Conceptual Content Management Application Development by Means of Storyboarding

Jolita Savolskytė

Submitted in partial fulfilment of the requirements for the degree
Master of Science in Information and Media Technologies.

supervised by:
Prof. Dr. Joachim W. Schmidt (STS)
Prof. Dr. Helmut Weberpals (TI6)
M.Sc. Sebastian Boßung (STS)

Institute for Software Systems
Hamburg University of Science and Technology

August 2006

Abstract

These days the union of two existing information systems or using one system to complement the other with its specific features becomes a very common issue. The first system does not have a mechanism to specify a user interface in the mode of web applications; however the other system is a web application specification language which allows to generate a prototype web application by means of Storyboarding. Therefore, it would be beneficial to connect them and develop a web application by means of Storyboarding.

The first approach is based on the objects which have both data and behaviour whereas the web application entities are stored in the relational database. The goal of this project is to map this two different data models by bridging the “impedance mismatch”. For that, the mapping is defined which converts one data model into another.

Declaration

I declare that:
this 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, 28 August 2006
Jolita Savolskytė

Acknowledgement

I would like to thank Professor Joachim W. Schmidt of Institute for Software Systems (STS) for supervising my Master Thesis and Professor Helmut Weberpals of Distributed Systems Department (TI6) for being the co-supervisor.

Special thanks go to M.Sc.Sebastian Boßung for the interesting topic. He was very patient in guidance and providing advice during the project.

Thanks also to Alexander Bienemann of the University of Kiel and. Dr. Hans-Werner Sehring for answering the questions about the project.

I am grateful to my fiancé Piotr for the inspiration and moral support he provided throughout my project.

I would like to thank my mother and sister and all whose direct and indirect support helped me completing my thesis in time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The bringing together two information systems is a common task since many years. The system can be fully integrated into other system or it can use only the beneficial features of the other.

Conceptual Content Management (CCM) systems are represented by content-concept pairs which are called *Assets*. The conceptual part is needed to explain the way content refers to an entity. Content serves as an existential proof of the validity of concepts [SeSc04]. CCMS are open and dynamic, in the sense that users are able to change the assets definition and the system dynamically reacts to these changes without human intervention.

The storyboarding SiteLang language does not have this feature of *openness* and *dynamics* but a means of web application. On the other hand, CCMS does not have a mechanism to specify web applications, which makes a conjunction between SiteLang and CCMS beneficial.

Web Information Systems (WIS) is database-backed information systems that are distributed over the web and accessed via web browsers. These information systems retrieve data from the database and present them in a structured form, pages with text, images etc. The *storyboarding* paradigm gives the way to model such web information systems in terms of story space, scenarios, scenes, media objects and dialogs definitions. The websites are generable on the basis of the specification which is advantageous for the information-intensive web applications and error prone. Using this model various prototypical web applications could be generated, which correspond to volatile infrastructure requirements. They allow specifying the user interaction flow and the database behaviour in parallel. *SiteLang* is a storyboarding language developed at the University of Kiel and its semantics are based on operational and axiomatic semantics of Abstract State Machines (ASM) [BiZ01]. Its operational semantics are based on entity-relationship structuring.

The storyboarding SiteLang language allows specification of the complete web application, i.e. of structuring (structure, static integrity constraints), behaviour (processes and dynamic integrity constraints), information support (views, units and

containers) and of the interaction and story space (scenes, media objects, dialogs and dialogue steps with transaction semantics) as well as the relationships between them. Using such a structuring, it is easier to modify the complex specifications.

To compliment the features of these two systems, the CCM asset model has to be made available to the SiteLang specification. A mapping between assets model and the SiteLang data model has to be defined. The CCM systems can be extended by means of the generator which converts the domain model defined by asset class definitions into the SiteLang application entities stored in the relational database. Then using the SiteLang specification we can generate an application which is able to communicate with the user, web server, and the underlying database. It accepts the user input and generates database transactions and appropriate user dialogues. The dynamically generated dialogues are filled with data that are retrieved from the database views which are called media objects.

1.2 Structure of this Thesis

This thesis is organized into six chapters. This chapter describes the problem of this project.

Chapter 2 introduces one of two main technologies used in this project – Conceptual Content Management Systems as well as the asset modelling language.

Chapter 3 introduces the web application specification language SiteLang and its data model.

Chapter 4 discusses how to map two different data models introduced in chapter 2 and chapter 3.

Chapter 5 describes two CCMS generators developed to implement the mapping between CCMS and SiteLang entities according to the mapping rules defined in chapter 4.

Finally, the thesis ends with a conclusion in chapter 6 summarizing the key results of the presented work and giving an outlook on future works.

Chapter 2

Conceptual Content Management Systems

Content Management Systems (CMS) provide a complete framework for creating, managing, organizing, and publishing of documents and other content. Often, they are web applications for managing websites and web content, which have a user interface and a permanent data repository (disks, databases) for storing the content. CMSs usually provide much less information about how the content of an entity is related to its concept, that explains the characteristics and rules for the content, and limited functionality for presenting and using the content, particularly multimedia content.

Conceptual Content Management Systems (CCMS) connect the content and concept of the application entities. These content-concept pairs are called Assets. Figure 2.1 represents an Asset Model. The conceptual part is needed to explain the way content refers to an entity and how the content serves as existential proof of the validity of concepts [HSS04], thereby the value of content is improved. These two parts of an asset cannot exist in isolation since they refer to the same entity.

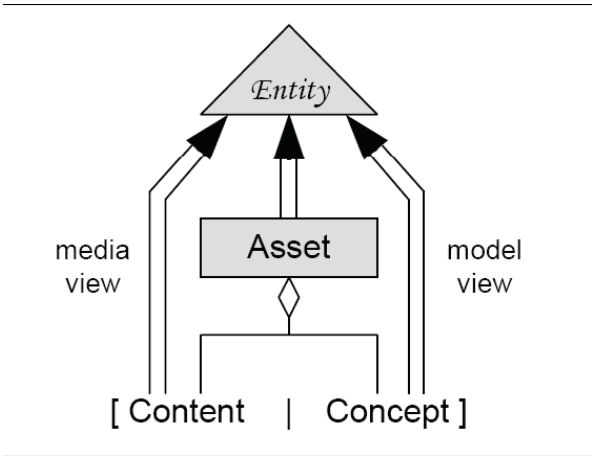

Figure 2.1: Representation of Asset Model [HWS03]

CCMSs allow a user to express his individual view on the application entities. The user is able to change the asset definitions. This property of the system is called openness, which is complemented by dynamics, i.e. the system reacts to the changes automatically without human intervention.

2.1 Asset Definition Language

The users may exchange information within the assets, therefore the definitions formulated have to be standardized and explicit. The modelling language Asset Definition Language (ADL) was developed for this purpose and is based on objects similar to the variables in object-oriented languages. The application entities are described by the asset classes, the structure of which is based on class definitions similar to the object-oriented world.

The structure of the asset classes will be described in more detail because it is a major subject of this project, and it will be used throughout the whole report.

The code in Figure 2.2 gives an example of the asset classes' definition. The definition of the asset model starts with the keyword **model** followed by its name and then includes all asset classes which belong to the model. The model has the ability to import asset definitions from other models using the keyword **import**.

```
model Person
from Department import Employe

class Picture {
    content contents: Image
    concept
        characteristic title: String
        characteristic placeOfCreation:Place
        relationship artist: Artist
        relationship topic: Subject*
        constraint artist.placeOfBirth = placeOfCreation
}
class Photo refines Picture {
    concept
        characteristic sizeX: int
        characteristic sizeY: int
}
```

Figure 2.2: Asset model definition example

The structure of the assets is similar to Java class definitions. They are introduced by the keyword **class** then follow the class name and its structure. It follows thence that the asset class can be addressed by the model and its name combination. The ADL language supports an inheritance of asset classes and this is implemented by the keyword **refines**. The asset class *Photo* subclasses the asset class *Picture*, i.e. as in object-oriented languages, all components of a base class will be inherited by a subclass or they may be overwritten.

As described above and shown in Figure 2.2, a body of the asset class is comprised of **content** and **concept** parts.

The content part of an asset has two assignments: firstly, it holds a reference to the asset location and access to its content. Secondly, it should provide preview information to the user so that he can imagine the content without seeing it. In the example above, the content *contents* of asset class *Picture* has a type of Java class *Image*. The content handles are defined by some object-oriented language type, standard or user-derived. Currently, the base language for handle types is Java language.

The conceptual part of an entity description is covered in the concept section of an asset class. It consists of two types of attributes and constraints:

1. **characteristic**. The characteristics describe the inherent properties of an entity. Each characteristic is identified by a name, and the type of their values is given after the colon. It could be either a Java class or a build-in data type. In the example, the type of the characteristic *title* in the asset class *Picture* is the standard Java class *String*.
2. **relationship**. It defines the relationships between assets. The relationship *artist* constrains a reference to the entity described by the asset class *Artist*. The asterisk (*) after the type *Subject* determines that the attribute topic refers to a set of asset *Subject* instances (many-to-many relationship).
3. **constraint**. The constraints restrict the values of the attributes of instances. In the example shown above, the *placeOfBirth* of the artist has to be the same as the *placeOfCreation* of the picture. The constraints can be expressed in many other ways which are not mentioned here but are found in [HWS03] and [RAD03].

2.2 ADL Compiler

In order to grasp the meaning of the generator used for data model mapping and, what it is used for, and where it fits in Conceptual Content Management Systems, this section will briefly introduce the architecture of the CCMS. About the generators is described in detail in [Sma04].

Because of the dynamics demand, the system should be able to adapt itself to changes. The model compiler generates the CCM systems based on the user defined domain model input. It is designed as a framework of generators, the structure of which follows the classical compiler architecture. It consists of a front-end and a back-end which communicate by interchanging the intermediate model.

The front-end compiler reads the user defined Asset Definition Language (ADL) input model, does the full lexical and syntax analysis of the model, and produces an intermediate representation of the input to produce an Intermediate model. This model is used by the back-end generators. For each asset class definition the Intermediate model creates the Java classes.

The back-end consists of the API generator which is a central generator of the system and produces the uniform module API and various module generators. The modules implement the functionality in order to offer the appropriate operations. “The

specific requirements of a concrete model are reflected by the asset parameters of the module's operations."[ScSe03]. Every generator is responsible for one kind of module. They receive input from the front-end. The model compiler starts the generators with the parameter Intermediate model. It decides itself the order in which generators should run.

Two generators which perform the mapping between the asset domain model and the SiteLang application entities are inserted in the CCMS architecture. The generators communicate with one another by means of symbol table. Every generator may read any number of symbol tables and produce exactly one symbol table.

The generated modules are combined into components which comprise the functionality of the CCM systems. The modules of the component are joined up to complete a specific task. Furthermore, the modules are arranged in the layers in the component, related to each other, and use the services supplied by the component. More detail description about the CCMS could be found in [HWS03].

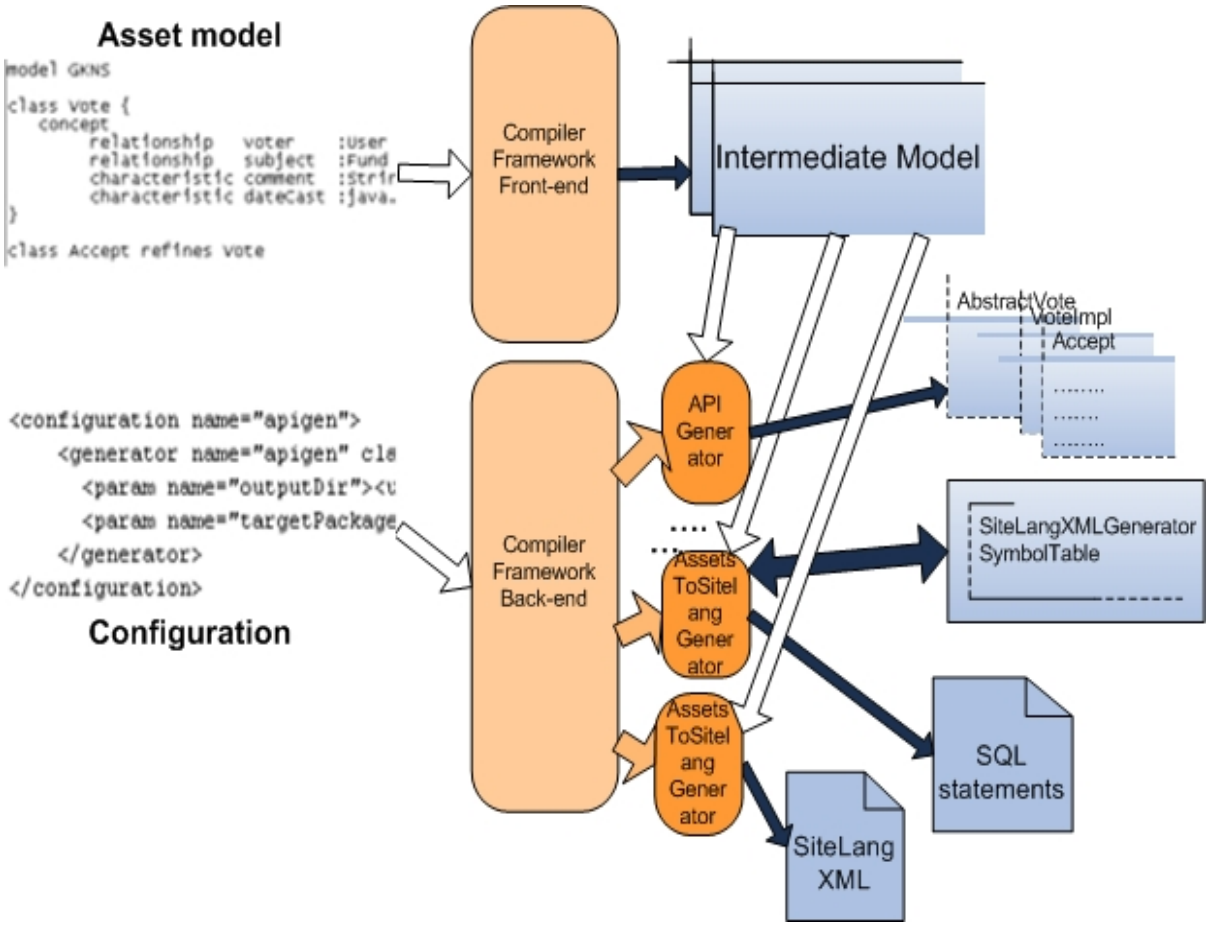

Figure 2.3: Model compiler architecture

Chapter 3

Storyboarding

This chapter will give a very brief introduction to the website specification language SiteLang that is used in this thesis. A more complete description can be found either in the [ABZ01] or in the Diploma Thesis of Ioannis Stragalis [IST05].

3.1 Overview of Website specification language SiteLang

The keyword Storyboarding comes from the movie industry. It comprises everything that will be contained in the website, e.g. what menu screens will look like, how the websites will be hyperlinked, in other words, everything that can be seen or heard or experienced by the end-user of the Web Information System (WIS). The SiteLang approach introduces a few new keywords to describe the WIS design. The whole WIS is understood as a story and the business logic flow of the application is represented by scenes. The user can specify the application story to his own needs by using the Storyboard editor.

The specification of the web application is stored in the meta database; the specification is developed by a sophisticated Storyboard editor, which integrates a generator that generates the XML files for the application and the files which provide the functionality and interactivity.

The storyboarding language SiteLang introduces several elements which describe the structure and semantics of the WIS. To get an overall picture, the main elements and terms will be described briefly below. Figure 3.1 is a graphical representation of the main storyboarding elements.

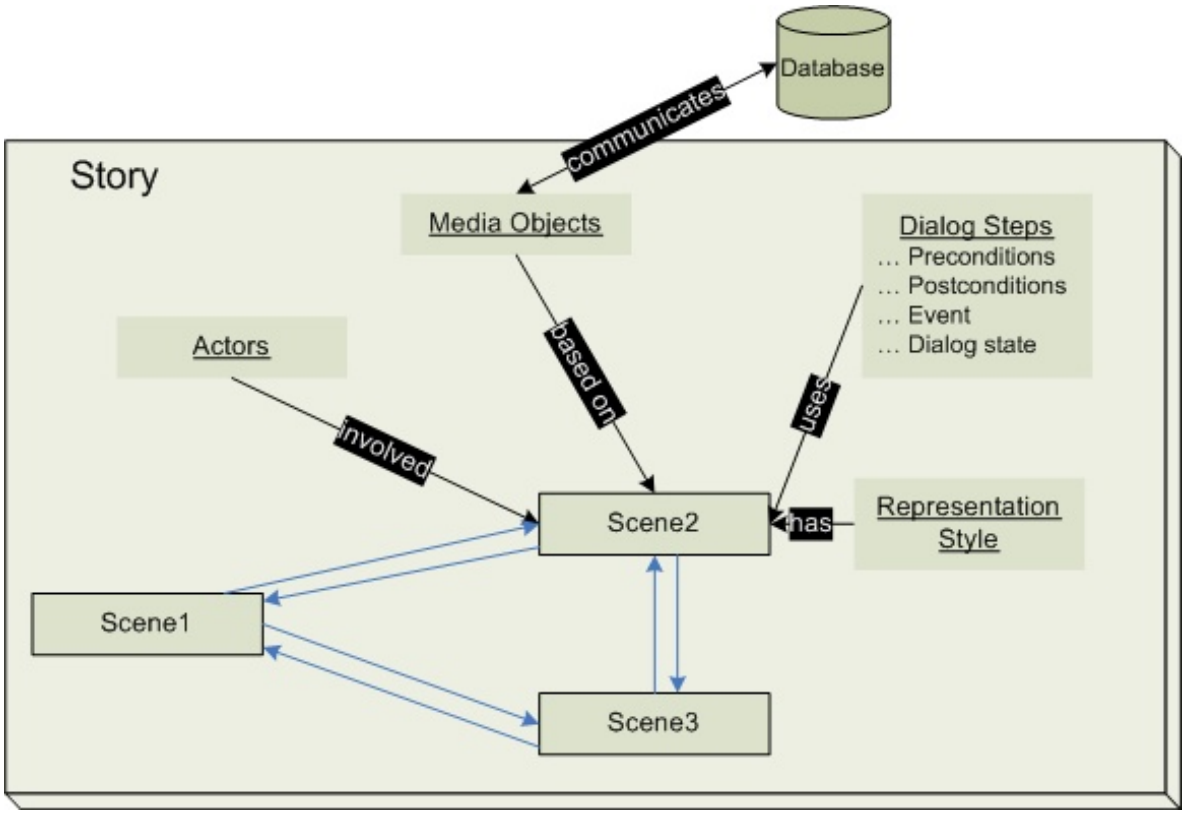

Figure 3.1: Representation of the main storyboarding elements

Story

An abstract layer, which describes the business logic of the Web Information System in terms of Storyboarding, is called a story. The subject of the story is the end-user, therefore while creating the story we have to take into consideration who will be using the system and then design the workflow.

Scenarios

While the user navigates in the WIS, he goes through a number of web pages to accomplish a defined task. This path is called a scenario (displayed by blue arrows in Figure 3.1). The story can be played in many scenarios which are separated into dialogues. The storyboarding often begins by modelling scenarios which are integrated into the story space.

Scenes

A scenario is composed of several scenes. The scene is the web page, which displays information to the user and offers an interface for interaction with the WIS. It is associated with involved actors, a media object, dialog steps and the representation styles (see Figure 3.1).

Actors

The users may have different behaviour, intentions and roles in relation to the Web Information System. They are grouped depending on their individual characteristics, assigned tasks, access to the system rights and roles. In order to distinguish the groups from single users, the user group profile is called an actor. The scenes are assigned to the actors who are allowed to access and execute the actions within them.

Dialog Objects

The end-user interacts with the Web Information System through a user interface which displays the information and accepts the input from the user. The user interface is characterized by dialog objects. They hold information which may be static and dynamic. The static elements display information to the end-user, such as simple text, labels of buttons, text input fields, and pictures, and the dynamic elements are linked with media objects which will be discussed in the next paragraphs. Other kinds of dialog objects trigger events depending on the user's action. A typical event is the user pressing a button. The actions triggered by users are called dialog steps which will also be discussed later.

Application Database

Today's Web Information Systems are database-backed systems to store the content of the application. The application database holds a complete description of the WIS specification in the SiteLang language. The SiteLang database contains a meta database for a full description of the application database structure which is based on the Higher-order Entity-Relationship Model (HERM) [BTH00].

In this Master's Thesis, mapping the CCMS data schema to the SiteLang database will be related to the Meta database. The schemata will be introduced in section 3.2.

Media Objects

The media objects are the mechanism to get the data from the application database. They are ordinary views on the application database.

Media objects may be created during the design stage of the application and parameterized by representation style, involved actors, and the context access. Other kinds of media objects are runtime media objects which are created dynamically depending on the information users enter, and all parameters are instantiated and extended by escort information [Tha00]. This information allows the user to see the history of performed steps.

The queries of media objects are in Query-by-Example (QBE) form. The basic syntax of QBE is introduced in [ABZ01], [JohQBE] and enhanced syntax in order to gain a better semantic and create complex queries in [IST05].

Dialog States

While the user interacts with the WIS, he executes a sequence of actions. The state of the system changes and it is associated with the scenario, e.g., based upon input

information provided by the user. The state of the system in Storyboarding is represented by dialog states.

Dialog Steps

As mentioned above, the dialog steps are the events triggered by user. They are the basic units of an action and use media objects. In a Web Information System, a webpage is a dialog step.

Dialog steps have pre- and postconditions, i.e. preconditions specify under which condition a dialog step may execute, and a dialog step may exit if postconditions are fulfilled. The dialog steps change the state of the system.

3.2 The SiteLang database structure

Nowadays, a database is necessary to hold and support almost every Web Information System with content data; therefore the web application prototype generated by SiteLang is not an exception. As mentioned above, it contains a meta database which part storing the application entities is depicted in Figure 3.2. For a complete description of the application database structure which data model is close to the HERM model see in [Tha00]. The used database management system is Sybase. The whole database structure is used to store the information about the structure and functionality, developed by a sophisticated Storyboard editor which also plays a major role in the consistent maintenance of database integrity constraints.

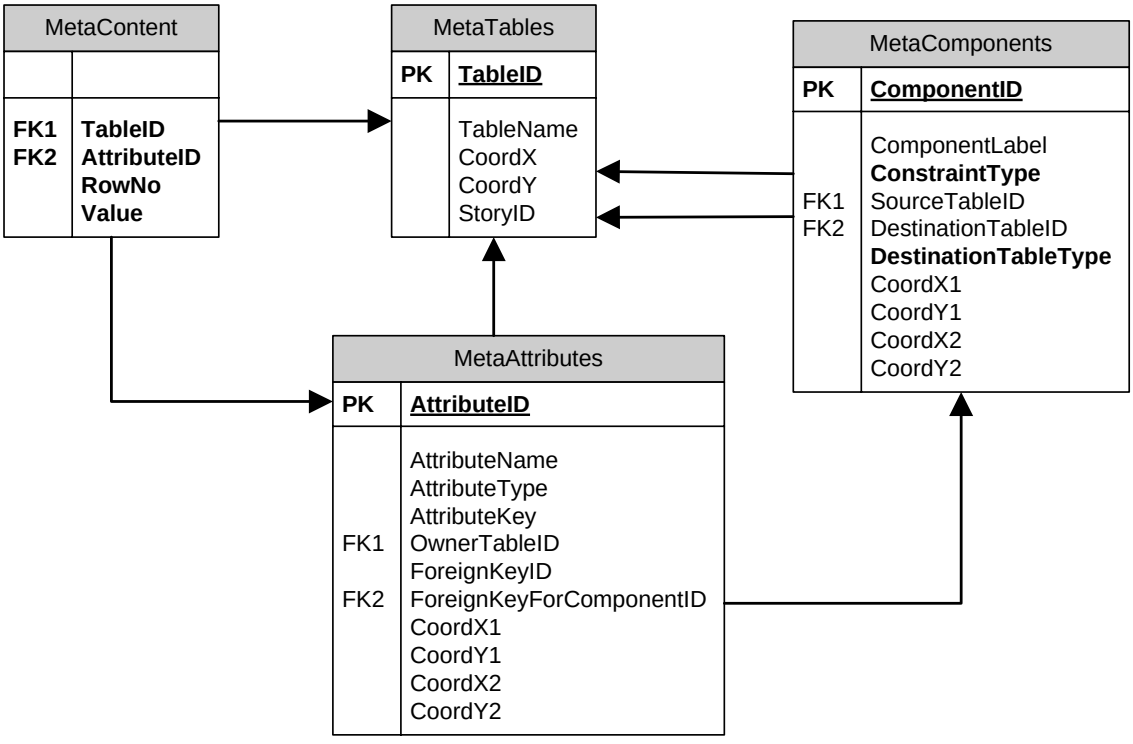

Figure 3.2: The structure of the part of the meta database

Table 3.1 explains briefly the tables shown in Figure 3.2 which comprise the meta database for the application entitties. Only these four tables will be used for the mapping, and they store all the application entities. The attributes and their meaning are listed in the table 3.2 as well as the cardinality constraints in case of a foreign key. The extended description of the tables will be given below the two tables.

Table	Explanation
MetaTables	Contains all tables of the application database structure
MetaAttributes	Contains the attributes of each table
MetaComponents	Contains the relations between the tables
MetaContent	Contains the content of each table

Table 3.1: Tables of the Meta database [IST05]

Table	Attribute	Explanation	Cardinality constraint
MetaTables	TableID	The ID of a table.	
MetaTables	TableName	The name of a table.	
MetaTables	CoordX	The X coordinate of the table.	
MetaTables	CoordY	The Y coordinate of the table.	
MetaTables	StoryID	The ID of the story.	(0,N)
MetaComponents	ComponentID	The ID of the relation.	
MetaComponents	ComponentLabel	The label of the relation.	
MetaComponents	ConstraintType	The constraint type.	
MetaComponents	DestinationTableType	The type of the destination table.	
MetaComponents	CoordX1	The X1 coordinate of the relation.	
MetaComponents	CoordY1	The Y1 coordinate of the relation.	
MetaComponents	CoordX2	The X2 coordinate of the relation.	
MetaComponents	CoordY2	The Y2 coordinate of the relation.	
MetaComponents	SourceTableID	The ID of the source table.	(0,N)
MetaComponents	DestinationTableID	The ID of the destination table.	(0,N)
MetaAttributes	AttributeID	The ID of the attribute.	
MetaAttributes	AttributeName	The name of the attribute.	
MetaAttributes	AttributeType	The type of the attribute.	
MetaAttributes	AttributeKey	Indicates whether the attribute is a primary key.	
MetaAttributes	CoordX1	The X1 coordinate of	

		the attribute.	
MetaAttributes	CoordY1	The Y1 coordinate of the attribute.	
MetaAttributes	CoordX2	The X2 coordinate of the attribute.	
MetaAttributes	CoordY2	The Y2 coordinate of the attribute.	
MetaAttributes	ForeignKeyID	The ID of the referred attribute, if the current attribute is a foreign key.	(0,N)
MetaAttributes	ForeignKeyFor ComponentID	The ID of the relation which created the foreign key.	(0,N)
MetaAttributes	OwnerTableID	The ID of the table which owns the attribute.	(0,N)
MetaContent	TableID	The ID of the table.	(0,N)
MetaContent	AttributeID	The ID of the attribute.	(0,N)
MetaContent	RowNo	The row index of the value.	
MetaContent	Value	The value.	

Table 3.2: Attributes of the tables and their meaning [IST05]

The *MetaTables* table contains all the tables of the application database. Since all the application elements can be created, managed and represented in the graphical Storyboard editor, each table has the attributes *CoordX* and *CoordY* for its rectangular representation. The tables are associated with a particular story.

The *MetaAttributes* table contains all attributes of each table defined in the *MetaTables* table. The attributes are identified by ID and the name. The contained value type is stored in the *AttributeType* column. The supported data types in SiteLang are integer, string, float, Boolean, image, audio and video. The attributes are represented in the editor as simple lines (see Figure 3.3), therefore this table must contain the coordinates of the starting and the ending points. The *AttributeKey* indicates whether an attribute is a primary key and its value is assigned to "1", if the attribute is a primary key, otherwise it is equal to "0". To represent the foreign keys, the *MetaAttributes* table needs two attributes the *ForeignKey* and the *ForeignKeyForComponentID*. "The *ForeignKey* holds the ID of the primary key to which the foreign key refers. It will be noticed, that the *MetaAttributes* table uses a recursive relation. The *ForeignKeyForComponentID* holds the ID of the relation for which the foreign key is created." [IST05]. It is not enough to represent the foreign key only by ForeignKeyID because between two tables may exist more than one relation. So each relation creates a different foreign key column.

The *MetaComponents* table contains all relationships between application tables. Each relation owns an ID and a label. They are represented as arrows in the editor so their coordinates are also included in this table. As a relation is defined between two tables, the attribute *SourceTableID* denotes that the table identified by the

DestinationTableID is a component of the table identified by the *SourceTableID*. The source table owns foreign keys which point to the primary key or the foreign keys of the destination table. The cardinality of the relationship is stored by the *ConstraintType* attribute. The possible integer values of cardinalities are showed in Table 3.3. The *DestinationTableType* attribute defines the type of the relation which could be Tuple, Set, or List and contains the integer values: respectively: "0", "1"," 2".

***ConstraintType* value**	**Meaning**	**Explanation**
0	(0,1)	An entity of a foreign key may exist max. 1 time.
1	(0,N)	An entity of a foreign key may exist unlimited.
2	(1,1)	An entity of a foreign key may exist exactly 1 time.
3	(1,N)	An entity of a foreign key may exist min. 1 time.

Table 3.3: Cardinality constraints between tables

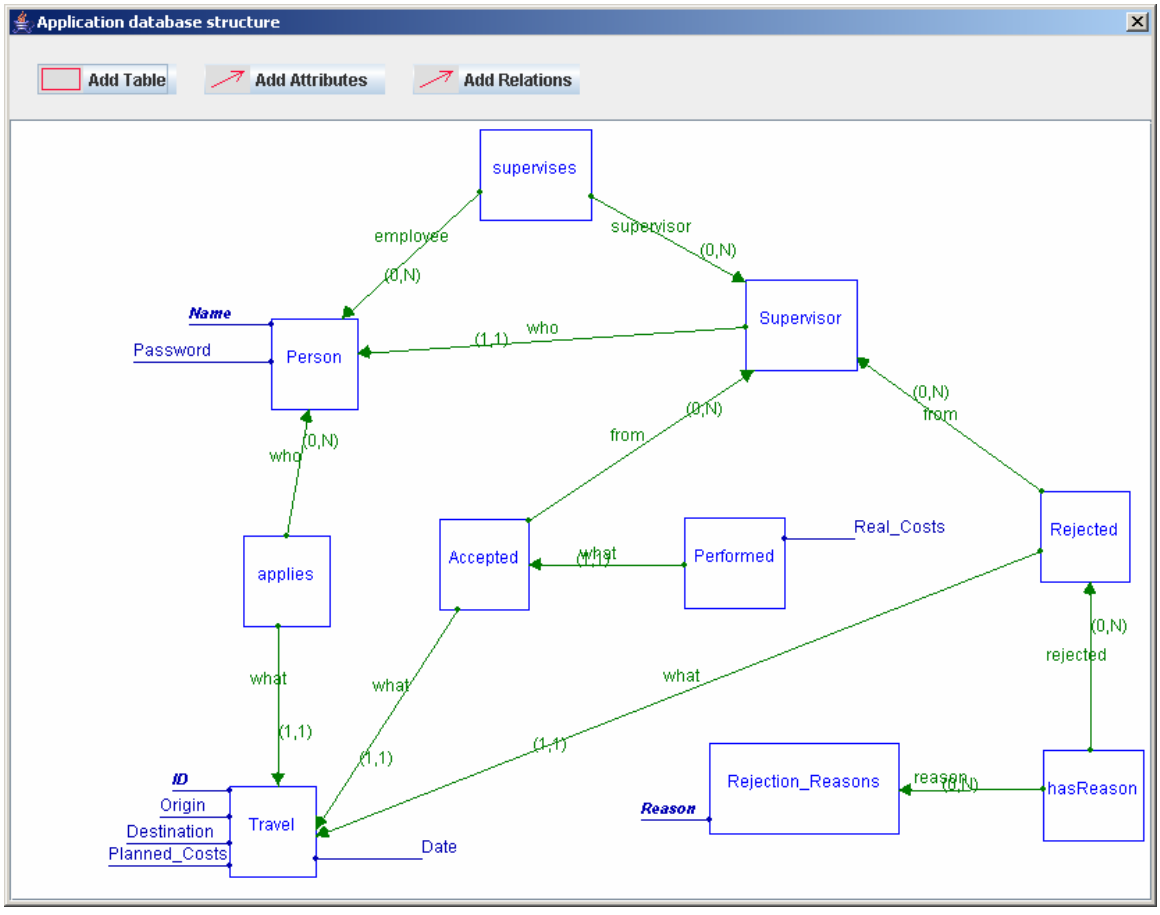

Figure 3.3: An example database structure of a business trip application (developed at the University of Kiel)

Chapter 4

Integration of Conceptual Content Management Systems into SiteLang

Both systems, Conceptual Content Management Systems and SiteLang, represent and store the application entities in different formats. The previous two sections introduced both data models, one of which is based on object-oriented languages, whereas the other stores the application entities in a relational database. In order to extend CCM systems in such a way that generator provides an output which corresponds to the format generated by the storyboarding specification. To achieve such a format, this project implements the mapping of the CCMS data model into the SiteLang entities stored in the relational database.

This section will briefly introduce the whole process to achieve the above described goal which is shown in Figure 4.1. The documents which are manually written are depicted by the orange colour symbols. The first part of this figure shows two generators integrated into the CCMS framework which perform the data model mapping and their input and output information. Other part of the figure depicts the way from the generated document to the WIS application. The process description will begin from the latter part because it is important to know what is needed to be generated and how the generated document is useful.

The Storyboard editor introduced in Chapter 3 is used to create the full SiteLang specification which is stored in the database as well as the whole application database structure. It is also used to format the specification and export it either into a single XML file or into the actual SiteLang code. Since the SiteLang code is based on ASM semantics, it would be quite complicated in this project to generate it without a prior knowledge about it. Therefore we confine us to use only an XML specification which could be translated via an XSLT [W3C01] engine into a textual SiteLang code. The XSL file for the translation of the XML file as well as the XML Schema for the XML file structure is available as a package together with the Storyboard editor. The XML file will not contain the complete specification of the web application but only the part which defines the application entities. For this purpose, two generators are developed.

The first generator, called *SiteLangXMLGenerator* and shown in the first part of Figure 4.1, generates the above mentioned SiteLang XML file. It generates an XML

document by taking a user defined asset domain model as input and applying the mapping rules which will be introduced in Chapter 4. The XML document conforms to the XML Schema (see [W3C02] for details) that has been created during the development of the Storyboard editor and can be found in the Diploma Thesis [Str05].

In order to see the application entities in the Storyboard editor, it is insufficient to generate an XML document because the editor does not support the functionality to import an XML document. Therefore, a second generator, called *SiteLangSQLGenerator*, is needed. Similar to the first generator, this one takes a user defined asset domain model as input and generates SQL statements. The same mapping rules and the SiteLang XML Schema are taken into consideration by the generator. The SQL statements could be executed directly from the generator or exported to a text file and later executed in the database management system. An implementation of both generators will be described in detail in Chapter 5.

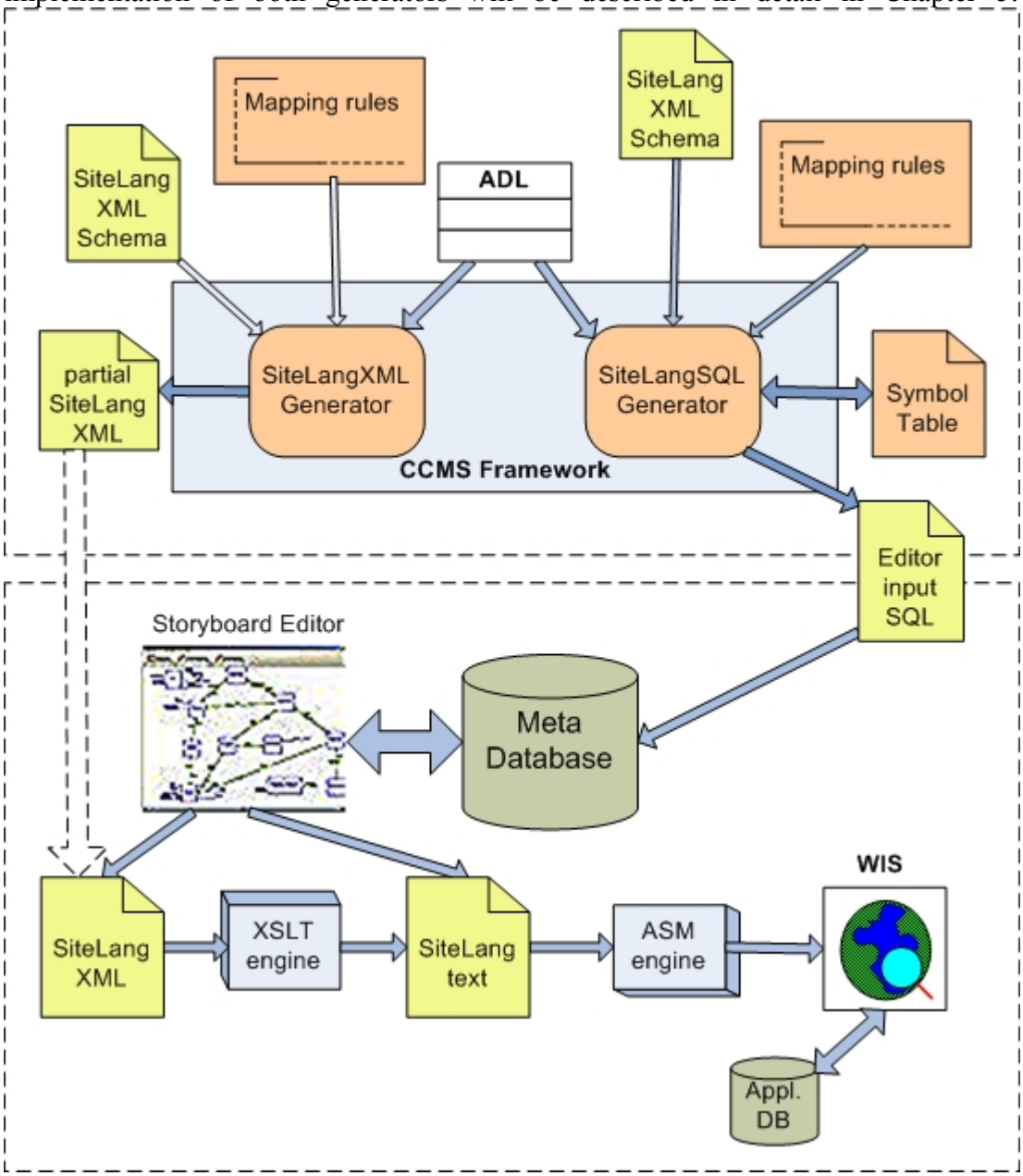

Figure 4.1: Mapping CCMS model into the SiteLang specification

4.1 Mapping CCMS data model to relational database schema

The basic goal is to achieve that the generator could automatically identify as many mapping rules as possible in order to convert between Conceptual Content Management Systems domain model and SiteLang data model. Since there is no direct mapping between asset classes based on the object-oriented model and the SiteLang entities stored in the relational database, this chapter will provide the mapping rules between these two paradigms which implementation is described in the next chapter.

As described above, the asset classes are based on object-oriented languages while the SiteLang entities are tuples in the relational database. The asset classes do not have the same properties and behaviour as objects but anyway the mapping between these two modelling approaches will be done using the object-relational mapping techniques.

While asset classes are related using direct references, the tables in relational databases are related via primary and foreign keys. Furthermore, the relational databases do not support an inheritance of data and behaviour as the asset classes do. The constraints in the asset classes set the restrictions on the values of instances; however the SiteLang does not specify any constraints on the data direct in the application database. The only possibility, the SiteLang language supports, is the media objects where the SQL queries filter the data. First, the mapping of the “body” of an asset class definition will be described, then the content part, its characteristics, the inheritance mapping strategies and finally relationships.

Figure 4.2 depicts an example of asset class definitions which are combined in a model and the tables which could be generated from the defined asset classes using the defined mapping rules. The parts of this example will be taken to demonstrate each mapping case throughout this chapter.

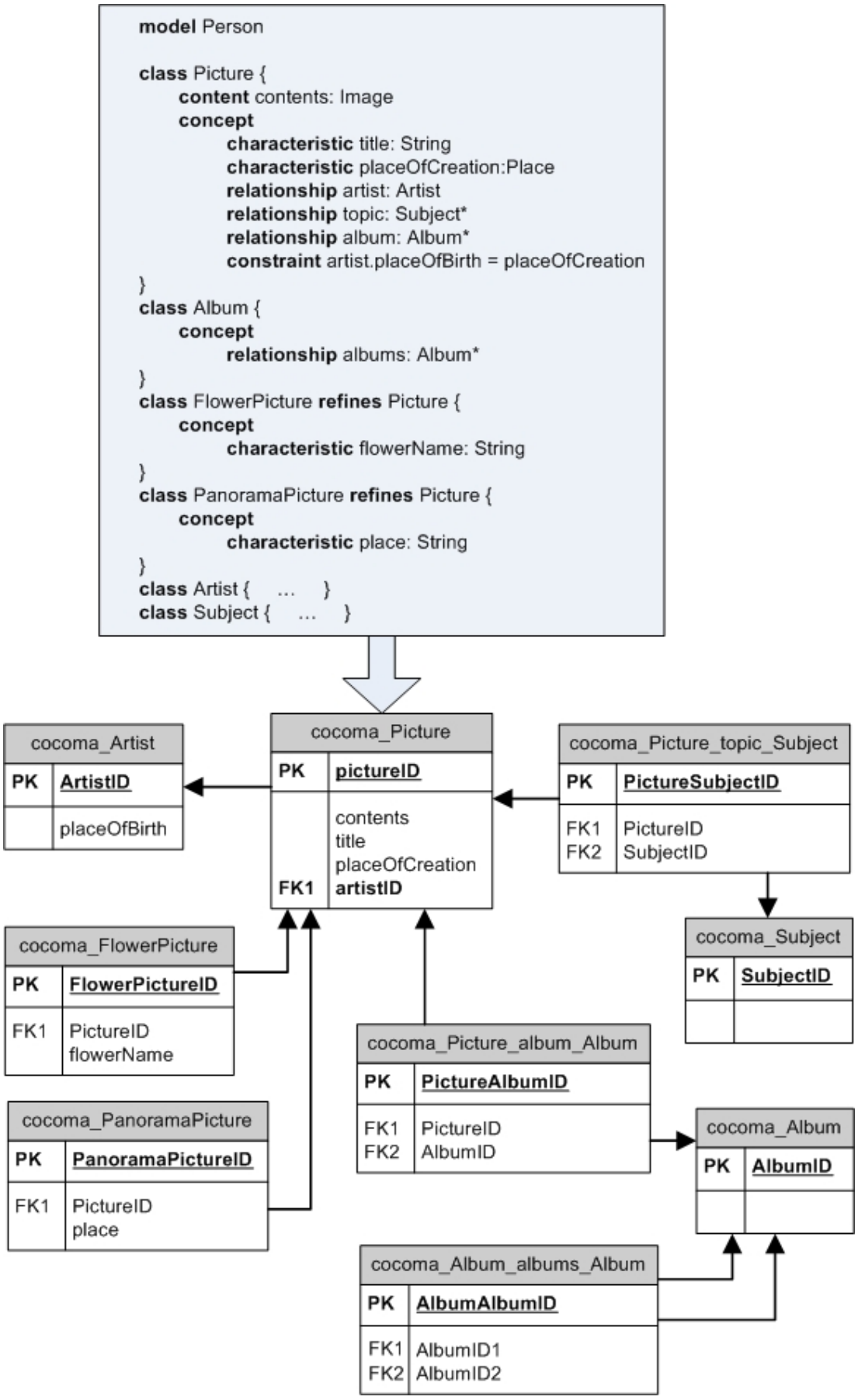

Figure 4.2: Mapping asset classes to the tables

4.2 Mapping asset class member

The mapping of asset classes begins with the asset class declaration. Each asset class in the model is represented in the SiteLang database by one or more tables. Different mapping solutions for the asset classes are described in the next section about inheritance and relationships mapping.

The mapped class is identified by the table name according to the pattern *cocoma_<AssetCalssName>*, e.g. the asset class *Picture* is mapped to the table *cocoma_Picture* (see Figure 4.2). The prefix *cocoma* is used in order to distinguish the mapped tables between already existing tables in the SiteLang database.

An identification column is automatically designated to each table which is also a primary key. This key will be later used as a foreign key reference in relationship mapping. The column is named *<AssetClassName>ID* and its value is of type *int*, e.g. the identification column of the table *cocoma_Picture* is *PictureID*. The stereotypes "PK" and "FK" in front of the column name (see Figure 4.2) indicate the primary and foreign keys respectively.

4.3 Mapping the content

The mapping of the asset class **content** part is rather straightforward. It is mapped to a column of the table and the name of the content handle is also a column name, e.g. the content *contents* of the asset class *Picture* is mapped to the column *contents* in the appropriate table. Since the *content* part explains a multimedia view of an entity, it is defined by some class object, e.g. *Image*. This content type has to be mapped to the value type of the table column which must be supported by SiteLang.

SiteLang language does not support binary type information. An ASM engine basically works with the data of type *string*. Since the SiteLang application entities are stored in the relational database it would be advantageous to represent multimedia objects (images, documents, audio files and so on) by a path where is the file located, instead of working with the binary form of the content. Eventually, the column corresponding to the content of an asset class definition always contains the values of type *string* which are the absolute or relative path to the media object (URL).

4.4 Mapping characteristics

Characteristics are defined in the conceptual part of an asset class under the keyword **characteristic**. They could be mapped to the relational database without difficulties similar like the content part. Each characteristic maps to single column in the table and the column is named the same as the handle of the characteristics, e.g. the characteristic *title* of type *String* from the asset class *Picture* maps to the column *title* in the asset corresponding table *cocoma_Picture* (see Figure 4.2).

The possible types for characteristics are determined by Java language. They may be all supported Java primitive data types or build-in Java classes. Each primitive data

type has a corresponding build-in container class. The conversion of the primitive types to the SiteLang data types is described in the next section. Some other common Java data types are also listed in Table 4.1.

4.5 Mapping data types

Since SiteLang language supports only a few data types and the characteristics of asset classes may have any Java data type, the conversion for these data types must be performed. The generator has to recognise and convert automatically as many tapes as possible. Java automatically converts a numeric type value to more general numeric type but in databases this should be foreseen before the column has been added to the table.

If the data types of the asset class instances and the objects of a target language mismatch, it could cause the problem firstly, by inserting them into the tables in the database, and secondly, much later when generating a web application by an ASM engine.

The task of this project is not the conversion of values of instances but the conversion of the column data types. Because the types of the characteristics of asset class are defined in Java language, it is obviously to appeal to type conversions in Java (see [GJSB05] for details). On the other hand, we are limited by the SiteLang supported data types. Type conversion of primitive numeric data is separated into two main categories according to the loss of precision: widening and narrowing conversions. For mapping numeric data types, we have only two choices in SiteLang, either *int* or *float*. Most likely conversion is a widening one where the order of types looks as following:

byte → short → char → int → long → float → double

Thus, the types *byte* and *short* and their corresponding built-in Java classes are mapped into *int* type and as it is a widening conversion these types do not loose any information. Conversion of a *long* type to *float* may result in loss of precision because the least significant bits of the value may be lost.

Although a *char* is a numeric data type but it is meaningful to map it to a *string* because its corresponding built-in Java object *String* is usually used in the defining asset class characteristics.

A narrowing conversion is applied only to a *double* type. Whereas a *double* is a floating-point numeric type, it is mapped to a *float* type. This conversion may cause a loss of precision of the least significant bits. In some cases, it would be possible to convert any type to a *string*. The summary of mapping all above mentioned data types and few others is listed in the table 4.1.

Java primitive data types	Java built-in data types	SiteLang data types
int	java.lang.Integer	int
short	java.lang.Short	
byte	java.lang.Byte	
long	java.lang.Long	
float	java.lang.Float	float
double	java.lang.Double	
char	java.lang.String java.lang.Character	string
boolean	java.lang.Boolean	boolean
	java.util.Date	string
	java.util.Time	
	java.util.Calendar	
	java.sql.Timestamp	

Table 4.1: Mapping between Java and SiteLang data types

4.6 Mapping inheritance structures

New asset classes can be defined regarding the existing asset classes using the keyword **refines** after the asset class name and by specifying the extended asset class. They inherit the definitions of all components (content objects, characteristics, relationships and constraints) that the parent class defines but the subclass may redefine them. In contrast to the asset classes, relational databases do not natively support inheritance therefore when mapping asset classes into relational databases it should be taken into consideration how to organize the inherited structures in the tables.

There are various strategies for representing an asset class inheritance in the relational database tables which could be used also in combination. The methods are based on object-relational mapping and more exhaustive description could be found in [Amb04] and [ORM31]. In this report, four mapping techniques will be introduced and one of them chosen for the implementation of the mapping asset classes to the SiteLang entities.

An example in Figure 4.3 is taken from Figure 4.2 and is be used to demonstrate different mapping patterns. For the sake of simplicity, each asset class has just one characteristic and that is enough to present the mapping. An inheritance tree is made up of the asset class *Picture* from which the asset classes *FlowerPicture* and *PanoramaPicture* inherit the characteristic *title*.

```
class Picture {
    concept
        characteristic title: String
}
class FlowerPicture refines Picture {
    concept
        characteristic flowerName: String
}
class PanoramaPicture refines Picture {
    concept
        characteristic place: String
}
```

Figure 4.3: Example asset classes used for mapping inheritance

Next subsections discuss different inheritance mapping approaches and by comparing them the most suitable one will be selected for the implementation of inheritance mapping.

4.6.1 One inheritance tree one table

Following this approach all attributes of the asset classes are mapped to a single database table. This table contains columns for all attributes of the parent class and subclasses and it is named as a parent class, e.g. the asset class *Picture* is a parent class for *FlowerPicture* and *PanoramaPicture* (see Figure 4.3) therefore all three asset classes are mapped to the table *cocoma_Picture* (see Figure 4.4). Each asset class instance stores its relevant data in one table row. The columns that are not relevant to the asset class are filled with a *null* value.

cocoma_Picture	
Column name	**Data type**
<PK> PictureID	int
type	int
title	string
flowerName	string
place	string

Figure 4.4: Mapping to one table

Two additional columns are created in the table *cocoma_Picture*: column *PictureID* and column *type*. The first one is the primary key column indicated by <*PK*> in front of the column name in Figure 4.4. Its creation is described in the section 4.2. The value of the *type* column is used to distinguish between subclasses therefore in [ORM31] this strategy is called filtered mapping. In this example, the *type* column contains the *int* values which could be equal to "1" for *FlowerPicture*s and equal to "2" for *PanoramaPicture*s.

4.6.2 Each asset subclass to different table

With this mapping strategy each asset subclass in the inheritance hierarchy is mapped to a separate table. Each mapped table includes the attributes implemented by the asset class and all inherited attributes from its parent classes. It means that any attribute in the superclass is duplicated across the tables of the subclasses.

Using this mapping approach the asset class definitions in Figure 4.3 are mapped to two tables as shown in Figure 4.5.

cocoma_Picture	
Column name	**Data type**
<PK> PictureID	int
title	string

cocoma_FlowerPicture	
Column name	**Data type**
<PK> FlowerPictureID	int
title	string
flowerName	string

cocoma_PanoramaPicture	
Column name	**Data type**
<PK> PanoramaPictureID	int
title	string
place	string

Figure 4.5: Mapping subclasses to the tables

The new tables get the names from the relevant asset classes and the additional primary key columns, e.g. the inheritance structure is mapped into three tables: *cocoma_Picture*, *cocoma_FlowerPicture,* and *cocoma_PanoramaPicture*. A parent asset class *Picture* is mapped to its own table in order to keep the instances of it. Each table corresponding to the asset subclass includes the column *title* of type *string* for the inherited characteristic *title.* In [ORM31], mapping each subclass to the different table is called a horizontal mapping.

4.6.3 Each asset class to its own table

This kind of mapping represents each asset class in the inheritance hierarchy with one table for each asset class (described as the vertical mapping strategy in [ORM31]). The attributes of the asset class map directly to the columns in the matching tables. Additionally, the primary key is assigned to each table for identification information and the tables corresponding to the asset subclasses contain the foreign key column that references the primary key of the parent table.

In the example tables of Figure 4.6, the foreign key column *PictureID*, which links to the parent table *cocoma_Picture*, is inserted in every subclass table *cocoma_PanoramaPicture* and *cocoma_FlowerPicture*. The foreign key in the figure is indicated by <*FK*> in front of the column name. The data of the asset class *PanoramaPicture,* as well as the asset class *FlowerPicture,* are stored in two tables therefore in order to retrieve an instance of the subclass requires a join of the *cocoma_Picture* and *cocoma_PanoramaPicture* tables.

cocoma_Picture	
Column name	**Data type**
<PK> PictureID	int
title	string

cocoma_PanoramaPicture	
Column name	**Data type**
<PK> PanoramaPictureID	int
<FK> PictureID	int
place	string

cocoma_FlowerPicture	
Column name	**Data type**
<PK> FlowerPictureID	int
<FK> PictureID	int
flowerName	string

Figure 4.6: Mapping each class to its own table

4.6.4 Mapping to the generic table structure

The last inheritance mapping strategy which is introduced in this paper is "a generic, sometimes called meta-data driven approach. This approach isn't specific to inheritance structures; it supports all forms of mapping" [Amb04]. The tables in the figure 4.7 represent the mapping according to this approach.

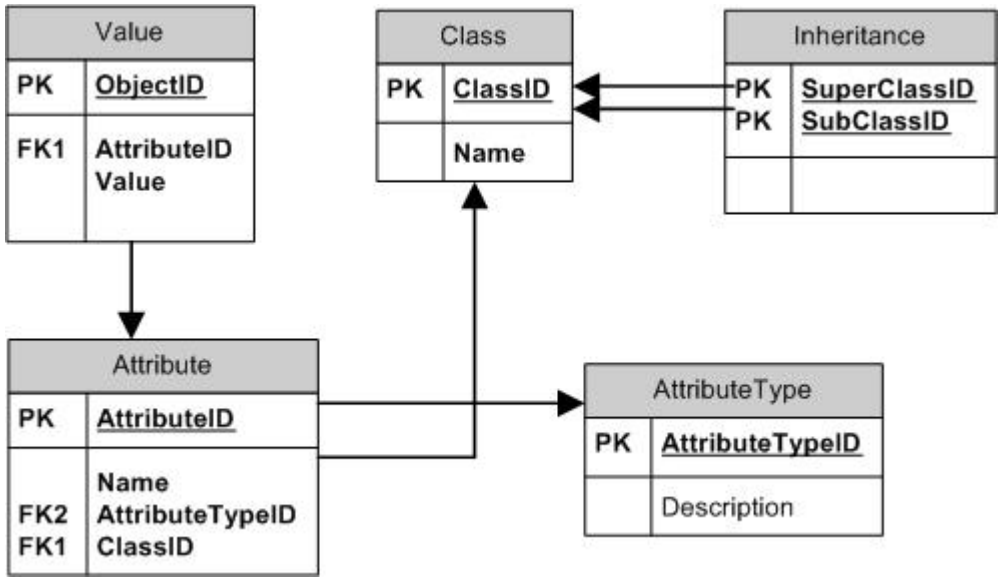

Figure 4.7: A generic data schema for storing objects [Amb04]

This kind of mapping is described here because of its resemblance to the SiteLang Meta database schema which was described in the section 3.2. The correspondence between tables would be as listed below:

Value	MetaContent
Class	MetaTables
Attribute	MetaAttributes
Inheritance	MetaComponents

There is a small inadequacy between the tables *Inheritance* and *MetaComponents* because the latter one stores the relationships between the tables which could be better suitable for the relationships (associations) between asset classes mapping.

The table *Class* contains all the asset classes defined in the domain model therefore the mapping has to begin with this table. The table *AttributeType*, in the case of this project, would contain all the SiteLang supported data types. The attributes of the asset class are stored in the table *Attribute* which column *AttributeTypeID* refers to the relevant column in the table *AttributeType*. The table also is linked to the table *Class* via *ClassID*. An inheritance between asset classes is defined in the table *Inheritance* where each leaf in the inheritance tree is represented by one row, e.g. *SuperClassID* would refer to the row in the table *Class* representing *Picture* and *SubClassID* to the asset class *PanoramaPicture* (see Figure 4.3).

4.6.5 Comparing the mapping strategies

The table 4.2 lists advantages and disadvantages of each mapping approach described above.

None of the mapping techniques is perfect, each of them has strengths and weaknesses. However, the each class to its own table meets the most of the requirements for mapping asset classes to the relational database. First of all, this approach conforms to the object-oriented concept best. It supports very well the polymorphism and facilitates modification and adding the tables conforming to the superclasses as well as subclasses. Consequently, this mapping strategy will be used for the inheritance mapping implementation. This choice will be also advantageous later when mapping relationships between assets.

The fourth mapping method to the generic table structure would be difficult to implement due to the fact that the SiteLang Meta database is organized according to this principle.

Strategy	Advantages	Disadvantages
One inheritance tree one table	+ Single table + No joins in retrieving data + Easy to add new asset classes to the hierarchy + Useful for simple hierarchies	- Not all fields are relevant – confusing - Space wasted in the database - Frequent locks on the table – may cause poor performance - Need to think about naming of fields because they can repeat among subclasses
Each asset subclass to different table	+ Instance retrieved by only one query + No irrelevant fields in the table + Good performance – spread access load among tables	- Difficult to implement the relationships to the parent asset class - In case of changes in the superclass, have to be changed every table
Each asset class to its own table	+ Conforms best to the object-oriented concept + No irrelevant fields in the table + Only one table needed to update when asset class is changed + Easy to add new asset subclasses + Flexible relationship mapping	- Many tables in the database - The tables corresponding to the superclasses have to be accessed often – may cause a bottleneck. - A join to access the tables corresponding to the subclasses.
Generic table structure	+ Could be extended to support relationship mapping + Easy to update, flexible	- Difficult reporting against the data because an instance of one asset class is stores in several rows. - Difficult to implement

Table 4.2: Comparing the mapping strategies

4.7 Mapping relationships

In asset modelling, the entities may be related to other entities. Not only asset classes must be mapped to the relational database but the relationships must be mapped too and the multiplicities must be kept the same as defined in the domain model. There are four kinds of relationships that an asset class could be involved with: inheritance,

one-to-one, many-to-many, and recursive relationships. The solutions how to mimic an inheritance to the relational databases are described in the section above. The three last types of relationships are based on the cardinalities between the entities and are defined in the asset class under the keyword **relationship**. Their mapping to the relational model will be discussed in the next sections.

Figure 4.8 depicts an example asset classes which will be used to represent relationships between assets in the database.

```
...
class Album {
    concept
        relationship albums: Album*
}
class Picture {
    concept
        ...
        relationship artist: Artist
        relationship topic: Subject*
        relationshiop album: Album*
}
class Artist {    ...    }
class Subject {    ...    }
```

Figure 4.8: Example asset classes for the relationship mapping

4.7.1 Mapping one-to-one relationships

As described above, the relationships between asset classes are defined in the **concept** part of asset class definition using the keyword **relationship**. The simplest type of relationships is a one-to-one relationship. Consider one-to-one class relationship *artist* of the asset class *Picture* to the asset class *Artist* as depicted in Figure 4.8, it means that each *Picture* is drawn by at most one *Artist*.

In the relational databases one-to-one relationship is implemented by means of a foreign key. Since in the section 4.5.5 was decided to implement an inheritance relationship using the strategy "each asset class to its own table", as a result each asset class is represented by its own table, as in Figure 4.9.

cocoma_Picture	
Column name	**Data type**
<PK> PictureID	int
<FK> ArtistID	int

cocoma_Artist	
Column name	**Data type**
<PK> ArtistID	int

Figure 4.9: Mapping one-to-one relationships

The foreign key column holds the value of the primary key of the row being referenced. The column *ArtistID* in the table *cocoma_Picture* is a foreign key column which holds the value of the primary key column *ArtistID* in the *cocoma_Artist* table. The relationship handle *artist* is not shown in the result of this mapping example but it is not lost since the reverse mapping should be also possible when required. That will be described in the implementation chapter 5.

Since the above described relationship is uni-directional (the Picture knows about its Artist but not the other way around) the foreign key column is embedded only in one table. If it would be a bi-directional relationship then the table *cocoma_Artist* have had a foreign key column *PictureID*.

4.7.2 Mapping many-to-many relationships

We cannot present a many-to-many relationship in the relational databases using a foreign key strategy described above because it does not have a single-valued end of the relationship to hold a foreign key. The asset class may reference to more than one asset class. The asterisk "*" after the relationship type identifies a referenced set of asset classes as depicted in Figure 4.8. The relationship *subject* indicates that each *Picture* could be classified by many *Subject*s.

In order to map a many-to-many relationship between asset classes into relational database we need three tables: two of them are derived from the mapping asset class definitions and the third is introduced as an associative table [Amb04]. The associative table maintains the relationships between two tables in the relational database. The name of the table is composed of the two asset classes involved in the relationship and the name of the relationship it implements as follows:

cocoma_<SourceAssetClass>_<RelashionshipName>_<TargetAssetClass>.

The asset classes involved in the relationship don't embed any new columns in their corresponding tables. However, the associative table contains two foreign key columns that point to each of the primary key IDs of the tables in the relationship. In addition, it has also a primary key column which is a combination of the names of the asset classes that the associative table joins and the suffix "ID" as shown in Figure 4.10. Each table according to the SiteLang language must have a primary key column.

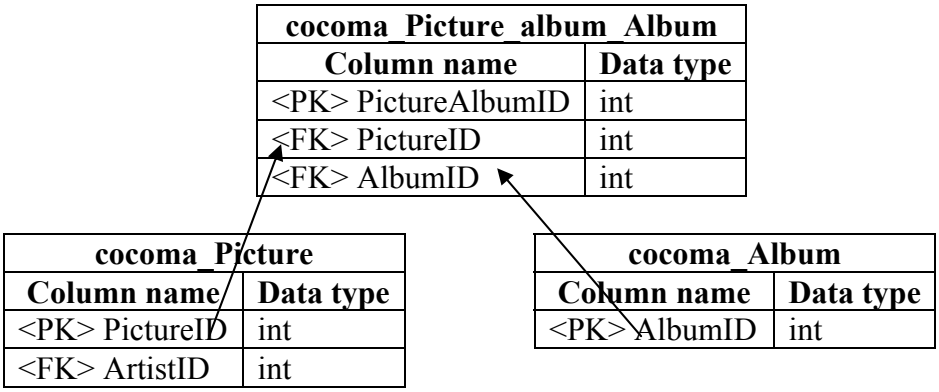

Figure 4.10: Mapping using an associative table

It is possible that both foreign key columns in an associative table point to a primary key on the same table. This case is discussed in the next section.

4.7.3 Mapping recursive relationships

In the previous section, the two types of relationships were introduced. The mapping strategies were applied for the relationships between two different asset classes. However, the asset class *Album* in the example in Figure 4.8 defines one more kind of relationship. The relationship *albums* relates the asset class *Album* to itself and is called recursive relationship [Amb04]. However, because the asterisk "*" follows the relationship type, it is a many-to-many recursive relationship. Figure 4.11 depicts its implementation into the relational database.

cocoma_Album	
Column name	**Data type**
<PK> AlbumID	int

cocoma_Album_albums_Album	
Column name	**Data type**
<PK> AlbumAlbumID	int
<FK> AlbumID1	int
<FK> AlbumID2	int

Figure 4.10: Mapping recursive relationship into the tables

The asset class defining a recursive relationship is mapped into two tables similar like in the many-to-many mapping described above. The only difference is that an associative table contains two foreign key columns which refer to the same table in the relational database. Since the table cannot have two columns with the same name, the foreign key columns will get the numerical suffix to the name as shown in Figure 4.10.

When the asset class is related to itself it does not mean that its instance is related to itself. However, an instance of this asset class is related to another instance of the asset class. An instance of *Album* can be a part of other *Album* instance.

Chapter 5

Mapping Implementation

The previous chapter introduced the mapping rules used to map the entities of Conceptual Content Management Systems - asset class definitions - to the application entities described in the web specification language SiteLang. This chapter describes the mapping implementation. The first generator maps asset classes to the SiteLang entities in XML format whereas the second maps to the SiteLang application database.

5. 1 SiteLang XML specification

This section briefly explains the entire structure of the XML file that contains the specification of the web application. This file is created by the Storyboard editor. The file generated by the CCMS generator *SiteLangXMLGenerator* introduced in the previous chapter and described in detail in Section 5.3 and its subsections contains only part of the full XML specification. This project deals only with the application entities of the WIS generated using the SiteLang specification; therefore, the part of the XML file to be generated is related to the application database definitions.

Before generating an XML file with the generator it is necessary to be familiar with its structure. Chapter 3 describes quite clearly all storyboarding elements. This includes the declarations of the events defined by the name, local and global variables and their initialization. Each scene description in the XML file includes all properties, local variables, dialog steps with their events, pre- and post-conditions and involved dialog objects, as well as media objects of the scene. Following these elements is the representation of the application database and its initial content. This part of the XML file is discussed in the next section in detail.

The application database structure in the XML file is enclosed within the `<MetaDatabase>` tag. A part of the table *cocoma_Picture* from Figure 4.2 is described in the XML format in Figure 5.1. Since the entities are stored in the relational database as tables, the database definition tag contains the declarations of the tables within the `<Table>` tags as child elements. The name of the table is declared in the `<TableName>` tag, e.g. *cocoma_Picture*. The columns of the table correspond to the elements `<Attribute>`, which define the name of the column by the `<AttributeName>` tag and the type of the column by the `<AttributeType>` tag.

Figure 5.1 describes two columns: *PictureID*, which is the primary key of the table; and the column *artist,* which is the foreign key, as explained below.

The `<PrimaryKeys>` tag contains all primary keys of the table. Each column that is a primary key of the table is declared within the `<PrimaryKey>` tag, where the name of the primary key column is an element text, e.g. *PictureID*.

The `<ForeignKeys>` tag contains all foreign keys of the table where each foreign key is enclosed within the `<ForeingKey>` tag. It encloses the `<ReferenceTable>` tag, which defines the table the foreign key refers to, while the `<ReferenceType>` tag contains the type of the referenced table. The `<Attributes>` tag declares the foreign key of the table and the `<ReferenceAttributes>` tag contains the primary key of the referenced table defined in the `<ReferenceTable>` tag. The column *artist* is a foreign key of the table *cocoma_Picture,* and it contains the value of the primary key *ArtistID* from the table *cocoma_Artist*.

```
<MetaDatabase>
   <Table>
          <TableName>cocoma_Picture</TableName>
          <Attribute>
                 <AttributeName>PictureID</AttributeName>
                 <AttributeType>int</AttributeType>
          </Attribute>
             ...
          <Attribute>
                 <AttributeName>artist</AttributeName>
                 <AttributeType>int</AttributeType>
          </Attribute>
          <PrimaryKeys>
                 <PrimaryKey>PictureID</PrimaryKey>
          </PrimaryKeys>
          <ForeignKeys>
                 <ForeignKey>
                        <ReferenceTable>cocoma_Artist</ReferenceTable>
                        <ReferenceType>Tuple</ReferenceType>
                        <Attributes>
                               <AttributeName>artist</AttributeName>
                        </Attributes>
                        <ReferenceAttributes>
                               <AttributeName>ArtistID</AttributeName>
                        </ReferenceAttributes>
                 </ForeignKey>
          </ForeignKeys>
   </Table>
   <Table>
      ...
   </Table>
</MetaDatabase>
```

Figure 5.1: A fragment of the application database specification in XML format

The XML file may contain more information than the Storyboarding supports, e.g. HERM modelling (according to how the application database schema is modelled), which provides the possibility to define the destination tables of the relations by tuple, set and list types, which Storyboard editor supports but the SiteLang language and ASM machine do not.

5.2 SiteLang XML Schema

An XML Schema describes the structure of the XML file. The Storyboard editor also includes with its source code the XML Schema for the exported storyboarding specification in XML format. The XML Schema meets the requirements of the storyboarding language. It defines all the constructs that encapsulate the story.

In order to generate the WIS specification in XML format, the supported XML Schema is very useful, even for the part related to the application database. Since the CCMS generator is written in Java language, it is necessary to have control over the structure and content of the generated XML document.

JAXB [JAXB20] is a Java technology that provides the possibility to bind the XML Schema components to Java classes and interfaces that reflect the rules defined in the schema. The Java classes generated with the JAXB represent the different top-level elements and top-level complex types in an XML Schema. An XML document that conforms to the XML Schema may be constructed from the Java classes. There is no need to worry about the XML syntax.

Figure 5.2 depicts an XML Schema for defining the application database SiteLang specification in XML format. Each class in the UML class diagram represents a complex type of XML Schema, which are also the Java classes generated by JAXB.

The fragment of the XML Schema that describes the meta database depicted in Figure 5.2 is generated by JAXB into nine Java classes corresponding to each top-level element. A factory class *ObjectFactory* consisting of methods to create new instances of the Java representation for XML content is also generated. To get a deeper look into the generated classes, the class *Table* is a good example because it is generated by the XML element `Table`, which is a complex element and encapsulates a sequence of four elements, as depicted in the diagram. The child elements are declared in the class as follows:

```
protected String tableName;
protected List<Attribute> attribute;
protected PrimaryKeys primaryKeys;
protected ForeignKeys foreignKeys;
```

The first element `TableName` contains only the text node; therefore, it is converted to a Java variable type *String*. The class *Table* includes the setter and getter methods to set and get the name of the table. Table 5.1 lists all methods of the class *Table*. The complex element `Attribute` can occur in the table as many times as the columns it contains. Since the `Attribute` is mapped to the Java class *Attribute,* the *getAttribute()* method returns a *List* by reference where the new `Attribute` element can be added to the table by the method of the class *java.util.List*. The element `Table` must contain only one element `PrimaryKeys`, which is also a complex element that means that the variable *primaryKeys* declared in the class *Table* is of Java class type *PrimaryKeys*. The getter method returns an instance of the class.

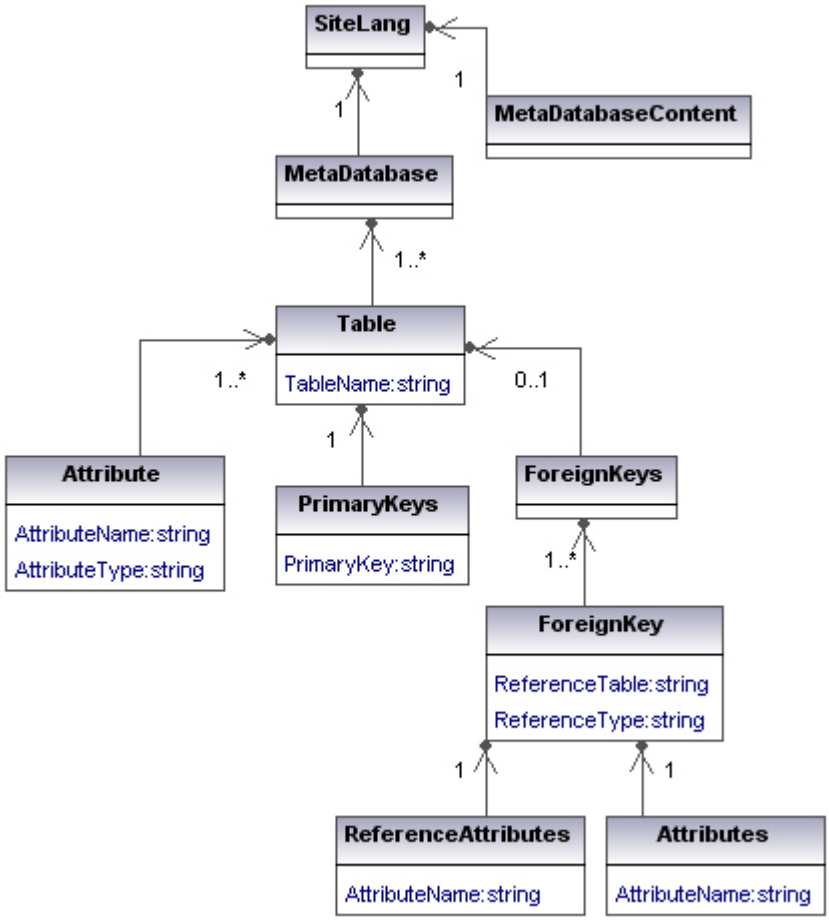

Figure 5.2: The partial diagram of the SiteLang XML Schema

Setter method	Getter method

`public void setTableName(String value) {`
`  this.tableName = value;`
`}`
	`public String getTableName() {`
	`  return tableName;`
	`}`

	`public List<Attribute> getAttribute() {`
	`  if (attribute == null) {`
	`  attribute = new ArrayList<Attribute>();`
	`  }`
	`  return this.attribute;`
	`}`

`public void setPrimaryKeys(PrimaryKeys value) {`
`  this.primaryKeys = value;`
`}`
	`public PrimaryKeys getPrimaryKeys() {`
	`  return primaryKeys;`
	`}`

`public void setForeignKeys(ForeignKeys value) {`
`  this.foreignKeys = value;`
`}`
	`public ForeignKeys getForeignKeys() {`
	`  return foreignKeys;`
	`}`

Table 5.1: An example of the setter and getter methods from the class Table

Section 5.3 discusses how the Java classes generated from the SiteLang XML Schema are integrated into the *SiteLangXMLGenerator* to marshal the generated content from asset class definitions into the SiteLang XML document.

5.3 Mapping asset classes to SiteLang entities in XML format

As described in chapter 3, to map the asset class definitions to the SiteLang entities two CCMS generators are developed. The first of them will be described in this section. The generator *SiteLangXMLGenerator* maps the asset classes to the SiteLang entities described in the XML format. Section 5.1 introduced the structure of the XML document, which is defined by the XML Schema. It is described in the previous section, as well as the binding of the element declarations to the Java classes.

5.3.1 Compilation

The mapping generator *SiteLangXMLGenerator* is an extension of the Conceptual Content Management Systems framework. It enables a uni-directional CCMS domain model mapping to the entities of the web application specified by the Storyboarding language SiteLang. The generators composing the framework are dependent on each other and the compiler controls the order of their execution by the requested Symbol tables. The generator is started by invoking the actual performance method

```
public SymbolTable generate(IntermediateModel im, SymbolTable[] symTabs,
                            Map<String, ? extends Object> params) method.
```

of which the input parameters are the Intermediate model introduced in chapter 2, symbol table created by the central API generator and the parameters defined in the configuration file. This method returns the symbol table it creates (this feature is common for every generator) and the mapping results in the XML file, which is a side effect of the generator, but the intention of what the generator is created for. In this case, the produced symbol table is empty, but more about symbol tables is discussed in Section 5.4.1, together with other generators that create and use a symbol table.

5.3.2 Marshalling to the XML file

The generator output XML document is created using the Java classes generated with JAXB from the XML Schema definitions, as described in section 5.2. The starting point of the marshalling of the XML document is the instantiating of the *JAXBContext* class, which takes care of the binding relationship between XML element names to the Java class. The package name containing the JAXB mapped classes as well as the *SiteLangXMLGenerator* class is given as a parameter. Once the domain object is initialized, the Marshaller object is created. The JAXBElement corresponds to the root complex element Sitelang of the generated XML document and wraps the content of this element. The generated XML content is formatted to

the human readable text and marshalled to the given file. The described marshalling code is as follows:

```
JAXBContext jc =
JAXBContext.newInstance("de.tuhh.sts.cocoma.compiler.generators.sitelang");
SitelangType slt = new SitelangType();
 … create the domain object …
Marshaller m = jc.createMarshaller();
JAXBElement<SitelangType> SLelement = (new
                                ObjectFactory()).createSitelang(slt);
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(SLelement, new FileOutputStream("SiteLangXMLMetaDatabase.xml"));
```

Figure 5.3: Marshalling sample code

5.3.3 Adding the content to the XML file

The framework parser translates the asset model to an internal representation in the form of an Intermediate model *IntermediateModel,* which contains:

- *AssetClass* (an asset class defined by its name, superclass and members)
- *Content* (a content attribute with its name and type)
- *Characteristic* (a characteristic attribute with its name and type)
- *Relationship* (a relationship attribute with its name, referred type and the cardinality)
- *Constraint* (a constraint with an associated rule. Currently, the constraints are not supported by the parser.)

As mentioned above, we get the asset class definitions from the Intermediate model. It returns an array of asset classes *AssetClass* by invoking its method *getClasses()*. By iterating through the array, we get all asset classes from the asset model that should be mapped to the relational database tables applying the mapping rules described in the previous chapter. This section describes the mapping to an XML document, but the mapping rules are the same as those defined for mapping to the tables because the structure of an XML document is based on the table definitions. The complete SiteLang XML file describing the application database generated by the *SiteLangXMLGenerator* can be found in Appendix A.

Asset class member

Each asset class in the array is mapped to the separate element `Table` in the XML file. The table is created immediately after getting the new asset class (see Figure 5.4).

```
ltbs.add(addNewTable(tbPrefix + a.getName(), a.getName() + "ID", "int"));

public static Table addNewTable(String tbName, String attrName,
                                String attrType) {
       Table tb = new Table();
       tb.setTableName(tbName);
       List<Attribute> attrs = tb.getAttribute();
       attrs.add(addNewAttribute(attrName, attrType));
       PrimaryKeys prKeys = new PrimaryKeys();
       prKeys.setPrimaryKey(attrName);
       tb.setPrimaryKeys(prKeys);
       return tb;
}
```

Figure 5.4: Method to add new table

where *tbPrefix* is the table prefix "*cocoma_*", and *a* is the current *AssetClass*. The new table is created by instantiating the class *Table*, which is generated by JAXB from the XML Schema definitions. It contains the methods to set and get all possible attributes. In the above listed code, the method *addNewTable(...)* creates the element `Table`, sets the table name according to the pattern *cocoma_<AssetClassName>*, adds the child element `Attribute` corresponding to the column *<AssetClassName>ID* (the method to add the `Attribute` is shown in Figure 5.5), here the type of the column by default is *int* and sets it as a primary key. The newly created *Table* is added to the list of `Table` elements *List<Table> ltbs*.

Content

The next step is mapping the content part of the asset class. An object of the *AssetClass* contains an array of the *Content* objects having the values of the name and the type of the content. Each content member is mapped to the column of the table, which means it is added to the `Table` element as an `Attribute` element. The column of the content attribute by default contains the values of the type *string* since the content type is a URL to the media. The methods to add the `Attribute` are as follows:

```
addTableAttr(neededTable, c.getName(), "string");

public static void addTableAttr(Table tb, String attrName, String attrType)
{
       List<Attribute> attrs = tb.getAttribute();
       attrs.add(addNewAttribute(attrName, attrType));
}

public static Attribute addNewAttribute(String attrName, String attrType) {
       Attribute attr = new Attribute();
       attr.setAttributeName(attrName);
       attr.setAttributeType(attrType);
       return attr;
}
```

Figure 5.5: Method to add an `Attribute` element to the `Table`

where *c* is a content object.

Characteristics

The characteristics are generated using the similar principle as the content part. The difference is only that the type of the characteristic is converted according to Table 4.1. For that, the generator includes a separate method that converts as many expected characteristic types as possible into SiteLang supported data types.

If the type of the characteristic is not found in the list then the message "The data type <*dataType*> is not recognized by the generator!" is returned. Such being the case, the generated XML file could be reviewed and either a text of the element `AttributeType` changed manually or the asset model edited so that the type would be one of those listed in Table 4.1 and the file generated once again. It is suggested that the second option be performed in order to avoid the same problem by generating the SQL statements.

Relationships

The method *getRelationships()* of the *AssetClass* returns an array of the *Relationship* objects. Each object contains the name of the relationship and the referred type *AssetType*. This type contains the referred *AssetClass* and the property indicating whether this relationship type refers to a collection of assets.

As described in Chapter 4, the relationships between asset classes are divided into three categories: one-to-one, many-to-many and recursive relationships. The previously defined mapping rules are applied and discussed in this section.

If the asset class is involved in the one-to-one relationship, the table implementing it adds an additional column for the foreign key value. The new `Attribute` with the child elements for the name value <*RelationshipName*> and the column type *int* is appended to the `Table` element. Since this column is a foreign key, it is defined in the new element `ForeignKey`. The code in Figure 5.6 shows how the new foreign key is added:

```
public static ForeignKey addNewForeignKey(Table tb, Relationship r,
                                          String refType, String nr) {

       ForeignKey frKey = new ForeignKey();
       frKey.setReferenceTable(tbPrefix + r.getReferredClass().getName());
       frKey.setReferenceType(refType);

       Attributes attrs = new Attributes();
       attrs.setAttributeName(r.getName() + nr);
       frKey.setAttributes(attrs);

       ReferenceAttributes refAttrs = new ReferenceAttributes();
       refAttrs.setAttributeName(r.getReferredClass().getName()+"ID");
       frKey.setReferenceAttributes(refAttrs);
       return frKey;
}
```

Figure 5.6: Method to add a `ForeignKey` element to the `Table`

where *refType* is a type of the table that is equal to *Tuple* (explained in Chapter 3). The property *nr* is used in the recursive relationship mapping and will be explained later.

The many-to-many relationship is implemented by a similar method as the one-to-one relationship. The difference is that it doesn't add an extra `Attribute` to the element `Table` but creates a new `Table` element, which is an associative table between two tables. Adding of the new Table element is described in the paragraph "Asset class member" of this section and the name of the table is combined, as defined in Chapter 4.

```
public static Table addNewUnionTable(String relName, Relationship r) {
	Table tb = new Table();
	tb.setTableName(tbPrefix + relName + "_" + r.getName() + "_"
				+ r.getReferredClass().getName());
	List<Attribute> attrs = tb.getAttribute();
	String sourceTb = "", targetTb = "";
	if (relName.equals(r.getReferredClass().getName())) {
		sourceTb = "1";
		targetTb = "2";
	}
	attrs.add(addNewAttribute(relName + r.getReferredClass().getName() +
"ID", "int"));
	attrs.add(addNewAttribute(relName + "ID" + sourceTb, "int"));
	attrs.add(addNewAttribute(r.getReferredClass().getName() + "ID" +
targetTb, "int"));
	PrimaryKeys prKeys = new PrimaryKeys();
	prKeys.setPrimaryKey(relName + r.getReferredClass().getName() +
"ID");
	tb.setPrimaryKeys(prKeys);
	ForeignKeys foreignKeys = new ForeignKeys();
	List<ForeignKey> frKeys = foreignKeys.getForeignKey();
	frKeys.add(addNewForeignKeyForUnion(tb, relName, "Tuple",sourceTb));
	frKeys.add(addNewForeignKeyForUnion(tb,
r.getReferredClass().getName(), "Tuple", targetTb));
	tb.setForeignKeys(foreignKeys);
	return tb;
}
```

Figure 5.7: Method to add new associative table

where *relName* is the name of the asset class holding the relationship, *sourceTB* and *targetTB* are indicators used for recursive relationships, and here they are equal to "".

The associative `Table` element encloses three `Attribute` elements, where one is a primary key and the other two are foreign key columns pointing to the primary keys of the tables participating in the relationship. The method to add a new `Attribute` *addNewAttribute(...)* is described in the paragraph "Content". The difference in adding a `ForeignKey` element from a one-to-one relationship is the name of the foreign key column. Here, it is the same as the primary key in the referenced table.

The last type of the relationships is a recursive relationship. Its mapping rules are described in Section 4.6.3. Its implementation is similar to the many-to-many relationship. The same method *addNewUnionTable(…)* listed in Figure 5.7 is used to append a new associative `Table` element. Since both foreign key columns in the associative table refer to the same primary key column, the suffixes *sourceTb* and *targetTb* are added to the foreign key column name to differentiate them. The recursive relationship is detected if the *AssetType* refers to a collection of assets and

the *AssetClass* name of the relationship type is equal to the name of the asset class implementing the relationship.

Inheritance

An asset class inheritance is implemented using the mapping strategy "Each asset class to its own table". The mapping rules of this strategy are introduced in Section 4.5.3 and its implementation is quite simple.

The *AssetClass* objects include a property that contains the value null if the asset class doesn't refine any asset class, otherwise it contains the asset superclass object *AssetClass*. If the method *getSuperClass()* returns an object then the inheritance is implemented in a similar way as the one-to-one relationship. The `Table` element adds the Attribute element, which is the foreign key column; therefore, the `ForeignKey` element is also appended. The `AttributeName` contains the text, which is the name of the asset superclass and a suffix ID. The `AttributeType` is by default set to int.

5.4 Mapping asset classes to the SiteLang database

The previous section discussed how the asset model is mapped to the SiteLang meta database in XML format and introduced the main methods to implement the mapping and their performing tasks in regard to an XML document.

This section describes the asset model mapping to the SiteLang database. The application entities are stored in the meta database, which consists of four tables. It is described in Section 3.2 in detail, and Figure 3.2 depicts the structure of the meta database. The mapping rules defined in Chapter 4 are implemented for the mapping asset class definitions to the relational database tables and the second generator *SiteLangSQLGenerator* is developed. Since the structure of the above described XML file reflects the tables in the database, the Java classes generated by JAXB from SiteLang XML Schema definitions are also integrated into this generator.

The generator creates the symbol table *SiteLangSQLGeneratorSymbolTable* and a text file with the SQL statements, which are later executed by the database. Since the task of this project is to map the data model, the SQL INSERT statements deal only with three tables of the SiteLang meta database. The table *MetaContent* contains the instances of the application entities.

5.4.1 Symbol table

The generator *SiteLangSQLGenerator* creates the symbol table, whose main purpose is to enable the communication between generators. The symbol tables contain all information about the generated tables, their columns, the relations between asset classes and tables, and some other information that will be described in the next sections. This generator includes the mapping methods from the first generator with small modifications. The first generator creates an output file in XML format and adds all generated `Table` elements to the `MetaDatabase` node. *SiteLangSQLGenerator* also creates the *Table* objects with all properties, similar to

the XML document, but adds them to the symbol table. It contains a number of *Map* objects with key-value pairs, e.g. the object *tables* associates the asset classes to the *Table* objects:

```
Map<AssetClass, Table> tables = new HashMap<AssetClass, Table>();
```

Each object in the symbol table has the getter and setter methods.

5.4.2 Mapping to the MetaTables table

The table *MetaTables* of the SiteLang meta database contains all tables of the application database. As described in Chapter 4, the asset class definitions are mapped to the tables. This table inserts a row for each asset class and also additional associative tables for the relationship mapping, which contains the number of the table, table name, coordinates for the graphical representation in the Storyboard editor, and ID of the story (Figure 3.2 depicts all the tables of the meta database). For this example, a new story is created by the editor, of which the ID is equal to "2".

SiteLangSQLGenerator implements the method *generateSQL()*, which generates the SQL statements from the given asset model. First, the content of the *MetaComponents* is generated. Then by iterating through the tables the statements for the *MetaTables* and their columns for the *MetaAttributes* table are generated. Actually, the generation order is not very important because all data needed for the SQL statements are stored in the symbol table's objects. This is important when creating the application entities by the Storyboard editor.

The analogous method to the *addNewTable(…)* listed in Figure 5.3 is used to create a new *Table* object, but here the created object is added to the symbol table, as shown in Figure 5.8.

```
getMyST().addTable(a, addNewTable(...));

Map<AssetClass, Table> tables = new HashMap<AssetClass, Table>();

public void addTable(AssetClass a, Table t) {
       tables.put(a, t);
}
public Table getTable(AssetClass a) {
       return tables.get(a);
}
public List<Table> getTables() {
       return new ArrayList(tables.values());
}

Map<Relationship, Table> unionTables = new HashMap<Relationship, Table>();
```

Figure 5.8: Methods to add and get *tables* of the symbol table

The *Map* object *tables* associates a *Table* object to the *AssetClass*. Since the associative tables are created when iterating throughout all relationships of the asset class, the *Map unionTables* associates each many-to-many and recursive *Relationship* to the associative table. These two objects contain all the tables that have to be inserted into *MetaTables*.

Since each table must have an ID, after adding a new table, a *Map* object is updated with the integer value for each newly created table. We assume that the tables of this project are identified by the IDs starting from 1000. The generated SQL statements can be found in Appendix B.

5.4.3 Mapping to the MetaComponents table

The table *MetaComponents* of the SiteLang meta database contains the relationships between tables defined in the *MetaTables*. The relation owns an ID, a label, a source and destination tables, the starting and ending coordinates of the arrow, the cardinality constraints, and the type.

The mapping into *MetaComponents* is divided into three groups according to the cardinality constraints:

1. One-to-one relationships mapping. This kind of relationship creates one row in the *MetaComponents* table. The relationship adds an object to the *Map* object in the symbol table where *Relationship* is associated with the ID. The relation label is the name of the relationship. The *Relationship* object contains the referred asset class; therefore, the source table is taken from the *Map tables* containing the *AssetClass* associated with the *Table* (see Figure 5.8). Since the relationship one-to-one has its own object, which relates the *AssetClass* to the *Relationship* object, the destination table is taken from this association. The IDs of the tables are stored in the *Map* object *tableIDs*.
2. Many-to-many and recursive relationships mapping. This type of relationship inserts two rows in the *MetaComponents* table because it is mapped using an associative table. This table includes two foreign key columns, which point to different tables or the same table in the case of the recursive relationship. The label of the relation is composed from the relationship name and the table the foreign key refers to: <*RelationshipName*>_<*TableName*>. The source tables are the tables corresponding to the asset class containing the relationship and the other is the table corresponding to the referred asset class. The destination table is an associative table, whose ID is stored in the *Map unionTables* (see Figure 5.8).
3. Inheritance mapping. If the asset class is involved in the inheritance with other asset classes it is added to the Map object mapping the AssetClass to the ID that belongs to the same numeration as the relationships described above. Each inheritance relationship contains one record in the MetaComponents table. The relation label is composed of the asset class name and its superclass name: <*AssetClassName*>_ <*AssetSuperClassName*>. The source table is the table corresponding to the asset superclass. The destination table is the table corresponding to the inheriting asset class. Their IDs are added to the separate Map object containing an ID for each asset class that implements a superclass.

5.4.4 Mapping to the MetaAttributes table

The table *MetaAttributes* contains the attributes of each table defined in *MetaTables*. Each column of the table corresponds to one row. The attribute owns an ID, a name, and a column type. The column *AttributeKey* indicates whether an attribute is a primary key of the table. It also includes the ID of the table that owns an attribute, and the coordinates of the line for the Storyboard editor. To represent a foreign key attribute, two columns are used. The *ForeignKey* holds the ID of the primary key that the foreign key refers to. The *ForeignKeyForComponentID* column has the ID of the relation for which the foreign key is created (see Section 3.2 for details).

The SQL statements to insert the attributes are generated in the same loop iterating tables. The attributes are kept in the *Map* object of the symbol table, whose key is composed of the table and attribute names map the *Attribute* object. The ID of each attribute is stored in the *Map*. The *Attribute* object encapsulates the properties for the name and type, which are returned by the methods *getAttributeName()* and *getAttributeType()*, accordingly. The *Map* object of the symbol table contains the object for all *ForeignKey*s, whose keys are the table name and attribute name. While adding foreign keys to the table the *Map* is created to hold the *Attribute* and the ID of the relationship it created, whose ID is a value of the column *ForeignKeyForComponentID.*

Chapter 6

Conclusions and Future Work

This chapter gives a brief assessment of this project. First of all, the evaluation of the achieved results is made and then the ideas on further work are given.

6.1 Conclusions

The intent of this project was to extend Conceptual Content Management Systems with the web application. The SiteLang language is a web specification language that allows the generation of applications by means of Storyboarding. Since it supports the structuring of the web specification, it is easy to modify the complex specification by separating it into parts. One of the important parts is a data model specification, which was chosen as a first step for the integration with CCMS, and the mapping of both data models is a subject of this project.

The CCMS entities are modelled by means of asset class definition whereas the SiteLang application entities are stored in the tables of the relational database. The mapping rules to map these two different paradigms were defined. They are based on object-relational mapping.

The asset class is mapped to the table in the SiteLang meta database. The content part and characteristics mimic the columns of the table. Since SiteLang only supports several data types (see Table 4.1) and the characteristics defined in the Asset Definition Language may have any type of Java language, if the narrowing conversion is applied to the value, it may loose its precision.

Two generators were implemented in the CCMS framework that convert the asset class definitions into the SiteLang entities by applying the defined mapping rules. First generator creates an XML document which contains the definitions of the tables in the XML format. In order to modify the entities and create the web specification, the second generator was developed. It converts the asset classes to the SQL statements which could be executed in the database.

The requirements of this project were successfully fulfilled. The asset class definitions are mapped into the SiteLang meta database tables. The future work, which could be done as an extension of this project, is offered in the next section.

6.2 Future work

This section provides a brief overview on further work to enhance the quality of the mapping. Since the asset class definitions are mapped to the relational database tables quite well, there are still several proposals.

First of all, the user should take care defining the asset classes if he has an intent to map them into the SiteLang entities. The data types supported by SiteLang language are described in Section 4.5 that is much less than the Asset Definition Language has. On the other hand, the generator may automatically recognize more data types as listed in Table 4.1.

The constraints on the characteristics and relationships are not yet supported by the CCMS parser. There could be found some way to integrate them into SiteLang data model.

Theoretically, the asset class may inherit from several asset classes. Now the CCMS parser therefore the mapping generators as well support the inheritance only from one asset class. As soon as the new parser is developed the generators also should be adapted.

After the SQL statements are executed in the SiteLang meta database the web application can be created using the Storyboard editor. The graphical representation of the entities may be adjusted in the editor because this aspect was not considered while implementing the mapping.

Bibliography

[Amb04] Scott W. Ambler. Mapping Objects to Relational Databases: O/R Mapping In Detail, Essay from Chapter 14 of Agile Database Techniques, 2004, http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm

[BiZ01] A. Binemann-Zdanowicz. Towards information system modeling on the basis of ASM semantics. In Computer Science Report I-12/2001, Brandenburg University of Technology at Cottbus, 2001.

[GJSB05] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. The Java™ Language Specification, 3rd Edition, Addison-Wesley, 2005

[JAXB20] Sun Microsystems. JSR-000222 Java™ Architecture for XML Binding (JAXB) 2.0, Proposed Final Draft, September 30, 2005, http://jcp.org/aboutJava/communityprocess/pfd/jsr222/index.html, JAXB 2.0.

[JohQBE] Samuel Johnson. Query-by-Example (QBE), http://www.cs.wisc.edu/~dbbook/openAccess/thirdEdition/qbe.pdf. Accessed 26 August 2006.

[ORM31] Objectmatter Inc., Mapping Tool Guide, Version 3.1, Chapter 2: Object Relational Mapping Strategies, http://www.objectmatter.com/vbsf/docs/maptool/ormapping.html

[ScSe03] Joachim W. Schmidt and Hans-Werner Sehring: *Conceptual Content Modeling 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.

[Seh03a] Hans-Werner Sehring: Konzeptorientierte Inhaltsverwaltung: Modell, Systemarchitektur und Prototypen, Dissertation, Hamburg University of Science and Technology, 2003.

[Seh03b] Hans-Werner Sehring. Report on an Asset Definition, Query, and Manipulation Language. Version 1.0. Technical report, Software Systems Department, Hamburg University of Science and Technology, Germany, 2003.

[SeSc04] Hans-Werner Sehring and Joachim W. Schmidt: Beyond Databases: An Asset Language for Conceptual Content Management. Proc. ADBIS 2004, 2004.

[Sma04] 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.

[Str05] Ioannis Stragalis. Storyboard editor for SiteLang. Diploma Thesis, Christian Albrecht University at Kiel, Institute of Computer Science and Applied Mathematics, Kiel August 2005.

[Tha00] B. Thalheim. Entity-Relationship Modelling – Foundations of Database Technology. Springer, 2000

[W3C01] World Wide Web Consortium, XSL Transformations (XSLT) Version 1.0, http://www.w3.org/TR/xslt. W3C Recommendation 16 November 1999.

[W3C02] World Wide Web Consortium, XML Schema, http://www.w3.org/XML/Schema, Recommendation 28 October 2004.

Appendix A

The generated meta database content in XML format

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Sitelang>
    <MetaDatabase>
        <Table>
            <TableName>cocoma_PanoramaPicture</TableName>
            <Attribute>
                <AttributeName>PanoramaPictureID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>place</AttributeName>
                <AttributeType>string</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>PictureID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>PanoramaPictureID</PrimaryKey>
            </PrimaryKeys>
            <ForeignKeys>
                <ForeignKey>
                    <ReferenceTable>cocoma_Picture</ReferenceTable>
                    <ReferenceType>Tuple</ReferenceType>
                    <Attributes>
                        <AttributeName>PictureID</AttributeName>
                    </Attributes>
                    <ReferenceAttributes>
                        <AttributeName>PictureID</AttributeName>
                    </ReferenceAttributes>
                </ForeignKey>
            </ForeignKeys>
        </Table>
        <Table>
            <TableName>cocoma_Artist</TableName>
            <Attribute>
                <AttributeName>ArtistID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>ArtistID</PrimaryKey>
            </PrimaryKeys>
        </Table>
        <Table>
            <TableName>cocoma_Picture</TableName>
            <Attribute>
```

```
                <AttributeName>PictureID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>contents</AttributeName>
                <AttributeType>string</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>title</AttributeName>
                <AttributeType>string</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>placeOfCreation</AttributeName>
                <AttributeType>The data type Place is not recognized
by the generator!</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>artist</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>PictureID</PrimaryKey>
            </PrimaryKeys>
            <ForeignKeys>
                <ForeignKey>
                    <ReferenceTable>cocoma_Artist</ReferenceTable>
                    <ReferenceType>Tuple</ReferenceType>
                    <Attributes>
                        <AttributeName>artist</AttributeName>
                    </Attributes>
                    <ReferenceAttributes>
                        <AttributeName>ArtistID</AttributeName>
                    </ReferenceAttributes>
                </ForeignKey>
            </ForeignKeys>
        </Table>
        <Table>
            <TableName>cocoma_Picture_album_Album</TableName>
            <Attribute>
                <AttributeName>PictureAlbumID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>PictureID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>AlbumID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>PictureAlbumID</PrimaryKey>
            </PrimaryKeys>
            <ForeignKeys>
                <ForeignKey>
                    <ReferenceTable>cocoma_Picture</ReferenceTable>
                    <ReferenceType>Tuple</ReferenceType>
                    <Attributes>
                        <AttributeName>PictureID</AttributeName>
                    </Attributes>
                    <ReferenceAttributes>
                        <AttributeName>PictureID</AttributeName>
                    </ReferenceAttributes>
```

```
            </ForeignKey>
            <ForeignKey>
                <ReferenceTable>cocoma_Album</ReferenceTable>
                <ReferenceType>Tuple</ReferenceType>
                <Attributes>
                    <AttributeName>AlbumID</AttributeName>
                </Attributes>
                <ReferenceAttributes>
                    <AttributeName>AlbumID</AttributeName>
                </ReferenceAttributes>
            </ForeignKey>
        </ForeignKeys>
    </Table>
    <Table>
        <TableName>cocoma_Picture_topic_Subject</TableName>
        <Attribute>
            <AttributeName>PictureSubjectID</AttributeName>
            <AttributeType>int</AttributeType>
        </Attribute>
        <Attribute>
            <AttributeName>PictureID</AttributeName>
            <AttributeType>int</AttributeType>
        </Attribute>
        <Attribute>
            <AttributeName>SubjectID</AttributeName>
            <AttributeType>int</AttributeType>
        </Attribute>
        <PrimaryKeys>
            <PrimaryKey>PictureSubjectID</PrimaryKey>
        </PrimaryKeys>
        <ForeignKeys>
            <ForeignKey>
                <ReferenceTable>cocoma_Picture</ReferenceTable>
                <ReferenceType>Tuple</ReferenceType>
                <Attributes>
                    <AttributeName>PictureID</AttributeName>
                </Attributes>
                <ReferenceAttributes>
                    <AttributeName>PictureID</AttributeName>
                </ReferenceAttributes>
            </ForeignKey>
            <ForeignKey>
                <ReferenceTable>cocoma_Subject</ReferenceTable>
                <ReferenceType>Tuple</ReferenceType>
                <Attributes>
                    <AttributeName>SubjectID</AttributeName>
                </Attributes>
                <ReferenceAttributes>
                    <AttributeName>SubjectID</AttributeName>
                </ReferenceAttributes>
            </ForeignKey>
        </ForeignKeys>
    </Table>
    <Table>
        <TableName>cocoma_FlowerPicture</TableName>
        <Attribute>
            <AttributeName>FlowerPictureID</AttributeName>
            <AttributeType>int</AttributeType>
        </Attribute>
        <Attribute>
            <AttributeName>flowerName</AttributeName>
            <AttributeType>string</AttributeType>
        </Attribute>
```

```
            <Attribute>
                <AttributeName>PictureID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>FlowerPictureID</PrimaryKey>
            </PrimaryKeys>
            <ForeignKeys>
                <ForeignKey>
                    <ReferenceTable>cocoma_Picture</ReferenceTable>
                    <ReferenceType>Tuple</ReferenceType>
                    <Attributes>
                        <AttributeName>PictureID</AttributeName>
                    </Attributes>
                    <ReferenceAttributes>
                        <AttributeName>PictureID</AttributeName>
                    </ReferenceAttributes>
                </ForeignKey>
            </ForeignKeys>
        </Table>
        <Table>
            <TableName>cocoma_Album</TableName>
            <Attribute>
                <AttributeName>AlbumID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>AlbumID</PrimaryKey>
            </PrimaryKeys>
        </Table>
        <Table>
            <TableName>cocoma_Album_albums_Album</TableName>
            <Attribute>
                <AttributeName>AlbumAlbumID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>AlbumID1</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <Attribute>
                <AttributeName>AlbumID2</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>AlbumAlbumID</PrimaryKey>
            </PrimaryKeys>
            <ForeignKeys>
                <ForeignKey>
                    <ReferenceTable>cocoma_Album</ReferenceTable>
                    <ReferenceType>Tuple</ReferenceType>
                    <Attributes>
                        <AttributeName>AlbumID1</AttributeName>
                    </Attributes>
                    <ReferenceAttributes>
                        <AttributeName>AlbumID</AttributeName>
                    </ReferenceAttributes>
                </ForeignKey>
                <ForeignKey>
                    <ReferenceTable>cocoma_Album</ReferenceTable>
                    <ReferenceType>Tuple</ReferenceType>
                    <Attributes>
                        <AttributeName>AlbumID2</AttributeName>
```

```
                    </Attributes>
                    <ReferenceAttributes>
                        <AttributeName>AlbumID</AttributeName>
                    </ReferenceAttributes>
                </ForeignKey>
            </ForeignKeys>
        </Table>
        <Table>
            <TableName>cocoma_Subject</TableName>
            <Attribute>
                <AttributeName>SubjectID</AttributeName>
                <AttributeType>int</AttributeType>
            </Attribute>
            <PrimaryKeys>
                <PrimaryKey>SubjectID</PrimaryKey>
            </PrimaryKeys>
        </Table>
    </MetaDatabase>
    <MetaDatabaseContent></MetaDatabaseContent>
</Sitelang>
```

Appendix B

The generated SQL INSERT statements into the meta database tables

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1004, 'artist', 0, 1002, 1003, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1008, 'albums_Album', 3, 1007, 1008, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1009, 'albums_Album', 3, 1007, 1008, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1005, 'topic_Picture', 3, 1003, 1005, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1006, 'topic_Subject', 3, 1009, 1005, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1002, 'album_Picture', 3, 1003, 1004, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1003, 'album_Album', 3, 1007, 1004, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1007, 'FlowerPicture_Picture', 2, 1003, 1006, 0)

INSERT INTO MetaComponents (ComponentID, ComponentLabel, ConstraintType, SourceTableID, DestinationTableID, DestinationTableType)VALUES (1001, 'PanoramaPicture_Picture', 2, 1003, 1001, 0)

INSERT INTO MetaTables VALUES(1009, 'cocoma_Subject', 450, 450, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1031, 'SubjectID', 'int', 1, 1009, -1, -1)

INSERT INTO MetaTables VALUES(1002, 'cocoma_Artist', 100, 100, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1006, 'ArtistID', 'int', 1, 1002, -1, -1)

INSERT INTO MetaTables VALUES(1003, 'cocoma_Picture', 150, 150, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1007, 'PictureID', 'int', 1, 1003, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1017, 'ArtistID', 'int', 0, 1003, -1, 1004)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1008, 'contents', 'Image', 0, 1003, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1010, 'title', 'string', 0, 1003, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1012, 'placeOfCreation', 'The data type Place is not recognized by the generator!', 0, 1003, -1, -1)

INSERT INTO MetaTables VALUES(1007, 'cocoma_Album', 350, 350, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1027, 'AlbumID', 'int', 1, 1007, -1, -1)

INSERT INTO MetaTables VALUES(1006, 'cocoma_FlowerPicture', 300, 300, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1022, 'FlowerPictureID', 'int', 1, 1006, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1025, 'PictureID', 'int', 0, 1006, -1, 1007)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1023, 'flowerName', 'string', 0, 1006, -1, -1)

INSERT INTO MetaTables VALUES(1001, 'cocoma_PanoramaPicture', 50, 50, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1004, 'PictureID', 'int', 0, 1001, -1, 1001)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1002, 'place', 'string', 0, 1001, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1001, 'PanoramaPictureID', 'int', 1, 1001, -1, -1)

INSERT INTO MetaTables VALUES(1008, 'cocoma_Album_albums_Album', 400, 400, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1028, 'AlbumAlbumID', 'int', 1, 1008, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1029, 'AlbumID1', 'int', 0, 1008, -1, 1008)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1030, 'AlbumID2', 'int', 0, 1008, -1, 1009)

INSERT INTO MetaTables VALUES(1005, 'cocoma_Picture_topic_Subject', 250, 250, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1019, 'PictureSubjectID', 'int', 1, 1005, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1020, 'PictureID', 'int', 0, 1005, -1, 1005)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1021, 'SubjectID', 'int', 0, 1005, -1, 1006)

INSERT INTO MetaTables VALUES(1004, 'cocoma_Picture_album_Album', 200, 200, 2)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1014, 'PictureAlbumID', 'int', 1, 1004, -1, -1)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1015, 'PictureID', 'int', 0, 1004, -1, 1002)

INSERT INTO MetaAttributes (AttributeID, AttributeName, AttributeType, AttributeKey, OwnerTableID, ForeignKeyID, ForeignKeyForComponentID) VALUES (1016, 'AlbumID', 'int', 0, 1004, -1, 1003)