result to a specific version of a model. There can be many types of manual relationships and the type plays a role in the propagation of changes.

The dependency relationships are created when storing the results of some tool into the repository. For example an AOM composition tool composes an aspect model with a primary model and the resulting composed model is stored in the repository. In this case a dependency relationship could be created between the input artifacts and the output artifacts. There could be a number of dependency relationships created depending on the granularity needed.

Event Notification Facility

The Event Notification Facility receives events from the repository and distributes them to interested parties. An action (Create, Retrieve, Update, Delete) to any artifact potentially generates an event. There are administrative controls that can limit the actual event generation to keep volume to a reasonable size. This facility exists to fulfill the notification and change propagation requirements.

An event from a model update action might cause various analysis tools to be run to check the model or to propagate the change to other artifacts in the repository. The decisions and the steps in the process are evaluated outside the repository. This gives the greatest amount of flexibility and makes the event response actions and processes orthogonal to the repository and tools.

Tool

The tool represents any software application that creates, deletes, updates or reads data from the repository. Some possible tools are a UML modeling tool like ArgoUML, an AOM composition tool, a UML model analysis tool, a UML to programming code translation tool, a viewer (browser-based), etc.

A tool can access any combination of artifacts that make sense for it to do its job. It could access all or part of the artifacts, an entire artifact cluster or all or some of the artifacts from multiple artifact clusters. The access mode could be read/write or read-only. The amount of data involved is controlled by the granularity of the artifact requested, data integrity rules of the interface layer and the type of access action. In most cases accessing the highest encapsulation of a structure involves all the components of the structure.

Tools are not part of the repository, but rather interact with it. It might make sense to have a representation of a tool in the repository so that it can be associated with a particular artifact. For example there could be a representation of two different Java compliers to be associated with different byte code results.

Tool Repository Interaction

At the highest logical view the tool repository interaction is message based. The tool sends a message to the repository with the action desired and the data necessary to carry out that action. Interpretation of the action takes place in the Repository Interface Layer. Illustration 2 shows a more detailed view of the interaction.

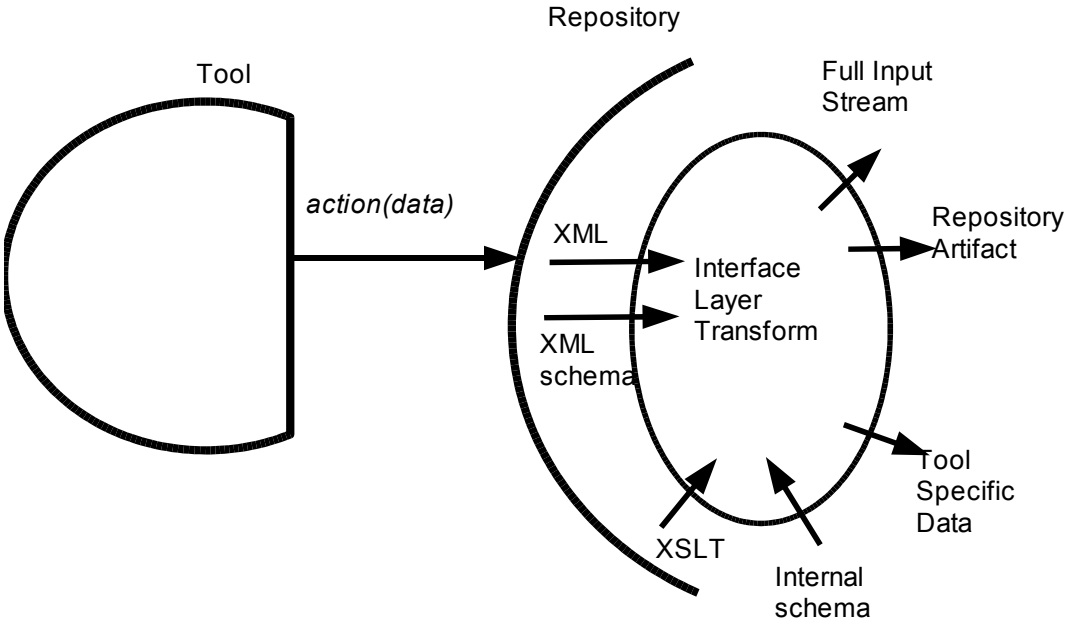

Illustration 2 Tool Repository Interaction

The tool sends an action request to the repository with the data needed. The data consists of an XML or XMI document and a pointer (URL) to the XML Schema describing the document.

The first action of the Repository Interface Layer is to validate the input and the second to parse the data and retrieve the schema. This information is sent to a transformation function for the third step.

The transformation function has the incoming data, usually in XMI format, along with the XMI schema which it combines with the repository internal schema and an XSLT transformation to produce repository artifacts and tool specific data chunks. The tool specific data chunks are related to the repository artifacts. Additional integrity checks can be performed after the transformation.

The output of the transformation is passed onto the action routine to carry out the requested action. Output and status of the action might be transformed on the way back to the tool. To provide for future extensions the full input is saved so that re-parsing after adding a new application is possible.

Repository Object Meta-Structure

Illustration 3 shows a high level view of the principle types of data entities within the repository. This is the highest level of abstraction for the principle entities,

administrative entities are ignored for clarity at this time. The various artifacts and artifact-relationships will need to be created, managed and searched.

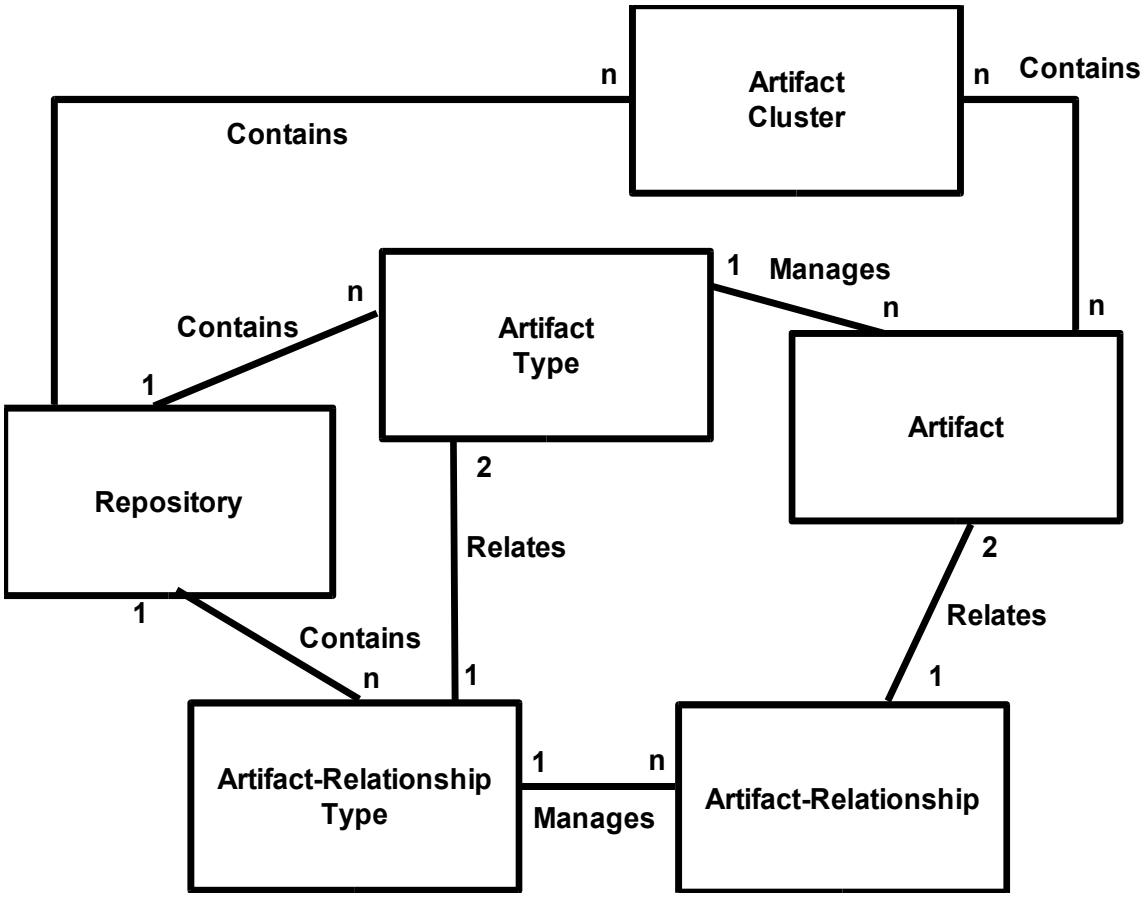

Illustration 3 Abstract Artifact Structure

For each artifact there is an **artifact type**. The artifact type represents the prototype for creating a new instance of that type of artifact. The artifact type can also be combined with a **Finder** capability, so that the **Factory-Finder** becomes a well-known object. The abstract artifact structure discussion reflects a creation approach that uses the **Prototype** pattern from Gamma [Gamma1995].

To achieve the extensibility requirements new artifact types need to be created. The pattern used to create artifact from artifact types can be repeated. There will be a well known object, **Artifact-Type_Factory**, that is a factory-finder for the creation of new artifact types. The prototype factory pattern is used here also. This object is also a principle object in implementation of the query capability.

Illustration 4 represents the factory and finder relationships between classes needed to support the creation and management of artifacts, as well as the basis for extensions.

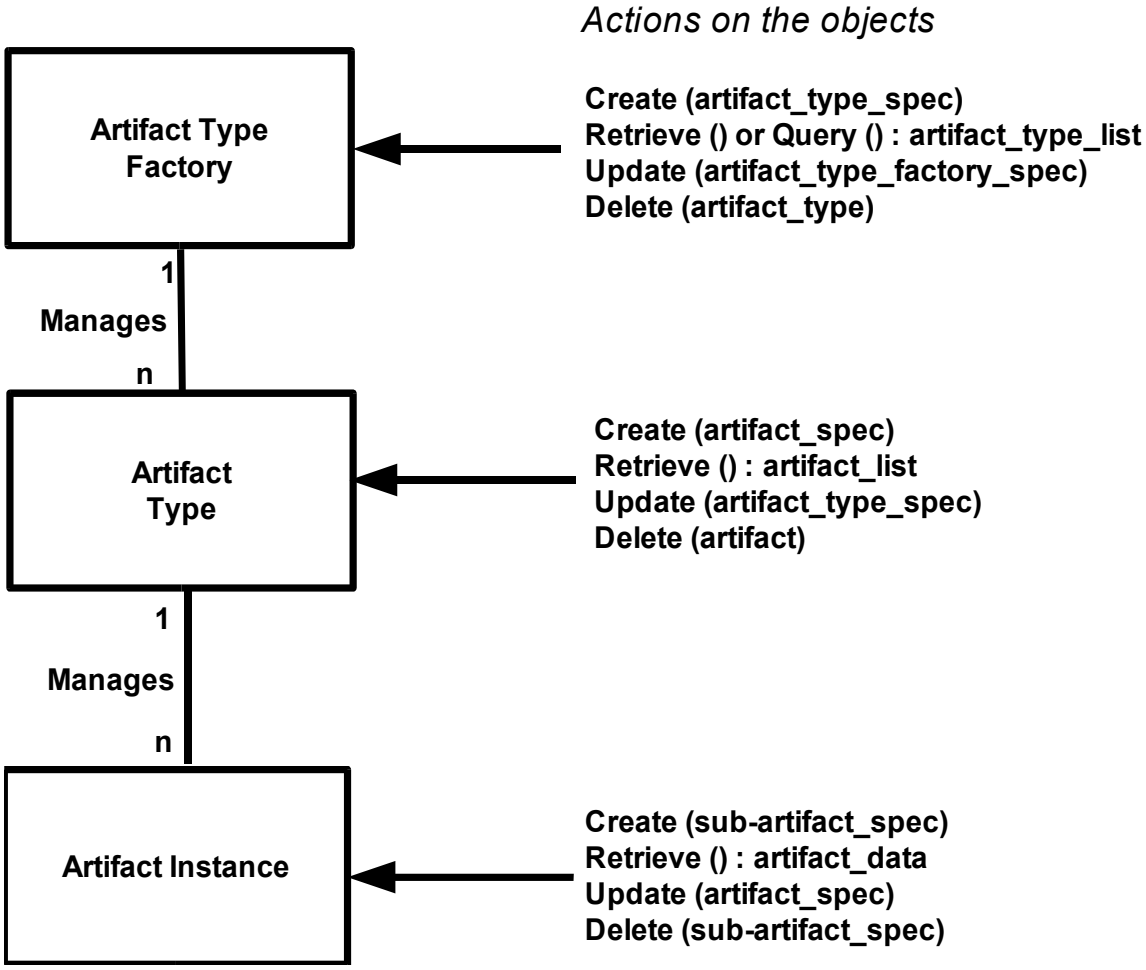

Illustration 4 Factory Pattern Structure

The basic CRUD actions on each of the entities is listed in Illustration 4. The notation like *Retrieve() : artifact_type_list* indicates the return type of the method. The *Artifact_Type_Factory* is a well-known object that always exists. A factory object uses the Prototype factory pattern to instantiate the instances it manages.

Structure and Versioning

There are requirements to support both the versioning of an artifact to track changes and the structuring of artifacts to create complex artifacts. The modeling solution ideas picked to support both of these goals come from OMG PDM Enablers specification[PDM2000]. Illustration 5 shows the core interface pattern from the PDM Enablers that is used to support versioning and

structuring. Attributes and methods are not shown to emphasize the relationships.

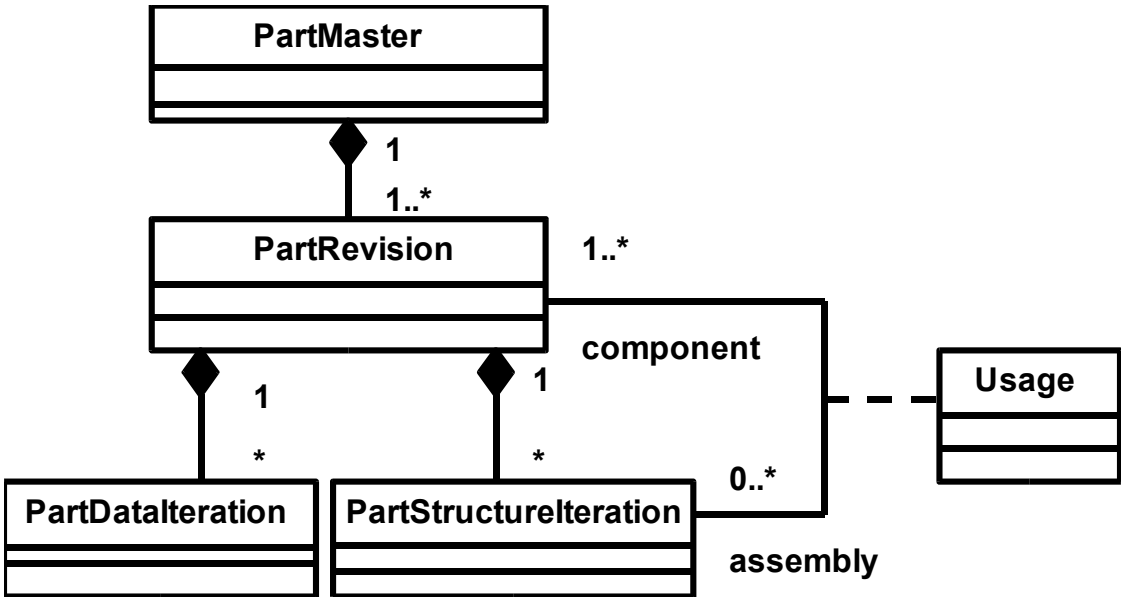

Illustration 5 Basic Part Structure

Each artifact, whether simple or complex, will be mapped onto this structure.

The **PartMaster** interface represents the unchanging attributes and characteristics of a part (artifact). This does not mean that some of the attribute values will not change over time.

The **PartRevision** interface reflects the controlled or approved changes to the part (artifact) definition.

The **PartDataIteration** interface is used to hold the definition of a simple part (artifact). The multiple iterations under a revision allow for the saving work-in-progress or design explorations.

The **PartStructureIteration** encapsulates the structure of a complex part (complex artifact or artifact cluster). The structure iterations under a revision serve the same purpose as with the PartDataIteration.

The **Usage** association class allow the role of the sub-part in the assembly to be qualified.

The **PartRevision**, **PartDataIteration**, and **PartStructureIteration** interfaces match the composition pattern from *Design Patterns* by Gamma [Gamma1995].

Experience by 3 of the authors applying commercial PDM products in a variety of solutions gives us confidence that the concepts can be successfully applied to a repository of software and model artifacts. The details of applying the PDM concepts, especially those beyond versioning, are beyond this paper. We will report on those experiences in a future paper.

4 Conclusions and Future Research

The similarities between the problems faced by the discrete manufacturing industry in managing the electronic information about parts and assemblies and the problems facing a model driven design approach are obvious. By leveraging the experience of our engineering colleagues and applying the lessons learned to our software technology environment many problems we face can be overcome.

There are many advantages to a PDM based repository approach for managing the model evolutions.

- Many of the basic questions in managing model changes can be answered.
- The structures provide many extension possibilities.
- The repository ensures the integrity of the models via its ACID (atomicity, consistency, integrity, dependability) capabilities.
- Allows for access to the model at different levels of abstraction or detail.
- Tool neutrality.
- Building upon existing solution experience.

Future Research

There remains work to be done. Through the construction of use cases or scenarios the desired extensions to the basic repository artifact structure can be explored. Some of the use cases are listed below:

- Store refinement decisions and rules for MDA to record the rationale.
- Indicate the role played by an element of an aspect and an element of the primary model.
- The AOM composition rules to be used in a specific case.
- Tracing capabilities to decompose situations where an aspect changes a property of a method, like public to private or a property of an association like the cardinality.
- The structure pattern and the indicators for an analysis tool to store suggested changes.
- Tagging of entities and values to distinguish between human selected changes and tool constructs for limiting propagation of changes.

What is the best granularity and representation for the atomic entities in the repository? An approach to solving this question is to experiment with different construction and analysis tools operating on the information. This should lead to the forms that require the fewest transformations from the repository format to the tool format. These experiments should help determine the optimum granularity for the atomic entities, such as class or association.

What is the best repository structure to support other UML models such as interaction diagrams and state diagrams? The answer to this question should be derived from both construction process activities, as well as an analysis activities.

There are questions about what traceability data is needed to explain the rationale behind a particular design decision.

We will attempt to answer these questions in our future work via a prototype currently under development.

References

[Clarke1999] S. Clarke, W. Harrison, H. Ossher, P. Tarr. Separating Objectives Throughout the Development Lifecycle. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Springer, 1999.

[FranceBieman2002] R. B. France, J. Bieman, Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software, Proceedings of the International Conference on Software Maintenance 2001, 2001.

[France2004] R. B. France, I. Ray, G. Georg, S. Ghosh, An Aspect-Oriented Approach to Design Modeling, to be published in IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2004.

[Georg2002] Geri Georg, Indrakshi Ray, Robert France. Using Aspects to Design a Secure System. In *Proceedings of the International Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, ACM Press. December 2002.

[Ghezzi2003] C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, 2nd Edition, Prentice Hall, 2003.

[MDA2001] www.omg.org/mda

[PDM2000] OMG Product Data Management Enablers formal specification http://www.omg.org/cgi-bin/doc?formal/2000-11-11

[STEP1999] STEP and OMG Product Data Management Specification, A guide for Decision Makers http://www.omg.org/cgi-bin/doc?mfg/99-10-04

[Gamma1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, Prentice Hall, 1995.

[Judson2003] Sheena Judson, Robert France, Doris Carver, Supporting Rigorous Evolution of UML Models, to be published in the Proceedings of the International Conference on Engineering Complex Computer Systems 2004 (ICECCS 2004), 2004.

[Rashid2003] Awais Rashid, Ana Moreira, João Araújo, "Modularization and composition of aspectual requirements", *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, 2003.

[IGES1995] Initial Graphics Exchange Specification 5.3, www.uspro.org/ or www.nist.gov/iges/

[STEP] Standard for the Exchange of Product Model Data, ISO 10303, www.uspro.org/

[XMI2003] XML MetaData Interchange Specification version 2.0, OMG formal/03-05-02, May 2003

[OMGAD040401] Revised submission to the QVT RFP, access limited to OMG members.