TUHH
Technische Universität Hamburg-Harburg

Generation of Mapping Code for Conceptual Content Management based on Model Matching

(Includes post submission fixes of spelling and punctuation.)

Master Thesis Report

Submitted in partial fulfillment of the requirement for the award of the Degree of

Master of Science in Information and Communication Systems

Submitted by:

Venkata Ravi Kishore Chayanam

Matriculation Number 23421

Hamburg, Germany

07 August 2006

Under the Esteemed Guidance of
Prof. Dr. Joachim W. Schmidt (STS)
Prof. Dr. Friedrich Mayer- Lindenberg (TI6)
Dr. Hans-Werner Sehring (STS)

Department Softwaresysteme
Hamburg University of Science and Technology

Abstract

Database schemata evolve continuously in real-world applications. Schema transformation modules facilitate the communication between incompatible systems with different Asset models. Because of the openness and dynamics properties of database schemata in Conceptual Content Management Systems, schema transformation modules have to be generated dynamically based on schemata changes. To do so, a Schema transformation module generator will be developed. Within the planned generator, model mappers are used to match outdated and revised database schemata using name based matching(similar to schema matching technique) and to compute the mappings between them. The Strategy pattern is followed to design an interface for a family of model mapping algorithms. One implementation for a model mapper algorithm is provided. The Adapter pattern is used to convert the interface of an outdated class to the interface of a revised class according to the chosen model mapping algorithm. Transformation code is generated for classes using Java meta model and after target code is generated in Java, the instances of schema transformation module symbol table are filled.

Keywords: Database schemata, schema transformation module, Conceptual Content Management System, Strategy pattern, Schema matching, model mapping, Adapter pattern, schema transformation module symbol table, Java meta model.

Declaration

I declare that:

This Master thesis work has been prepared by myself, all literal or content based quotations are clearly pointed out, and no other sources or aids than the declared ones have been used.

Hamburg, 07 August 2006
Venkata Ravi Kishore Chayanam

ACKNOWLEDGEMENTS

I take this opportunity to thank all those magnanimous persons who rendered their full services to my Master thesis project work.

It is with lots of happiness I express gratitude to my supervisor Dr. Hans-Werner Sehring, for timely and kind help, guidance and valuable suggestions whenever I used to digress away from the aim of the project work, giving me innovative ideas during the design and implementation phases of the thesis work. Also provided me with the most essential materials required for the completion of this report. This inspiration up to the last moment had made things possible in a nice manner.

I would like to thank Prof. Dr. Joachim W. Schmidt, Head of the Department of SoftwareSysteme and the staff members of the Department especially M.Sc Sebastian Bossung for the kind cooperation and valuable suggestions he gave me while revising the report, with out which it would not have been possible for me to get a good and challenging research topic for the Master thesis.

Special acknowledgements to Prof. Dr. Friedrich Mayer- Lindenberg, Head of the Department of Computer Technology for supervising the Master thesis as a second examiner.

The blessings of the God Almighty has been a continuous source of strength and confidence through out the thesis work, so I take it as a special privilege to sincerely express my gratitude to the God for everything and also to my family members.

Contents

LIST OF FIGURES

1. Introduction

1.1 Conceptual Content Management Approach

Users can work collaboratively on information systems which are based on a common conceptual content model. In many applications users need to have a personal, subjective view of the world. Such applications require openness of the model so that users can change it according to their needs. The system must then react to these changes dynamically without intervention of a developer. Information systems usually do not address the issue of having users who do not have to agree on a common conceptual content model in order to work collaboratively. According to Cassirer [1, 2] and others, entity modelling processes, in order to be successful have to meet the following goals:

- Openness: Users can express their personal view of the application domain by openly changing the conceptual model underlying the system.

- Dynamics: Users can create instances of their views in the system based on their personal model with out the involvement of a developer.

Conceptual Content Management achieves openness and dynamics with the three main contributions given below:

1. Asset Definition Language: Content is represented by multimedia documents. The concept consists of “characteristic” properties of the entities, its “relationships” with other entities, and “constraints” on these characteristics and relationships. Based on the observation that neither content nor concept can exist in isolation, conceptual content management is based on the joint representation of pairs of content and concept. Each pair is called an “Asset”.

2. Asset model Compiler: A typical Conceptual Content Management System (CCMS) instance is generated by an Asset model Compiler which is designed as a framework. The framework follows the typical structure of compilers consisting of frontend and backend components, where generators form the backend. Several generators are developed and each generator creates a small part of the CCMS.

3. System Architecture: The Conceptual Content Management System architecture consists of modules, components and systems. The Compiler framework generates modules. Each module is a self-contained unit. Sets of modules all of which use the same model are combined to form components. These modules achieve a certain task. Components are combined to form systems. Distinct components are necessary as the same system often uses multiple models.

More detailed and specific information related to openness, dynamics, Asset definition language, model Compiler and system architecture can be found in [3].

1.2 Typical Conceptual Content Management System Modules

Modules of a Typical Conceptual Content Management System allow us to keep configuration flexible. Modules are self contained units and they are smallest building block of a CCMS. There are different types of modules for a typical CCMS as given below:

- Client Modules
- Server Modules
- Schema Transformation Modules
- Mediation Modules
- Distribution Modules

The five kinds of modules of a Conceptual Content Management System listed above are described in the following subsections:

1.2.1 Client Modules:

Client modules map all calls they receive onto some third party system. They do not use any further modules. The most common use is to store Asset instances in a database.

1.2.2 Server Modules:

Server modules are complementary to client modules. They use modules on the layer below them. A server module can be used for a web application interface.

1.2.3 Schema Transformation Modules:

The two interfaces for these modules conform to different schemata. The generators for these modules either are provided with information on how to map these schemata or they use schema matching techniques.

1.2.4 Mediation Modules:

Mediator is used to provide a homogenous access to different information sources.

1.2.5 Distribution Modules:

Distribution modules offer remote communication between components.

Asset models evolve continuously in real world applications. As described above, schema transformation modules facilitate the communication between incompatible systems with different Asset models. We generate a schema transformation module and it uses schema matching technique to compute the differences between two Asset models. A schema transformation module is developed to achieve Asset model personalization and model evolution.

More specific and detailed information about CCMS modules can be found in [3].

1.3 Functionality of Schema Transformation module

Public Asset model is an Asset model which is made publicly available to the general users and private Asset model is an Asset model for which access is allowed to specific users and private Asset model is a result of personalization. Asset class definitions are enclosed in the keyword model which is defined in the Asset definition language. Schema transformation module has the following functionalities:

- A schema transformation module generator transforms the public Asset model up/down to the private Asset model.
- A schema transformation module also provides rights control access to private Asset model.

Other important functionalities of Schema transformation modules can be found in [3].

1.4 System Evolution

CCMSs evolve over time following the dynamics property. They can evolve due to model evolution, model personalization, application domain combinations or software evolution. Model evolution allows users to redefine their Asset models. Model personalization allows users to personalize a publicly available model. New Asset classes can be derived by combining specialized application domain models.

To dynamically adapt Conceptual Content Management Systems to changing models they are recompiled at runtime. The demand for dynamics leads to system evolution and the specific information related to dynamics of a system can be found in [4].

The evolution of Conceptual Content Management Systems has two aspects:

- the software needs to be modified, and
- existing Asset instances need to be maintained.

Typical issues with respect to these two aspects of evolution are:

- Changes performed on behalf of individual users should not have any impact on others. Therefore, dynamic support for system evolution must not prevent continuous operation of the software system.
- On the one hand, Assets as representations of domain entities cannot automatically be converted in general. On the other hand, manual instance conversion is not feasible for typical amounts of Asset instances.
- If a user personalizes Assets for his own needs, he still will be interested in changes applied to the original. Through awareness [3] measures he can be informed about such changes. To be able to review the changes, access to both the former and the current versions are needed. That is, revisions of Assets and their schemata need to be maintained.

A detailed description about CCM System Evolution can be found in [6].

1.5 Organization of this Thesis

The remainder of this thesis report is organized as follows.

In *Chapter 2*, the problem that is being solved by the development of the schema transformation module generator is described.

Chapter 3 deals with explaining several significant concepts of Conceptual Content Management Systems and the evolution steps - model evolution and model personalization.

In *Chapter 4,* the reader will be introduced to the design and implementation of generated code for various imperative adapters.

In *Chapter 5,* the design and implementation aspects of schema transformation module generator are discussed.

Chapter 6 deals with the evaluation of the developed transformation module generator and the test cases will be developed and they will be executed against the developed module generator.

In *Chapter 7, a* summary of the work done and future work that can be done are discussed.

2 The Problem with Evolving Models

2.1 Problem Description

The dynamics property of CCMSs necessitates that the systems have to evolve continuously during their life cycle. The several reasons existing for evolution of systems are discussed in detail in the section 2.1.1 Model Evolution and section 2.1.2 Model Personalization. Most evolution steps are handled by adding modules through application of the mediator architecture [7].

2.1.1 Model evolution

Models evolve during their lifetime and the dynamics property of CCMSs will help address this evolution. Users can redefine their Asset model because of the openness property of the model. In order to explain the model evolution step, the above figure 1 is considered. If the model M_1 is an outdated model and model M_2 is updated version, queries are wrapped through the transformation module, which will remove those parts of a query that apply to the model changes only, and will translate back the results.

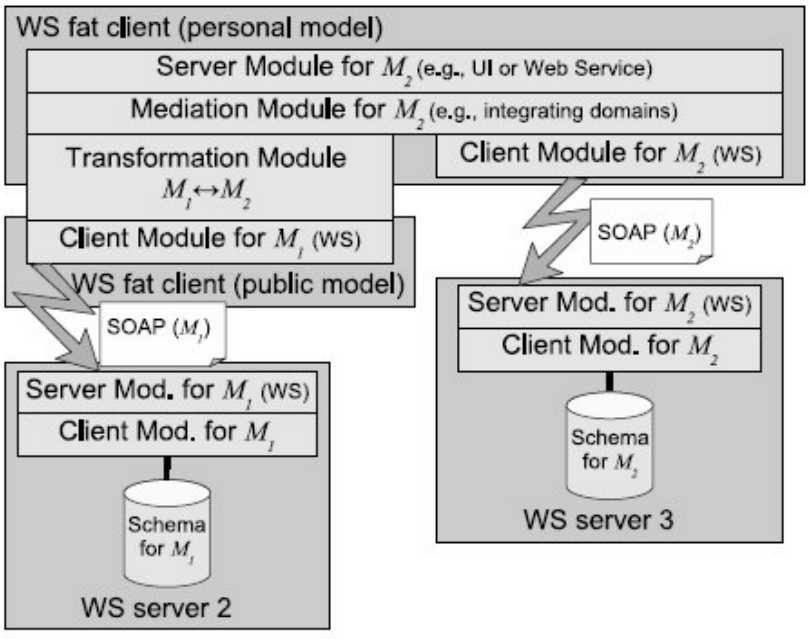

Figure 1 Components implementing a fat client scenario [8]

When querying for Assets, mediation module will concurrently retrieve from newer component of M_2 and the older one for M_1. Query results are merged, since the Assets have already been transformed to M_2. New Assets are always created in the component for M_2 and no Asset is retrieved twice.

Specific details about model evolution explained above can be obtained from [8]

2.1.2 Model personalization

Model personalization allows the adaptation of data and its representation. Personalization of characteristics and relationships of an Asset class in an Asset model is supported by the Schema transformation module.

2.1.3 Overview of CCM Module API and Asset object API

Asset models change over time rendering the current Asset classes outdated. For each Asset class *A,* API Generator creates CCM Module API which has the following interfaces:

- Interfaces for the states/roles of an Asset object
- An iterator interface for collections of Asset objects
- A factory interface to create Asset objects
- An interface for query classes
- A visitor interface to distinguish between subtypes
- A visitor interface for Asset object states

All of the above mentioned interfaces except visitor interfaces will change whenever the Asset class definition changes. The current implementation needs to be adapted to suit the modified Asset class definition.

Asset roles are reflected in interfaces of Assets objects, i.e. the Asset object API. There are five such interfaces in the Asset object API for an Asset class *A*, namely

- Abstract*A*
- *A*
- AbstractMutable*A*
- Mutable*A*

- New*A*

for the possible states/roles of Asset objects.

More Specific description about the CCM Module API and Asset object API introduced above is given in the section 5.1 and any further information can be found in [9].

2.1.4 Members of an Asset

Each Asset class in an Asset model can change over time. Members of an Asset class are Content handles, Characteristics, Relationships and Constraints. When an Asset class changes we can end up with the following types of members in the modified Asset class:

- New members
- Unused members
- Unchanged members
- Changed members

New members: These are the members that do not exist in the outdated Asset class and are added to the revised Asset class. Default initial values should be assigned to these members when they are used for the first time.

Unused members: These are the members that exist in the outdated Asset class and do not exist in the revised Asset class. References to these members must be removed when transformation is done. All constraints to these members should also be removed.

Unchanged members: These are the members that exist in both the outdated Asset class and the revised Asset class. All references to these members must be delegated to the outdated Asset class.

Changed members: These are the members whose properties changed when the Asset class is modified. All references to these members should address the changes made to the members.

2.2 The Proposed solution

A Schema transformation module generator has to compute the differences between outdated and revised Asset models. This process is known as model matching or Schema matching [10]. Schema matching is the process of identification of correspondences, or mappings between schema objects. Any combination of content, characteristics and relationships of an Asset class can change over time.

Manual schema matching has its limitations:

- It is generally impossible for a human to validate and modify Asset schemata.
- Manual schema matching is generally time consuming and may be unfeasible especially with large schema definitions.

An automated model mapper needs to be developed which computes these differences between an outdated Asset class and the current Asset class and hence obtain the mappings between them. There are several possible implementations of model mappers – *matching model mapper*, and any other automated model mapping algorithms.

Matching model mapper is the mapper which computes the differences between two Asset classes in two models by comparing them by name. Matching model mapper computes new members, unchanged members, changed members and unused members between an outdated Asset class and the current Asset class.

Schema transformation module generator generates the following classes for each changed Asset class in a model:

Adapter – Adapter is generated using Object Adapter [11, 12] Design pattern. Adapter implements the interfaces for the states/roles of an Asset object. This class implements the interfaces A, MutableA, and NewA for an Asset class A. This adapter also contains an implementation of the outdated Asset class as a delegate. Delegate method calls are made for all unchanged members of an Asset class, since these members are unchanged. Initial values are assigned for new members of the Asset class. This is the class where initial values for content objects and characteristics need to be assigned. These values need to be assigned for all the members that are added to an Asset class, because these did not exist in the outdated Asset class. These values to be assigned depend on the definition of the Asset model.

Factory – This class provides an implementation for the factory interface to create Asset objects. This class implements the interfaces AFactory for an Asset class A. This adapter also contains an implementation of the outdated factory implementation as a delegate. Delegate method calls are made for all unchanged members of an Asset class, since these members are unchanged.

Query – This adapter class is generated using the Object Adapter [11, 12] Design pattern. This adapter also contains an implementation of the outdated query class as a delegate. Delegate method calls are made for all unchanged members of an Asset class, since these members are unchanged. Constraints to unused members are removed. Constraints to new members are added.

IteratorAdapter – This adapter is generated using Object Adapter Design pattern. An iterator adapter implements the iterator interface for collections of Asset objects. This class implements the interface AIterator for an Asset class A. The outdated iterator implementation is included in the adapter as delegate. Delegate method calls are made for all unchanged members of an Asset class.

3 Conceptual Content Management Systems

3.1 Introduction

In Conceptual Content Management Systems (CCMSs), entities are modelled by content-concept pairs called Assets. Peirce [13] and others identified different perspectives that entity descriptions have to satisfy for modelling these types of entities. The three different perspectives are given below:

- the inherent characteristics of an entity
- Relationships between entities
- the systematics behind the above two perspectives

3.2 Asset Model

Assets represent intimately allied content-concept pairs which represent and signify application entities. The content and concept of an Asset are defined below:

- The content aspect of an Asset may contain an image of the entity.
- The concept aspect describes the entity by its characteristics and by relationships to other entities.

The representation of an Asset as a Content-Concept pair is shown in figure 2. Referring to the figure 2, Content aspect of an Asset represents a media view of the entity while the concept aspect represents its allied Concept view. A more detailed description about Asset modelling can be found in [3].

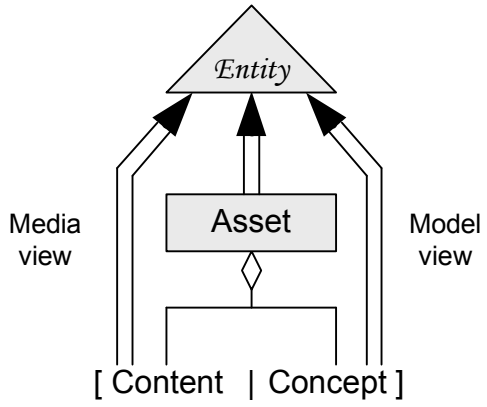

Figure 2 Asset as a Content-Concept pair [3]

3.3 Asset Systems Implementation – Asset Model Compiler

For Asset systems implementation in Conceptual Content management a two-step approach is adopted: The first step is driven by an Asset model Compiler, the second is based on a module configurator. The model Compiler translates ADL definitions into a set of modules of different kinds which form the basis for implementing that model. The configurator creates the modular structure of the executable target system for Asset modeling and management. This way, the goal of an open dynamic Asset system is achieved without paying the performance penalty for runtime interpretation, which – as experience from other projects show – could be prohibitively high.

The Asset model Compiler is designed as a framework. Java code generation toolkit [3] is a domain independent meta-programming [14] infrastructure. For generating the target code in a Schema transformation module generator Java code generation toolkit is used. A typical CCMS is created by instantiating the model Compiler framework, using an appropriate set of generators.

The definitions of API Symbol Table and Symbol Table are detailed in the following paragraphs:

API Symbol Table: Usually, the first generator run is API Generator, which produces Java interfaces (APISymbolTable) for the unique interface all modules have to fulfill.

Symbol Table: Data interdependencies are passed between generators using data structures called symbol tables. Each generator may read any number of symbol tables and produces exactly one symbol table.

The compiler itself controls the order in which generators are run and the data flow between the generators. Each generator produces a symbol table and data is passed between generators using this symbol table.

The basic structure follows the classical Compiler [15] architecture consisting of a frontend and a backend which communicate by exchanging some intermediate model.

3.4 Frontend of the Asset Model Compiler

The frontend of the model Compiler includes parsing and checking the Asset definitions. ADL Grammar and Intermediate Asset model are two significant contributions in the Frontend of the model Compiler.

3.4.1 ADL Grammar

Asset language syntax is described by the ADL grammar i.e. Asset definition language, Asset query and manipulation language. It is written for the ANTLR Parser Generator [16] which is used for the Asset model Compiler.

ANTLR, ANother Tool for Language Recognition, is a parser generator that accepts grammatical language descriptions such as Asset language definitions in the context of CCMS, and generates programs that recognize sentences in those languages. As part of a translator, one can augment his grammars with simple operators and actions to tell ANTLR how to build Abstract Syntax Trees(ASTs) and how to generate output. ANTLR knows how to generate recognizers in Java, C++, C#, and so on. ANTLR uses top-down parsing (or LL parsing).

ANTLR knows how to build recognizers that apply grammatical structure to three different kinds of input: (i) character streams, (ii) token streams, and (iii) two-dimensional trees structures. Naturally, these correspond to lexers, parsers, and tree parsers.

The lexical analyzer or lexer scans the input character stream and breaks up it into a stream of tokens and then the parser applies grammatical structure (syntax) to the token stream. Tree parsers are very much helpful in navigation through the nodes in parse trees.

The grammar of the Asset definition language i.e. ADL grammar is defined using some variant of EBNF, Extended Backus Naur Form [17].

A more detailed and specific introduction about ANTLR parser generator can be found in [16].

3.4.2 Asset Definition Language

Asset definition language is the language, which is used to define Asset classes. An example code for an outdated Asset model definition written in Asset definition language is shown in figure 3. This Asset model definition was outdated due to Model evolution.

Definition of an Asset class is organized in the model Statue using the keyword *model*. Asset class SomeEquestrianStatue is defined by the keyword *class* which has a *content* part image of type *java.awt.Image* and a *concept* part which has *characteristic* attributes sex and title of type *java.lang.String* and paintedAt characteristic attribute is of type *java.util.Date*. Two *relationship* attributes artist and requestBy are defined in the concept part of the Asset class SomeEquestrianStatue. Relationship attributes artist and requestBy have one-one relationship with another classes Artist, Ruler respectively.

An Asset class EquestrianStatue is the subclass of SomeEquestrianStatue, it extends the base class SomeEquestrainStatue under the keyword *refines*. The sub class EquestrianStatue inherits the content part, characteristic and relationship attributes from the concept part of the base class SomeEquestrianStatue. The sub class EquestrianStatue defines two characteristic attributes placeOfCreation of type *java.lang.String* and registrationNo of type *int*.

Two reference classes are also defined in the above Asset definition of model Statue, they are class Artist and class Ruler. Class Artist defines two characteristic attributes name and nationality of type *java.lang.String* in the concept part. Class Ruler contains concept part which defines a characteristic name of type *java.lang.String*.

```
model Statue

class SomeEquestrianStatue {
content
      image: java.awt.Image
concept
      characteristic sex: java.lang.String
      characteristic paintedAt: java.util.Date
      characteristic title: java.lang.String
      relationship artist: Artist
      relationship requestBy: Ruler
      relationship depicted: Person := self.requestBy

}

class EquestrianStatue refines SomeEquestrianStatue {
concept
      characteristic placeOfCreation : java.lang.String
      characteristic registrationNo: int
      }

class Artist{
 concept
   characteristic name      : java.lang.String
   characteristic  nationality : java.lang.String
}

class Ruler{
 concept
   characteristic name      : java.lang.String
}

class Person{
 concept
   characteristic name      : java.lang.String
}
```

Figure 3 Example code of an outdated Asset model definition

3.4.3 Asset Query and Manipulation Language

In addition to the Asset definition Language, there is also the Asset query and manipulation language. The language offers means to query on Asset instances using the lookfor command, creates new instances using the create command. The Asset instances can be modified by using the modify command, it is also possible to delete the existing Asset instances using the delete command. Each of these query and manipulation commands is available in several constraints in order to deal with single instance or set of instances at a time. In a complex task like the one shown below, the commands are combined.

modify
  lookfor Artist {name = “Picasso”} {nationality := “Spain”}

More details can be found in [18].

3.4.4 Intermediate Asset Model

Intermediate Asset Model: The parser in the frontend produces an internal representation of the Asset model to be translated in the form of an intermediate Asset model. Figure 4 illustrates the meta model of the intermediate Asset model. This intermediate Asset model is passed to the generators. The intermediate model contains the following:

- AssetClass: an Asset (name, superclass, members)
- Content handle: an object managing media data
- Characteristic: a characteristic attributes
- Relationship: a relationship attribute
- Constraint: a constraint which poses value restrictions on characteristic and relationship Attributes.
- Content handles, Characteristics, relationships, and constraints are called Members of an Asset class.

The terms introduced above related to intermediate model are detailed in the following paragraphs:

AssetClass has a content handle named as name of type String type String, Characteristic and Relationship are subclasses of Attribute.

Characteristic – Characteristic is of type *java.lang.Class* and every Characteristic can be assigned an initial value which is of type *ObjectExpression*.

Relationship: Relationship has a targetType – AssetClass and every relationship can have an initial binding of type AssetExpression.
Constraint:

1. Constraint term: Constraint rules can be set up on Characteristics and relationships using seven types of operators.

Possible comparators in constraint expressions test for equality (“=”), lesser (“<”), greater (“>”), different (“#”), or similar (“~”) values or bindings.

2. Compound Constraint: Compound constraint is created by combining two constraints and AND, OR connectors.

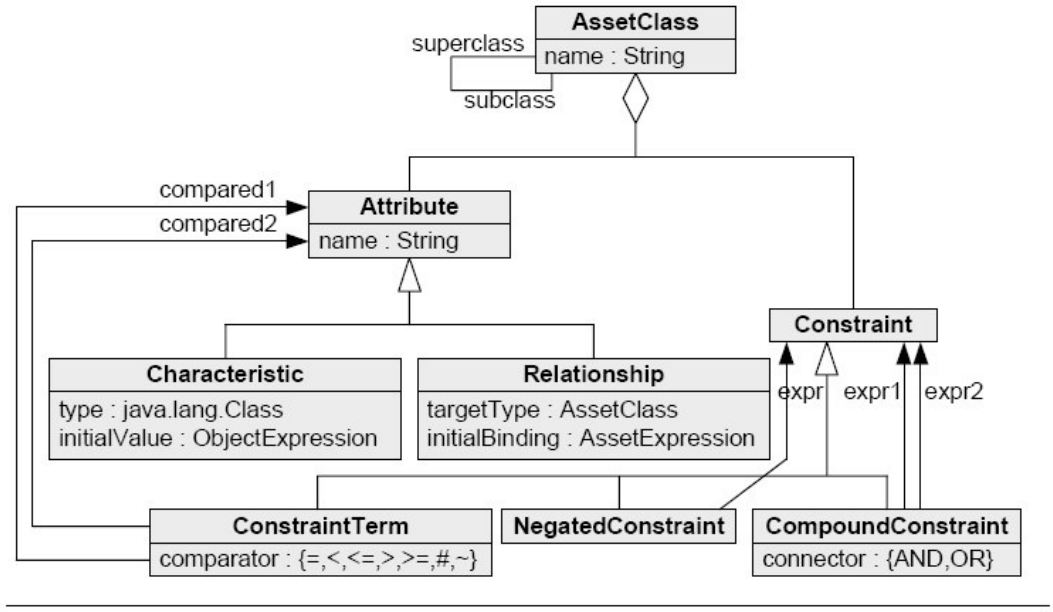

Figure 4 Meta model of the Intermediate Asset model [3]

A detailed description of the section 3.3.1- Asset model Compiler and section 3.4- Frontend of the model Compiler can be found in [3].

3.5 Backend of the Asset Model Compiler

There are generators for the various types of modules in the backend of the model Compiler: client modules, transformation modules, mediation modules, distribution modules, and server modules. The two main contributions in the backend of the model Compiler are CCM Module API, Asset Object API and Module Generators which are discussed in the section 5.1.

3.6 Typical System Architecture of Conceptual Content Management Systems

3.6.1 Components, Modules, Systems

A CCM system consists of a set of components reflecting one model each. These components are broken down into modules. The model Compiler creates modules, which are the basis of a domain –specific software architecture suitable for dynamic system generation [19]. The functionality of a component is defined by a component configuration. Figure 5 illustrates different abstraction levels of a Conceptual Content Management System. The implementation of these different abstraction levels will be provided as shown in figure 5.

Software-Technical Unit	Implementation through
Systems	Components, Cooperation
Components	Module, Service Interfaces
Modules	Assets, Standard operations
Assets	Objects, Methods
Objects	Data, Standard functions

Figure 5 Abstraction levels of a Conceptual Content Management System [3]

A CCM system consists of Components. Each component can contain several modules. Each module operates on several Assets.

3.6.2 Module Kinds – Separation of Concerns

In order to cope with the current purposes, five kinds of modules have been identified (as already introduced in section 1.2) in a typical CCM System are given below:

- Client Modules
- Server Modules
- Distribution Modules
- Mediation Modules
- Transformation Modules

Asset Management System with typical CCMS Module kinds- Server module, mediation module, distribution module, transformation module and a client module and the architectural overview is illustrated in figure 6.

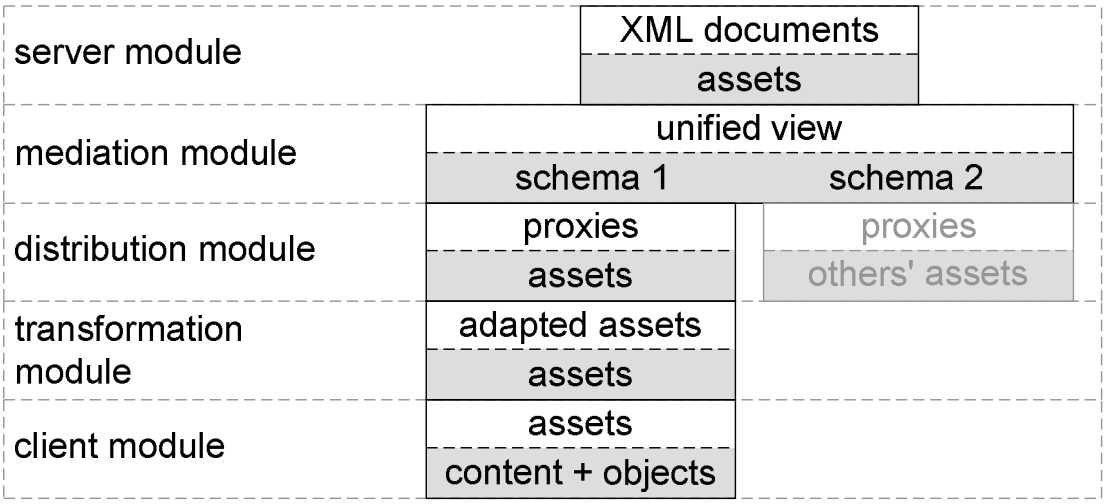

Figure 6 Asset Management System: Typical CCMS Module Kinds and Architectural Overview [20]

3.6.3 Recombinability and Reusability of Modules

The architecture of Conceptual Content Management supports the dynamic combination of instances of the various module kinds. These modules share some similarities with components [21, 22] (combinability, statelessness...), but in contrast to these they are generated for a concrete software system. Modules constitute the minimal compilation units of the generated software which the compiler can add or replace. The architecture enables reuse on several levels. In addition, any further description related to different levels of reuse in software can be found in [23]. A detailed description about combinability of modules can be found in [24].

4 Design and Implementation of a Code Generator for Model Mapping

4.1 Generator Options

All generators create code from Asset definition from one and only one Asset model. Transformation module generator, by definition, needs to work with the Asset definition from two Asset models.

A generator can be adapted to work with the Asset definition from two Asset models. This can be achieved in a couple of ways. They are described below:

4.1.1 Inner Generator

An inner generator is implemented inside a generator, the main generator (or the outer generator) works with the Asset definition from one Asset model and the inner generator works with the Asset definition from the second Asset model.

4.1.2 Inner Compiler

A compiler runs a generator which works with the Asset definition from one Asset model. An inner compiler is executed as a new process launched from the main compiler runtime. This inner compiler works with the Asset definition from the second Asset model and generates the APISymbolTable using the APIGenerator and it has a second generator to communicate the APISymbolTable to the main compiler.

The symbol table generated by the inner compiler is communicated back to the main compiler by using RMI in the form of a marshalling text file containing mappings for all the methods. The size of the marshalling text file to be communicated is very big as the SymbolTable class contains many methods, thus we have to provide all the mappings present in SymbolTable to marshalling text file.

The complexity of this approach lies in the file size and the innumerable mappings which have to be provided for all the methods present in SymbolTable class so that they can be converted into marshalling text file.

Inner Generator approach does not have this file communication issue as the entire involved between two processes. With Inner Generator approach, the entire configuration for the generator is available in a single file. With Inner Compiler approach, the configuration for the generator is spread out across two files, which is more difficult to maintain. So, Inner Generator is the choice of implementation for this thesis.

4.2 Generator Configuration Customization

A configuration file is used to configure the compiler framework. This file is used to specify the scanner, parser used by the compiler. This file also lists the generators in the compiler. This file also lists all the configuration parameters for each generator like output directory where the output of each generator is stored.

The configuration file for the compiler is modified to accept more parameters which are necessitated by using an inner generator. If an inner compiler were used, the configuration file would need less parameters.

The inner generator option necessitates that we capture more parameters like the location of the outdated Asset definition file.

4.3 Working with outdated and revised Asset models

A Compiler is run using the main method in Compiler class. An Intermediate Model is generated from the revised Asset model definition as part of this Compiler run. The intermediate model for the outdated Asset model definition needs to be generated when running the schema transformation module generator. To accomplish this, an extra input configuration parameter is used to specify the path to the outdated Asset model definition. Another input configuration parameter is used to specify the directory of the dictionary of the outdated Asset model definition.

The path to the outdated Asset model definition is set as the source for an *ADLScanner* to read the Asset definitions and statements from. The scanner is used for generating a token stream for an *ADLParser*. The ADLParser then parses the outdated Asset model definition and generates the Intermediate Model for the outdated Asset model definition.

4.4 Model Mapping Strategy / Asset Model Matching

There are various methods to compare and contrast two Asset models. The model matching algorithm which is used to match two Asset models in this thesis work is called Matching Model Mapper.

A program needs to be designed using which the mapping algorithm can be changed as required or as desired. The program also needs to be flexible enough to accept newer algorithms as they are developed. The program also should allow the algorithms to be modified as required. These algorithms are interchangeable and any one of them can be chosen at any given time.

A behavioral Design pattern, the Strategy [11, 12] pattern is used to design and develop a model mapper Context in which all the mapping methods are encapsulated. The mapping method to be used by the Transformation Module is captured as a configuration parameter. Figure 7 illustrates the UML [25] class diagram for the Design of Model Mapper following the Strategy pattern.

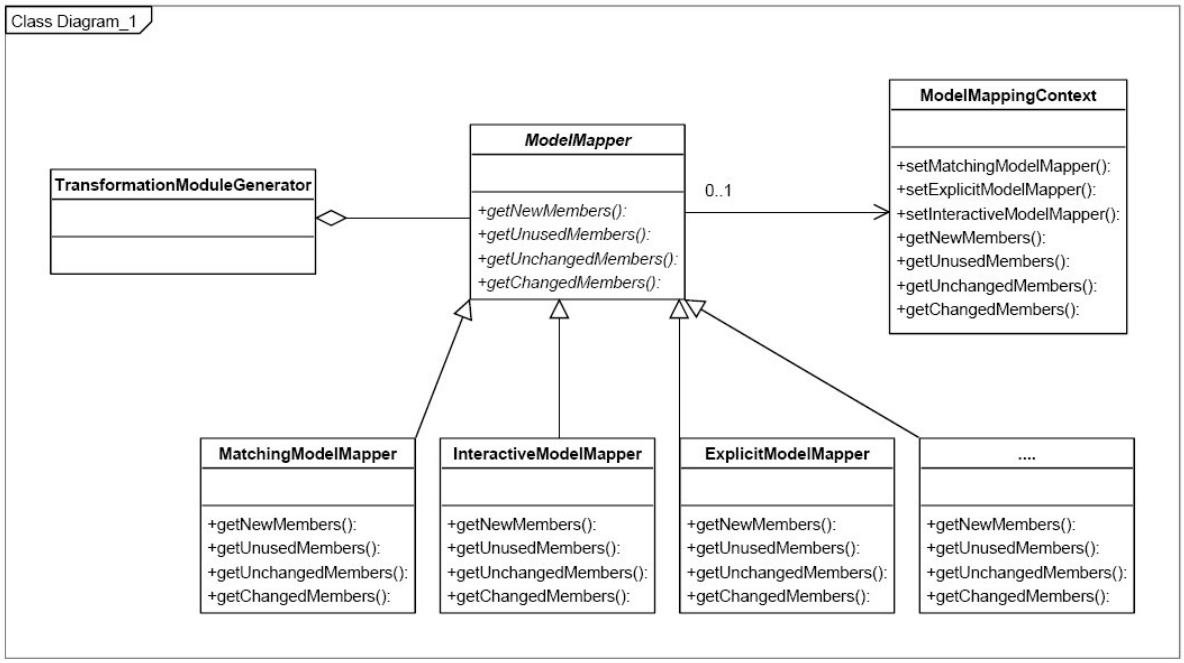

Figure 7 Design of Model Mapper following the Strategy pattern

The figure 8 shown below gives a clear idea about how Attributes can evolve from an outdated to a revised Asset models. Model matching does not consider the cases when a characteristic changes to relationship and when a relationship changes to a characteristic in an Asset model. All the other possible changes to Attributes are considered while Asset model matching is done in the Model mapping Strategy of a code generator.

conversion to from	Characteristic	Relationship
no Attribute	New characteristic	new relationship
Characteristic	another type	Asset set value instead
Relationship	projecting Assets on characteristics	change class references of Assets

Figure 8 Change of Attributes in an Asset model [3]

4.4.1 New Members

New members are defined as the members, both characteristics and relationships, which are found in the revised Asset model and which are not found in the outdated Asset model. A characteristic found only in the revised Asset model is a new member. A relationship found only in the revised Asset model is a new member.

To compute the new characteristics in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getNewMembers method. Each Asset class that exists in outdated and revised model is compared for characteristics by name. If a characteristic is found in an revised Asset class whose name does not match the name any of the characteristics in the outdated Asset class, then this characteristic is added to a hash table data structure whose key is the name of the Asset class.

To compute the new relationships in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getNewMembers method. Each Asset class that exists in outdated and revised model is compared for relationships by name.

If a relationship is found in an revised Asset class whose name does not match the name any of - the relationships in the outdated Asset class, then this relationship is added to a hash table data structure whose key is the name of the Asset class.

An example of a model written in Asset definition language is listed in figure 9. This is the updated version of the outdated model listed in figure 3.

New members in the revised Asset model in figure 9 when compared to the outdated Asset model in figure 3 are the following: The characteristic shortTitle in the Asset class EquestrianStatue and characteristic placeOfBirth in the Asset class Artist

4.4.2 Unused Members

Unused members are defined as the members, both characteristics and relationships, which are found in the outdated Asset model and which are not found in the revised Asset model. A characteristic in the outdated Asset model found missing from the revised Asset model is an unused member. A relationship in the outdated Asset model found missing from the revised Asset model is an unused member.

To compute the unused characteristics in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getUnusedMembers method. Each Asset class that exists in outdated and revised model is compared for characteristics by name. If a characteristic is found in an outdated Asset class whose name does not match the name any of the characteristics in the revised Asset class, then this characteristic is added to a hash table data structure whose key is the name of the Asset class.

To compute the unused relationships in a revised intermediate mode, all the Asset classes in outdated and revised models are collected in getUnusedMembers method. Each Asset class that exists in outdated and revised model is compared for relationships by name. If a relationship is found in an outdated Asset class whose name does not match the name any of the relationships in the revised Asset class, then this relationship is added to a hash table data structure whose key is the name of the Asset class.

Unused members in the revised Asset model in figure 9 when compared to the outdated Asset model in figure 3 are the following: The characteristic nationality in the Asset class Artist

4.4.3 Changed Members

Unused members are defined as the members, both characteristics and relationships, which are found in the outdated Asset model and which are not found in the revised Asset model. A change in a characteristic's type makes a characteristic a changed member. A change in a relationship's cardinality makes a relationship a changed member.

To compute the changed characteristics in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getChangedMembers method. Each Asset class that exists in outdated and revised model is compared for characteristics by name. If a characteristic is found not to be a new member, the type of characteristic in revised class is compared to the type of characteristic in the outdated class. If the type does not match, then this characteristic is added to a hash table data structure whose key is the name of the Asset class.

To compute the changed relationships in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getChangedMembers method. Each Asset class that exists in outdated and revised model is compared for relationships by name. If a relationship is found not to be a new member, the cardinality of relationship in revised class is compared to the cardinality of relationship in the outdated class and the type of relationship in revised class is compared to the type of relationship in the outdated class. If the cardinality or type does not match, then this relationship is added to a hash table data structure whose key is the name of the Asset class.

Changed members in the revised Asset model in figure 9 when compared to the outdated Asset model in figure 3 are the following:

The characteristic paintedAt in the Asset class SomeEquestrianStatue because the characteristic type changed from java.util.Date to java.util.Calendar

The characteristic registrationNo in the Asset class EquestrianStatue because the characteristic type changed from int to long

```
model Statue1
class SomeEquestrianStatue {
content
      image: java.awt.Image
concept
      characteristic sex: java.lang.String
      characteristic paintedAt: java.util.Calendar
      characteristic title: java.lang.String
      relationship artist: Artist*
      relationship requestBy: Ruler*
      relationship depicted: Person := self.requestBy
     }

class EquestrianStatue refines SomeEquestrianStatue {
concept
      characteristic placeOfCreation : java.lang.String
      characteristic registrationNo: long
characteristic shortTitle  : java.lang.String
}
class Artist{
 concept
    characteristic name         : java.lang.String
   characteristic placeOfBirth : java.lang.String
}

class Ruler{
 concept
    characteristic name      : java.lang.String
}
class Person{
 concept
    characteristic name      : java.lang.String
}
```

Figure 9 Example Code of a revised Asset model definition

4.4.4 Unchanged Members

Unchanged members are defined as the members, both characteristics and relationships, which do not differ from outdated Asset model and to revised Asset model. A characteristic which exists in both the outdated and revised Asset models and whose type did not change is an unchanged member. A relationship which exists in both the outdated and revised Asset models and whose cardinality did not change is an unchanged member.

To compute the unchanged characteristics in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getUnchangedMembers method. Each Asset class that exists in outdated and revised model is compared for characteristics by name. If a characteristic is found not to be a new member, the type of characteristic in revised class is compared to the type of characteristic in the outdated class. If the type matches, then this characteristic is added to a hash table data structure whose key is the name of the Asset class.

To compute the unchanged relationships in a revised intermediate model, all the Asset classes in outdated and revised models are collected in getUnchangedMembers method. Each Asset class that exists in outdated and revised model is compared for relationships by name.

Unchanged members in the revised Asset model in figure 9 when compared to the outdated Asset model in figure 3 are the following:

- The characteristic sex in the Asset class SomeEquestrianStatue
- The characteristic title in the Asset class SomeEquestrianStatue
- The relationship artist in the Asset class SomeEquestrianStatue
- The relationship requestBy in the Asset class SomeEquestrianStatue
- The relationship depicted in the Asset class SomeEquestrianStatue
- The characteristic placeOfCreation in the Asset class EquestrianStatue
- The characteristic registrationNo in the Asset class EquestrianStatue
- The characteristic name in the Asset class Artist
- The characteristic name in the Asset class Ruler
- The characteristic name in the Asset class Person

If a relationship is found not to be a new member, the cardinality of relationship in revised class is compared to the cardinality of relationship in the outdated class and the type of relationship in revised class is compared to the type of relationship in the outdated class. If the cardinality and type matches, then this relationship is added to a hash table data structure whose key is the name of the Asset class.

4.5 Model Mapping Strategy

4.5.1 Matching Model Mapper

Strategy [11, 12] design pattern is followed to implement model mappers. One of the supported algorithms is Matching Model Mapper. Matching model mapper matches two intermediate models and computes the new, changed, unchanged and unused members.

Collections of all the Asset classes in the revised and outdated intermediate models are created. Characteristics and Relationships of each Asset class in these collections are then examined to determine if they are new, unused, changed or unchanged members. The member name and the Asset class it belongs to are added to a hash table data structure. Matching Model Mapper returns these hash table data structures as the output of the mapping methods to compute the new, changed, unchanged and unused members.

4.5.2 Other Model Mappers

Any new implementation of a model mapper can be added easily as facilitated by the Strategy design pattern. All model mappers need to provide implementation for four mapping methods to compute the new, changed, unchanged and unused members.

4.6 Adapters for all Asset classes

Assets need to be converted from outdated to revised Asset model. This is achieved by following the Adapter [11, 12] design pattern. Adapters are generated for states/roles interface, query interface, and factory interface of the each Asset class so that they can be used with the data from outdated Asset model.

4.7 Target Code Generation of Asset objects using Java Code Generation Toolkit

Java code generation toolkit is used to generate all the adapters mentioned in 4.6. The adapters are generated as Java classes and they are created in the same output folder used by the API generator to generate the interfaces for the updated Asset model. The package name for these adapters is the same as the package name for the updated Asset model.

5. Design and Implementation of Generated Code for Adapters

5.1 CCM Module API and Asset Object API

There is one API generator which defines the Java interfaces, which has to be implemented by every module. CCM Module API and Asset object API are already introduced in Chapter 2, Section 2.1.3. The class diagram of different interfaces for representing the Asset life cycle is shown in figure 10.

The getX () method in AbstractA interface is used to retrieve the current value of a characteristic attribute defined in the concept part of an Asset model. The setX () method in the interface AbstractMutableA is used to set the value for a characteristic attribute defined in the concept part of an Asset model. accept () methods are defined for Asset classes which are derived from another Asset class.

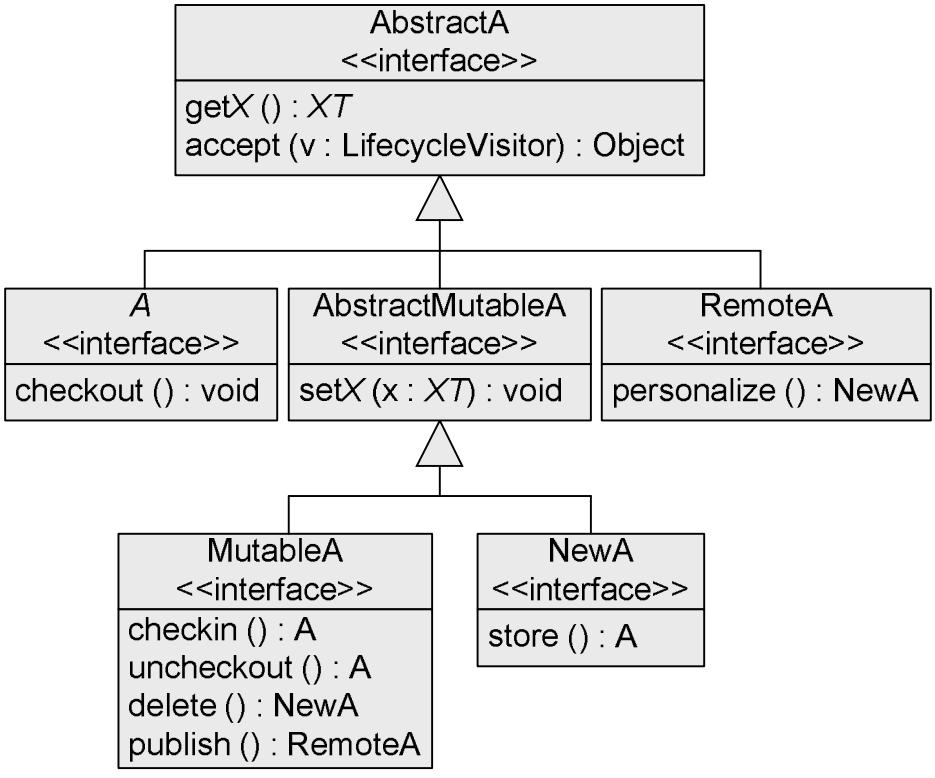

Figure 10 Class Diagram of Interfaces for illustration of Asset object life cycle. [3]

State Diagram for the Asset Instance Life Cycle is illustrated in figure 11.

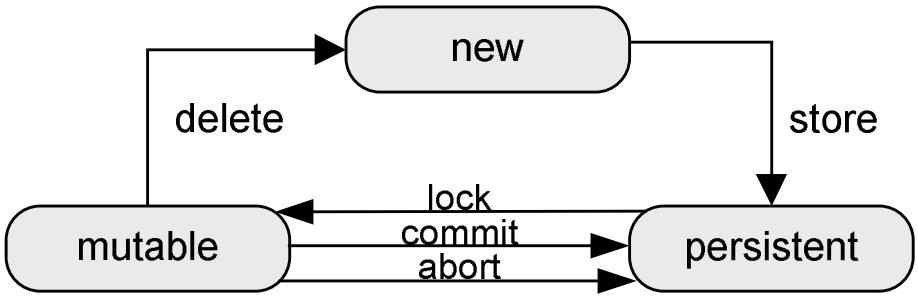

Figure 11 State Diagram for illustration of Asset object life cycle [6]

The state of an Asset object is changed by the following life cycle methods given below.

- lock ()
- commit ()
- abort ()
- delete ()
- store ()

If an Asset object is in locked state, it fulfils the interface MutableA which extends AbstractMutableA and the generic interface MutableAsset. It defines methods to change state:

- commit () method is used to make changes persistent, then the Asset object is in persistent state,
- abort () method is used to discard changes; then the Asset object is in a persistent state, and
- delete () method is used to transfer the Asset object to volatile state.

In Java:

public interface Mutable*A* extends AbstractMutable*A*, Mutable*B*

{

A abort () ;

A commit () ;

NewA delete () ;

} *// interface MutableA*

A newly created Asset object is in volatile state, indicated by the interfaces NewA, Subtype of AbstractMutableA and NewAsset (or NewB). This interface defines one method store () to transfer an Asset object to persistent state.

In Java:

```
public interface NewA  extends AbstractMutableA, NewB
{
  A store () ;
} // interface NewA
```

As long as it is in volatile state the Asset object is not persistent. Instead, it resides in the main memory only.

A detailed description about CCM Module API, Asset object API and Asset instance life cycle can be found in [1, 4].

5.2 Adapters

Asset class definitions change as Asset models evolve over time. The interface of an Asset class needs to be converted to match the interface of the changed Asset class. An instance of the Adapter [11, 12] design pattern can be used to convert the interface of an outdated Asset class to the interface of the corresponding revised Asset class.

Adapters can be of two types:

1. A new class is derived from the original class and methods are added to make the new class conform to the original class. This is called a class adapter [12].

2. The original class is included inside the new class and methods are created to adapt to the changes in the class. This is called an object adapter [12].

Figure 12 illustrates the Design for Generation of Transformation code following the Adapter pattern. An Asset class Artist which is introduced in the section 3.4.2 is used in this figure.

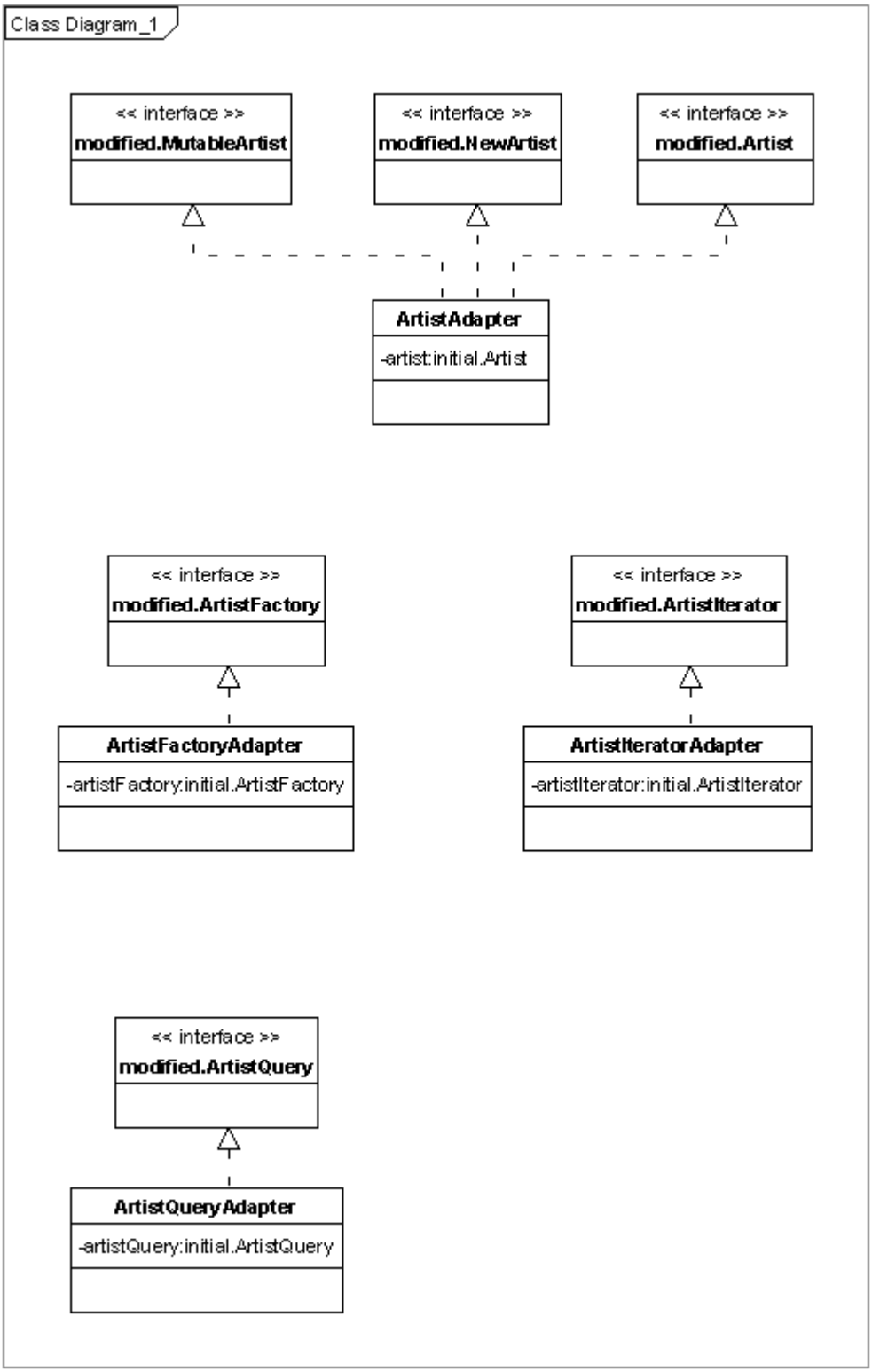

Figure 12 Design for Generation of Transformation code following the Adapter pattern

The generated adapter for an Asset class A implements the MutableA, NewA and A interfaces generated by the API generator. Setter and getter methods are implemented for each content type in an updated Asset class. These methods are delegated to the corresponding methods in

the outdated Asset interface. The set method sets the content value only when the Asset object is mutable.

Set and get methods are implemented for each characteristic in an updated Asset class. If the characteristic is a new member, a logical initial value is assigned to the characteristic in the set method. If the characteristic is a changed member, the set method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast to the outdated characteristic's type. If the characteristic is an unchanged member, the set method is delegated to the corresponding method in the outdated Asset interface with the unchanged member as the input parameter. The set method sets the value of the characteristic only when the Asset object is mutable.

If the characteristic is a new member, a logical initial value is returned in the get method. If the characteristic is a changed member, the get method is delegated to the corresponding method in the outdated Asset interface and the output is type cast to the updated characteristic's type when returning the value. If the characteristic is an unchanged member, the get method is delegated to the corresponding method in the outdated Asset interface.

Set and get methods are implemented for each one to one relationship in an updated Asset class. If the relationship is a new member, a logical initial value is assigned to the relationship in the set method. If the relationship is a changed member, the set method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast to the outdated relationship's type. If the relationship is an unchanged member, the set method is delegated to the corresponding method in the outdated Asset interface with the unchanged member as the input parameter. The set method sets the value of the relationship only when the Asset object is mutable.

If the relationship is a new member, a logical initial value is returned in the get method. If the relationship is a changed member, the get method is delegated to the corresponding method in the outdated Asset interface and the output is type cast to the updated relationship's type when returning the value. If the relationship is an unchanged member, the get method is delegated to the corresponding method in the outdated Asset interface.

Has, add and remove methods are implemented for each one to many relationship in an updated Asset class. If the relationship is a new member, the has method is not delegated to the outdated Asset class. If the relationship is an unchanged or changed member, the has method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast where required.

If the relationship is a new member, a logical initial value is assigned to the relationship in the add method. If the relationship is a new member, the add method is not delegated to the outdated Asset class. If the relationship is an unchanged or changed member, the add method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast where required. The add method adds a relationship only when the Asset object is mutable.

If the relationship is a new member, a logical initial value is assigned to the relationship in the remove method. If the relationship is a new member, the remove method is not delegated to the outdated Asset class. If the relationship is an unchanged or changed member, the remove method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast where required. The remove method removes a relationship only when the Asset object is mutable.

The implementation for all the other methods that need to be implemented for this class is provided by delegating to the outdated Asset class.

5.3 Factories

Object Adapter Design pattern is followed for designing an adapter for Factory class. Adapted Factory classes are generated to initialize any new members are added to an Asset class in the revised Asset model definition when compared to the outdated Asset model definition.

A factory adapter is generated for all Asset classes which has at least one subclass. The implementation for all the required methods that need to be implemented for this class is provided by delegating to the outdated Asset class. A create method is generated for each sub Asset class of the current Asset class.

5.4 Iterators

Object Adapter Design pattern is followed for designing an adapted Iterator class. Adapted Iterator classes for sets of objects are generated to return Adapter objects instead of the plain objects supplied by an underlying Iterator.

An iterator adapter is generated for all Asset classes. The implementation for all the required methods that need to be implemented for this class is provided by delegating to the outdated Asset class. The methods hasNext, next, remove, nextAsset, getLength, nextA are generated for each Asset class A and the implementation for these methods is provided by delegating to the outdated Asset class. A couple of constructors are also implemented in this adapter.

5.5 Query Objects

Object Adapter Design pattern is followed for designing an adapted Query class. Adapted Query classes are generated to remove constraints to any unused members found in the revised Asset model definition when compared to the outdated Asset model definition.

A query adapter is generated for all Asset classes. Seven methods – Equal, NotEqual, LessThan, LessOrEqual, GreaterThan, GreaterOrEqual, and Similar - are implemented for each characteristic in an updated Asset class. An overloaded method for each of these seven methods is also implemented by the adapter. If the characteristic is a new member, the method is not delegated to the outdated Asset class. If the characteristic is a changed member, the method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast to the outdated characteristic's type. If the characteristic is an unchanged member, the method is delegated to the corresponding method in the outdated Asset interface with the unchanged member as the input parameter.

Seven methods – Equal, NotEqual, LessThan, LessOrEqual, GreaterThan, GreaterOrEqual, and Similar - are implemented for each relationship in an updated Asset class. An overloaded method for each of these seven methods is also implemented by the adapter. If the relationship is a new member, the method is not delegated to the outdated Asset class. If the relationship is a changed member, the method is delegated to the corresponding method in the outdated Asset interface and the input parameter is type cast to the outdated relationship's type. If the relationship is an unchanged member, the method is delegated to the

corresponding method in the outdated Asset interface with the unchanged member as the input parameter.

A constraint on newly introduced attributes is checked against the default value for the corresponding attribute. If the default value matches the constraint, then the usual logical rules can be applied. An empty result set is returned for un-satisfiable query constraints.

5.6 Visitors

The implementation of Visitors is done following the procedure explained below:

- The interface *AbstractA* generated for an Asset class A in the Asset model defines an accept method which has to be implemented in all the subclasses of Asset class A.

- For Asset classes with no explicit base classes in the Asset model, a visit method is added to the generic interface *de.tuhh.sts.model.AssetVisitor* and the accept method is defined accordingly.

Referring to the outdated and revised Asset model definitions illustrated in figure 3 and figure 9 respectively we have the following Asset Classes:

SomeEquestrianStatue, EquestrianStatue, FemaleEquestrianStatue, MaleEquestrianStatue, Artist, Ruler and Person.

Therefore, the interface AbstractSomeEquestrianStatue generated for an Asset class SomeEquestrianStatue in the Asset model Statue defines an accept method which has to be implemented in the subclass of SomeEquestrainStatue i.e., Asset class EquestrianStatue. Similarly the interface EquestrianStatue generated for an Asset class EquestrianStatue in the Asset model Statue defines and accept method which has to be implemented in the subclasses of EquestrianStatue i.e., Asset classes FemaleEquestrianStatue and MaleEquestrianStatue.

Finally, for Asset classes Artist, Ruler and Person which do not have any explicit base classes in the Asset model Statue, a visit method is added to the generic interface *de.tuhh.sts.Statue.AssetVisitor* and the accept method is defined accordingly.

5.7 Inventing Initial Values

When a new member is added to an Asset class in the revised Asset model definition, the member needs to be initialized with a relevant value. If initial values are not supplied, there are three possible choices to choose from as follows:

- Produce an error
- Use zero for integers, null for objects and false for Boolean variables.
- Use arbitrary values for characteristics and arbitrary bindings for relationships.

The generated adapters use zero to initialize new members whose type is primitive integer, null to initialize new members whose type is java.lang.Object, and false to initialize new members whose type is Boolean. A null value is set as the binding for new relationships in the generated adapters. The cases mentioned just now are just few specific cases for which new members can be of type primitive integer or java.lang.Object or Boolean and there can be many other cases conceptually, depending on the type of the each member of an Asset class that can be present in the Asset model definition.

5.8 Content handles

Implementation for Content handles is provided to handle media data in the content part of an Asset model definition.

Set and get methods are implemented for each content in an updated Asset class. The set method is delegated to the corresponding method in the outdated Asset interface with the unchanged member as the input parameter. The set method sets the value of the content only when the Asset object is mutable. The get method is delegated to the corresponding method in the outdated Asset interface.

5.9 Module class

Module class is created by a Component according to its configuration. A module has to provide a default constructor for its creation. A new module instance is initialized by a call to init (Component, String, Map). Init method is used to get hold of parameters that are required

in the start method. During its lifetime, the module class should be activated and deactivated by calls to start () and stop () methods. A module instance should allocate and free resources accordingly.

5.10 Target Code Generation of Complete module using Java Code Generation toolkit

Target code for complete module i.e., Asset classes are generated using Java Code Generation toolkit according to the definitions in the revised API symbol table. To reflect the actual Java code instances of a symbol table for the schema transformation module have to be filled.

The Java Code Generation Toolkit enables convenient code production by programs. It offers classes comparable to those found in the package *java.lang.reflect* of the standard Java libraries [26]. In contrast to those the toolkit allows to manipulate and create classes.

The symbol table generated for the schema transformation module is a data structure which stores all the information regarding the schema matching results of the outdated and revised Asset models generated by the model mapper and the outdated and revised intermediate models, and the symbol table also stores all the generated Adapters which are generated by the transformation module.

Therefore, all other modules of Conceptual Content Management System can obtain any necessary information related to schema matching results generated by the model mapper and also about the Adapters generated by the schema transformation module from the schema transformation module symbol table.

6. Evaluation

6.1 Test Cases

6.1.1 GKNS as a test application

Geschichte der Kunstgeschichte im Nationalsozialismus (GKNS) [27] web application is developed using Conceptual Content Management. The usage of this application when the Asset model for this application evolves is a use case for transformation module generator.

When the Asset model used for GKNS is redefined, transformation module is used to generate the required adapters which convert the Asset classes from outdated model to Asset classes of the updated model.

6.1.2 Another sample test application

Figure 8 lists all the possible ways in which an attribute can evolve in an Asset model. An updated Asset model is defined for an outdated Asset model which will have all the possibilites listed in figure 9. The updated Asset model is listed in figure 9 and the outdated Asset model is listed in figure 3.

Transformation module generator is run on these two models and the generated adapters are verified manually for accuracy of results.

6.2 Results

6.2.1 Results for GKNS as a test application

The results of the generated Adapters for the use case GKNS web application are not listed in the report due to space restrictions.

6.2.2 Results for sample test application

The list of generated adapters for the sample test application mentioned in Section 6.1.2 is ArtistAdapter: The characteristic placeOfBirth is a new member and the characteristic name in the Asset class Artist is an unchanged member. Selected parts of the generated adapter code are listed in figure 13 – the code for an unchanged attribute and for a new attribute is listed

here. Set and get methods are verified for placeOfBirth and name characteristics in this adapter. It is also verified that placeOfBirth has a default initial value. The generated code is verified for the implementation details explained in Section 5.2.

```
// generated by the generator
(de.tuhh.sts.cocoma.compiler.generators.ex.TransformationModuleGenerator)
// (C) 2005-2006 STS
package de.tuhh.sts.statue1;
public class ArtistAdapter extends java.lang.Object implements
                de.tuhh.sts.statue1.Artist, de.tuhh.sts.statue1.MutableArtist,
                de.tuhh.sts.statue1.NewArtist {
        private de.tuhh.sts.statue.AbstractArtist delegate;
        …
        public void setName(java.lang.String name) {
                final java.lang.String namefinal = name;
                java.lang.Object e = delegate
                                .accept(new de.tuhh.sts.cocoma.generic.LifeCycleVisitor() {
                                        public java.lang.Object visit(de.tuhh.sts.statue1.Artist a)
                                        {
                                                return new java.lang.IllegalStateException(
                                                                "Delegate is not mutable!");
                                        }

                                        public java.lang.Object visit(
                                                        de.tuhh.sts.statue1.MutableArtist m) {
                                                m.setName(namefinal);
                                                return null;
                                        }

                                        public java.lang.Object visit(
                                                        de.tuhh.sts.statue1.NewArtist n) {
                                                n.setName(namefinal);
                                                return null;
                                        }
```

```
                                        private void setName(de.tuhh.sts.statue.AbstractArtist a)
                                        {
                                                ((de.tuhh.sts.statue.AbstractMutableArtist) a)
                                                                .setName(namefinal);
                                        }

                                        public java.lang.Object visit(
                                                        de.tuhh.sts.cocoma.generic.Asset a) {
                                                return null;
                                        }
                                        public java.lang.Object visit(
                                                        de.tuhh.sts.cocoma.generic.NewAsset n) {
                                                return null;
                                        }
                                public java.lang.Object visit(
                                                        de.tuhh.sts.cocoma.generic.MutableAsset
m) {
                                                return null;
                                        }
                                });
                        if (e != null)
                                throw new java.lang.IllegalStateException(
                                        "Delegate is not mutable!");
                }

                public java.lang.String getName() {
                        return delegate.getName();
                }

                public void setPlaceOfBirth(java.lang.String placeOfBirth) {
                }
```

```
public java.lang.String getPlaceOfBirth() {
        return "";
    }
    …
}
```

Figure 13 Generated Java code for ArtistAdapter

ArtistIteratorAdapter: The characteristic placeOfBirth is a new member and the characteristic name in the Asset class Artist is an unchanged member. Selected parts of the generated iterator adapter code are listed in figure 14. The generated code is verified for the implementation details explained in Section 5.4.

```
// generated by the generator
(de.tuhh.sts.cocoma.compiler.generators.ex.TransformationModuleGenerator)
// (C) 2005-2006 STS
package de.tuhh.sts.statue1;

public class ArtistIteratorAdapter extends java.lang.Object implements
        de.tuhh.sts.statue1.ArtistIterator {
    private de.tuhh.sts.statue.ArtistIterator iter;

    …

    public de.tuhh.sts.cocoma.generic.AbstractAsset nextAsset() {
        return new de.tuhh.sts.statue1.ArtistAdapter();
    }

    public int getLength() {
        return iter.getLength();
    }
    public de.tuhh.sts.statue1.AbstractArtist nextArtist() {
        return new de.tuhh.sts.statue1.ArtistAdapter();
    }
}
```

Figure 14 Generated Java code for ArtistIteratorAdapter

ArtistQueryAdapter: The characteristic placeOfBirth is a new member and the characteristic name in the Asset class Artist is an unchanged member. Selected parts of the generated query adapter code are listed in figure 15 – the code for an unchanged attribute is listed here. The generated code is verified for the presence and correct implementation of Equal, NotEqual, LessThan, LessOrEqual, GreaterThan, GreaterOrEqual, and Similar methods for all characteristics and relationships as explained in Section 5.5.

```
// generated by the generator
(de.tuhh.sts.cocoma.compiler.generators.ex.TransformationModuleGenerator)
// (C) 2005-2006 STS
package de.tuhh.sts.statue1;

public class ArtistQueryAdapter extends java.lang.Object {
        private de.tuhh.sts.statue.ArtistQuery delegate;

        private boolean isFalsification;
        …
        public void constrainNameEqual(java.lang.String name) {
                if (!this.equals(name))
                        isFalsification = true;
        }

        public void constrainNameEqual(java.lang.String name, byte connector) {
                boolean match = this.equals(name);
                if (match && (connector == 2))
                        isFalsification = false;
                else if (!(match && (connector == 1)))
                        isFalsification = true;
        }

        …

        public de.tuhh.sts.statue1.ArtistIterator execute() {
                if (isFalsification == true)
                        return (de.tuhh.sts.statue1.ArtistIterator) delegate.execute();
```

```
                else
                        return (de.tuhh.sts.statue1.ArtistIterator) delegate.execute();
        }
}
```

Figure 15 Generated Java code for ArtistQueryAdapter

EquestrianStatueAdapter

EquestrianStatueFactoryAdapter: Factory adapters are generated for Asset classes with sub classes. Selected parts of the generated factory adapter code are listed in figure 16. The generated code is verified for the implementation details explained in Section 5.3.

```
// generated by the generator
(de.tuhh.sts.cocoma.compiler.generators.ex.TransformationModuleGenerator)
// (C) 2005-2006 STS
package de.tuhh.sts.statue1;

public class EquestrianStatueFactoryAdapter extends java.lang.Object implements
                de.tuhh.sts.statue1.EquestrianStatueFactory {
        public de.tuhh.sts.statue1.NewMaleEquestrianStatue createMaleEquestrianStatue() {
                return new de.tuhh.sts.statue1.MaleEquestrianStatueAdapter();
        }

        public de.tuhh.sts.statue1.NewFemaleEquestrianStatue
createFemaleEquestrianStatue() {
                return new de.tuhh.sts.statue1.FemaleEquestrianStatueAdapter();
        }
}
```

Figure 16 Generated Java code for EquestrianStatueFactoryAdapter

- EquestrianStatueIteratorAdapter
- EquestrianStatueQueryAdapter
- FemaleEquestrianStatueAdapter
- FemaleEquestrianStatueIteratorAdapter
- FemaleEquestrianStatueQueryAdapter

- MaleEquestrianStatueAdapter
- MaleEquestrianStatueIteratorAdapter
- MaleEquestrianStatueQueryAdapter
- PersonAdapter
- PersonIteratorAdapter
- PersonQueryAdapter
- RulerAdapter
- RulerIteratorAdapter
- RulerQueryAdapter
- SomeEquestrianStatueAdapter
- SomeEquestrianStatueFactoryAdapter
- SomeEquestrianStatueIteratorAdapter
- SomeEquestrianStatueQueryAdapter

All of these adapters listed above are verified for compilation errors and then accuracy of the generated methods and their implementation.

7. Summary and Future Work

7.1 Summary

The dynamics property of CCMSs necessitates that the systems have to evolve continuously during their life cycle. Models evolve during their lifetime and the dynamics property of CCMSs will help address this evolution. Users can redefine their Asset model because of the openness property of the model. Model Personalization allows the adaptation of data and its representation. Personalization feature of only data is considered in this thesis work.

Schema transformation modules facilitate the communication between incompatible systems with different Asset models. The Schema transformation module generator computes the differences between outdated and revised Asset models using Asset model matching.

The source code for the outdated and revised Asset models is written in Asset definition language. Asset model matching is done to compute the model changes between outdated and revised Asset models. The model mapper is designed following the Strategy pattern. The model mapper computes the differences by matching classes and members by name. A matching model mapper is implemented in the model mapper. Other implementations of model mapping can be added to this model mapper.

Interfaces of classes need to be adapted to address the changes in an Asset model. The Object Adapter pattern is followed to design the required Adapters. Adapters are generated to adapt an outdated Asset model to the revised Asset model.

Java code generation toolkit is used to generate query objects, factories and iterators for all the Asset classes in a model. When a new member is added to an Asset class, the member needs to be initialized with a relevant value. For this purpose, initial values are invented for new members.

Target code is generated in Java, for the full module according to the definitions in the target symbol table and transformation symbol table instances are filled.

7.2 Usefulness of the Schema Transformation Module Generator

Schema transformation module generator developed in this thesis work has the following uses:

- Schema transformation module is used for dynamic model evolution and model personalization. When models evolve, the existing Assets should be adapted to work with the revised Asset model.

- Schema transformation module is used for transforming an outdated Asset model to a revised Asset model.

- Schema transformation module is used to provide rights control access to a private Asset model.

7.3 Future Work

In future developments, Asset model matching can be done by implementing any of the matching algorithms i.e., *explicit model mapper* or *interactive model mapper* or any other new matching algorithm. The advantage of the model mapper that is designed using the Strategy pattern lies in the fact that it is designed in such a way as to support any number of matching algorithms that might be developed in the future for computing the Asset model matching.

Referring to the figure 8, Asset model matching can be extended in future, to other cases i.e., when a characteristic changes to relationship and also when a relationship changes to a characteristic in an Asset model.

References

[1] Ernst Cassirer. Die Sprache, Das mythische Denken, Ph¨anomenologie der Erkenntnis, volume 11-13 Philosophie der symbolischen Formen of Gesammelte Werke. Felix Meiner Verlag GmbH, Hamburger Ausgabe edition, 2001-2002.

[2] Donald Verene, editor. Ernst Cassirer: Symbol, Myth, and Culture. Essays and Lectures of Ernst Cassirer 1935-1945. Yale University Press, 1979.

[3] Hans-Werner Sehring.: Konzeptorientierte Inhaltsverwalting:Modell, Systemarchitektur und Prototypen. Doctoral thesis, Technische Universität Hamburg-Harburg, 2004.

[4] Giorgio De Michelis, Eric Dubois, Matthias Jarke, Florian Matthes, John Mylopoulos, Joachim W. Schmidt, Carson Woo, and Eric Yu. A Three-Faceted View of Information Systems. Communications of the ACM, 41(12):64–70, 1998.

[5] P. Dourish and V. Bellotti. Awareness and Coordination in Shared Workspaces. In Proceedings of ACM CSCW 92 Conference on Computer-Supported Work, pages 107–114, 1992.

[6] Hans-Werner Sehring and Joachim W.Schmidt: Beyond Databases: An Asset Language for Conceptual Content Management. In: András Benczúr, János Demetrovics, and Georg Gottlob (editors), Proceedings of the 8th East European Conference on Advances in Databases and Information Systems, ADBIS 2004, Budapest, Hungary, September 22-25, volume 3255 of Lecture Notes in Computer Science, pp. 99-112. Springer-Verlag, 2004.

[7] Wiederhold G.: Mediators in the Architecture of Future Information Systems. IEEE Computer, 25:38–49, 1992.

[8] Sebastian Bossung, Hans-Werner Sehring, and Joachim W. Schmidt. Conceptual Content Management for Enterprise Web Services. In Jacky Akoka, Stephen W. Liddle, Il-Yeol Songand Michela Bertolottoand Isabelle Comyn-Wattiau, Willem-Jan vanden Heuvel, Manuel Kolp, Juan Trujillo, Christian Kop, and Heinrich C. Mayr, editors, *Perspectives in Conceptual Modeling: ER 2005 Workshops CAOIS, BP-UML, CoMoGIS, eCOMO, and QoIS*,

volume 3770 / 2005 of *Lecture Notes in Computer Science*, pages 343 – 353. Springer-Verlag, 2005.

[9] Hans-Werner Sehring: COCoMaS Module API: http://www.sts.tu-harburg.de/~hw.sehring/cocoma/projs/api/Module_API.pdf , 2004.

[10] Matteo M., Nikos R., Peter M.B., Danilo M.: Schema Integration Based on Uncertain Semantic Mappings. Conceptual Modelling – ER 2005. In proceedings of 24th International conference on Conceptual Modelling, pages 31-46, Klagenfurt, Austria, October 24-28, 2005.

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

[12] James W. Cooper: Java Design Patterns, A Tutorial, Addison-Wesley Professional, 2000.

[13] C.S. Peirce. Collected Papers of Charles Sanders Peirce. Harvard University Press, Cambridge, 1931.

[14] Yannis Smaragdakis, Shan Shan Huang, and David Zook. Program Generators and the Tools to Make Them. In PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pages 92–100. ACM Press, 2004.

[15]. Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman: Compilers: principles, techniques, and tools, Addison-Wesley Publishing Company, 1986

[16] Terence Parr, http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html , 2006

[17] Marcus A., Ivan P.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels, 2003

[18] Martin Abadi and Luca Cardelli. A Theory of Objects. Monographs in Computer Science. Springer-Verlag New York, Inc., 1996.

[19] Sharon Andrews White and C. Lemus. Architecture Reuse Through a Domain Specific Language Generator. In Proceedings of the Eighth Workshop on Institutionalizing Software Reuse, 1997.

[20] Joachim W. Schmidt and Hans-Werner Sehring: Conceptual Content Modelling and Management. In: Manfred Broy and Alexandre V. Zamulin (editors), Perspectives of System informatics, volume 2890 of Lecture notes in Computer Science, pp. 469-493. Springer-Verlag,2003.

[21] U. Aßmann. Invasive Software Composition. Springer-Verlag, 2003.

[22] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.

[23] Ted J. Biggerstaff. A Perspective of Generative Reuse. Ann. Software Eng., 5:169–226, 1998.

[24] Sebastian Bossung, Hans-Werner Sehring, Michael Skusa, and Joachim W. Schmidt: Conceptual Content Management for Software Engineering Processes. Advances in Databases and Information Systems. In proceedings of 9th East European Conference, ADBIS 2005, pages 309-323, Tallinn, Estonia, September 12-15, 2005.

[25] Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling User Guide, Addison-Wesley, 1999.

[26] Java 2 5.0 API, http://java.sun.com/j2se/1.5.0/docs/api/index.html , 2006.

[27] Geschichte der Kunstgeschichte im Nationalsozialismus, http://www.welib.de/gkns/

Appendix

Glossary

Adapter Pattern. Used to convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

ANTLR Another Tool for Language Recognition. It is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions.

AST Abstract Syntax Tree. A syntax tree in which each node represents an operator and the children of the node represent the operands and it is a useful starting point for thinking about the translation of an input string.

Compiler. Is a program that reads a program written in one language – the source language and translates it into an equivalent program in another language – the target language.

Design Pattern. Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.

EBNF Extended Backus Naur Form. It is an extension of the basic Backus–Naur form (BNF) metasyntax notation. The most commonly used variants of EBNF are currently defined by standards, most notably ISO-14977.

Factory Method Pattern. Used to provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Intermediate code. It is an explicit intermediate representation of the source program generated by some compilers after syntax and semantic analysis in the front end of the compiler.

Iterator Pattern. Used to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Lexer (Lexical analyzer). The stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

Metaprogramming. Metaprogramming is the writing of programs that write or manipulate other programs (or themselves) as their data or that do part of the work that is otherwise done at compile time during runtime. In many cases, this allows programmers to get more done in the same amount of time as they would take to write all the code manually.

Parser (Syntax analyzer). It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.

Strategy Pattern. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Symbol table. It is a data structure containing the record of each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

Top-down Parser (LL Parser). The top-down construction of a parse tree is done by starting with the root, labeled with the starting nonterminal, and repeatedly performing the following two steps:
1. At node n, labeled with nonterminal A, select one of the productions for A and construct children at n for the symbols on the right side of the production.

2. Find the next node at which a subtree is to be constructed.

Tree parser. It is used for navigation through nodes in parse trees.

Visitor Pattern. Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.