Uppsala Master's Theses in
Computing Science No. 302

An Object-Relational Meta-data Manager for Picture Files

Jakob Elmiger

23rd October 2005

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

The Object-Relational Meta-data Manager for Picture Files is an extension to the Amos II (Active Mediator Object System) database. It implements a wrapper for pictures. Meta data, more specifically a subset of meta data defined by the Exif (Exchangeable Image File Format) standard, is made accessible. Using AmosQL, a relationally complete functional query language, the Object-Relational Meta-data Manager for Picture Files is a powerful tool offering a sophisticated access to query pictures.

Supervisor: Tore Risch
Examinator: Tore Risch

Passed:

Contents

1 Introduction

1.1 What is a Mediator?

The Latin word "mediator" stands for a person who acts as a middleman in a dispute between two or more parties [4].

In Computer Science a *mediator* is a design pattern, described in e.g. [9]. Such a mediator also acts in the role of a middleman. The goal of applying the mediator pattern is to simplify the communication or the interaction between two or more objects. This is achieved by centralization. In a system of m objects where there is heavy interaction between the objects, each object has to know the interfaces of up to $m-1$ objects. As a result, changing the interface of a single object can lead to changes in all other objects. By adding a mediator that takes control of the communication and interaction between the objects each object only has to know the mediator's interface. The number of communication paths and the interdependency between the objects can thereby be heavily reduced.

1.2 What is a Wrapper?

A *wrapper* is another design pattern. It is also known by the name "adapter" [9]. The goal of a wrapper is to change the interface of an object (without changing its functionality). This is achieved by linking the commands for the new interface to the commands of the object's original interface. The wrapped object can be easily accessed by a user who only knows the interface provided by the wrapper. Transparently to him[1], his request will be translated and the original commands will be called. When having to provide a new interface wrapping an existing object is often easier than creating a new one from scratch.

1.3 The Amos II Approach

Amos II (Active Mediator Object System) is developed by the UDBL (Uppsala DataBase Laboratory) group at the Department of Information Technology of Uppsala University. It is a database which allows access to heterogeneous data. This is achieved by following a mediator/wrapper approach [15]. It combines the two patterns described in 1.1 and 1.2.

An Amos II peer is a mediator between the user[2] wanting to access data and different data sources (e.g. relational databases, XML files, Internet search engines) providing data. The user only has to know the Amos interface (e.g. AmosQL[3]) and does not have to know the interfaces of any other

[1]For language simplicity I assume the user is male.

[2]The user does not necessarily have to be human. It can also be a program or even another database.

[3]AmosQL is a functional query language which is relationally complete.

data sources he transparently accesses.

An Amos II mediator contains wrappers to access external data sources. A wrapper is a program to translate and process data between Amos II and the external data source.

1.4 The Aim of this Work

The aim of this work is to extend the functionality of Amos II by providing a wrapper for picture files, more specifically for pictures taken with a digital camera and stored as JPEG-files. The meta data should be made accessible. Additionally, the user should be able to display pictures directly from Amos II using a simple viewer. The picture files however remain within the original file system and are not to be stored within Amos II itself.

In order to achieve this goal general considerations about the availability of meta data for pictures are necessary. Then, a suitable database schema for the meta data of pictures can be defined. Finally, the wrapper can be implemented.

2 Meta Data for Pictures

Within the last few years digital cameras have become more and more popular. There is little doubt digital cameras will eventually replace most traditional film cameras [17]. One of the biggest advantages of digital cameras and a main reason for users' choosing a digital instead of a traditional camera is the fact that they can take almost as many pictures as they want. Which pictures one wants to keep or even to have developed can be decided after having taken them and having examined the result. There is no need "to buy a pig in a poke" as you have the chance to see (and even manipulate) your pictures before ordering or printing them.

With the widespread use of digital cameras, the extremely low marginal costs of taking pictures, and the virtually unlimited storage space comes an accumulation of picture data. It is not unusual for digital camera users to have thousands – or even tens of thousands – of digital pictures stored on their computer hard disks or other storage devices. The number of a user's stored pictures tends to increase and the picture data can become as hard to handle as a shoe box stuffed with hard copy pictures. Hence, searching for a certain picture can be a long and tiring quest.

In order not to have to browse through different folders for hours it would be much appreciated to have methods to access picture data similar to those provided for common data stored in a database (e.g. the birth date of a certain employee within a company's database). Whereas this kind of data is usually referred to as *structured* data, picture data is said to be *unstructured* [7]. From a data modeler's point of view an employee of a company is much easier to describe than a picture is because of the nature of the data structure. At first, one might think there is not too much meta data available which could be of any help for describing a picture (and thereby structuring the unstructured data) but – in fact – there is quite a lot.

2.1 What is Exif?

Exif stands for "Exchangeable Image File Format". It is a standard for the image file format to be used by digital cameras. It was written by the Japan Electronic Industry Development Association (JEIDA) and its most current version (version 2.2) dates from April 2002 [8]. The standard is widely supported by most digital camera manufacturers. However, the differences in their implementation of the standard vary considerably.

The meta data tags defined by the Exif standard cover e.g. the original date and time the picture was taken or digitized, the flash usage, the exposure time, and the focal length. Moreover, there are tags to store Global Positioning System (GPS) data. However, there are only a few camera models which support this kind of data at present.

Although initially intended for achieving better printing results, the meta data defined by the Exif standard can be used for further purposes. In [3] it is shown that semantic picture classification can be done referring to Exif meta data. In particular, they demonstrated that the classification of indoor/outdoor scenes using only the flash information provided by Exif meta data easily outperforms the traditional approach of accessing a picture's raw data in respect to both performance and accuracy. As far as accuracy is concerned, best results are achieved by combining the two methods and taking additional meta data into account.

2.2 Other Picture Meta Data

Exif meta data is by far not the only meta data available for digital pictures. However, most other meta data is content related. For instance, the International Press Telecommunications Council (IPTC), Adobe, and others agreed on a standard to store content meta data within pictures about, for instance, its caption, keywords, and copyrights [11]. This meta data can be viewed and edited by different programs such as [1] or [12].

Besides this standard there are many other ways of assigning content meta data to a picture. One can think of a (proprietary) schema used by applications such as a picture database.

A simple form of content meta data lies within the folder structure set up by any (average) digital camera user to store picture files. Generally, users put their pictures into folders according to the event they refer to or the time they were taken.

2.3 Content Related Meta Data

Obtaining content related meta data generally requires a lot more effort by the user than obtaining non-content related meta data. For instance, the Exif meta data, which is non-content related, is automatically written to the picture file when it is created. However, once content meta data is available it offers more user friendly access methods[4].

In order to maximize the user friendliness while minimizing the required user effort, solutions have been proposed to automatically derive some content related meta data out of non-content related (Exif) meta data. [5] describes a sophisticated method for automatic event clustering using the original date and time stored within the Exif meta data.

[4]e.g. searching for pictures of a certain birthday party

3 Data Model and Schema

First, this section describes the data model used within Amos II. Second, the database schema will be presented. Third, how the schema can be perceived from a user's point of view[5] is described. Finally, the procedures defined within the schema will be discussed.

3.1 The Amos II Data Model

The Amos II data model is a functional data model. It basically consists of *objects*, *types*, and *functions.*

[15] states "everything in Amos II is represented as objects". An object is either a *surrogate* or a *literal.* Whereas surrogates have an object identity and are explicitly created and deleted by the user, the literals are "self-described system maintained objects" and are deleted by the garbage collected as soon as they are are no longer referenced. Literals are e.g. strings, numbers, or collections (i.e. bags or vectors).

Types are used as a classification of objects. They are organized in a supertype/subtype hierarchy and multiple inheritance is allowed. A type inheriting from more than one supertype makes all of its objects (i.e. the extent of the type) being instances of all of its supertypes as well. (In other words, the extent of a subtype is a subset of the intersection of the extents of all of its supertypes.) Types themselves are meta-objects of the meta-type `Type`. The extent of the type `Userobjects` holds all objects of user-defined types. To support a role concept objects can dynamically change their type.

Functions are used to model both properties of and relationships between objects. Furthermore, functions can be used for a broad range of computations. Every function itself is an instance of the system type `Function`. A function consists of two parts: the *signature* and the *implementation.* The signature defines the types of the arguments and the result. The implementation specifies the computation. As in Java, functions (more commonly called "methods" in Java) can be *overloaded* (same function name, but different parameters) and *overridden* (same function name and same parameters, but different implementations in supertype/subtype). As a difference to overridden functions in Java, the resultant type of overridden functions in Amos II can be different.

Functions are classified into

- *stored functions* (stored directly within the Amos II database),
- *derived functions* (implemented as an AmosQL query without side-effects),

[5]This virtual database schema actually is a subset of the complete schema.

- *foreign functions* (implemented in any other (supported) programming language)
- *database procedures* (implemented as an non-side-effect-free AmosQL procedure)

In addition to this classification and to distinguish between foreign functions with and without side-effects I will use *foreign functions* for foreign functions without any side-effects and *foreign procedure* for foreign functions with (intended) side-effects. Similarly, *derived procedures* and *derived functions* shall be used to distinguish between derived functions with and without side-effects.

3.2 The Schema

Defining an appropriate schema is always a crucial task. It should be reasonably stable, so it does not have to be changed too often. The schema must fulfill the requirements specification and one should even take into account the needs of (any not yet specified) future features as far as possible and justifiable. To simplify the writing and maintaining of applications referring to the database, the main logic should lie within (or even better: be inherent to) the structure of the data (i.e. the schema). Still, simplicity is very desirable.

During the development of the database schema a critical point was the fact that the meta data properties could (and should) not be specified right from the start. Though it was clear that only some out of many available meta data properties should be included, the design had to be as open as possible in order to include additional meta data properties at any later point. In other words, the answer to the question of which meta data should be included is dynamic. Therefore, the goal of developing the schema was not to develop a schema for a *specific* kind of meta data (such as, for instance, Exif meta data), but to develop a schema for *any* kind of meta data. Hence, such a schema could be called "*meta*schema". I will not use this terminology but will continue using the simple term "schema" instead.

The central datatype of the schema illustrated in figure 1 on the following page is `PICTURE`. It represents a picture which is stored not in the database but in the file system. The `file` attribute of `PICTURE` is a link to the referred picture file. It is implemented as a stored function with the following signature:

```
file(PICTURE key)->CHARSTRING key fileString
```

This attribute is comparable to a key attribute in a relational database. It is not possible to store two pictures referring to the same file. Furthermore, it should not be possible to load a picture with a string pointing to a non-existing or incorrect file or directory (see 4.4.1 on page 20).

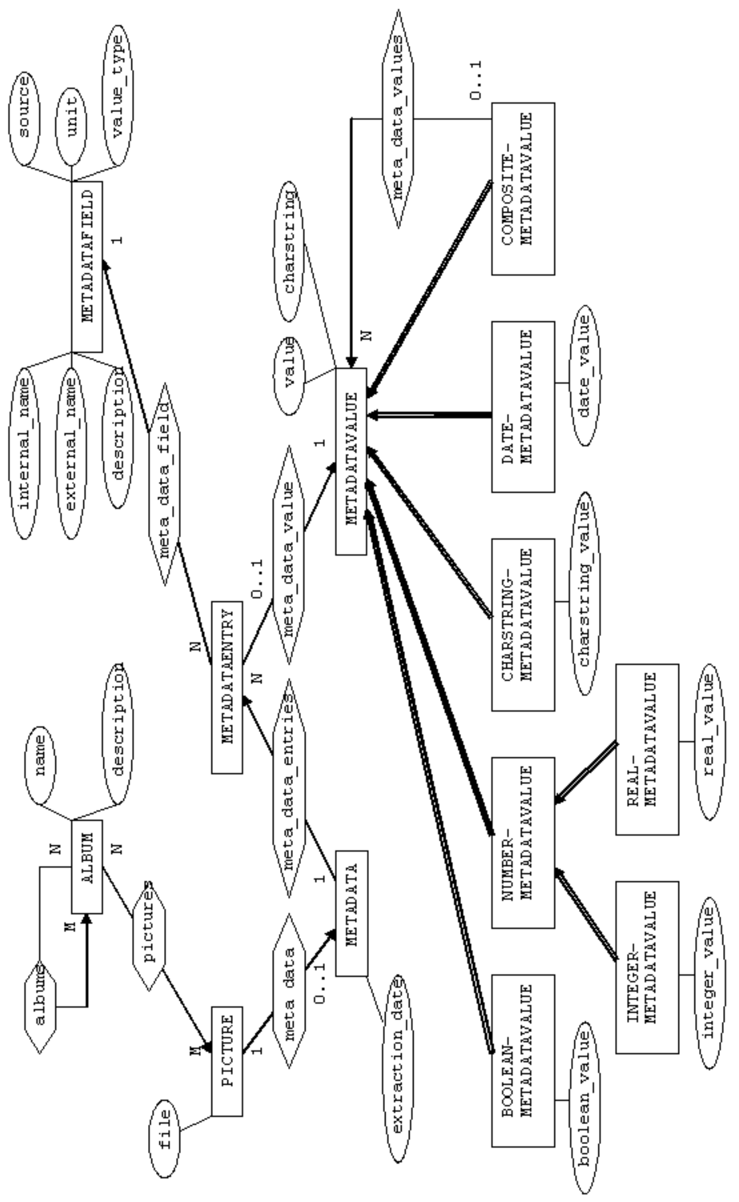

Figure 1: the database schema

The logical structuring of the meta data into the four following types seemed to be a good solution (see figure 1 on the preceding page):

1. METADATA is decoupled from PICTURE and can contain more than one METADATAENTRY.

2. METADATAENTRY is associated with exactly one METADATAFIELD and contains exactly one METADATAVALUE. The application building up the structure (for instance, when a picture is loaded into the database) has to make sure (the subtype of) the contained METADATAVALUE corresponds with the type required by value_type property of the associated METADATAFIELD.

3. METADATAFIELD stores information about a meta data property (e.g. internal_name, unit, description, value_type).

4. METADATAVALUE stores the actual value of the property. The subtypes define the actual type of the (encapsulated) value (e.g. boolean, string, date, integer). To allow (more) complex value types (e.g. a bag of strings for the IPTC attribute “keywords”), the composite pattern [9] was applied: i.e. a COMPOSITEMETADATAVALUE can consist of other METADATAVALUEs.

Hence, the schema presented follows a property/value approach. Meta data properties can be added dynamically without having to change the schema. This increases both stability and maintainability of the application. The database schema is independent from the actual and specific meta data it is used for. However, to increase user friendliness some specific meta data access methods have been implemented. They will be described in 3.3.

As it requires only minimal user effort the main focus of this work is on non-content related meta data. Still, to take into account some content related information the schema offers a structure for organizing pictures within albums (see figure 1 on the preceding page). Between the datatype ALBUM and PICTURE an $m{:}n$ relationship called pictures is established. This means that an an ALBUM can not only contain more than one PICTURE but also a PICTURE can be part of more than one ALBUM. Within ALBUM a recursive $m{:}n$ relationship called albums is defined. It allows an ALBUM to contain (and be contained in) other ALBUMs.

3.3 The User’s View of the Schema

As aforementioned, the user’s view of the schema is in fact nothing but a subset of the full database schema. The objects and functions which do not belong to the user’s view are not truly hidden from the user. A user still can see, query, and access them. Nonetheless, they can be considered as

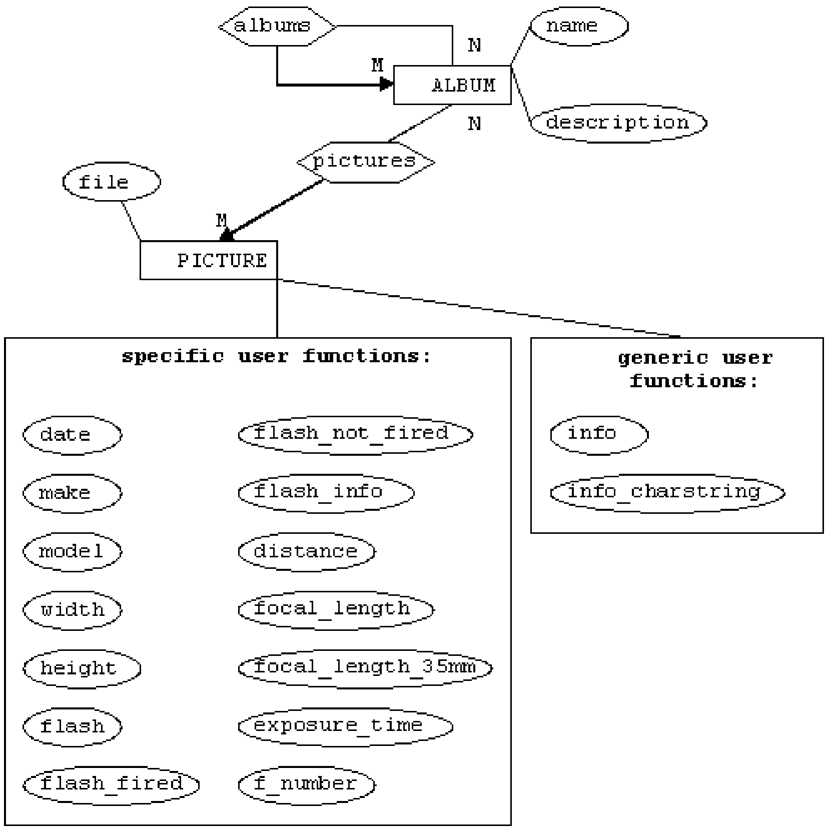

Figure 2: the database schema as it can be perceived by the user

system objects and functions. This adds a (virtual) layer of abstraction and transparency for the user.

The user's view of the schema is shown in figure 2 on the previous page. Only the types and functions relevant to the user are shown. In addition to figure 1 on page 9, the functions implementing specific meta data access are shown (see box "specific user functions"). Moreover, "generic user functions" (i.e. they are independent from any specific meta data properties) are also shown in a separate box. Table 1 on the next page and table 2 on page 14 describe these user functions in more detail.

In order to minimize the "hard coding" of specific meta data structure the methods to directly access specific meta data values (see table 1 on the next page) are not implemented as stored (or foreign) functions, but as derived functions[6]. These functions are the only functions that require some specific knowledge about the type of meta data (to be) stored within the database.

They are implemented using the helper function

```
get(PICTURE p key,
    CHARSTRING internalFieldName key)
    ->OBJECT value
```

to get the value of the meta data property as an `OBJECT`. Afterwards, this `OBJECT` is cast as the appropriate type (logically defined by the `METADATA-FIELD`'s `value_type` property). As an example, the implementation of the user function `date` is given:

```
create function date(PICTURE p key)->TIMEVAL as
       select cast(get(p, 'originalDate') as TIMEVAL);
```

The underlying helper function `get` itself is generic and uses yet another (generic) helper function:

```
meta_data_matrix(PICTURE p key)
                 -><METADATAFIELD nonkey,
                    METADATAVALUE key>
```

This latter helper function provides a simplified view of the structure of the meta data of a picture. Both the `get` and the `meta_data_matrix` functions are not intended to be used (directly) by the user. The `get` function is primarily intended for the administrator to easily implement new user functions once new meta data properties have been made accessible. The `meta_data_matrix` function is also used for the implementation of the generic user functions described in table 2 on page 14.

[6]By the way, the generic user functions (see table 2 on page 14) are also implemented as derived functions.

function name	return type	description
`date`	`TIMEVAL`	the date and time the picture was taken
`make`	`CHARSTRING`	the name of the manufacturer of the camera
`model`	`CHARSTRING`	the name of the camera model
`width`	`INTEGER`	the width of the picture in pixel
`height`	`INTEGER`	the height of the picture in pixel
`flash`	`CHARSTRING`	answers the question whether or not the picture has been taken using the digital camera's flash with `'yes'` or `'no'`
`flash_fired`	`BOOLEAN`	returns `TRUE` if the picture has been taken using the digital camera's flash
`flash_not_fired`	`BOOLEAN`	returns `TRUE` if the picture has been taken without the digital camera's flash
`flash_info`	`CHARSTRING`	gives additional information about the flash settings (e.g. red eye removal, auto-mode)
`distance`	`REAL`	the distance of the subject of the picture in meter (calculated by focal length and therefore might not be very accurate. Note: this method has not yet been tested!)
`focal_length`	`REAL`	gives the focal length of the lens in millimeter
`focal_length_35mm`	`REAL`	the same as `focal_length` but on a 35mm basis
`exposure_time`	`REAL`	gives the exposure time in seconds
`f_number`	`REAL`	gives the F-Number

Table 1: the specific user functions

function name	return type	description
`info`	`<CHARSTRING,OBJECT>`	returns a set of tuples of all available meta data properties with the external name of the `METADATAFIELD` and the value of the `METADATA-VALUE`
`info_charstring`	`<CHARSTRING,CHARSTRING>`	returns a set of tuples of all available meta data properties with the external name of the `METADATAFIELD` and the value of the `META-DATAVALUE` converted to a `CHARSTRING` including the corresponding unit (if available)

Table 2: the generic user functions

3.4 The Procedures

Within the database schema procedures are defined to execute the following tasks:

- loading pictures
- refreshing the meta data of a picture (which has already been extracted)
- displaying a picture by adding it to a simple viewer

The actual implementation of these procedures will be discussed in section 4 on page 17. In the following, only the signatures of the procedures will be given along with some information about their usage.

3.4.1 Loading Pictures

A user can load new pictures into the database by using the procedure

```
load_picture(CHARSTRING fileOrDirectory)
                ->Bag of PICTURE pictures
```

The obvious side-effect of this procedure is to create one (or more) `PICTURE` object(s) and the corresponding meta data objects described in 3.2 on page 8.

For adept users the more sophisticated procedure

```
load_picture(CHARSTRING fileOrDirectory,
             BOOLEAN refreshExisting,
             BOOLEAN includeSubdirectories)
             ->Bag of PICTURE pictures
```

provides two more options[7]. The option `refreshExisting` indicates whether or not the meta data of any existing picture(s) should be refreshed or not. The other option `includeSubdirectories` will only be taken into account when `fileOrDirectory` refers to a directory. If `fileOrDirectory` does refer to a directory, the option `includeSubdirectories` indicates whether or not pictures stored within subfolders should be loaded as well.

In both cases, the procedures return references to the loaded `PICTURE` objects.

3.4.2 Refreshing the Meta Data

The meta data objects of an existing picture can explicitly be refreshed using the foreign procedure

```
refresh(PICTURE)->CHARSTRING answer
```

Unlike the naming pattern of the procedures before this procedure is not named `refresh_`*`picture`* as the argument makes the function name non-ambiguous. The implementation of the procedure is quite simple. Without checking for changes the meta data is extracted once again and the old one is deleted (see 4.4.4 on page 22 for more details). However, the procedure is still powerful. Using the queries e.g.

```
for each picture p
         refresh(p);
```

and

```
for each picture p where like(file(p),'sample_directory*')
         refresh(p);
```

all pictures or all pictures within a specific directory respectively can be refreshed (see [2] for explanation of the `like` function). The reason for using `for each` (instead of `select`) is to control the order of the execution of the foreign procedure (see 3.4.3 on the next page).

The returned string informs the user of the procedure's success in refreshing the meta data of the picture.

[7]In fact, the procedure described before is a derived procedure that calls this (foreign) procedures with the two additional options (`refreshExisting` and `includeSubdirectories`) both set to `TRUE`.

3.4.3 Displaying a Picture

The function

```
display(PICTURE)->CHARSTRING feedback
```

is also a foreign procedure. The side-effect of this function is not to (delete and) create objects within the database, but to display a picture by a simple viewer.

It is important to remember that this function has a side-effect (which is to add a picture to the viewer). For instance, if we have the query

```
select display(p)
from   picture p
where  hour(date(p)) < 12;
```

we cannot be sure that all the pictures displayed within the viewer indeed satisfy the restriction (i.e. they were taken in the morning). With the query formulated as above we cannot control the order in which it is executed. If the procedure `display` is executed before the restriction, we get a wrong result within the viewer as all pictures will be added to it (anyway, the feedback strings are correct). Because this function has a side effect, we have to make sure the restriction is evaluated before the function. To do so, we formulate our query as

```
for each picture p where hour(date(p)) < 12
         display(p);
```

The returned string gives a feedback that the picture has been added to the viewer.

4 Implementation of the Procedures

While section 3 on page 7 discussed the Amos II data model and schema, this section focuses on the implementation of the foreign procedures. This implementation is done entirely in Java.

First, a basic overview will be given. Then, the implementation will be described in more detail. Finally, a step-by-step walk-through shows how everything works together.

4.1 Basic Overview

[6] describes two interfaces: the Amos II `callin` and `callout` interfaces. The former interface provides access from Java to Amos II and is comparable to ODBC or JDBC. Using this interface within a Java program Amos II functions can be called and data can be accessed and manipulated. Moreover, schema definition and manipulation can be done as well. The latter interface provides access from Amos II to Java. Generally, it allows the definition of foreign functions with Java, but we can also use this interface for the implementation of foreign procedures.[8]

The implementation uses a combination of the two interfaces described above. The foreign procedures call a Java method through the `callout` interface. The Java methods call Amos II back through the `callin` interface. The `callin` and `callout` interfaces are encapsulated within the two Java classes `PictureManagerOut` and `PictureManagerIn` respectively.

Java as the language to implement the foreign procedures has been chosen not only because of the described and already existing interfaces but also due to its object oriented features and wide range of support. To take advantage of the benefits object oriented programming offers a good modularization has to be made, i.e. high coherence within the modules and low cohesion between them [10]. Careful attention has to be paid to the interfaces, the information hiding principle and to documentation in order to write (re)usable and maintainable code and – eventually – to achieve a good solution.

4.2 The Representation of the Database Schema within Java

The schema presented in section 3 on page 7 has to be represented within Java as well. For each type of the Amos II schema we define an interface in Java. The classes which implement these interfaces are separated. Instances of the classes implementing the interfaces represent database objects[9] (see proxy pattern in [9]).

[8]In contrast to foreign function foreign procedures do have side-effects (see 3.1 on page 7).

[9]Furthermore, all interfaces used for representing database objects extend the interface `DatabaseObject`, see 4.4.3 on page 21.

We have to keep in mind there is an important difference between Amos II objects and Java objects. Whereas Amos II objects are persistent (i.e. they can be saved on disk) Java objects are not. They are transient, which means they are lost as soon as the system is exited. As a consequence the Java objects generally do not exist as long as the Amos II objects do. The Java objects are only created and used when a foreign procedure is called (e.g. when extracting the meta data from a picture file). As the meta data should be available persistently we have to save it in Amos II objects.[10]

The actual relationship between the Amos II objects and the Java object is only known to the Java object itself. For instance, the implementation decides whether it uses a cache or not. If it does, it reads out the status of an object once, stores it in local fields, and returns their values whenever it receives a request for such action. By using no cache it has to perform a database query whenever it receives a request about the status (the value) of a property.

It was not clear from the beginning which approach might be more effective, so using interfaces was very suitable. All objects dealing with meta data objects (including meta data objects that contain or are associated with other meta data objects) only know their interfaces[11]. This makes the design independent from the actual implementations. Furthermore, one can have different implementations of the same interface and test them against each other.

`MetaDataFieldImpl` is the only class to use *direct representation.* The `METADATAFIELD` objects, which are stored within the database, are extracted and represented by `MetaDataFieldImpl` objects (using no cache). All other meta data objects (`MetaDataImpl`, `MetaDataEntryImpl`, `MetaDataValue-Impl`) are *indirect representations*, i.e. they are designed to be first created as Java objects and saved to the database at some later stage. After being saved to the database there is no more need for those objects (with indirect representation) to be represented within the Java environment. The Java application is only used for extracting or refreshing the meta data of a picture[12] but not for any analytical purposes. Queries (using only side-effect free functions) only access the picture meta data stored within Amos II and do not use the Java environment.

Table 3 on the following page gives an overview of the main representations for the Amos II types and the corresponding Java interfaces and classes. The Java interfaces and classes representing the subtypes of `METADATAVALUE` are not listed in this table but they are defined and named analogously.

[10]How this is done will be explained in 4.3 on the following page.

[11]The class `ImplementationOracle` is an exception and will be discussed in 4.3 on the next page.

[12]or to display a picture

Amos II type	Java interface	Java class
`PICTURE`	`Picture`	`PictureImpl`
`METADATA`	`MetaData`	`MetaDataImpl`
`METADATAENTRY`	`MetaDataEntry`	`MetaDataEntryImpl`
`METADATAFIELD`	`MetaDataField`	`MetaDataFieldImpl`
`METADATAVALUE`	`MetaDataValue`	`MetaDataValueImpl`

Table 3: Java interfaces and classes representing the Amos II types

4.3 The Implementation in More Detail

The class `PictureManagerIn` is the starting point for the Java application. It encapsulates the `callout` interface as mentioned in 4.1 on page 17. All calls from Amos II to Java access a static method within the `PictureManagerIn`. The `PictureManagerIn` reads out the arguments of the Amos II call. Then it checks whether or not the main objects of the Java application have already been initialized. If they have not yet been initialized, it calls the initialization method of the `Main` class. If the objects have (already) been created, the environment is set up and ready to be used. Once the objects are initialized, the `PictureManagerIn` forwards the request to the `PictureDirector`.

The `PictureDirector` is the logical center of the Java application and in charge of handling the request(s). The `PictureDirector` is in a manager's position – apart from delegating work, it does not actually do anything.

The `ImplementationOracle` is the fugleman of the `PictureDirector`. Whereas the `PictureDirector` only knows about the interface of most objects, the `ImplementationOracle` knows the actual classes behind them (i.e. those that implement the requested interfaces).

The class `PictureManagerOut` encapsulates the `callin` interface as mentioned in 4.1 on page 17. All calls from Java to Amos II must access this class. It is often necessary for the Java objects to access the Amos II database (reading out the meta data properties, storing new objects, etc.). This database access is provided by the `PictureManagerOut`.

The actual extracting of the meta data from a picture file is done by a `MetaDataHandler`. There are many ways to extract the meta data. The class implementing the `MetaDataHandler` interface has to decide on the details. The interface is also separated from its actual implementation as described in 4.2 on page 17. The `MetaDataHandler` is the interface, `DefaultMetaDataHandler` and its subclass `DrewNoakesMetaDataHandler` implement the `MetaDataHandler` interface.

Sun's Java API has not yet offered a suitable solution to extract the (Exif) meta data of a picture file. Therefore, an open source solution provided by Drew Noakes [14] has been integrated. It is implemented within the class `DrewNoakesMetaDataHandler`. Once a suitable solution is available by

Sun's Java API, a `JavaMetaDataHandler` could be implemented.

4.4 An Example of How it All Works Together

According to 3.4 on page 14 procedures have to be implemented for:

- loading pictures
- refreshing the meta data of a picture (which has already been extracted)
- displaying a picture by adding it to a simple viewer

The implementation of the first two procedures and some parts of the third one are quite similar. Therefore, only the first one will be discussed in detail. For the other two I will only shortly point out the main differences. This will be done in 4.4.4 on page 22 and 4.4.5 on page 23.

To describe the interactions between the objects and the dynamics within the Java program we have a look at the following example:

We assume the database schema has been set up properly (according to section 3 on page 7). The user enters the command

```
load_picture('sample.jpg')
```

within the Amos II environment. As seen in 3.4.1 on page 14, `load_picture` is a derived procedure which calls a foreign procedure. This foreign procedure is connected to a corresponding static method in the Java class `PictureManagerIn`. The `PictureManagerIn` forwards the request to load a picture to the `PictureDirector`[13].

The tasks that have to be done (or delegated) by the `PictureDirector` in order to load a picture are the following three:

1. create a persistent picture within the Amos II database (`PICTURE`) and a transient Java representation (`Picture`) thereof.
2. extract the meta data from the picture file
3. save the meta data to the Amos II database

4.4.1 First Task: Creating the Picture Objects

As mentioned, the `PictureDirector` only knows the interface of a picture (`Picture`). To fulfill the first task, it will therefore ask the `Implementation-Oracle` to return an object implementing this interface. The `Implementa-tionOracle` knows that `PictureImpl` implements the `Picture` interface (see

[13]Before doing so, the `PictureManagerIn` has to make sure the main objects are initialized according to 4.3 on the previous page.

also table 3 on page 19), so it returns an instance thereof. The constructor of `PictureImpl` makes sure to actually save the picture within the Amos II database. To do so, it calls the `PictureManagerOut`[14].

The `PictureManagerOut` will save the picture persistently (i.e. it creates a `PICTURE` object within the database) and set the `file` attribute (see 3.2 on page 8) to the (absolute) pathname. In case of any errors (e.g. file not found), no picture will be loaded and the `PictureDirector` will generate an error message.

4.4.2 Second Task: Extracting the Meta Data

This task is delegated to an object that implements the interface `MetaDataHandler`. The `PictureDirector` asks the `ImplementationOracle` to provide him with a `MetaDataHandler`. The `ImplementationOracle` returns an instance of `DrewNoakesMetaDataHandler`. `DrewNoakesMetaDataHandler` uses the open source software of Drew Noakes [14] to extract the meta data.

After having received a `MetaDataHandler` object the `PictureDirector` asks it to extract the meta data. Following the strategy of its superclass `DefaultMetaDataHandler` the `DrewNoakesMetaDataHandler` will go through all defined meta data properties[15]. For the properties it knows (by their internal name) it extracts out the values. The corresponding `MetaDataField` object encapsulates them into the appropriate subtype of `MetaDataValue`. The `MetaDataField` and the (newly created) `MetaDataValue` will be put together in a `MetaDataEntry`. All `MetaDataEntry` objects together build up the `MetaData` object that will be returned to the `PictureDirector`.

4.4.3 Third Task: Saving the Meta Data in Amos II

The third task is to make the transient data persistent, i.e. to save the meta data represented by Java objects in the Amos II database. The `PictureDirector` initializes this by setting the `MetaData` object to the `Picture`, i.e. the `PictureDirector` calls the `setMetaData` method of the `Picture`. The `PictureImpl` has to make sure that all meta data related objects are saved to the database.

All interfaces representing the types of the database schema inherit from `DatabaseObject`. So, the classes implementing those interfaces have to know how to save themselves to the database. This is usually done by calling the `addToDatabase` method of the `PictureManagerOut` class.

So calling the `setMetaData` method of the `Picture` triggers all the meta data related objects to call their individual `save` method. The `PictureImpl`

[14]The `PictureImpl` has to ask the `ImplementationOracle` to get a reference to the `PictureManagerOut`

[15]It receives a `MetaDataField` array from the `ImplementationOracle`.

first saves the MetaData object. Before saving itself, the MetaData object (which in fact is a MetaDataImpl object) saves all MetaDataEntry objects by iterating through them and calling their save method. Every MetaDataEntry (or actually MetaDataEntryImpl) object calls the save method of their associated MetaDataField and MetaDataValue object before saving itself. Whereas the execution of the save method of the MetaDataFieldImpl does not have any effect[16], the save method of the MetaDataValueImpl saves itself and stores the value it represents in the appropriate property (which is a stored function, e.g. boolean_value or integer_value, see figure 1 on page 9).

After all meta data objects have been saved to the database, the PictureImpl has to link the METADATA object with the PICTURE within the database. Since it is a database operation the PictureImpl forwards the request to the PictureManagerOut. The PictureManagerOut first deletes all previously existing meta data objects stored within the database that are related to the referred picture by calling the Amos II derived procedure

```
delete_meta_data(PICTURE)->BOOLEAN
```

Then, it defines the link to the newly created METADATA object by setting the stored procedure

```
meta_data(PICTURE)->METADATA
```

accordingly. Deleting all existing meta data objects related to the picture is very important when refreshing the meta data of a picture (see 4.4.4). However, when we load a new picture to the database, there are no previously existing meta data objects referring to the picture. So, calling the delete_meta_data derived procedure will have no effect (i.e. it will not delete any object).

4.4.4 What is Different When Refreshing Meta Data?

The implementation for refreshing the meta data of a picture is very similar to the one for loading a new picture. Task two and three (see 4.4.2 on the previous page and 4.4.3 on the preceding page) are identical. Task one is slightly different: The PictureManagerIn does not receive a string pointing to a picture file but, instead, a handle[17] linked to a PICTURE object stored within the Amos II database. This handle is the parameter that is exchanged between the PictureManagerIn, the PictureDirector, the ImplementationOracle, and the PictureImpl. The main difference to loading a (new)

[16]The MetaDataFieldImpl uses the *direct representation* discussed in 4.2 on page 17 representing an already existing METADATAFIELD.

[17]an Oid object defined within the callin interface

picture is that no persistent PICTURE object has to be created. Nonetheless, a transient Java Picture object representing the PICTURE must be created[18].

Whether or not the meta data of the picture has changed will not be checked. The meta data will simply be extracted once again. If the meta data of the picture is successfully extracted, it will be stored within the database and any previously existing meta data objects associated with the picture will be deleted as discussed in 4.4.3 on page 21. In case the meta data cannot be extracted (e.g. the file string stored within the file property of the PICTURE has become invalid) any previously existing meta data objects will not be deleted but kept. The reason for this is that users might have some of their pictures stored on external storage devices (e.g. CD, DVD)[19]. When trying to refresh the meta data of all pictures deleting those meta data objects is most probably not intended.

4.4.5 What is Different When Displaying a Picture?

The Java Picture object representing the picture to display is created in exactly the same way as it is when trying to refresh the meta data. The PictureManagerIn gets a database handle, forwards it to the PictureDirector who receives a Picture object by forwarding the handle to the ImplementationOracle.

Afterwards, the PictureDirector simply calls the display method of the Picture object. The Picture object, which more specifically is a PictureImpl object, adds itself to the PictureViewer. It is a very simple viewer and basically follows the design of [13].

The PictureViewer allows browsing through a composition of pictures. Internally, this composition is represented as a Vector. Once the PictureViewer is closed the Vector is set to null again. The PictureViewer reopens itself when a Picture is added to it. There are no methods implemented to preload images or to check that a Picture cannot be added more than once.

When displaying a picture no meta data has to be extracted, i.e. task two and three will not be executed.

[18]A Java object representing this meta data surely existed before when e.g. the PICTURE was created. However, when the system exited it was lost again.

[19]This feature is not yet fully supported. Further adjustments have to be made.

5 Conclusion and Future Work

5.1 Summary

Amos II follows a mediator/wrapper approach [15]. It mediates between a data accessor and the data source. External data sources can be made accessible using wrappers. This work is about the development of a wrapper for pictures to make their meta data accessible.

There are different standards for different kinds of meta data of a picture. The meta data defined by the Exif standard [8] (which is widely applied within most modern digital cameras) seems suitable. Unlike content related meta data it does not require a lot of user effort.

After giving an overview of the functional data model of Amos II the schema is elaborated. The schema is intended to be non meta data specific but very generic. It follows a property/value approach, where properties and values are separated and can be dynamically defined without changing the schema. A (small) part of the schema also deals with content related data. The composite pattern is applied to allow more complex values. A virtual user layer of the schema abstracts from underlying details and thereby creates some transparency.

Pictures are loaded into the database using foreign procedures. Foreign procedures are defined for loading pictures, refreshing the meta data of pictures, and displaying pictures within a simple viewer. The foreign procedures are written entirely in Java. They use both the Java `callin` and the Java `callout` interface for Amos II which are encapsulated in the classes `PictureManagerOut` and `PictureManagerIn`, respectively. Most types defined within the schema are represented in the Java application. Interfaces are separated from their actual implementations which makes the implementations dynamically exchangeable. The class `ImplementationOracle` is the only class to know which interfaces are implemented by which classes. The `PictureDirector` is at the heart of the application controlling the logic of the execution. The `MetaDataHandler` extracts the meta data from the picture file. A `PictureViewer` is used for displaying pictures.

5.2 Conclusion

This work extends the functionality of Amos II with a powerful tool to access the meta data of pictures. The generic structure of the schema allows the integration of different meta data standards. It also ensures a better maintainability for the Java application in charge of implementing the foreign procedures. New meta data properties can easily be added.

Some meta data (flash usage, exposure time, etc.) that have been made accessible have high explanatory power for semantic picture classification [3]. Especially the original date and time a picture was taken are of high significance for approaches dealing with auto sorting of pictures [5]. Providing an easy and fast access to this meta data might help to improve those applications as well.

Following the principles of object oriented programming [10] the implementation of the foreign procedure is well modularized. The software is intended to be maintainable and extensible.

5.3 Limitations

At the moment, the integrated open source library [14] only provides read access but no write access to the meta data stored within the picture file. However, other applications exist that make this meta data editable as well (e.g. [12]).

The picture viewer is extremely simple. It only covers the very basic feature of displaying a picture and browsing through a collection. As there are no sophisticated methods implemented to preload (raw) image data it is slow for pictures big in size.

Although the composite pattern [9] is defined within the schema and the Java application, it has not yet been tested. So far, no meta data with complex value structure have been integrated. Actually, adjustments might be necessary to make complex meta data accessible.

5.4 Future Work

Generally, this work can be extended by integrating further meta data (other Exif meta data properties, IPTC meta data, etc.).

Once GPS data is more widely processed by digital cameras a broad range of interesting extensions are imaginable including, for example, a graphical representation of the distribution of pictures within a map [16].

Furthermore, algorithms for picture comparison could be implemented, whereby users could search for similar picture etc. An efficient way to do this even on the compressed JPEG format is proposed in [18].

To deal with issues mentioned in 5.3 there are the following tasks for future work:

- implementing methods for writing the meta data within the picture file
- extending the viewer with more sophisticated features or linking external picture viewing software
- including meta data with a complex value structure[20]

[20]e.g. the IPTC property “keywords”

References

[1] Adobe Photoshop. http://www.adobe.com/products/photoshop/-main.html, Oct 2005.

[2] Amos II Release 7 User's Manual. http://user.it.uu.se/˜udbl/amos/doc/amos_users_guide.html.

[3] Matthew Boutell and Jiebo Luo. Photo Classification by Integrating Image Content and Camera Metadata. In *Proceedings of the 17th International Conference on Pattern Recognition*. IEEE Computer Society, 2004.

[4] Brockhaus Homepage. http://www.brockhaus.de, Jun 2005.

[5] Matthew Cooper, Jonathan Foote, Andreas Girgensohn, and Lynn Wilcox. Temporal Event Clustering for Digital Photo Collections. *ACM – Association for Computing Machinery*, (1-58113-722-2/03/0011):364–373, November 2–8 2003.

[6] Daniel Elin and Tore Rish. Amos II Java Interfaces. Technical report, Dept. of Information Science,Uppsala University, Uppsala, Sweden, Aug 2000. http://user.it.uu.se/˜torer/publ/javaapi.pdf.

[7] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, third edition, 2000.

[8] Exif and Related Ressources Homepage. http://www.exif.org, Jun 2005.

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

[10] Martin Glinz. Vorlesungsskript zur Kernvorlesung "Software Engineering". http://www.ifi.unizh.ch/req/courses/kvse/, SS 2004.

[11] International Press Telecommunications Council Web Page. http://www.iptc.org, Jun 2005.

[12] Irfan Skiljan. IrfanView. http://www.irfanview.com, Oct 2005.

[13] The Java Tutorial: How to use icons. http://java.sun.com/docs/books/tutorial/uiswing/misc/icon.html, Jun 2005.

[14] Drew Noakes. Open Source Library: Metadata Extraction in Java. http://www.drewnoakes.com/code/exif/, 2003.

[15] Tore Rish, Vanja Josifovski, and Timour Katchaounov. Functional Data Integration in a Distributed Mediator System. In Peter M.D. Gray, Larry Kerschberg, Peter J.H. King, and Alexandra Poulovassilis, editors, *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, chapter 9, pages 211–238. Springer, 2003. http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf.

[16] Kentaro Toyama, Ron Logan, Asta Roseway, and P. Anandan. Geographic Location Tags on Digital Images. *ACM – Association for Computing Machinery*, (1-58113-722-2/03/0011):156–166, November 2–8 2003.

[17] Wikipedia - The Free Online Encyclopedia: Digital Camera. http://en.wikipedia.org/wiki/Digital_camera, Jun 2005.

[18] Hong Heather Yu. Visual image retrieval on compressed domain with Q-distance. Technical report, Panasonic Information and Networking Technology Lab.