Department of Computer Science
Chair IV, Software and Systems Engineering

Model Evolution for a Distributed UML Software Engineering Workbench based on CORBA and Java

Frank Keienburg
Tutor: Andreas Rausch
Prof. Manfred Broy

Date:05.03.2000

Contents

Illustrations

PREFACE

Foreword

This document was developed in scope of a System Development Project at the Technische Universität Munich – Department of computer science chair IV, software and systems engineering. The document and it's belonging code should be an extension of the existing software engineering workbench AutoMate [http://automate.informatik.tu-muenchen.de], which is used to support the development of distributed systems with CORBA [OMG 99] and Java [SUN] based on UML [OMG 99b] class models. The development of this extension is used as a case study to explain the general problems of model evolution and a possible solution attempt. As a prerequisite the reader should be familiar with the concepts of UML, CORBA and the Java programming language.

Introduction And Motivation

Why Scheme Evolution ?

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 a lot of changes to 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 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 again. 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.

For that reasons, development tools like AutoMate that facilitate concurrent engineering, reuse, and componentware should support model changes. This evolution should also include data migration of data based on old models to new models and data access with code based on different model versions, or shortly preserving persistent data across scheme changes. Technically this means to support scheme evolution.

Structure Of This Document

This document is structure into three main parts.

Part 1: In the beginning of the document a theoretical overview of model scheme evolution is presented. It gives an introduction to the theoretical aspects of model evolution. It concerns the parts of a model, the change operations on these parts, the related basic runtime problems of model evolution and how these problems can be classified and solved.

Part 2: The intention of this part is to document the requirement analysis process and it's results for the concrete model scheme evolution application AutoMate. It is based on the question, what the primary tasks and requirements for the system extension are. To achieve this goal the informal and formal requirements are specified and or described. To explain the main problem and functionality two scenarios are used. The "State of the Art scenario" describes the actual status, and according to this the actual behavior of the system. The "Future scenario" should explain the desired behavior of the system.

Part 3: As next step this part describes based on the results of part one and two how the future model evolution architecture of AutoMate can be achieved and will be transferred from the theoretical part. This part shows which design decisions have been made and why.

Part 1 - Model Evolution Theoretical

Today developing software systems is tool aided because of the inherent system complexity. Especially CASE tools based on the Unified Modeling Language (UML) are used to support the difficult development process. With assistance of this tools, models and interfaces of a desired system are specified. These models serve as interfaces, abstraction and communication foundation during the development process and help to understand the nature of the problem. With focus on further automation of the development process this models are also used for generating application and database access code.

1.1 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 Illustration 1. A Component is constructed according to the composite pattern of [Gamma] and is a single implementation or a compound of other components.

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.

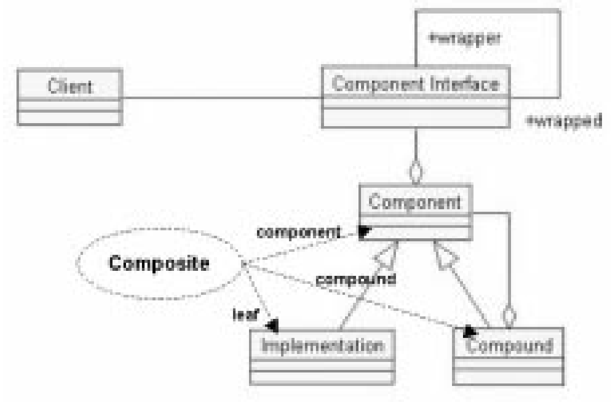


Illustration 1 – A Component Model

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 solution approach is modeled in Illustration 1. The latest version of the component interface exactly corresponds to the actual implementation, all other interfaces are wrappers or adapters in the sense of [Gamma] 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.

Illustration 2 explains how a wrapper or adapter for components can be modeled according to [Gamma] 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 Illustration 1. 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.

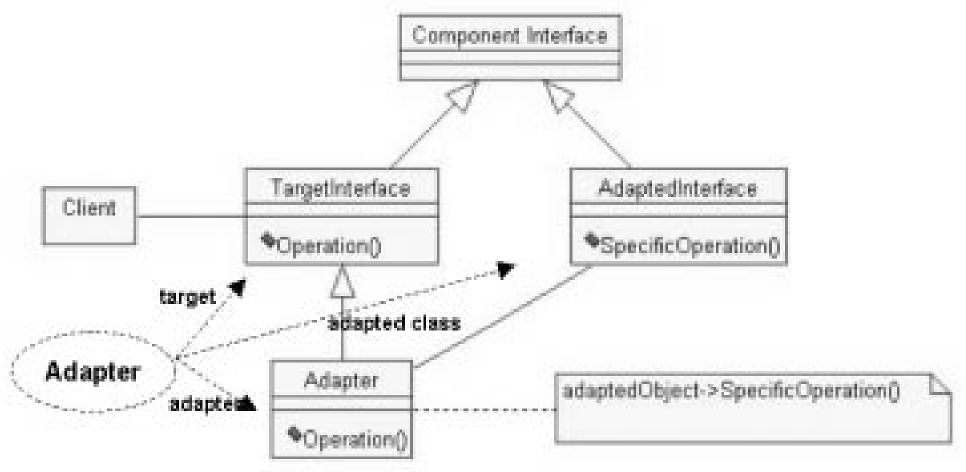


Illustration 2 – A Wrapper Model

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.

The idea of this document is to provide a flexible workbench that supports the software development process with automatically generating necessary interfaces and wrappers based on model changes. These workbench should provide additional functionality like instance conversion and delegation to the right adapted interface. The functionality demand requires automatic component code and instance evolution. This are the concept of model evolution.

1.2 Evolution Of UML Models

As mentioned before, today interfaces and their relationships are often described with graphical description techniques and afterwards the according Java code, CORBA IDL or database adapters are generated. A common graphical description technique is the usage of UML class models. 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. In the further document only the basic parts of UML class diagrams are taken into consideration

The most important entities of a UML class model diagram are illustrated below in Illustration 3 according to the [UML] specification.

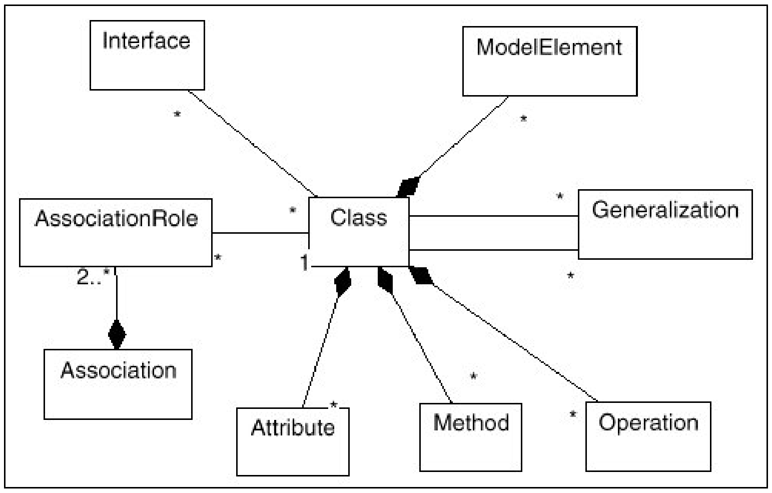


Illustration 3 – Entities of UML Class Diagrams

The relevant one's for this document are:

- classes,
- attributes,
- methods,
- and relations (association, aggregation, generalization, …).

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. These changes will be called update primitives in the further document and are introduced in more detail in the later.

1.3 Model Evolution

The last two sections described how a client can work with different interface versions of a component at runtime (1.1) and what possible changes of a component interface described in UML are (1.2). This section describes an architecture how possible changes can be organized and applied on a model.

1.3.1 Model Change List

Model evolution means proceeding a ordered list of model change primitives on an existing component model and therefore 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 Illustration 4. 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.

1.3.2 Model Change Macro

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 Illustration 4 according to the composite pattern of Gamma. 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.

1.3.3 Change Execution

Change execution can be modeled with the help of Gamma's command pattern [Gamma]. 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 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 4.

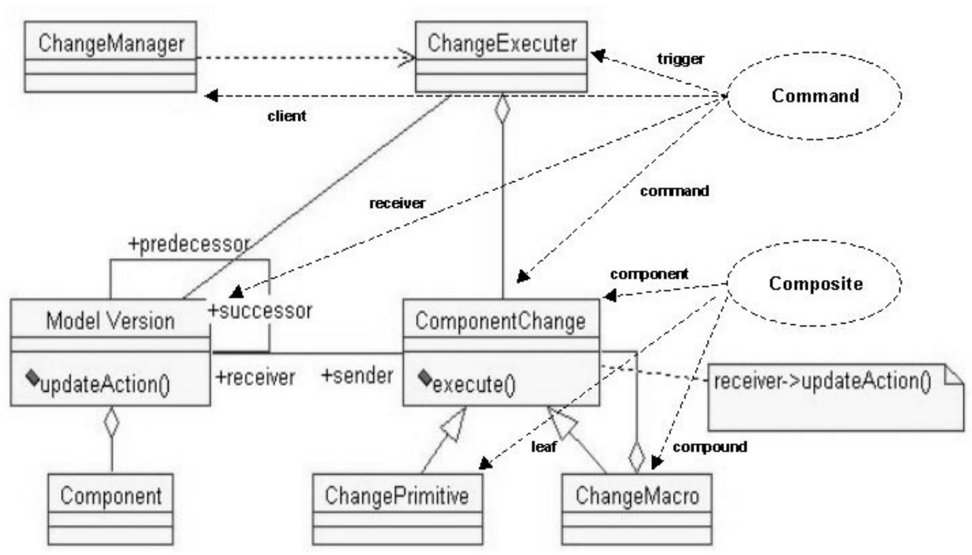


Illustration 4 – Evolution Model

1.4 Model Changes And Classification

Let us now consider in more detail the different UML class model update primitives that are relevant for Scheme 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 now. The below shown Illustration 5 summarizes possible change primitives.

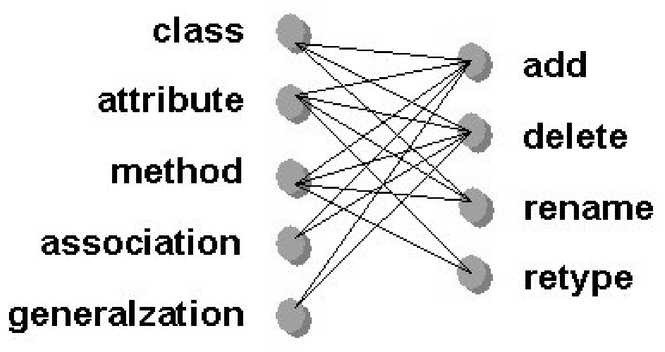


Illustration 5 – Model Entities And Changes

An interesting fact to recognize is that 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.

The introduced primitives 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. Remark: 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.

Concluding these four groups are explained in the following illustration.

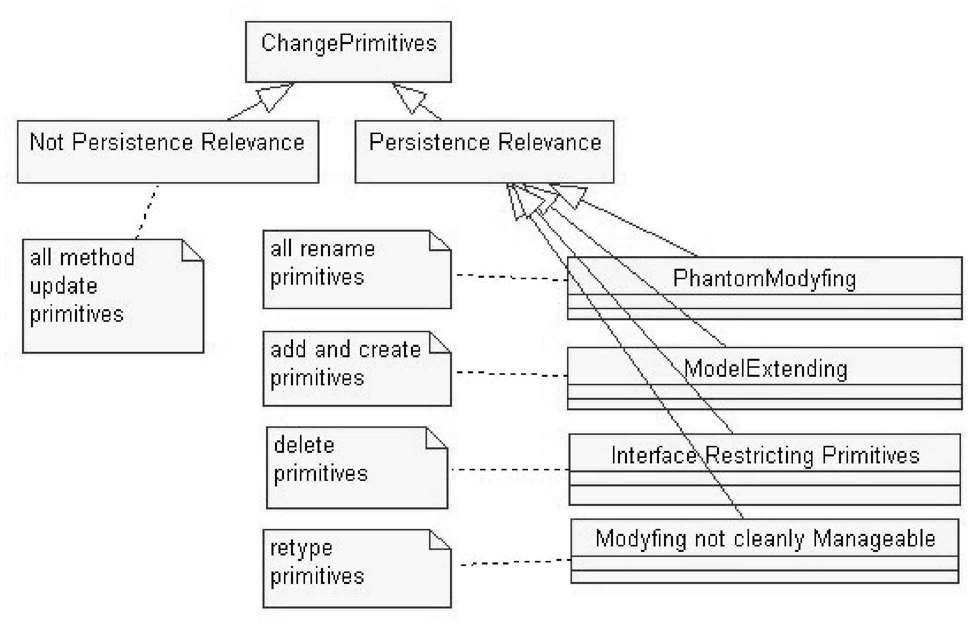

ChangePrimitives
Not Persistence Relevance
Persistence Relevance
all method update primitives
all rename primitives
add and create primitives
delete primitives
retype primitives
PhantomModyfing
ModelExtending
Interface Restricting Primitives
Modyfing not cleanly Manageable

Illustration 6 – Model Change Primitives

However, after discussing the different categories of changes it's interesting to know what the consequences of not cleanly manageable changes are and how these consequences are handled. Exactly this is done in the next paragraph.

1.4.1 Treatment Of Not Cleanly Manageable Model Changes

In this section a simple model is presented for supporting model scheme 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 1st one will be named as the "Convert Scenario" and the 2nd one as the "Extend Scenario". In the later the problems, advantages and disadvantages of each scenario are described.

The Convert 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 scheme changes and the data is converted too.

Now comes a short explanation of this scenario. At time $\mathbf{t_0}$ the class **C** in **Model** $\mathbf{t_0}$ has an attribute **a:integer**. The model which includes **C** is changed and now at time $\mathbf{t_1}$ the class **C** has an attribute **a:real**. After the model change two versions (Model $\mathbf{t_0}$ and Model $\mathbf{t_1}$) of the Model exist. The scheme 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 scheme and provides integer output to the code based on model $\mathbf{t_0}$. Illustration 7 provides you a visual overview of the explained scenario

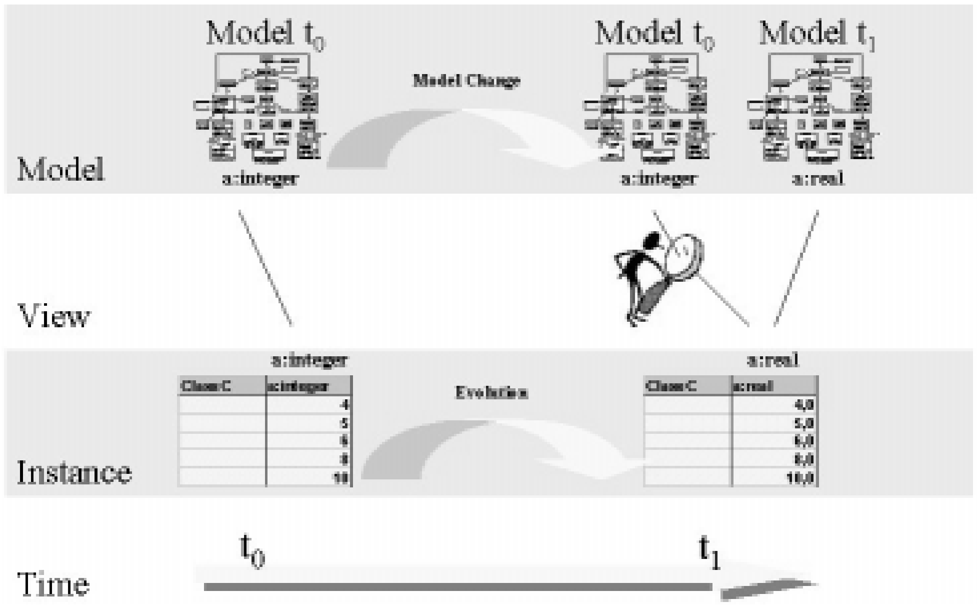


Illustration 7 – The Convert Scenario

The “Extend Scenario” chooses another way to keep consistency. This philosophy of this scenario is as follows. Every time a model is changed the scheme will be extended. The new scheme 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 scheme with the help of wrappers, because each model uses only a subset of the scheme.

Now the attribute **a** of Class **C** in Model **t$_0$** is changed from type **a:real** to **a:integer**. The most important difference is the way instances are treated. In this scenario not the attribute type of the scheme is changed, but a new attribute with type real extends the scheme. 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 Scheme Evolution changes are visible in Illustration 8 below.

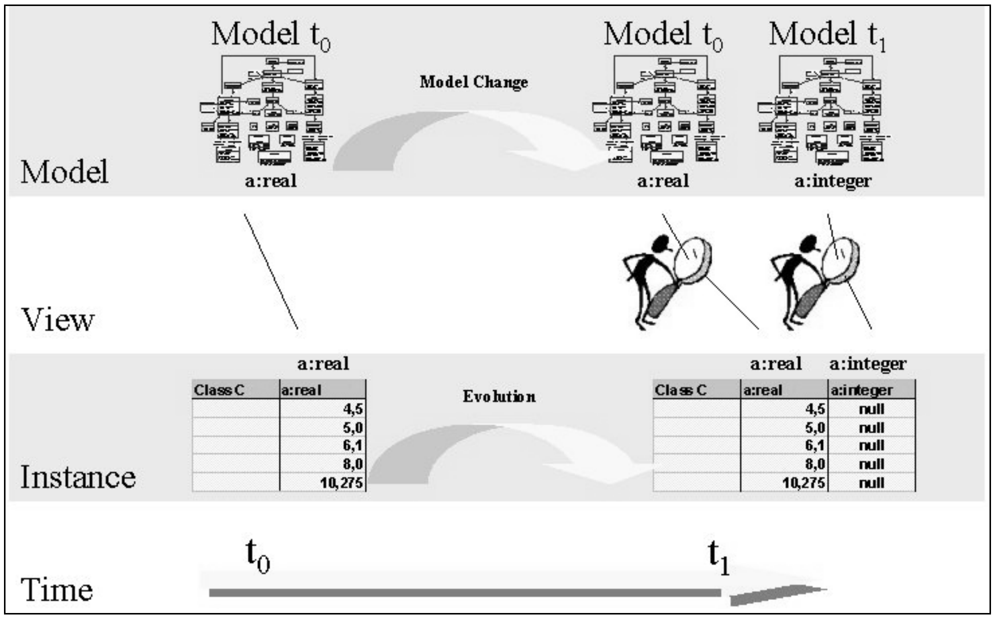


Illustration 8 – The Extend Scenario

Now we have discussed two scenarios of “Scheme 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. The next section comments the situation of not cleanly manageable update primitives and delivers the mathematical foundation.

1.4.2 Mathematical Foundation For Model Changes

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.

1.4.3 Treatment Of Generalization Changes

This problem is not solved already in this paper, but there is an approach to treat this problem. According to [Gamma] and [Ostermeier] delegation can be used to simulate inheritance. The effects of inheritance can be adjusted using aggregation. This provides the theoretical background for a flexible and changeable treatment of a inheritance mechanism during run-time. The problem is that’s in some languages it’s very difficult to map these model to the programming language (for example: Java).

Part 2 - REQUIREMENTS ANALYSIS FOR AUTOMATE

This part deals with the question what the development requirements for model scheme evolution in the existing AutoMate system are. Therefore this part is divided into three paragraphs. The 1st part or the "State of the Art" paragraph describes the actual system environment and explains how the actual system handles changes in the development model. The 2nd paragraph or the "Visionary" part illustrates how the system should work after the development and analysis the concrete requirements to the system. The final paragraph shows how the system demands can be solved.

2.1 State Of The Art

However, after discussing the general principles of model evolution and possible changes theoretically, it's interesting to point out the concrete requirements for a existing system. As a first step the existing AutoMate system has to be analyzed and the current development process has to be viewed to point out the weak points. Exactly this is done in the next sections.

2.1.1 The Development Process With AutoMate

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). Illustration 9 shows such an architecture. A popular approach to build such a system is using Java and CORBA. Because development of such a system is very complex it's helpful to use tools that support the development process. For explaining the benefits of developing distributed system with AutoMate in the beginning the standard steps of a development process are explained.

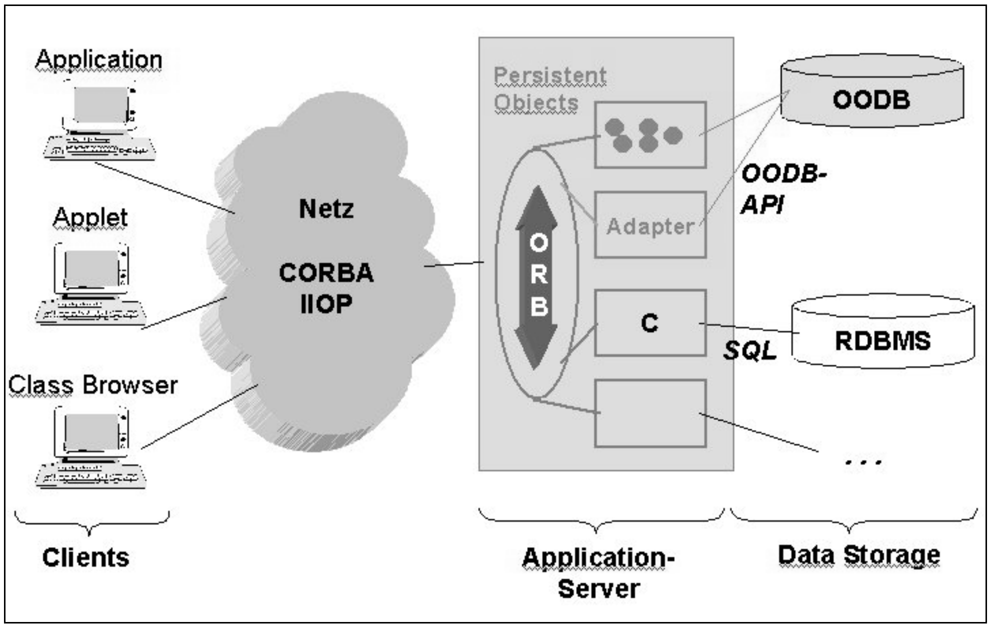


Illustration 9 – The 3 Level Architecture of a typical Distributed Systems

Normally following steps are involved in the development of a CORBA based distributed system:

- Create an Application Model (with a CASE Tool)
- Define or generate CORBA IDL interfaces based on the model
- Implement the application server according to the IDL interfaces
- Realize persistence for the application server instances
- Implement the client application(s) in a programming language like Java

Some of these standard work can be automated by a tool to become a continuous development process based on a consistent model. That's the point where AutoMate comes in the big picture. 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. See Illustration 10 for more details.

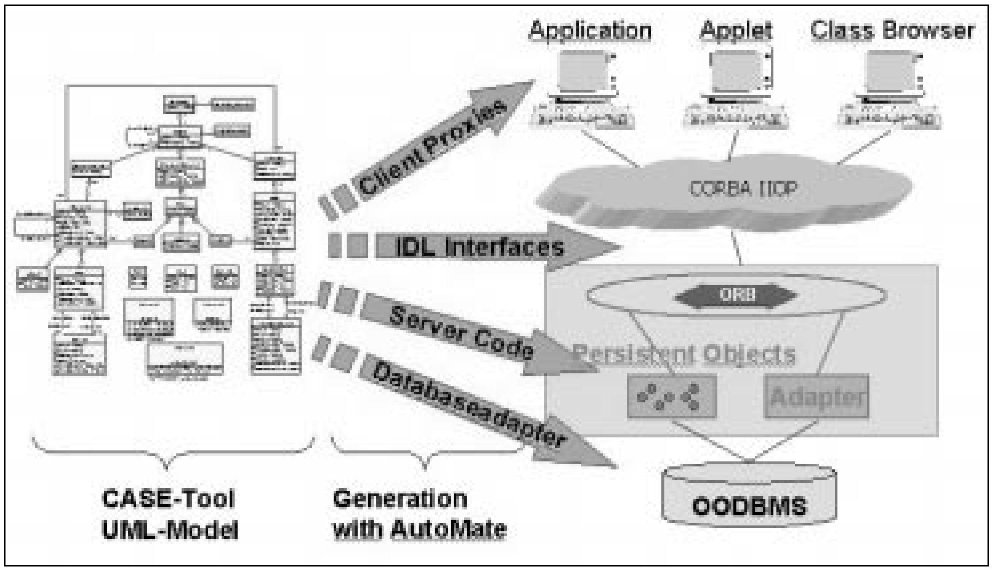


Illustration 10 – The Development Process with AutoMate

The current AutoMate version relieves you of generating application code, database access and delivers some standard work. But one thing currently missing in AutoMate is model evolution. This fact delivers the motivation for developing and discussing model evolution in the context of AutoMate.

Improvement Possibility

At the moment after changing a class model, new code overwrites old code, a new database scheme 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. Remember the introduction: 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 scheme evolution is a useful extension to the existing system. The next paragraph describes the possible extension of AutoMate.

2.1.2 What Should Happen After A Model Change ?

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.

To achieve the needed consistency, you have to care about two things, keeping your application code up to date and reorganize the interfaces and wrappers. And secondly reorganize of the object instances which means in the AutoMate environment database scheme and object instance changes and conversion.

Keeping Code Up To Date

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 created 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.

Reorganize Data Instances

Dynamic aspects concern the run-time 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.

Persistent objects and the database scheme are related to a specific code version. That's just why code changes cause also database scheme 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.

2.2 Desired Model Evolution and Application Behavior

To clearly and easily explain the requirements of model evolution in the AutoMate system, 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.

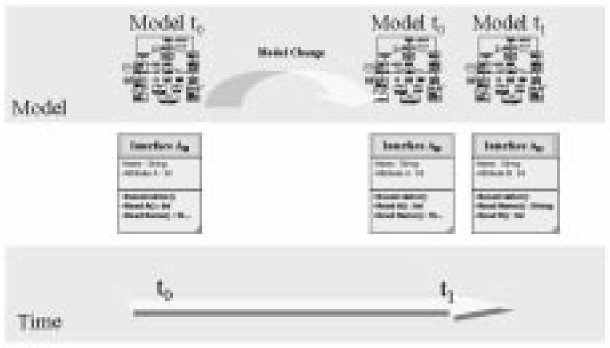


Illustration 11 – Model Change

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

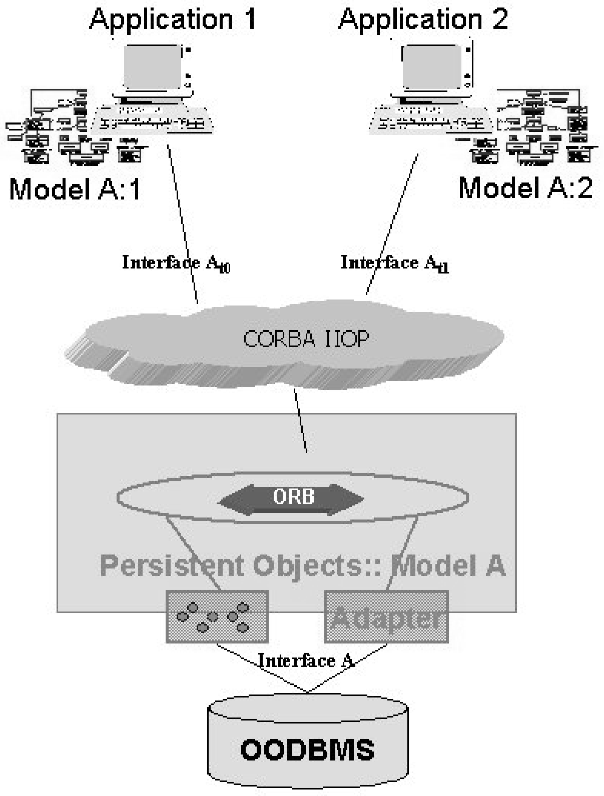


Illustration 12 – System with Different Model Versions

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 2nd tier consists of CORBA servers for the different interface versions and the 3rd 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. Based on this architecture a typical usage scenario in form of a message sequence chart is introduced.

2.2.1 A Typical Usage Scenario

This section describes from the users point of view how working with different model versions should be possible in the flight booking system created with AutoMate. Illustration 13 – Sequence diagram example for working with different model versions above shows how different clients in a distributed system can communicate and interact with the database and the persistent object instances. Clients can create instances of a model and select existing instances of a model from the database. In general this is nothing new, but the instances of a model can be based on different model versions. The Evolution Manager cares about the different versions and the database scheme and makes the access of the applications totally transparent. The Manager also cares about the version management or scheme evolution and provides something similar to a view in relational databases on the model instances to the applications. As a prerequisite two client versions based on Model A exist, the application server is simplified and presented by a “Evolution Manager” and no data based on Model A is already stored in the database server. The following scenario is a typical usage scenario for the above introduced flight system:

- Client one or Application1 creates a instance of a flight reservation based on Model A version one.
- To make this flight reservation persistent this instance has to be stored in the database. The database storage procedure should be totally transparent for the user, therefore a black box “Evolution Manager” is doing this work. This manager cares about creation of the database scheme and storage of the object instance.
- Client two also creates a instance of a flight reservation. But the difference is that Application 2 is based on code according to Model A version two. For this reason a instance based on Model A version two is created.
- The “Evolution Manager” is responsible for storing this second flight reservation in the database. Therefore the manager has to change or extend the database scheme and the already stored flight reservation totally transparent for the users.
- Now user of Application1 wants to see all flight reservations or instances of Model A. As a result the “Evolution Manager” presents the user two flight reservation instances according to a Model A version one view. Therefore the EM selects all instances of Model A and adapts them to Model A version one.
- If user of App 2 wants to see all instances of Model A version one, the same steps as before happen. As a result user 2 views two flight bookings based on model A version two.

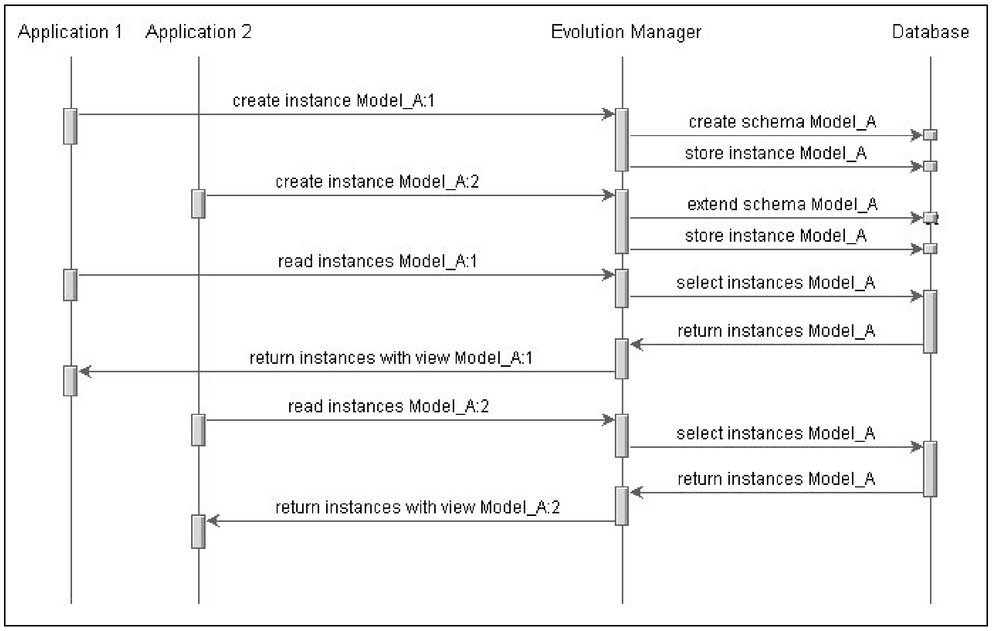


Illustration 13 – Sequence diagram example for working with different model versions

The above introduced usage scenario explains a desirable system behavior. Both consistency parts are fulfilled and you can access consistent data with clients based on the hole model life cycle. To create this future behavior something like an “Evolution Manager” has to be developed that should realize model evolution.

The above introduced scenario is a scenario nearly at the end of a model’s life cycle. It is a scenario using completely build applications. But model evolution can be useful over the hole model life cycle, and therefore for example also in the application development phase for reasons of a new application design or added functionality. Model changes are usual at all phases of the object oriented development process and afterwards at all phases of usage and maintenance.

Remark: In general it’s not imperative for all model changes to cause schema evolution in the database (for further details refer to UML class model changes). Additionally model evolution is only necessary if applications based on an old model version are existing and the developer wants to keep compatibility with these applications. Therefore it should be possible for the developer to explicitly decide keeping the old version or deleting it inclusive the according persistent objects.

After introducing the future system behavior and a usage scenario we can summarize system requirements for model evolution.

2.2.2 Summary of Requirements

One important result of a requirements analysis process is an overview of requirements. Something like a list of extensions, changes and new development functionality. Therefore a list is presented with the topics that model evolution for AutoMate should cover.

- Creation, changes and deletion of packages, classes, methods and attributes,
- migration of existent persistent data based on an old model to a new model,
- organization of automatically created code from all models of the model life cycle,
- transparent access to persistent data from all models of the model life cycle,
- application transparency, this means unchanged behavior of application during the hole model life cycle,
- and a decision possibility for the developer to handle not cleanly manageable model evolution. This can be for example the decision to extend a current model and migrate the data of the persistent objects or to delete the old data and create a new model.

After explaining how a system should react and listing up the requirements it’s interesting to deal with the question how to achieve the expectations. The last paragraph of this part shows in a very global and informal way how model evolution can be achieved.

Part 3 - System Design and Architecture For AutoMate

The following part of this document presents an architecture for solving the introduced requirements of the part before. In the following will be specified how necessary type casts will be handled in general, how models and model changes will be specified, how model changes will be transformed to scheme and instance changes in the database and last but not least how the server application code will be organized and updated according to the database scheme changes.

The theoretical evolution model will now be filled with the needed functionality.

3.1 General Thing's

3.1.1 Architecture for handling Type Casts

One important thing is handling of type cast needed for scheme evolution. A frame work for type conversations is needed on the one hand for casting database instances according to the new scheme during scheme evolution in the database and on the other hand for casting database instances backwards to older scheme versions to achieve the wanted application transparency.

The intention of this cast framework is to deliver a possibility to cast between all needed types. This will be done by using a neutral intermediate format. Every expression type that should be casted must deliver the functionality to cast it into the neutral format and the other way around from the neutral format to the expression format. An model example for simple types is shown in the illustration below.

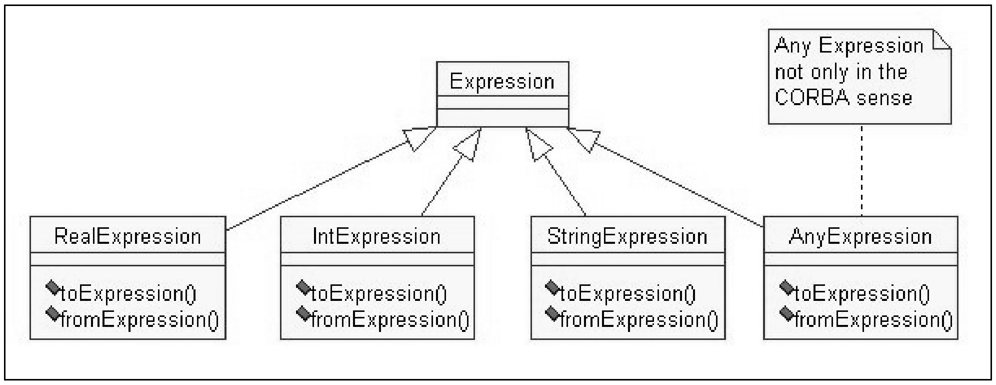


Illustration 14 – The Expression Model

The above specifiation of a cast environment is a minimum specification. It is possible to enhance these specification with additional functionality like user extensibility or different methods to cast a special type to or from the expression type.

3.2 Specification for Model's and Change's

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 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 (difference between old and new 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. (New = Old + Changes). 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 4 from the theoretical part 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 scheme.

The next section describes the model and change specification format and explains it with easy examples:

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.

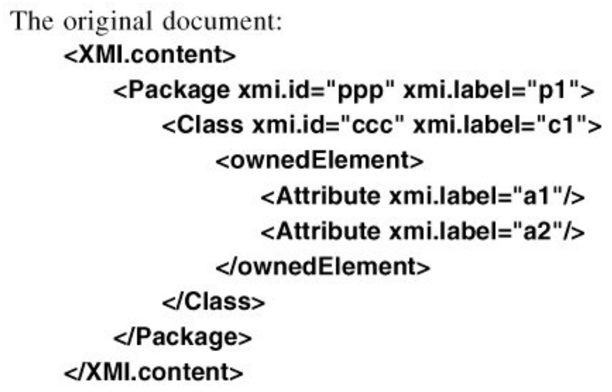
- XMI.delete: (reference to deleted element)
  The delete element is contained by a difference element. It’s link attributes contain a link to the element from the original document to be deleted and specifies a removal of the referenced element and all of it’s contents.

- 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 (The tags are shortened for clarity).

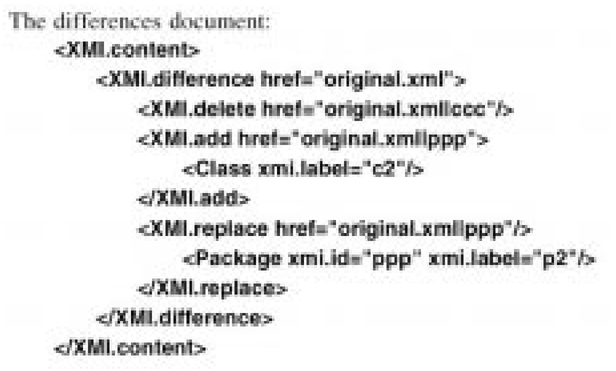
Original document: “original.xml”



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>
```

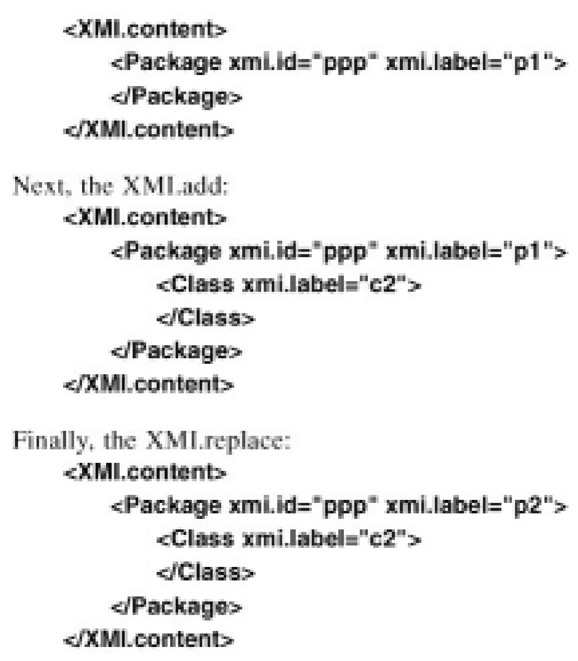
The change document with references to the original document.



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>
```

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>
```

The introduced model version and component change specification according to the evolution model are the first part of the model evolution framework. The next part concerns the question how the model changes can be transferred to the database.

3.3 Architecture of Model Evolution Framework

Now that we have introduced how models and model changes are specified, the next important thing is the architecture for handling these model changes. The first thing for handling model changes is to recognize them. This is the work of the change manager. The change manager parses a difference file and triggers the change executor to fulfill it's task.

As said before model evolution causes changes to the object instances and the application code. Therefore model changes made in a model must be forwarded to the according database and result in scheme and instance changes.

In addition to this there should exist a framework where the user can specify what happens to already existing instances and application compatibility. Then the user can specify for example if existing instances will be casted and transferred to the new scheme or if they will be dropped. But for more details refer to the later documentation. The next thing that will be introduced to you are the different change policies that are important for the database modifications and the later introduced code updates.

3.3.1 Change Policies

Change policies specify how scheme changes will be treated in the database and how the CORBA application code will be modified. These change policies are part of the change manager. According to the users needs and wishes the different policies can be set in the scheme change framework. These change policies can be mapped to the software life cycle. You can easy imagine that in early development phases a lot of model changes are applied and that it might not be important to keep compatibility to prior code versions and keep created instances, but the further you become in the software life cycle the more important it is to keep compatibility to older code versions and keep existing instances alive. In the following the scheme change policies are described a little bit more in detail.

- DropInstancesKeepCompatibility

  Using this policy all existing instances of a scheme will be dropped, but compatibility of applications based on older scheme versions to the modified scheme will be kept.

- DropInstancesDropCompatibility

  The laziest policy specifies that all existing scheme instances will be dropped and only applications based on the newest scheme version can get access to existing instances, otherwise an error message will be reported to the old application.

- KeepInstancesKeepCompatibility

  This policy delivers the most flexibility and is the most diligent policy. All existing instances will be casted according to casting rules to the new scheme and compatibility of applications based on older scheme versions will be kept.

- KeepInstancesDropCompatibility

  The last policy is responsible for casting all existing scheme instances to the new scheme version, but specifies that applications based on older scheme versions have no more access to the new scheme versions.

The illustration demonstrates the connection between database change policy, importance of instances, number of model changes and time in software life cycle.

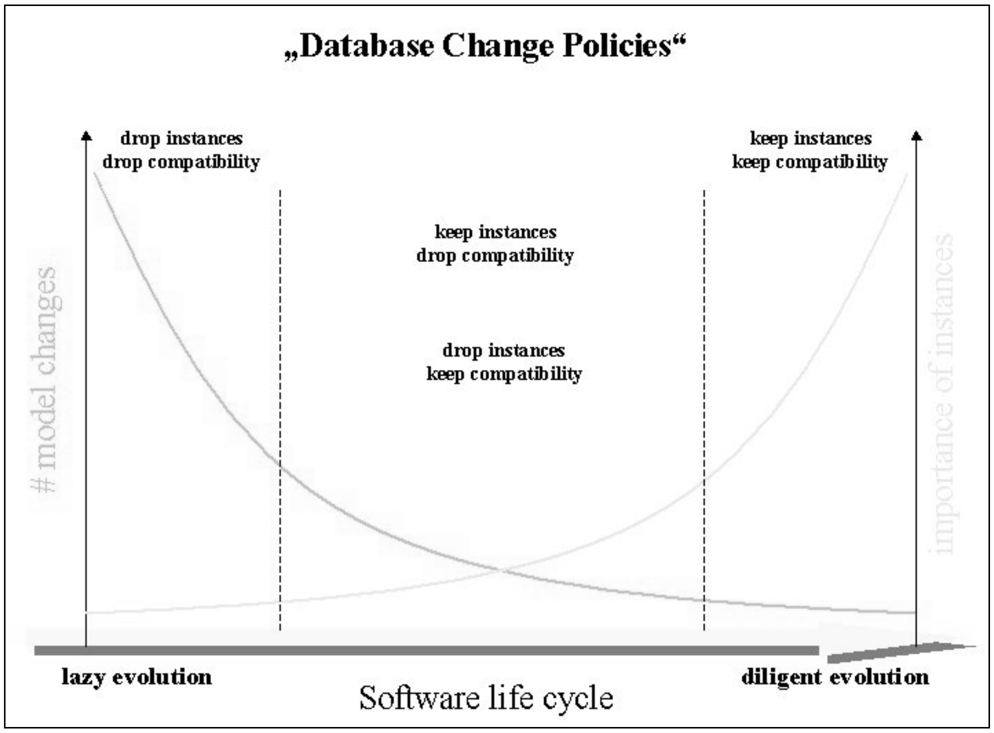


Illustration 15 – Database Change Policies

3.3.2 Mapping Model Changes to Scheme Changes

For future compatibility and independence of Case Tool vendors the model changes are transferred to the scheme cast framework according to XMI standards. Now a XML parser will recognize differences applied to a model and cause necessary changes to be carried out in the database and the access code. In the first only database changes will be handled. The different model change primitives will be treated according to the classification made above. “Scheme Extending” primitives will cause scheme add-on’s in the database scheme, “Real Modifying” primitives will cause scheme changes and instance casting and “Phantom Modifying” will cause renaming of scheme parts, but no real instance changes. But before the explanation of the change primitive treatment the Model Manager will be introduced.

3.3.3 The Change Parser and the Model Manager

As mentioned before XMI only specifies the data transfer format and not the algorithm for proceeding the changes. This algorithm will be described in the following paragraph (Treatment of scheme changes). But before you can proceed your model changes you have to extract them from the XMI-change document. This will be done with the help of a standard XML parser (XML4J from IBM). Based on this parser a component will be developed that extracts the model changes from interest and delivers it in form of a change tree to the change proceeding algorithm. The entity that specifies and embodies these algorithm will be called Change Manager in the following. This manager is responsible for the application logic that updates the database scheme, casts the instances and reorganizes the application code.

At this moment it’s time to add the information introduced to you to the well known model of the change framework.

In the following will be explained how the Scheme Manager handles the different scheme change primitives.

3.3.4 Treatment of Scheme Change Primitives

As mentioned before the different scheme change primitive classes will be treated differently and cause different actions for the scheme evolution framework. In the later scheme changes and instance casting will be discussed according to the most diligent change policy, because this is the most difficult and interesting one.

- Scheme Extending Primitives
  This changes will cause an extension of the existing scheme. For example an attribute will be added or something like that. Because of the fact that we add something new or additionally to the scheme there is no need to cast any old instance. The most databases deliver standard functionality to fulfil these simple scheme updates.
- Real Modifying Primitives
  Real modifying primitives are the most interesting ones. Supplementary to scheme changes, the existing primitives have to be casted to the new scheme. Both follow the principle of modifying the scheme to the latest changes. This means the scheme will be updated exactly to the latest changes and the instances will be casted to this new scheme version afterwards. The instance cast will be operated with the help of the cast framework and the instance access will be covered by code that fits exactly the new database scheme. Access from older code versions will be wrapped to the latest code version, but this is part of the last architecture feature described later.
- Phantom Modifying Primitives
  This changes like renaming an existing attribute will be maintained outside the database with the help of Java property files. A property file contains name value pairs that assign the original entity name to the actual entity name. These property files have to be updated every time the name of a entity changes. For reasons of simplicity this information is stored in property files but of course this information can also be stored in a database with the help of a separate object or table. Every time an object with an old entity name wants to get access to the database the new entity name will be looked up and the access will be carried out with the according new name totally transparent for the client object. For more details refer to the last part of the architecture that specifies in more detail how model changes are handled with the server code.

3.4 Architecture For Handling Code Changes With CORBA

This last part of the architecture is responsible for delivering a transparent client access to object instances from every model version of the object specification at runtime. To achieve this result three main things have to be developed and maintained during model evolution. The first one is the access code to the actual scheme, the second one are the wrappers that wrap the access code and deliver object access for every model version and the last one is the Portable object adapter needed to achieve the necessary server side flexibility.

3.4.1 The Access Code – Or The Delegated

The access code is that code that every time fits exactly to the scheme specified in the database. This code specifies only the actual fields (name and type) of a class. This access code will be used as a delegate from the wrappers to get access to the database instances. This delegated together with the POA can also be used for the transaction logic, but this is not content of this document. As implicit said before this code has to be updated every time the model and the according database scheme changes.

All delegated classes inherit from the class Delegated. This class offers essentially two methods, a method to set a field and a method to get a field. The setField method gets as parameters the field name and an object of type expression and returns nothing. The other getField method gets as parameter the field name and return as return value an object of type expression. These two methods are used to manipulate the field values and therefore accordingly the persistent instance values. Both methods use the reflection mechanism to get the needed information.

3.4.2 The Wrapper Code – Or The Delegate

Like the headline assumes this piece of code is responsible for two things, wrapping instance access and delegating calls to the right delegated. All delegate classes have to implement a constructor method that gets as parameter a reference to the delegated class and inherit from Delegate. Together with the earlier mentioned property file delegate classes therefore provide the functionality of name and type transparency.

- Name Transparency
  Name transparency will be achieved with the following mechanism. The attribute getter and setter methods of the delegate will lookup with the help of the parent lookup method the actual name of this attribute in the database. With this information and the delegated reference the fields of the delegated can be accessed and manipulated easily by calling the setField or getField method on the delegated.

- Type Transparency
  The second type of transparency is achieved in the same attribute getter and setter methods. Inside this methods the setField and getField methods are called as described above. The getField expression result has to be casted back to the wanted type and can then be returned to the client. The other way around the value that the client wants to be set has to be converted into the neutral expression format and can then be passed to the delegated.

Confused ? – There is no need to, here is the class model specifying the information from above.

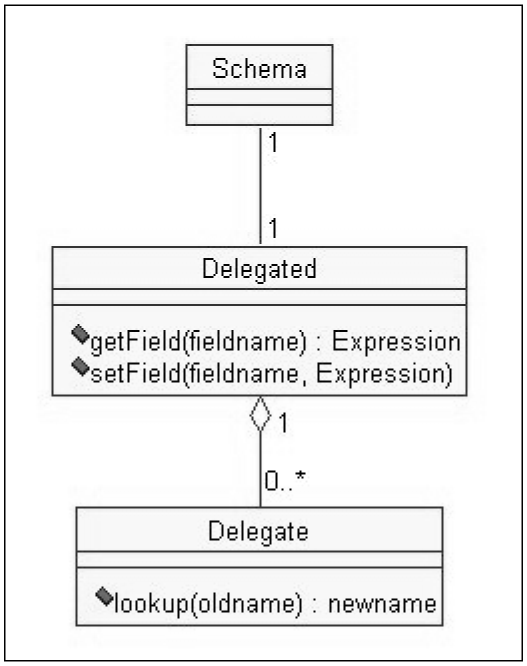

Schema
1
1
Delegated
getField(fieldname) : Expression
setField(fieldname, Expression)
1
0..*
Delegate
lookup(oldname) : newname

Illustration 16 – Delegation

3.4.3 The Object Adapter

The two things described before are responsible for the server side data access, but happens between the client and server objects? How is the right object version determined? How is the persistence of the object achieved? This last questions will be solved now.

The goal is to provide a architecture that handles the following scenario.

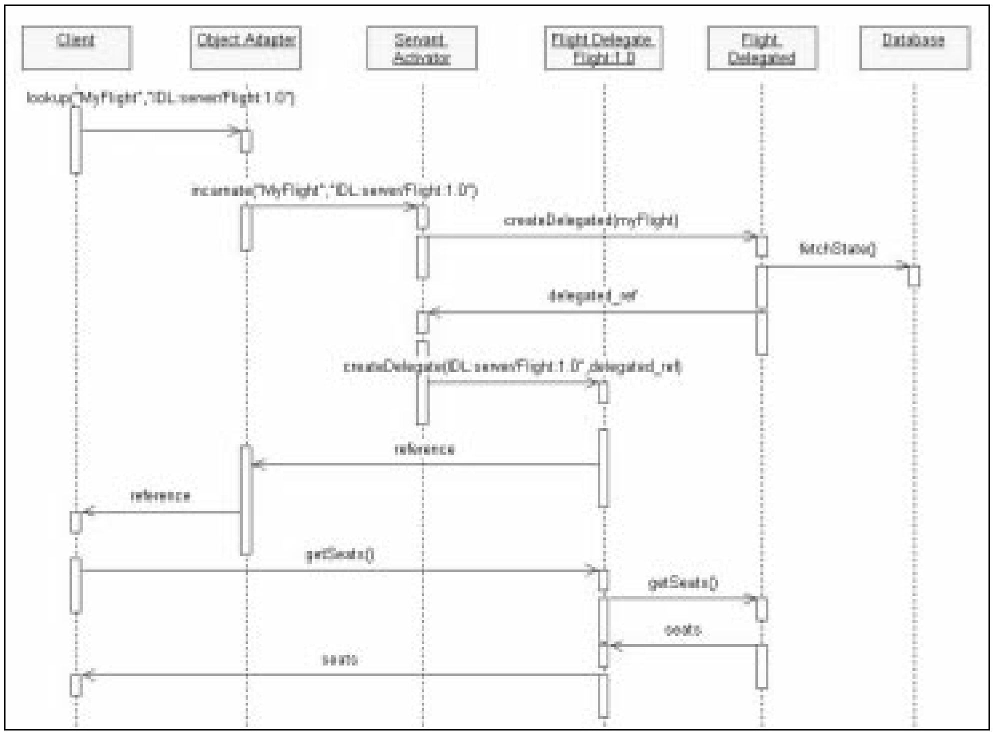

Client
Object Adapter
Servant Activator
Flight Delegate Flight 1.0
Flight Delegated
Database
lookup("MyFlight","IDL:server/Flight:1.0")
incarnate("MyFlight","IDL:server/Flight:1.0")
createDelegated(myFlight)
fetchState()
delegated_ref
createDelegate(IDL:server/Flight:1.0",delegated_ref)
reference
reference
getSeats()
getSeats()
seats
seats

Illustration 17 – The Delegate Sequence Chart

The precondition for this scenario is the existence of a client with a reference for a object with which no servant is associated at the time the client makes a request on the reference. It is the responsibility of the Orb, in collaboration with the POA and the server application to find or create an appropriate servant and perform the requested operation on it. The version of the wrapper can be determined with help of the standard CORBA helper classes. These classes specify a id method that returns the id or Idl version of the current client. This method can be used to create the correct server version.

The main steps:

1st: Identification of the object version with CORBA standard code

2nd: Request to the portable object adapter for a object with the given reference and the needed object version

3rd: The object adapter launches the creation of a delegated

4th: The delegated fetches the object with the given reference from the database.

5th: The object adapter activates a new wrapper and tie object with the needed object version and the reference of the delegated object.

6th: Exporting the tie/wrapper object to the POA

7th: Exporting the reference to the client

8th: The client performs requests on the tie/wrapper object.

Part 4 - Prospects

Possible extensions for the AutoMate System and some future work are listed below:

- Support for different Database Management Systems with specified interfaces and services for model evolution.
- The Treatment and Implementation of Generalization is a yet unsolved problem
- Embedding of change managers into CASE-Tools. With the help of vendor neutral model specifications (XML and XMI) it’s possible to enhance already existing CASE tools.
- Definition and Implementation of a common CORBA Evolution service
- Definition of user specific conversion function to extend the automatic conversion for standard types
- Enhancement of model lists to model trees. In the moment only a strong sequential list of changes is handled by the system.

Part 5 - Literature References

[BRK 98]	Dr. Klaus Bergner, Andreas Rausch and Karsten Kuhla. *Schnelle Schichten – Transparenter Zugriff auf ODBMS über CORBA*. IX11/1998, p. x
[BRS97]	Klaus Bergner, Andreas Rausch and Marc Sihling. *Using UML for Modeling a Distributed Java Application*. TUM-I9735 Technische Universität München 1997
[BHRS97]	Klaus Bergner, Franz Huber, Andreas Rausch, Marc Sihling. *Component-Oriented Redesign of the CASE-Tool AutoFocus*. TUM-I9752 Technische Universität München 1997
[HKLP]	Gerd Hillebrand, Patricia Krakowski, Peter C. Lockemann, Dietmar Posselt. *Integration-Based Cooperation in Concurrent Engineering*. IPD, Universität Karlsruhe
[Orfa]	Orfali. *Client/Server Programming with Java and CORBA*
[OMG 99]	Object Management Group (OMG): *Common Object Request Broker Architecture Specification*. www.omg.org (1999)
[SUN]	*The Java Programming Language*. www.sun.com
[OHE]	Orfali, Harkey and Edwards *The Essential Client/Server Survival Guide*
[OMG 99b]	*The Unified Modeling Language*. www.omg.org
[Busch]	Buschmann. *Pattern oriented Software Architecture*
[Gamma]	Gamma, Entwurfsmuster – Elemente wiederverwendbarer Software. Addison-Wesley
[XML]	XML, Specification. www.w3c.org
[XMI]	XMI, OMG Specification. www.omg.org
[Oest]	Oesterreich. Objektorientierte Softwareentwicklung
[SERC]	*Software Engineering with reusable components*
[Vers]	*Versant's Database Manual*
[JBR]	Jacobsen, Booch and Rumbaugh. *The Unified Software Development Process*

Part 6 - APPENDIX

6.1 The Database Environment

One of the most interesting parts in the AutoMate architecture for scheme evolution is the used database environment and the usage of this database. Currently AutoMate is using Versant's OBDMS. This object oriented database delivers some standard settings for Scheme Checking and Evolution. In the current version of AutoMate this functionality is not used, because as said before every time a change of the model occurs the persistent objects will automatically deleted. But it's still very interesting for the later development how the database handles scheme checking and evolution. For this reason the database functionality is described below.

The first time an instance of a class is stored persistently in the database the scheme of the class of an object get implicitly defined and stored in the database. Objects of the class can be stored in the database only after the scheme for that class has been defined in the database. Whenever an object is being stored in the database, for the operation to complete successfully, the Java language binding representation (i.e. the Java class) has to be compatible with the database representation (i.e. the database scheme). An incompatibility could arise when the scheme for a particular class has been previously defined in the database and the corresponding Java class differs from what it was when the scheme was implicitly defined. The incompatibility check between the database scheme and the corresponding Java class is very strict. Even if the same underlying storage size is used in storing attributes of different Java language types, the transparent binding detects an incompatibility. There are three ways in which an incompatibility might arise:

- **N**ew **A**ttributes: A Java class could differ from the database scheme if it contains new attributes not present in the database scheme
- **L**ack **A**ttributes: A Java class could lack attributes that are present in the database scheme
- **D**ifferent **T**ype: A Java class could have an attribute of a different Java type compared to the attribute type when the scheme was originally defined in the database.

All of this three conditions constitute an incompatibility or a mismatch between the database scheme and the Java class in the transparent Java binding. Logically a mismatch can also occur if there is a combination of this three incompatibility types. For instance a mismatch might be due to a missing attribute and additionally to a type changed attribute.

The question is now how to handle this mismatches. To specify what you want to happen if a mismatch occurs, you can set the setSchemeOption() method in a database Session. Versants ODBMS offers three different parameters with different results for this method.

- SCHEMA_ADD_DROP_ATTRIBUTES: "If the Java class contains a different number of attributes than the database scheme, then modify the database class definition so that the database class has the same attributes as the Java class"
  Remark: Type changes of an attribute can be modelled as an attribute deletion with a following attribute creation.

Database Reaction	Consequence
Add attributes	Existing database instances will be initialized with null values for each of the newly added attributes
Drop attributes	The values for those attributes are deleted from the database

↳ SCHEMA_FORCE_DROP_DATABASE: "If there is any mismatch between the Java class and the database class, drop the database class entirely"

Database Reaction	Consequence
Drop database class entirely	Existing database instances of the database class will be deleted

↳ SCHEMA_THROW_EXCEPTION_ALWAYS: "If there is any mismatch between the Java class and the database class, raise an exception"
Remark: This is the default option if you do not call setSchemeOption(). When you invoke the setSchemeOption(), your choice of scheme handling options affects the current session only. Each time you begin a new session the default value will be chosen.
If a exception is thrown it has the following attributes

Exception Attribute	Description
AttrName	A Java String which is the name of the attribute in the class for which the exception is thrown
Category	A Java Integer with one of the following values: Attribute_Missing_in_DB, Attribute_Missing:in_Java or Scheme_Java_Type_Mismatch
Cls	The Java Class due to which this exception is thrown
DbType	A Java String that represents the Java type of the attribute when the scheme was defined in the database
JavaType	A Java String that represents the current Java type of the attribute in the Java class

These are the possibilities that the object oriented database system from Versant offers. It is possible to use this functionality directly in their AutoMate architecture to achieve the targets of Scheme Evolution, but this solution would be a solution especially for this database and not a preferred general database independent solution.

6.2 Pre- and Post conditions for Evolution

In this section the effects of model evolution operations or primitives on consistency and compatibility are discussed. Based on the theoretical part, the model change primitives for a basic class model are surveyed with their pre conditions and consequences on static and dynamic aspects.

Create an Attribute:

- Pre Condition: An attribute to be created should not already be defined in the class where it is to be inserted. Anyway, if this is allowed the new definition replaces the old one. This is treated like a rename or a retype of an attribute.

- Consequence: If an attribute was already inherited to the class, it will now be re-defined.
- Post Condition: If an attribute is (re)defined, the new attribute is propagated to all subclasses where it is added as an inherited attribute, as long as it is not already defined locally there.

Create an Attribute:

- If an created attribute already exist in the class or one of it's subclasses, no matter locally defined or inherited, all methods accessing the attribute may change their behavior or become invalid, if the attribute definition is not a generalization of the old one. This behavior must be handled by attribute change.
- If an attribute is completely new, no conflicts with old applications can occur .

Create a Method:

- If an added method is a redefinition of an inherited method and the new method has another semantic than the old one then there is no access to the scope of the old method anymore. This might lead to behavior changes.
- If other methods refer to the new method by a different signature the method call will be invalid.

Create a Class:

- Class names have to be unique, this has to be preserved when adding a class
- A class can be inserted anywhere in the inheritance path, with the default position as a new subclass from a phantom class
- If super classes are specified the new class inherits all attributes and methods of this classes, but redefines not unique definitions.
- If subclasses are specified, this classes inherit all attributes and methods and the changes are propagated through the whole inheritance graph until there is a locally redefinition.

Create a Class:

- Class names have to be unique, if a class overwrites an old one then the application behavior may change.
- If a class creation changes the inheritance graph and redefines a method than the new method is chosen from the application and not the old one. This may cause behavioral inconsistency.
- Otherwise creating a class is behavioral consistent

Add a super class to the inheritance list:

- The super class has to be defined
- A new link is not allowed to add a cycle in the inheritance graph
- All attributes and methods are inherited from the super class and propagated to subclasses of the inheritance graph, until they are redefined

Add a super class to the inheritance list:

- Adding an inheritance link leads to addition of methods and attributes to the subclasses

Delete an Attribute:

- An attribute to be deleted has to be defined or redefined in the class from which it shall be deleted.
- It is not possible to delete an inherited attribute.
- If an attribute was redefined, the inherited feature now replaces the redefined on. This can be treated like an attribute rename or retype.
- The deleted attribute also has to be deleted from all subclasses.

Rename an Attribute:

- The attribute has to be locally defined
- For all name changes the new name should not introduce any new conflict, i.e. it should be unique within the class. Therefore a name check has to be performed.
- When the change is accepted, it has to be propagated to all subclasses as long as it is not redefined there.

Retype an Attribute:

- The attribute has to locally defined
- For all type changes the new type should not introduce any type conflict
- When the change is accepted, it has to be propagated to all subclasses as long as it is not redefined there.

Delete an Attribute:

- If a deleted attribute is replaced by an inherited one with the same type nothing changes.
- If a deleted attribute is replaced by an inherited one with a different type, methods using this attribute might change their behavior or become invalid
- If a deleted attribute is not replaced, methods referencing this attribute become invalid, the same for references to the attribute in subclasses which where formerly available through inheritance

Rename an Attribute:

- If an attribute is renamed and replaced by an inherited one with the same type nothing changes on application side.
- If an attribute is renamed and replaced by an inherited one with a different type the application might change their behavior.
- For a name change without replacement, the access to the attribute under the old name becomes undefined, therefore all methods using the old name are invalid too.
- Renaming might change the validity and behavior of inherited classes

Retype an Attribute:

- If an attribute is retyped and replaced by an inherited one with the old type nothing changes on application side.

- If an attribute is retyped and replaced by an inherited one with a different type the application might change their behavior.
- For a type change without replacement, the access to the attribute with the old type becomes undefined, therefore all methods using the old type are invalid too.
- Retyping might change the validity and behavior of inherited classes

The Evolution of methods in the context of structural consistency is exactly the same as for attributes

Delete a Method:

- If a method is deleted and not replaced with an equivalent inherited one then the application method calls are invalid
- If a method is deleted and replaced with a not equivalent inherited one then the application changes their behavior.
- If a method is deleted and replaced by inheritance with a equivalent one this means name, type and signature then the application behavior might neverthe-less change because of a different method semantic.

Rename a Method (Change the name):

- If a method is renamed and not replaced by an inherited one then application method calls for the old name are invalid
- If a method is renamed and replaced by a syntactic and semantic equivalent one via inheritance than the application behavior remains the same, otherwise it might change.

Retype a Method (Change the signature):

- If the signature of a method is changed then the only way to keep the behavior of an application is that it is replaced by an equivalent one via inheritance, oth-erwise the behavior of the application might change
- Method calls from application to the old signature become invalid

Delete a Class:

The effects of a class deletion depend on its position in the inheritance graph

- The class has to be defined
- If the class is a leaf node, i.e. it does not have any subclasses, then it can be deleted without affecting other class structures
- If the class has subclasses, then all inheritance links have to be deleted. With this deletion all references to attributes and methods of this class become inva-lid.

Rename a Class:

- The class has to be defined
- All subclasses inheriting from the old class become invalid, if the name change is not propagated to the subclasses or a equivalent class is defined.

Delete a Class:

- All references to the class, it’s attributes and it’s methods become invalid, thus, all methods referring the class become invalid
- All references to attributes and methods passed to subclasses become invalid too, as long as they are not locally redefined.

Rename a Class:

- All method code referring explicitly to the class by it’s name become invalid.

Remove a class from the inheritance list:

- The class must be new established in the inheritance graph either must inherit from the phantom class or the predecessor of the deleted class.
- All attributes and methods which have been inherited from the super class and which are not redefined have to be deleted from the subclass definition. This changes have to be propagated to the inheritance graph.

Remove a class from the inheritance list:

Deleting a link has the effect that all methods and attributes which have been inherited from the super class are no longer available for the subclasses. Therefore code referring to them becomes invalid.