Using XML/XMI for Tool Supported Evolution of UML Models[1]

Frank Keienburg, Andreas Rausch
Technische Universität München, Arcisstr. 21, D-80290 München, Germany
{keienbur|rausch}@in.tum.de

Abstract

Software components developed with modern tools and middleware infrastructures undergo considerable reprogramming before they become reusable. Tools and methodologies are needed to cope with the evolution of software components. We present some basic concepts and architectures to handle the impacts of the evolution of UML models. With the proposed concepts a infrastructure to support model evolution, data schema migration, and data instance migration based on UML models can be realized. To describe the evolution path we use XML/XMI files.

1. Introduction

Technical innovations in communication and information processing, permanent organizational changes, international business networks and virtual organizations lead to a new business competition landscape. Today's development of new products takes place under immense time pressure. Ever shorter technology time cycles lead to ever shorter product life cycles and shorter development time cycles. "Time-to-market" has become one of the most important success factors for new products to survive this competition. The question is, how can you shorten product development time to be successful at the market. Modern concepts of software engineering should support this improvement process. In the last decade, the object-oriented paradigm gained a great success covering almost all steps of software development and it's life cycle.

Concurrent engineering in software engineering shortens software development time. Thereby, the traditional sequential development process with its consecutive steps requirements analysis, design, implementation, quality insurance and service is replaced by a more concurrent one. This process should be iterative, incremental and therefore more cyclic than the old one. As soon as possible the results from earlier phases should be passed to later ones. This leads to a nearly parallel and therefore shorter process with early information exchange between the different phases, but also to changes in the original development model and the related data based on it. The new software development process and the supporting tools should cover all steps of the software life cycle and therefore also model changes.

Another demand of modern software engineering to shorten development time for being successful at the market, is to support development with regards to prior product versions and components with the help of reuse and componentware. These concepts differ to starting every time from scratch. For a efficient development process you need a tool that supports the creation of new software based on older versions and components. As an additional benefit, components can help to avoid software redundancies and provide interoperability if they are used by more than one software application.

Modern programming concepts like Java Enterprise Beans [1] or CORBA Components [2] and corresponding middleware infrastructures implementing these concepts offer a rich support of reuse and component based development. But, currently they do not support model changes and model evolution at all. This should also include data migration based on old models to new models and data access with code based on different model versions, or shortly preserving persistent data across schema changes. Technically this means to support schema evolution and data migration.

This paper provides some concepts and design solutions of a tool supporting schema evolution and data migration. In the fist section we introduce the user requirements of a tool for model evolution. We show how this tool should fit into modern middleware infrastructures like Java Enterprise Beans. In section three we show a simple model of component based applications. With this model we allow runtime mapping from various interfaces to a single implementation. The next section discusses the different entities of UML models and the operations an designer can perform on these models [4,5,6]. Based we are able to come up with the architecture of a tool supporting schema evolution and data migration. In the next two next sections we discuss four different kinds of change primitives and some problems to be solved with the not cleanly manageable model changes. Finally, we provide concepts for developers to specify the model

[1] This work originates form the research project *FORSOFT, supported by the Bayerische Forschungsstiftung*.

evolution based on XML and XMI [11]. A short conclusion rounds the paper up.

2. What You Need is What You Get

Modern distributed systems are based on a 3-tier or multi-tier client server architecture. A 3-tier architecture mostly consists of client (tier 1), application server (tier 2) and database server (tier 3) [13]. The right side in Figure 1 shows such an application. A popular approach to build such a system is using Java Enterprise Beans [1], CORBA [3], or DCOM [12]. Because development of such a system is very complex it's helpful to use tools that support the development process.

Developing a distributed system means usually to perform the following steps [14]:

- Create an application model with a CASE tool.
- Generate interfaces of the distributed components based on the model.
- Implement the application server according to the interfaces.
- Realize persistence for the application server instances.
- Implement the client applications.

have developed our own tool: a simple environment called AutoMate to easily generate 3-tier applications from UML models [9,10]. Using AutoMate you can concentrate your development on implementing the server functionality and the client code. Everything else is automatically done for you according to the class model. AutoMate generates IDL interfaces, client proxies, server code and adds the whole database functionality including transaction logic and other things, as Figure 1 shows.

As already mentioned, tools like AutoMate relieves you of programming application code, database access, and some more standard work. But model evolution is not supported in these tools.

At the moment after changing a class model, new code overwrites old code, a new database schema overwrites the old one and old object instances are deleted. This means only clients based on the newest model version can for example create or select persistent objects. All previous work is lost, you can't access persistent objects with an old client version anymore. Every time you change your model you have to start from scratch again. But, model changes are quite usual in a concurrent development environment and every time starting from scratch again is not very efficient. That's the reason why

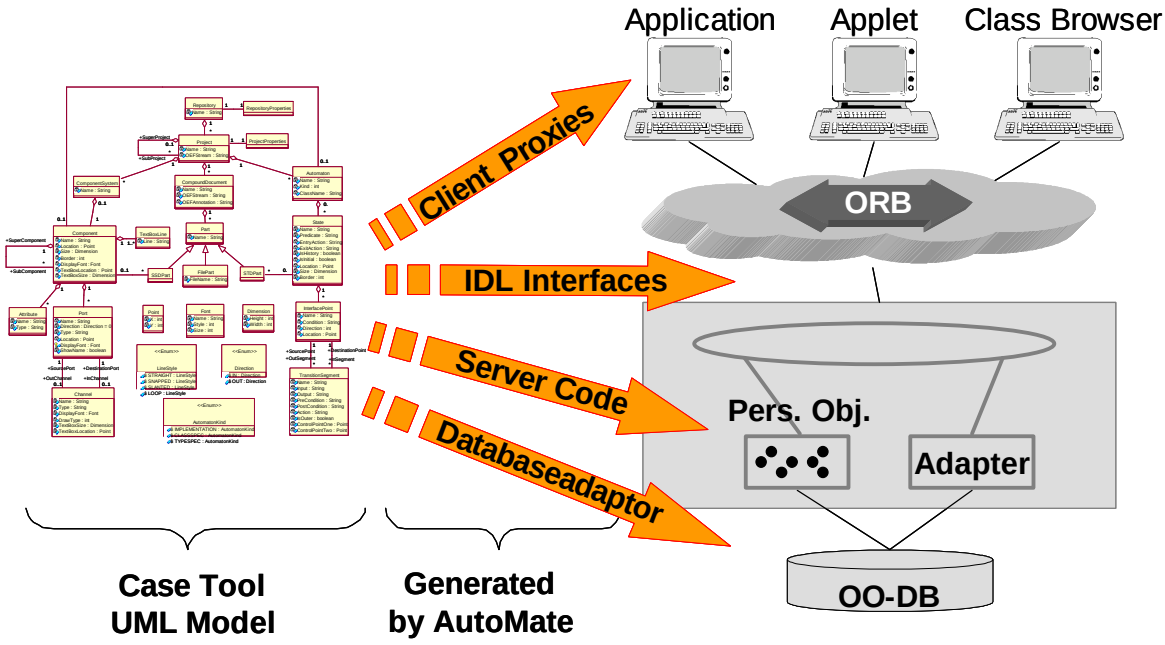

Figure 1: The Development Process with AutoMate

Some of these standard work can be automated by a tool to become a continuous development process based on a consistent model. Nowadays there are many commercial tools available like for instance various application server. For our own research activities, we

schema evolution is a useful extension to existing middleware infrastructures like for instance AutoMate.

To cover the hole model life cycle from analysis to test you have to ensure consistency. This consistency can be divided into static and dynamic aspects. You have to

maintain both, static aspects which are dealing with keeping application code consistent and dynamic aspects which are dealing with keeping object instances and their behavior consistent. In AutoMate or in any other tool with an integrated database support this means especially to perform object instance changes and conversion.

Static aspects deals with the definition of classes including it's attributes, method signatures, types and inheritance graphs and the static relation between such classes. The framework has to ensure that no type or interface inconsistencies occur.

After a UML model has changed you have to update your code that has already been generated by the tool, e.g. with AutoMate. This means for a three tier architecture realized with CORBA, you have to adapt IDL interfaces, client stubs, server code and database access in a way that clients based on old and new model version work together with your database and behave consistent over their whole life cycle.

Dynamic aspects concern the runtime behavior of instances when clients proceed method calls on them. These client calls based on a specific model version have to deliver the same results (behave consistent) over the hole model and application life cycle.

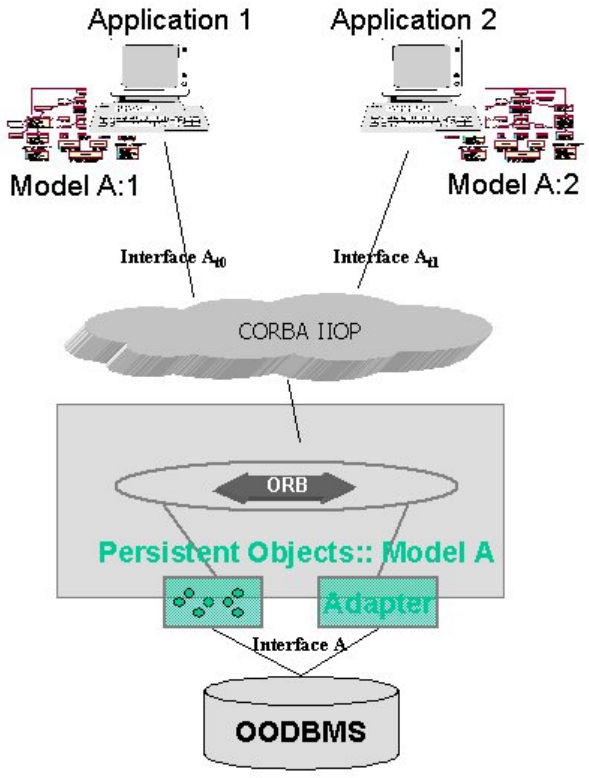

Figure 2: System with Different Model Versions

Persistent objects and the database schema are related to a specific code version. That's just why code changes cause also database schema and object instance changes. Both should be reorganized in a way that the data is consistent and accessible with client code based on any model version.

To clearly and easily explain the requirements of model evolution in a system like AutoMate, the desired model evolution behavior is introduced. In the beginning a model is introduced that changes over time, these change means a change of a special interface. After this change there are two versions of the model and accordingly two versions of the interface.

The desired behavior should cover how model evolution is handled in the future system from the client application point of view. The illustration shows a distributed system that is build on the before introduced model versions.

This software system is based on a three tier architecture, which may be created with AutoMate. The presentation tier of this system architecture consists of client applications build on basis of interfaces from different model versions of model A. The second tier consists of CORBA servers for the different interface versions and the third tier consists of a object oriented database with persistent objects based on a general interface A and the necessary wrappers that delegate the work to the general interface.

3. Components, Evolution and Runtime Mapping

As already mentioned software systems are usually very complex. For reasons of reuse and concurrent engineering these systems are partly constructed of components. A component client communicates and interacts with a component via it's interface. Such components with it's belonging set of interfaces can be modeled like shown in the UML class diagram in Figure 3. A Component is constructed according to the composite pattern [7] and is a single implementation or a compound of other components.

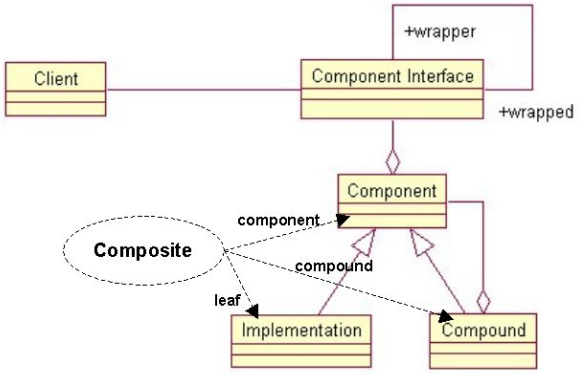

Figure 3: A Component Model

Normally development of components is a iterative and incremental process as surrounding conditions of the system or it's desired behavior changes during it's life cycle. Therefore a typical development scenario for a component with the help of a CASE tool can be like this: At the beginning of the development process a UML model of the interface will be designed. Based on this interface model the implementation will be realized or generated. In the later life cycle of the component, it's interface model will be changed and the implementation has to be updated accordingly.

This causes the necessity to support automatic component interface changes. Most of the time it is not possible or desired to update all applications that already use an existing component at the moment the interface of a component changes. The resulting problem is to handle more than one component interface of a single component. With the necessity to support component evolution a problem of interface incompatibility is born and the motivation to solve this problem with the help of schema evolution is introduced.

As a solution approach each component provides a set of interfaces mapping to it's versions, but only one actual implementation. This approach is modeled in Figure 3. The latest version of the component interface exactly corresponds to the actual implementation, all other interfaces are wrappers or adapters [7] that encapsulate the functionality of the component and provide translation and delegation. Each time a component model changes a new interface version has to be generated, the implementation has to be changed accordingly and the old interfaces have to be converted into wrapper to the latest interface.

Figure 4 explains how a wrapper or adapter for components can be modeled according to the corresponding pattern [7] and clarifies the principle of delegation from the target interface to the adapted interface. This model is a refinement of the interface wrapper relationship in Figure 3. All interfaces are generalizations of the before introduced component interface. All target interfaces are old versions of component interfaces and a adapted interface is the latest interface of the component. Therefore every component would have a 1-to-1 relationship with the adapted interface and a 1-to-* relationship with possible old target interfaces.

If a client calls an operation on a target interface a adapter will delegate this target interface operation to a operation on the adapted interface. This delegation mechanism with the adapter is responsible for matching operations and casting results at runtime.

4. Evolution of UML Models

Nowadays interfaces and their relationships are often described with graphical description techniques. The according Java code, CORBA IDL or database adapters are generated. A common graphical description technique is the usage of UML class models (cf. [4,5,6]). To understand the problems that are related to evolution of such models, it's important to get an overview of all possible class model entities in the beginning. Only the basic parts of UML class diagrams are taken into consideration

The most important entities of a UML class model diagram are specified in [4]. For schema evolution and data migration are only the following entities relevant:

- Classes
- Attributes
- Methods
- and relations (association, aggregation, generalization, etc.)

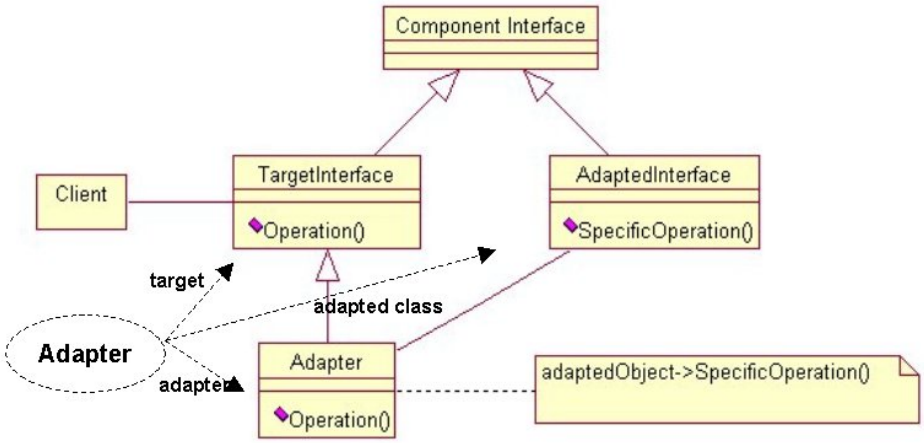

Figure 4: A Wrapper Model

As you can easy image it is possible for a developer to change every entity of such a class model diagram in a CASE tool. Basically a developer can perform the following operations on these entities. We call these operations update primitives:

- add
- delete
- rename
- retype

A deeper discussion of the topic of the various update primitives can be found in [15]. Note, that the this primitives guarantee completeness [16]: This set of operations cover all possibilities for schema modifications. But the correctness of the schema modifications is an undecidable problem [17,18]. As a consequence, we do not consider about methods updates. We only take care about the classes and their attributes and relations. Thus we can not support runtime evolution. You have to shut down the system run the evolution tool and than you can restart the system again. Usually that is an acceptable restriction if you want to support schema evolution and instance evolution for distributed systems

5. Supporting Model Evolution

In the last two sections we have described how a client can work with different interface versions of a component at runtime and what possible changes of a component interface described in UML are. This section elaborates the basic concepts and architecture to organize and apply possible changes on a model.

Model evolution means proceeding a ordered list of model change primitives on an existing component model and finally create a new model version. Afterwards the next interface versions and wrapper have to be generated, the implementation has to be changed and the old interfaces (target interfaces) have to be maintained in a way that they can work as wrapper to the latest version.

The different model versions are organized in a model list in Figure 5. The first element of such a list is the first model version, the last element is the latest model version. Recursively the successors are derived from their predecessors according to a set of change commands or update primitives. Every list item beside of the first and the last has exactly one predecessor and one successor. Each model version is aggregated of a set of model components, that are equivalent to the one's introduced before.

In principle every model change primitive leads to a new model version in the model change list, but most of the time it makes more sense to group update primitives together to a set of update commands or model change macros. These macros are modeled in Figure 5 according to the composite pattern [7]. A component change consists of at least one change primitive or a compound of change primitives. Proceeding such a set of update commands or a component change object on a model leads to a new model version.

Change execution can be modeled with the help of the command pattern [8]. The client of the command pattern is the so called change manager, the trigger or executor is a change executor and the commands themselves are the above introduced component changes. Last but not least, the receiver of the changes are the different model versions which are organized in a ordered model list.

A change manager is very similar to a parser for recognizing model changes. The change executor is responsible for the change logic. It's task is to apply a

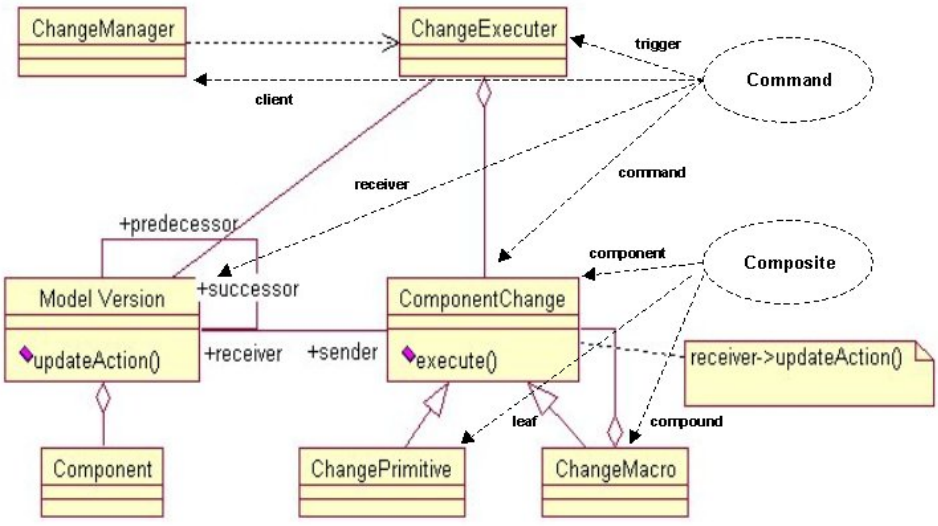

Figure 5: Supporting Evolution Model

component change on a model version. This task can include database changes, code changes and the organization of the new model version. Putting everything together delivers the following evolution model introduced in Illustration 5.

6. Model Changes and Classification

Let us now consider in more detail the different UML class model update primitives that are relevant for schema evolution. The following provides a short overview of the different model update primitives and the changes that can occur to them.

Different possible changes have different consequences that have to be reflected. But on the other side, certain model item changes could be replaced by a sequence of other model primitives. For example the change of a attribute name. This change could be compensated through a attribute deletion with a following creation with the new name.

Change primitives, we introduced in section 3, can be organized in different categories of model update primitives. This different groups of update primitives are different in the way how changes are evaluated for the model, the code, the instances or the wrapper. Generally the primitives above can be subdivided in primitives which are relevant for persistence and primitives which are not relevant for persistence. This categorization is possible because of the two different characteristics of objects, state and behavior. Everything that describes the state of a object like attributes is relevant for persistence, anything that describes the behavior like methods is not relevant for database persistence. Note, also there are some primitives that are not relevant for persistence this primitives have to be handled for code updates.

The primitives that are relevant for persistence can be further divided into four groups. These groups are ordered ascending to the difficulty of implementation:

- First Group: Phantom Modifying Primitives
  These primitives are all renaming primitives. With name parameterization this primitives can be handled very easily by a wrapper that looks up the actual name at runtime. Only the wrapper code has to be changed not the implementation nor the instances.
- Second Group: Interface Restricting Primitives
  This category is especially for deletion primitives. The consequence of a deletion primitive is only the creation of a new wrapper that restricts the range of the original implementation. There have to be no changes to the instances or the code, only restricting wrapper have to be implemented.
- Third Group: Model Extending Primitives
  These primitives are the create or add primitives. Essentially these primitives can be executed by an enhancement of existing wrapping interfaces, implementation code and additionally instance enhancement.
- Fourth Group: Not cleanly Manageable
  This group is for all retype primitives. This primitives are critical for reasons of information and exactness losses and indetermination of user's wishes. The treatment of such changes will be described in the next paragraph.

These four groups are explained in Figure 6.

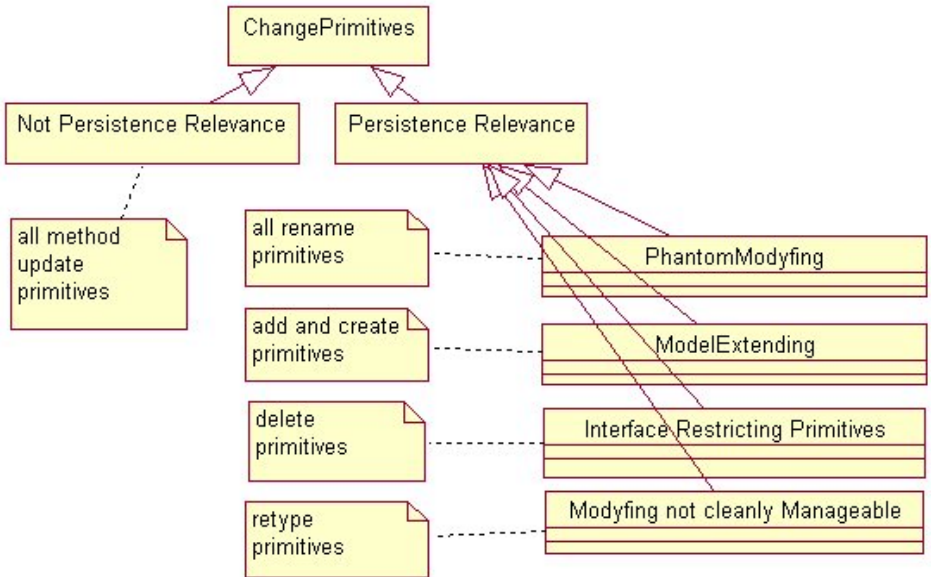

Figure 6: Classification of Change Primitives

7. Treatment of Not Cleanly Manageable Model Changes

In this section we present a simple model for supporting model schema evolution. This simple model is only used to explain the general behavior of not cleanly manageable model changes.

The target of model evolution is to maintain correctness after a model change. This means both the static relation of interfaces and types and cooperation between interfaces and the behavior of instances has to be consistent after a change. In the following two possible scenarios for handling model evolution are described. The first one will be named as the "Convert Scenario" and the second one as the "Extend Scenario". We describe the problems, advantages and disadvantages of each scenario.

In the "Convert Scenario" the old data of persistent objects is converted and adapted according to the new model specification. Every time the model changes the schema changes and the data is converted too.

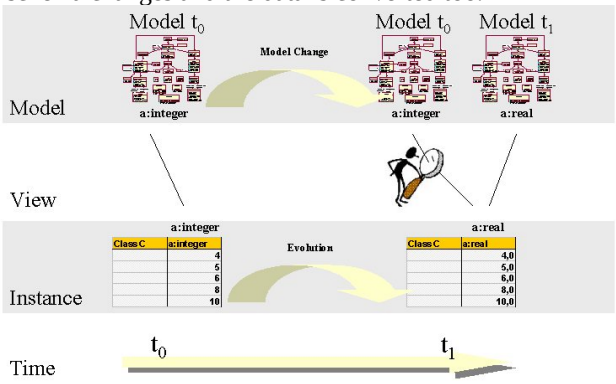

Figure 7: The Convert Scenario

Now comes a short explanation of this scenario. At time t0 the class C in Model t0 has an attribute a of type integer. The model which includes C is changed and now at time t1 the class C has an attribute a of type real. After the model change two versions (model t0 and model t1) of the model exist. The schema is changed according to the new model and the data has to be converted into the new format. This means all old values of attribute a have to be converted from integer to real. Additionally to the value converting a new view has to be created, because one of the requirements is that it should be possible to access the converted data with an old model version. This view is responsible for this transparent access and will be realized with a wrapper. At the moment you should imagine that this view is a black box that gets real value input from the database schema and provides integer output to the code based on model t0. Figure 7 provides you a visual overview of the explained scenario.

The "Extend Scenario" chooses another way to keep consistency. This philosophy of this scenario is as follows: Every time a model is changed the schema will be extended. The new schema is a union of the old and the new model. Newly added attributes for example will be initialized with null. The different applications (including the latest) access the schema with the help of wrappers, because each model uses only a subset of the schema.

Now the attribute a of Class C in Model t0 is changed from type real to integer. The most important difference is the way instances are treated. In this scenario not the attribute type of the schema is changed, but a new attribute with type real extends the schema. The old values are still accessible as integers but the values are not converted from integer to real. Instead of the attribute conversion the new attributes are initialized with a null reference. Each code equal if it is based on the new or the old model must now use a view to access the persistent data from the database. The schema evolution changes are visible in Figure 8 below.

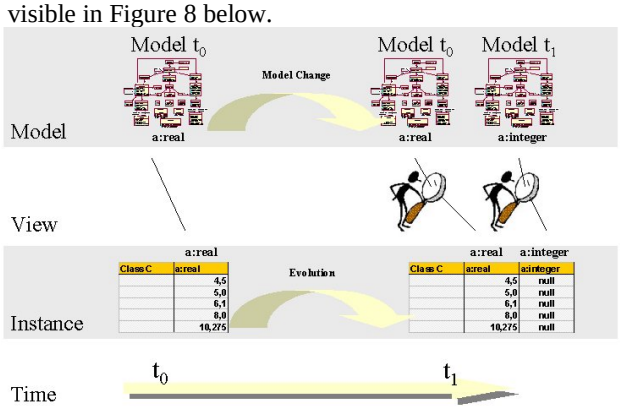

Figure 8: The Extend Scenario

Now we have discussed two scenarios of schema evolution, but at what time it makes sense to convert instead of extend and at what time the other way around or rather what wants the developer? The answer is it depends on the situation and on the users demands.

There is a very easy mathematical foundation for the problems of not cleanly manageable changes. The reason for the problems is that not every type cast with the related instance conversion is a bijective function. This means there is no identical way to convert the data from one representation into the other. The only way to achieve a general conversion possibility is to store every instance in a container that has type any, but this has the big disadvantage of no typing and data conversions.

8. XML/XMI based Specification of the Model Evolution

In the following the architecture for specifying UML models and model changes will be introduced. One needed important thing for delivering transparent model changes is a neutral model specification format. For

reasons of currently becoming a respected standard and being adopted by a lot of UML Case Tools vendors, XMI is chosen in this architecture as a neutral exchange format between different Case Tools. In addition there is a explosion of tools for handling XML documents very comfortable. The XMI standard [11] specifies with a Document Definition Type (DTD), how UML models are mapped into a XML file. Besides this functionality XMI also specifies how model changes can be easily mapped into an XML document. Therefore XMI is a very good solution for solving some of the requested requirements for UML model evolution.

As said before XMI specifies a possibility for transmitting metadata differences. The goal is to provide a mechanism for specifying the differences between documents in a way that the entire document does not need to be transmitted each time. This is especially important in a distributed and concurrent environment where changes have to be transmitted to other users or applications very quickly. This design does not specify an algorithm for computing the differences, just a form of transmitting them. Only occurring model changes are transmitted. In this way different instances of a model can be maintained and synchronized more easily and economically. The idea is to transmit only the changes made to the model together with the necessary information to be able to apply the necessary changes to the old model. With this information you have the possibility for model merging. This means you can combine difference information plus a common reference model to construct the appropriate new model. A important remark to this topic is that model changes are time sensitive. This means changes must be handled in the exact chronological order for achieving the wanted result.

According to Illustration 5, that specifies the evolution model, the model versions are represented as XMI files and the component changes are also XMI files that only specify the model changes. Each model version has a predecessor model from that it is derived (except if the model is the first version), a XMI document that represents the actual UML specification of this model. Each component change has a XMI-change document that specifies how a model version was constructed from the predecessor schema.

As introduced before not only the UML models will be specified according to the XMI standard, but also model changes. The following elements are used to encode the for this paper important model differences:

- XMI.difference: (reference to the old model)
  The XMI.difference element is contained by the XMI.content section of the XMI document. There can be zero or more difference elements and each difference element can contain zero or more particular differences. The difference element optionally links to the original document (the parent model) to which the changes are applied.
- XMI.delete: (reference to deleted element)
  The delete element is contained by a difference element. It's link attributes contain a link to the element from the original document to be deleted and specifies a removal of the referenced element and all of it's contents.
- XMI.add: (new element content)
  Like the delete element the add element is contained by a difference element. The content of a add element specifies the element and it's contend to be added to the original model.
- XMI.replace: (reference to replaced element, replacement content)
  The last element is also contained by a difference element. The content of replace is the element to replace the old element with. The old element will be specified in the link attributes of the replace element.

Here is an example how the UML model data and the changes can be coded according to the XMI standard, Note, the tags are shortened for clarity.

The original document:

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p1">
        <Class xmi.id="ccc" xmi.label="c1">
            <ownedElement>
                <Attribute xmi.label="a1"/>
                <Attribute xmi.label="a2"/>
            </ownedElement>
        </Class>
    </Package>
</XMI.content>
```

Figure 9: The Original Document

The change document with references to the original document.

The differences document:

```
<XMI.content>
    <XMI.difference href="original.xml">
        <XMI.delete href="original.xml#ccc"/>
        <XMI.add href="original.xml#ppp">
            <Class xmi.label="c2"/>
        </XMI.add>
        <XMI.replace href="original.xml#ppp"/>
            <Package xmi.id="ppp" xmi.label="p2"/>
        </XMI.replace>
    </XMI.difference>
</XMI.content>
```

Figure 10: The Evolved Version of the Document

And finally how the differences steps change the document if they are applied

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p1">
    </Package>
</XMI.content>
```

Next, the XMI.add:

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p1">
        <Class xmi.label="c2">
        </Class>
    </Package>
</XMI.content>
```

Finally, the XMI.replace:

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p2">
        <Class xmi.label="c2">
        </Class>
    </Package>
</XMI.content>
```

Figure 11: Three Samples of XMI Based Evolution Description

9. Conclusion

In this paper we have shown that modern middleware infrastructures for the development of distributed applications provide rich support for model based development and code generation. But there is almost no support in case of model evolution. We have introduced some concepts and architectures to realize a tool supporting model evolution and data migration and to integrate this tool in modern infrastructures. To specify the model evolution the developer should use an XMI based difference description.

Based on this concepts we have already implemented a first prototype. This is a very primitive version but it is already integrated in our framework AutoMate. Based on this experience we have realized the new version of the tool called ShapeShifter. ShapeShifter is now a stand alone tool supporting model evolution and data migration on top of Versant's object-oriented database. With ShapeShifter you specify the model difference in XMI and the model and the database are automatically migrated. ShapeShifter is now used in a first industrial project.

The next step will be a complete integration in a CASE tool. Currently one can export and import XMI model files from some CASE tools. But for a full integration of ShapeShifter we need more sophisticated tools to generate the XMI difference file from to XMI based model versions. Moreover we plan to integrate ShapeShifter into several Enterprise Java Beans Container.

10. References

List and number all bibliographical references in 9-point Times, single-spaced, at the end of your paper. When referenced in the text, enclose the citation number in square brackets, for example [1]. Where appropriate, include the name(s) of editors of referenced books.

[1] JavaSoft. Enterprise Java Beans Specification 1.1. http://www.javasoft.com. 2000.

[2] OMG. CORBA Components. http://www.omg.org. 2000.

[3] OMG. Common Object Request Broker Architecture 2.0. http://www.omg.org. 2000.

[4] OMG. OMG Unified Modeling Language Specification (Version 1.3). http://www.omg.org, document number: 99-06-08.pdf. 1999.

[5] Grady Booch, Ivar Jacobson, James Rumbaugh. The Unified Modeling Language User Guide. Addison Wesley Publishing Company. 1998.

[6] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual. Addison Wesley Publishing Company. 1998.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing. 1995.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Son. 1996.

[9] Klaus Bergner, Karsten Kuhla, Andreas Rausch. Schnelle Schichten: Transparenter Zugriff auf ODBMS über CORBA. iX No. 11. 1998.

[10] AutoMate. AutoMate, Technische Universität München. http://automate.informatik.tu-muenchen.de. 2000.

[11] OMG. The XMI Specification 1.0. http://www.omg.org. 2000.

[12] Frank E. Redmond III. DCOM: Microsoft Distributed Component Object Model. Microsoft Press. 1997.

[13] Dan Harkey, Robert Orfali. Client/Server Programming with Java and CORBA, Second Edition. John Wiley & Sons. 1998.

[14] Robert Orfali, Dan Harkey, Jeri Edwards. Client/Server Survival Guide, Third Edition. John Wiley & Sons. 1999.

[15] Eduardo Casais. Managing Class Evolution in Object-Oriented Systems. In Object-Oriented Software Composition, Prentice Hall. 1995.

[16] Fabiano Cattaneo, Alberto Coen.-Porisini, Luigi Lavazza, Robert Zicari. Overview and Progress Report of

the ESSE Project: Supporting Object-Oriented Database Schema Analysis and Evolution. In Proceedings of the 10th TOOLS Converence, Prentice Hall. 1996.
[17] Hyoung-Joo Kim. Algorithmic and Computational Aspects of OODB Schema Design. In Object-Oriented Databases with Applications to CASE, Prentice Hall. 1991.
[18] Emmanuel Waller. Schema Updates and Consistency. In DOOD'91 Proceedings, Springer Verlag. 1991.
[19] Robert Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In Building an Object-Oriented Database System – The Story of O2, Morgan Kaufmann. 1992.