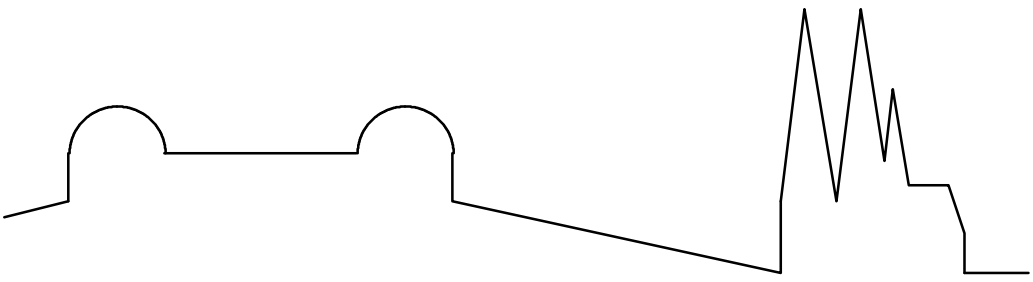
Transparent Java access to mediated database objects

Mattias Bendtsen & Mats Björknert



Thesis for the Degree of Master of Science
Majoring in Computer Science, 10 credit points
Spring 2001

Department of Information Science
Computer Science Division
Uppsala University
P.O. Box 513
S-751 20 UPPSALA
Sweden

Abstract

At Uppsala Database Laboratory (UDBL), a main memory object-relational database system, AMOS II, has been developed. AMOS II has common database facilities and a powerful query language. In addition, the system is based on the mediator / wrapper approach, which makes it possible to combine data from heterogeneous data sources.

Custom applications that handle AMOS II mediator data can be developed in several programming languages. Today, call-level interfaces, comparable to JDBC, exist for C, Lisp and Java. In the Java interface, classes exist for representing simple data types such as strings and numbers. Complex data types, that describe real-world entities, are represented with a generic class for object proxies. A consequence of this generalized representation is that the structure of data and relationships between objects, e.g. inheritance, is lost.

This report presents a prototype we developed, called Amos Class Exporter. It extends the existing interface by translating information between an AMOS II database schema and corresponding classes and methods in Java. It dynamically constructs an equivalent data structure in Java with classes, inheritance, methods for creating objects and methods for manipulation of attributes. Our solution generates Java source files that can be used by an application programmer when developing AMOS II database applications. The files are generated using meta-queries to the AMOS II type system.

Our solution offers a better way of writing AMOS II database applications in Java, as developers can write Java programs to retrieve data from an AMOS II database as semantically rich Java objects. These objects can be modified, which in turn modify the database objects. Modifying persistent data is therefore as simple as modifying any other Java data.

In our work we have addressed problems related to the extraction of necessary information to build Java source files from an AMOS II database using meta-queries. This has concluded in a new approach on how to handle and use custom object proxies for dissimilar types defined in an AMOS II database. We have presented solutions for different problems that we have identified as related to the investigation of this issue.

Table of contents

1. Introduction

Utilization of heterogeneous data sources, such as the Internet, is common today when it comes to collecting important information. With the increased use of Internet and other communication networks, computing environments turn out to be more and more distributed. This brings along a growing need for combining appropriate data from several and potentially different data sources. The integration of data from distributed and heterogeneous data sources presents a number of technical difficulties. One method used to handle these difficulties is the use of mediator technology [26].

AMOS II is the name of a main memory object-relational database system developed at Uppsala Database Laboratory (UDBL). AMOS II is a mediator system, which integrates multiple potentially different and distributed data sources [26]. With a query language called AMOSQL, users can execute object-oriented queries over these heterogeneous data sources [20].

Handling and further processing of mediator data is often accomplished using different custom applications. AMOS II provides several ways of interfacing an AMOS II database system with applications written in different programming languages [17]. One of the existing interfaces is the AMOS II Java interface [8]. The interface consists of two main parts, the callin interface and the callout interface. The callin interface allows programmers to execute AMOSQL statements from the Java programming environment, while the callout interface makes it possible to define AMOS II functions that calls Java methods [8]. These functions, called foreign functions, can then be used in database queries from AMOS II.

With the existing interface between AMOS II and Java it is possible to manipulate database objects by Java methods [8]. This interface is on a low level and therefore many lines of code are required to accomplish easy tasks of database object manipulation. The greatest shortcoming with the existing interface is that the objects retrieved from AMOS II always are in a predefined set of interface classes. With the exception of classes representing numbers and strings, the generic class Oid represents most objects retrieved. The Oid class is a generic depiction of all surrogate objects, i.e. objects that cannot be described using literals only. As a consequence of the current representation, structure of data and relationships between objects, e.g. inheritance, is lost.

The aim of our project is to build a higher-level interface that translates types and functions retrieved from AMOS II to the corresponding classes and methods in Java. That is, dynamically construct an equivalent data structure in Java with classes, inheritance, methods for creating objects and methods for manipulation of attributes. The interface will be built on top of the existing interface and the generated classes will facilitate user-friendly transparent access to the objects in the database. Hence, the database objects will appear as instances of an imported class library. From a programmers point of view this leads to simplification and ease of maintenance, since manipulation of database objects is handled by Java methods. The need for writing code for database communication is also eliminated.

The aim of our report is to describe our project and give an overview of underlying theory. Results of our work and differences between similar solutions are discussed.

There are issues that have not been addressed in this project. This solution does not handle problems related to concurrency and transaction control. Multiple inheritance is not supported in Java and therefore generated classes in our solution only inherit from one supertype.

Our *method* used in this project has been to implement a prototype based on an existing requirements specification. During this process, several technical problems were identified. These technical problems have been carefully discussed, solved and then the solutions have been applied to the prototype. The main problems addressed by our work are extraction of data for Java class definitions using meta-queries and handling of custom object proxies for specific types. Among the more specific issues are the need for a down-casting mechanism to a more specific type, easy access to the first value in a result set, management of multi-valued functions using scan objects, handling of scan objects for a specific type, handling of methods (functions) that cannot be derived to a specific type and management of null values in the result of a query. As a complement to the practical work, we have conducted a study of existing literature, research and solutions in this field. This material is used as a theoretical foundation in this report when our work is discussed.

Several techniques have been developed to manage database objects from Java. The approach of the Amos Class Exporter (ACE) interface differs in several ways from solutions presented in the theoretical section of this report. ACE is not an interface on the same level as a call-level interface. A call-level interface is used on a lower level compared to this solution. This solution provides generation of source code, which uses the AMOS II Java call-level interface. Therefore, it can be seen as an intermediate layer built on top of the existing interface between an AMOS II database and a Java application. Complete data structures from the database are translated into data structures for the Java language that provide transparent access to the database. ACE provides database object querying and manipulation capabilities through standard Java methods. This is achieved with the meta-data querying capabilities of AMOS II. The interfaces generated are completely built upon the data retrieved using meta-queries to the type system, which is an important aspect of how ACE is different from other solutions.

Our report is divided in two main sections. The first part provides the underlying theory on which the discussion of our work is based. In this part, a short overview of different database systems and their evolution is given. This is followed by an overview of the AMOS II system. Other issues discussed in this part are ways of database connectivity from a programming language, both call-level interfaces and binding solutions for the Java language. This includes a summary of the AMOS II Java interface. In the second part of the report, our work is presented in detail. The architecture of ACE is explained, benefits of this solution are presented and a comprehensive example of how it works is given. The report is concluded with our findings of this work.

2. Background and related work

This chapter gives an overview of the evolution of different database systems, the mediator system AMOS II and different technologies used for database access from a programming environment with emphasis on Java.

2.1 Database systems

Database systems have evolved rapidly during the last decades. This section provides a short overview of different types of database systems, differences in how they store data and their main advantages and disadvantages. New requirements such as the ability to handle complex objects and to integrate heterogeneous data are presented. One solution to the new requirements is the mediator / wrapper architecture, which concludes this section.

2.1.1 RDBMS

Most of today's database market consists of relational databases based on the principles described by E. F. Codd in the late 1960s and early 1970s. E. F. Codd, a member of the IBM Research Laboratory in San Jose, California, laid out the basics of relational database systems in a now famous paper [5]. In the early 1970's several experimental database management systems, based on the relational technology, were developed. Commercial products became available in the late 1970's. [4] Widely used products in large multi-user environments today are Oracle and DB / 2 [13].

In a relational database system (RDBMS) the database is thought of as a set of relations, which are physically represented as two-dimensional tables (see figure 1). Each relation contains a set of distinct rows, called tuples, which in turn contains a number of fields. Each field has a data type, e.g. character, string, time, date, number etc. Any attribute (field) of a tuple can store only a single value from a predefined domain. A domain is the set of allowable values for an attribute. The tuples of a relation must be composed of the same attributes. [9]

Customers

Customer_ID	Customer_ Name	…
C001	Andersons Inc.	…
C002	Elon Corp.	…

Orders

Order_Number	Customer_ID	Product_ID	Quantity	…
55254	C002	GT5	100	…
55255	C001	PF6	500	…

Figure 1: A simple example of two relations, customers and orders

The relational model is based on a set of mathematical operations that can be applied to the relations in the database. This mathematical basis, called relational algebra, offers data independence and is used for specifying different requests on the database. [3] Relational algebra is also important for query processing and optimization, but there are few commercial RDBMSs today, which are based directly on the relational algebra. Most RDBMSs provide a high-level declarative language interface, e.g. SQL. With a

query language like SQL users can specify the results desired rather than how to obtain the results, hence leaving the actual optimization and decisions on how to execute the query to the DBMS. [9]

Relational modeling focuses on the information in the system and not on the behaviour of the data. The strengths of a RDBMS are primary its simplicity, its strong theoretical foundation and its support for data independence. But relational systems also have weaknesses and are inadequate for certain types of applications, e.g. high-speed web applications. When developing such applications using a RDBMS, the data models in the database and the application are incompatible. This requires a conversion of data from relational to object every time the application accesses data, which results in poor performance. [6]

There is an ongoing trend in the database technology towards extended support for complex data types, e.g. images, audio, video or any large unstructured object, which are not directly supported in a RDBMS. Many RDBMSs handle complex data types simply via a reference to a file, or stored as binary large objects (BLOBS). These solutions lack much of the functionality and many of the protection mechanisms naturally offered by a RDBMS. [6]

2.1.2 ODBMS

An object database management system (ODBMS) constitutes of database capabilities, as concurrency control and data recovery integrated with an object-oriented programming language. This is different from databases using SQL, which is a separate language that defines, retrieves and manipulates data. SQL databases are two-level stores, separating memory from persistent storage, while ODBMSs overcomes this impedance mismatch. In an ODBMS memory and persistent storage are the same thing, which provide better performance and versatility. The first ODBMS prototypes were developed in the early 1990's. [6] Popular ODBMSs today are Jasmine, Gemstone, O2 and Object Store. [13]

In an object-oriented database the information is stored as objects. Each object is uniquely identifiable in the database by an object identifier (OID). An object can be described as a real-world entity that contains information about its current state (attributes) together with the actions that can be taken on the object (methods). An advantage with this approach is that the external aspects of an object are separated from its internal details. The internal details are hidden from the outside world and can be changed without affecting the application programs that use it. This solution with information hiding is referred to as encapsulation and consequently provides data independence. [6]

ODBMSs emerged in response to the increasing complexity of database application. The concept of objects allows the real world to be modeled more naturally. It is appropriate for managing complex data types and relationships, and provides better maintainability and reusability of code. An object-oriented architecture leads to a high level of congruence between the data models for the applications and the database. With an

object-oriented language programmers can write complete database applications with a great deal of simplicity. [13]

A disadvantage with ODBMS is that the systems are more complex and more difficult to use in comparison to RDBMSs. The biggest drawback though, has been the lack of standards concerning a data model and an object-oriented query language. [6] The SQL standard is one of the reasons for the success of commercial RDBMS. However, the object database management group (ODMG), a consortium of ODBMS vendors, has specified de facto standards for an object model, an object query language (OQL), and the bindings to object-oriented programming languages. [9] The ODMGs latest object database standard, version 3.0, was published in January 2000 [16]. In the future more ODBMSs are likely to add querying capabilities similar to those of RDBMSs. [9]

2.1.3 ORDBMS

Relational supporters claim that complex applications can be handled by extensions to the relational model and that the relational functionality is a necessary part of a DBMS. [6] Object-relational database management systems (ORDBMS) have evolved from relational database technology, and supports some of the object extensions needed to deal with complex data. [9]

Informix is the market leader in object-relational databases. Other products in the ORDBMS market are Oracle 8.x, Universal Database (DB/2 Extenders) and UniSQL/X [13]. Many of the products work as RDBMSs with enhanced object-oriented functionality, while others like UniSQL/X were developed from the beginning as an ORDBMS product. [9]

Many of today's relational database products are beginning to, or will eventually, deliver object-oriented extensions to their products. SQL-99, also known as SQL3, is considered an important step towards a future standardization of object-relational databases. SQL-99 provides object-oriented features such as abstract data types (ADTs) and inheritance. ADTs allow users to define new structures for user-defined types and support the object-oriented concepts of encapsulation. Inheritance is the ability to create new objects (new ADTs) based on existing ones. [9]

Information in an ORDBMS is like in a RDBMS stored in tables of rows and columns, but tabular entries can with the support for ADTs contain richer data structures. With ADTs, operations and functions associated with new data types can be used to index, store, and retrieve records based on the content of the data type. Since the information is stored in tabular form, many of the advantages and disadvantages of a RDBMS also apply to a ORDBMS. For example, from a programmer's point of view, translation between objects and tables is still necessary. [13]

2.1.4 The wrapper/mediator approach

With the increased use of communication networks, such as the Internet, computing environments turn out to be more and more distributed. There is a growing need for combining appropriate data from several and possibly different data sources. The integration of data from distributed and heterogeneous data sources presents a number of technical difficulties. One way of overcoming these difficulties is the use of the mediator / wrapper approach [26]. In the wrapper / mediator architecture, functionality can logically be divided into two different parts, a mediator and a wrapper [20]. The wrapper part offers access to data from different sources with possibly different formats and the mediator has functionality to process and combine the data retrieved. A wrapper concept was introduced in a recent paper [14] as a new part of SQL, called SQL / Management of External Data. SQL / MED contains language constructs for retrieving data from a foreign server using a foreign-data wrapper [14].

An interface between a workstation and a database server that merely defines communication protocols and formats of database elements is not sufficient to manage representation problems in existing data sources. In order to handle similar problems, the interfaces should be active. Wiederhold [26] describes the process of the active interfaces as mediation. This includes the processing needed for the interfaces to work, the transformation of data to information and needed storage. In order to provide mediation, Wiederhold proposes a specific category of software that mediate between an application and data resources making the application independent of these resources [26]. According to Wiederhold, mediators are defined as "software modules that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications". Three layers can be identified in a conceptual view of the mediator architecture:

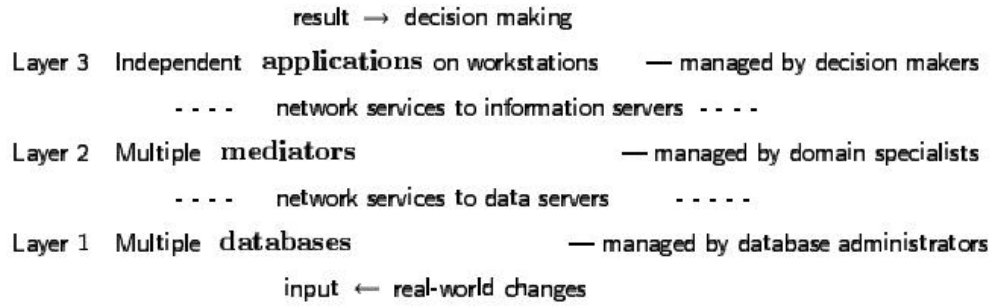


Figure 2: Mediator architecture [26].

In a mediator / wrapper architecture, a wrapper is a software component that translates data from a specific data resource, e.g. a website. The translated data can then be used in the mediator system and applications using the mediator. A common practice today when developing wrappers for different data sources is the use of a wrapper generator toolkit. An example of a wrapper generator is the World Wide Web Wrapper Factory (W4F). With this toolkit, the user specifies extraction rules for how the desired data should be extracted and W4F then generates Java class files for use in the applications. [1]

2.2 AMOS II

AMOS II is a second-generation object database with a complete query language, i.e. an object-relational database system. The main difference compared to an ODBMS, where an integrated object-oriented language is used to define, retrieve and manipulate data, is the use of a declarative query language with syntax similar to SQL. AMOS II is a system based on the mediator / wrapper approach [21]. The purpose of the AMOS II research project is to develop a mediator architecture to make it possible to combine and analyze data from several different data sources [18]. Several other systems with functionality related to AMOS II exist, such as DISCO, Garlic and Multibase [12].

2.2.1 Architecture

AMOS II is a distributed system consisting of several mediator servers communicating with a TCP / IP based protocol. Other protocols are used for communication with non-AMOS II systems, e.g. data sources that communicate using ODBC or HTTP. Every AMOS II server is an object-oriented, lightweight and extensible DBMS with common DBMS functions built in. The DBMS part of an AMOS II server consists of a storage manager, a recovery manager, a transaction manager, a disk backup manager and a query processor for the query language of AMOS II, AMOSQL. [20]

AMOS II is designed as a main-memory DBMS for optimal performance [22]. It can be used as a single-user database as well as a multi-user server to applications and to other AMOS II systems [20]. AMOS II runs under Windows NT, consisting of about 1500 KB meta-data and 350 KB of code [21]. The architecture of AMOS II is illustrated below.

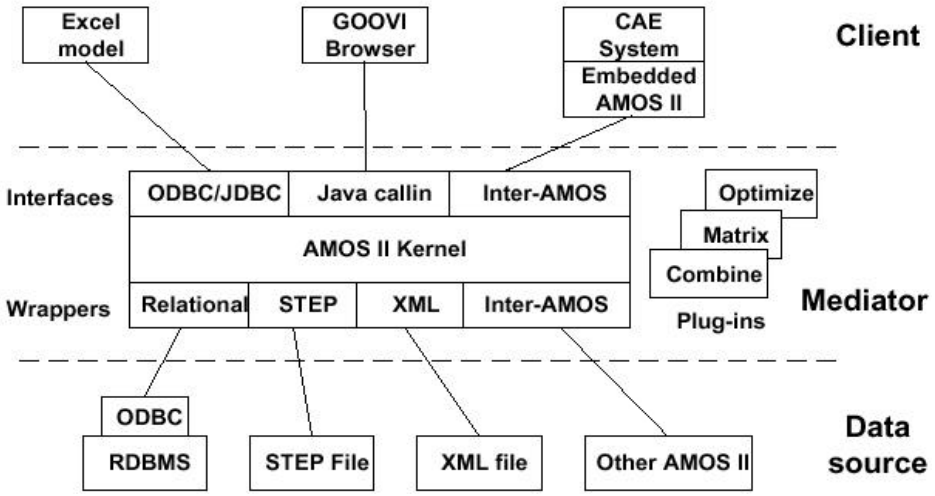


Figure 2: AMOS II Architecture [20].

2.2.2 AMOS II Data Model

Everything in an AMOS II database is represented as objects, both user-defined and system objects. Two categories of objects exist, literal objects and surrogate objects. All surrogate objects are assigned OIDs, which are managed by the system. Examples of

literal objects include objects that represent strings and numbers. The more complex surrogate objects usually represent something from the real world, such as a car, a person or a soccer game. Meta-objects, e.g. types and functions, are also represented as surrogates. This gives the user the possibility to query the system objects in AMOS II, with the opportunity to obtain for example the structure of a type defined in the system. The feature of meta-object querying allows a user to query the entire structure of a complete mediator server. [20]

2.2.3 Types

An object is always an instance of at least one type. The term type is similar to class in object-oriented programming. The types are arranged in a supertype/subtype hierarchy. This means that if an object is an instance of a type, it is also an instance of all its supertypes. AMOS II also supports multiple inheritance. Four categories of types exist in AMOS II: stored types, derived types, proxy types and integration union types. [21]

The stored types are the ordinary types with the definition stored in the mediator server. Instances of a stored type are managed by the user of the system. Types are created using special AMOSQL statements. The general syntax of such a statement is: [20]

```
create type <typename>
```

If a type inherits characteristics from a previously defined type, the general syntax looks like:

```
create type <typename> under <previously defined type>
```

2.2.4 Functions

In AMOS II, queries on objects, attributes of objects, methods of objects and relationships between objects are represented by functions [21]. Basic functions can be organized into five categories: stored functions, derived functions, foreign functions, proxy functions and database procedures [20].

A stored function represents characteristics of an object in the database. For example, common properties of an object of type person are name and age. A call to the stored function name on such an object returns the current value of the attribute name. The general syntax for creating stored functions is written as: [20]

```
create function <functionname(type)> -> <return type> as stored
```

It is possible to update the value of a stored function. The query language AMOSQL supports this functionality. The general syntax for updating a stored function is: [20]

```
set <function name(object instance)> = <value>
```

Derived functions are functions defined with the use of other previously defined AMOSQL functions. It is not mandatory for a function in AMOS II to have arguments and there is no upper limit on the number of arguments that a function can have. [20]

Foreign functions are functions called from AMOS II that are defined in a programming language. Currently, interfaces exist for the languages Java, C and Lisp [8][17]. A foreign function is similar to a method in an object-oriented database. A foreign function written in Java is defined in AMOS II as:

```
create function generate_interfaces(vector t, vector f) -> integer as foreign
“JAVA:AmosClassExporter/aceMain”;
```

In this example, the function `generate_interfaces` calls the method `aceMain` in the Java class `AmosClassExporter`. It takes two vectors as parameters and returns an integer. In order for such functions to work, the AMOS II kernel has an interface to the Java Virtual Machine implemented using the Java Native Interface [20], which makes it possible to call non-Java code from Java [7]. The foreign function feature provides enhanced functionality to the AMOS II system and it is possible to write large programs in Java linked to AMOS II.

2.2.5 AMOSQL

The declarative multi-database query language AMOSQL is an extended subset of the OO parts of SQL-99 [20]. AMOSQL is based on OSQL and DAPLEX with functionality added for mediation, multi-directional foreign functions, late binding and active rules [20]. The syntax of AMOSQL is similar to the syntax used in SQL. A general example of a query statement follows:

```
select <result> from <declarations of local variables> where <condition>
```

In a number of steps taken by the query optimizer, queries are carefully optimized before execution. [21]

AMOSQL can be used to issue queries on the meta-objects that constitute a database. Several functions are built in to the AMOS II system for this particular purpose. For example, attributes of a type can be obtained with a single function call. Furthermore, entire structures of inheritance and relationships between objects in the hierarchy can be extracted using a combination of such queries. Details on the name of a function, the number of parameters and results, data types of parameters and results are also possible to access using similar techniques. [10]

2.3 Call-level interfaces for database connectivity

In this section related ways of database connectivity through call-level interfaces (CLI) are presented. An overview is given of ODBC and JDBC [2] followed by a presentation of the AMOS II Java interface.

2.3.1 ODBC and JDBC

The ODBC and JDBC APIs have similarities in their architecture and will therefore be jointly outlined in this section. Both are call-level interfaces based on the X/Open SQL CLI specification that stipulates how client-server interaction is to be implemented in databases [2]. JDBC and ODBC are CLIs for relational databases.

ODBC is more or less considered to be the standard CLI for many database engines. It exists on several platforms, including Windows, UNIX and the Macintosh platform. ODBC was designed for maximum interoperability, which makes it possible for a single application to access different DBMSs with the same source code. The database applications call functions in the ODBC interface, which are implemented in database-specific drivers. Drivers are handled by the driver manager, which is an intermediate layer between the application and the driver in use. The use of a driver layer isolates applications from database-specific calls. The ODBC architecture has four layers:

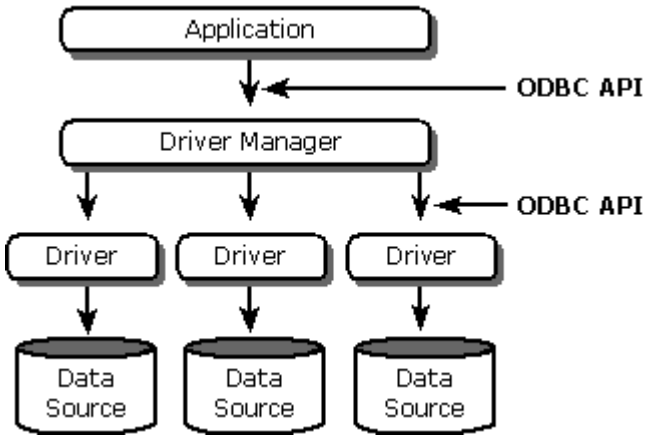


Figure 3: ODBC Architecture [15].

The architecture of JDBC, which is the Java community's contribution to a standardized CLI, has an architecture identical to the one used by ODBC. Their common architecture is a result of the underlying specification. Just like ODBC, JDBC uses a separate layer between the application and the database driver, called the JDBC driver manager. [25]

2.3.2 AMOS II Java interface

The AMOS II system provides an interface between AMOS II databases and the programming language Java. Essentially, there are two ways to communicate between AMOS II and Java:

- Calls are made to AMOS II from Java with the callin interface.
- Foreign functions are defined in AMOS II that calls Java methods with the callout interface. [8]

It is possible to use a combination of these two interfaces [8]. When combining the features of the callout interface with the functionality offered by callin, a developer can

write programs, which are started from AMOS II and then use the callin interface to communicate with AMOS II.

Communication through the callin interface can be accomplished in two different ways. With the embedded query interface, AMOSQL query strings are passed to AMOS II and Java methods are used for accessing the results. With the fast-path interface, functions predefined in AMOS II are called using standard Java methods. The fast-path interface is, as its name implies, much faster than the relatively slow embedded query interface. [8]

In the AMOS II Java interface there are several useful methods available to the programmer. Methods exist for accessing literal elements, executing queries, traversing through rows in a record set, creating/deleting objects and updating AMOS II functions [8]. In the AMOS II Java API the following classes constitute the core:

- `Connection`. Represents a connection to an AMOS II database.
- `Scan`. A result set of Tuples (rows) returned from the database.
- `Tuple`. Corresponds to a single row in a `Scan`.
- `Oid`. Object proxy for surrogate objects.

When a query is executed through the Java callin interface database objects are returned in a `Scan`. The `Scan` consists of several `Tuples`, each of them containing one or more objects, literal or surrogates. When retrieving surrogate objects they are always represented using the class `Oid`. This is currently a deficiency in the AMOS II Java interface. Object proxies are Java objects corresponding to objects in an AMOS II database. In the AMOS II Java interface, object proxies are represented by the class `Oid` [8]. Object proxies can correspond to any kind of data stored in an AMOS II database.

2.4 Binding solutions between databases and Java

There are several different approaches of transparently mapping database data to objects in programming environments. This section focuses on technologies related to the mapping between databases and the programming language Java.

2.4.1 ODMG Java Binding

The Object Data Management Group (ODMG) is a non-profit standards consortium with both members from the industry and academic members. ODMG addresses problems related to object storage. In the Internet-driven development environment of today, there is a need for Java developers to store objects, i.e. to give them persistence [2]. The ODMG Java Binding is a standard that adds persistence capabilities to the Java programming language.

The standard provides native Java object storage and application portability between different storage products. It is an extension to the Java language consisting of database programming features that makes it possible for developers to write database

applications entirely from within Java. The ODMG Java Binding specifies a tight integration between the application and the database, united in a single data model. For developers, it gives the possibility to store objects in relational, object-relational or object databases with no worries for the actual storage mechanisms involved. This requires of course that the underlying storage is ODMG-compliant. [2]

Database transparency was the fundamental principle in the design of the ODMG Java Binding. This means that the application developers' persistent classes and the database schema are one and the same. An ODMG-compliant database manages an object cache where Java objects are stored when retrieved from the database. Objects that have changed in the object cache are automatically written to the database. [2]

Two languages are included in the ODMG Java Binding. One of them is the Java Object Manipulation Language (OML), a set of classes with support for collections, transactions and databases. Also included is the Object Query Language (OQL), which provides object querying in Java and returns Java objects as a result. OQL does not handle data definition and data manipulation, which is handled by OML.

Compared to a JDBC-based solution, where a developer has to maintain equivalent data structures in both the database and Java, an ODMG compliant solution has major advantages. The most important advantage with such a solution is that the developer does not have to be fluent in both the database language, e.g. SQL, and the programming language used. [2]

2.4.2 Java Blend

Java Blend is a software product from Sun Microsystems co-operatively developed with Baan that provides translation between Java objects and relational databases. It implements the ODMG Java Binding on relational databases [2]. The product is available on the Sun Solaris and Microsoft Windows NT platforms and consists of a tool and runtime environment. Java Blend is intended to be used for overcoming the impedance mismatch between the relational data model and object-oriented data models. This is accomplished using an object-oriented view of relational data. [23]

With Java Blend, it is possible for developers to write programs entirely with Java objects. Mapping is a technique used when translating between different data models, e.g. the relational data model and the object-oriented data model. Java Blend automatically maps database records to Java objects, or Java objects to databases, so developers can avoid translating programming language data structures to database tables. [24]

Since Java Blend is an implementation of the ODMG Java Binding, advantages that apply to ODMG-compliant solutions also apply to Java Blend. An advantage of Java Blend is that developers do not need to be proficient in SQL or have a detailed understanding of database schemas. Java Blend makes it possible to write entire database applications using Java objects only. Before Java Blend was available, developers had to be acquainted with both SQL and the programming language.

Furthermore, they had to have extensive knowledge of how application data was represented, including the database schema and the data structures in the programming language. The developer also had to write commands to translate the SQL representation of data into the programming language representation. This is an example of the impedance mismatch between the relational database and the object-oriented programming language. With Java Blend, data retrieval is accomplished using Java classes and modification of data in the underlying database is carried out in a way analogous to modification of other Java objects. [24]

Another advantage with Java Blend is object querying. Queries can be written using the Object Query Language (OQL), which has syntax similar to SQL-92 and the Java programming language. OQL queries understand the relationships between classes which make it possible to execute queries in a more natural language compared to SQL. When a query is executed on a specific class, the query returns all matching objects as a set of records. From the programmer's point of view, it is possible to iterate through this collection of objects and eventually carry out operations on particular objects. [24]

Java Blend software is built on top of the JDBC API, which is used for its database connectivity [24]. The main reason for using JDBC technology as a foundation for the product is that JDBC provides a level of database independence. With the use of JDBC, and technologies based on this approach such as Java Blend, it is easy to switch between different underlying database systems [23].

Software products similar to Java Blend that are worth mentioning in this context exist. Products with related functionality are CocoBase, ROF and Top Link. A shared capability of all these products is that they allow Java programmers to work with Java objects and let a translator use the JDBC interface to store and retrieve data in a relational database.

2.4.3 Java Data Objects

ODMGs Java Binding is a basis for the Java Data Objects (JDO) specification. The main objective of JDO is to provide support for transparent persistence of Java objects. JDO is important because prior to this standard there was no Java platform specification with an architecture for storing Java objects in a transactional data store, i.e. in a database. [11]

Applications using JDO will be independent of the underlying data store used by a specific implementation. Among data stores for which an implementation is planned are different file systems, hierarchical databases, relational databases and object-oriented databases. [11]

JDO allows for almost any class to become persistent. Included in the list of classes that cannot be made persistent are classes implemented using the Java Native Interface or subclasses of Java system classes. Every persistent instance of a specific class is identified using a unique identifier provided by JDO. The normal definition of two equal objects in Java is that the references refer to the same object in memory. This is not

a sufficient definition for use in JDO since several Java Virtual Machines in different transactions may reference the same database object. JDO provides a class enhancer mechanism that processes existing class files and outputs an enhanced version of the class that is persistence capable. To be more specific, the processed classes implement the interface PersistenceCapable and methods contained within this interface. [11]

3 The Amos Class Exporter project

Amos Class Exporter (ACE), is an extension to the mediator system AMOS II and provides a high degree of transparency when manipulating AMOS II database objects using the AMOS II Java interface. ACE provides automatic generation of Java classes. This makes it possible to represent objects retrieved from an AMOS II database as objects more semantically rich than objects represented using the generic class Oid. In this chapter the requirements, the architecture, and noteworthy design and implementation details are described. The chapter is concluded by an example on how generated source files can be used.

Several techniques, outlined in the previous chapter, have been developed to manage database objects from the Java programming environment. The approach of ACE differs in various ways from the technologies presented above, but similarities can be recognized. It cannot be directly compared to a call-level interface, since complete data structures in a database are translated into data structures for a programming language that provide transparent access to the database. A call-level interface is used on a lower level compared to this solution. This approach has similarities with Java Blend, although Java Blend handles database objects in a different way. ACE provides database object querying and manipulation capabilities through standard Java methods. This is achieved with the meta-data querying capabilities of AMOS II. The database schema in an AMOS II database can be queried as any other database objects. The interfaces generated are completely built upon the data retrieved using meta-queries to the type system.

ACE is an example of a program that uses a combination of the callin and the callout interface. The program is started using standard AMOSQL query syntax, supplying the necessary parameters. With extensive use of the callin interface, essential information is extracted from the database using meta-object querying. ACE utilizes a feature of the AMOS II Java Interface called the tight connection where AMOS II works as an embedded database. This provides the fastest possible connection between ACE and AMOS II since they both run in the same address space. When the program is called via the callout interface, the address space is always shared between the application and the AMOS II server. [8]

The AMOS II Java interface is well suited for minor extensions to AMOS II. This is supported by the following:

- One issue always raised in discussions about Java is the performance of the Java language [7]. Performance is always important in applications programming but the tasks carried out by ACE are not time critical. If this were the case, a programming language such as C would be more appropriate.

- A primary goal of the Java language is safety [7]. It is safer to use the AMOS II Java interface than for example the C interface, since errors in the Java programs are not fatal to AMOS II. In the C language, common programming errors are related to direct access to system memory through the pointer mechanism, functionality not available in Java [7].

- Several libraries for common functions are available to Java programmers, which eases the implementation of an application [7].

We have identified the following benefits of extending AMOS II with Amos Class Exporter:

- Objects are retrieved from the database as semantically rich objects.
- Simplified access to AMOS II database objects from Java.
- Modifying persistent data is as simple as modifying any other Java data.
- Improved reuse and maintainability.
- Homogenous implementation of generated classes.
- Time used for developing AMOS II database applications in Java can be reduced.

3.1 Purpose

The aim in the ACE project is to build a higher lever interface that translates types and functions retrieved from AMOS II to the corresponding classes and methods in Java. The query-driven interface will dynamically construct an equivalent structure in Java with classes, inheritance, methods for object creation and deletion and methods for manipulation of attributes.

The interface will be built on top of the existing interface and the generated classes will facilitate user-friendly transparent access to the objects in the database. Objects retrieved from the database will appear as instances of an imported class library. From a programmers point of view this leads to simplification and ease of maintenance.

3.2 Requirements

The existing AMOS II Java interface offers some functionality for manipulation of database objects by Java methods. Each time a programmer writes a Java program that will manipulate objects in an AMOS II database these difficulties exist:

- The existing interface is fairly complex and the Java programmer is required to have extensive knowledge about AMOSQL in order to be able to manipulate objects in the database.

- Surrogate objects retrieved from AMOS II are always represented using the generic class Oid. This representation of AMOS II data structures has the consequence that the structure of objects and relationships between objects are lost.

- It takes many lines of code for a programmer to accomplish easy tasks.

In order to solve the problems presented we developed some key requirements for our solution. It is important that the interface is easy to use and requires a minimum of AMOSQL knowledge. Objects retrieved from an AMOS II database should be

represented by semantically rich Java objects that in detail describe the properties of the object as it is specified in the database. The results of an AMOS II database query, executed through a Java method, should be handled in a homogeneous manner disregarding the type of the objects retrieved. To create new database objects, a standard Java constructor method should be used. Manipulation of attributes should be available using standardized object-oriented practice with get and set methods. It should be possible to add and remove values from multi-valued AMOS II functions and to delete a database object.

With the intention of fulfilling the requirements presented, problems arose during our discussions of the solution. The main issues addressed by our work are extraction of data for Java class definitions using meta-queries and handling of custom object proxies for specific types. These two issues can be managed with solutions to the following problems:

- Down-casting mechanism.
  When retrieving existing objects from an AMOS II database, the objects are always represented with the class Oid in the Java interface. For types more specific than Oid, e.g. Person, it is necessary to extend the functionality provided by Oid in order to take advantage of the existing features. All object proxies, also more specific ones, need the behaviour offered by the implementation in the AMOS II Java API. Therefore, objects retrieved from the database need a down-casting mechanism for translation of the Oids to a more specific type.

- Scan objects for a specific type.
  All data retrieved from AMOS II is in the Java API put together in a collection called a scan object. Scan objects can contain various kinds of data: e.g. strings, numbers and surrogate objects. Java programmers writing AMOS II applications are used to this practice and it would be desirable if a similar structure could be used. However, in order to retrieve semantically rich Java objects that in detail describe the properties of the object as it is specified in the database, a more specialized type of scan objects is needed.

- Easy access to the first value in a result set.
  The result of an AMOS II query often consists of a single data object. Therefore, a mechanism to easily extract the first element in the collection of results is desirable. Perhaps the most natural approach to this issue would be to return just one object of the current data type, e.g. a string (String in Java) or a number (int or double in Java). However, in order to maintain a consistent programming practice and treat all return values in the same way, even single-valued functions should be returned in a collection with a method for accessing the first element.

- Management of multi-valued functions using scan objects.
  If the result of an AMOS II function returns more than one value, it may be needed to access other values than the first value in the current result set as described above. The scan object has a method for advancing to the next position in the scan and a procedure for determining the end-of-scan state. This functionality is also needed in scan objects for a specific type.

- Handling of methods that cannot be derived to a specific type.
  A function can be defined in AMOS II to have more than one argument and more than one return type. In addition, AMOS II functions can be defined with no arguments. Functions with less or more than one argument cannot be naturally connected to any specific type and thus, a solution for managing those methods is needed.

- Management of null values in the result of a query. A possible result of an AMOS II query could be that no objects are returned. In these situations the result set cannot be accessed since there are no results to obtain. Some kind of programming construct is needed to handle null results.

3.3 Architecture

Amos Class Exporter is built on top of the existing AMOS II Java interface. It translates types and functions retrieved from AMOS II to a corresponding structure in Java. The Java files generated acts as a user-friendly interface to an AMOS II database, supposed to be used by programmers when building database applications. Figure 4 illustrates this functionality.

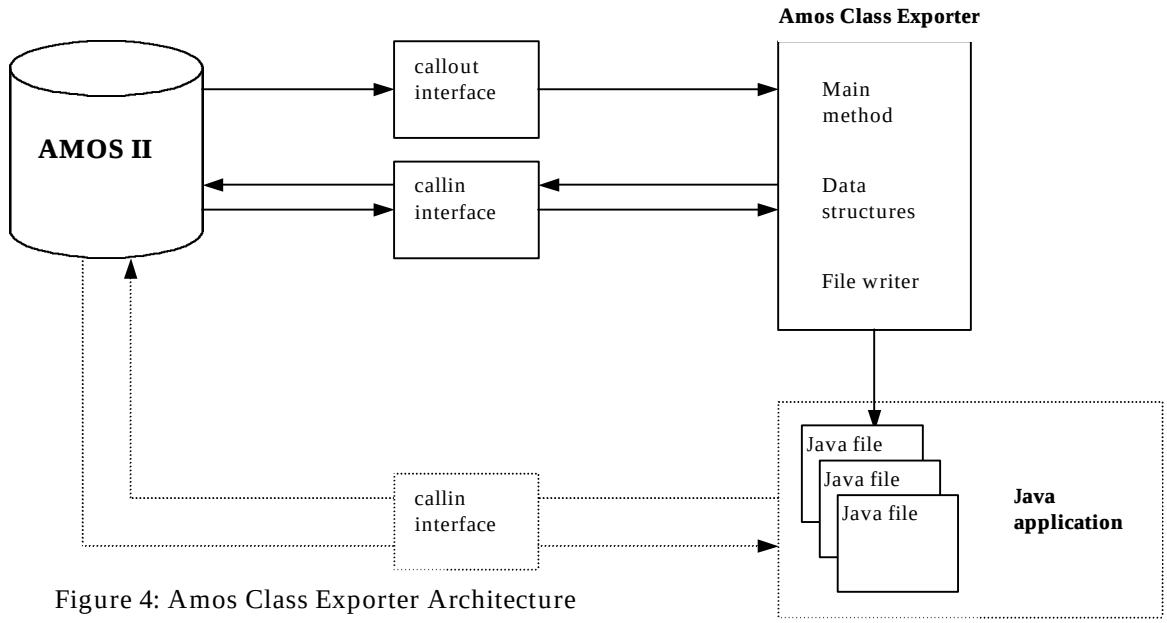


Figure 4: Amos Class Exporter Architecture

ACE is implemented as an AMOS II foreign function and is called by a user from AMOS II using a statement on the AMOSQL prompt. The types and functions a user wants to generate are supplied as parameters in the function call. If a foreign function called `generate_interfaces` has been previously defined, a call to ACE can be made with:

```
generate_interfaces({t1, t2}, {f1, f2, f3});
```

The arguments to the function must be of the AMOS II data type vector. In the above example a vector of the types `t1` and `t2` and a vector of the functions `f1`, `f2` and `f3` are supplied. The order of the arguments is important. ACE assumes that the first

parameter is the type vector and the second parameter is the function vector. Arguments sent to ACE can be constructed using AMOSQL query statements, making it possible to easily process all types and functions in an AMOS II database.

The program issues queries about the supplied types and functions in order to translate the database structure to a corresponding Java class structure. Queries are sent to the database using the AMOS II Java callin interface. Required data to build the Java structure for types is type name, associated functions and information about inheritance. Necessary information for construction of the functions is the function name, its parameters, return values and if it is possible to update the function in AMOS II.

Java source files are written to a file when all data structures have been created and populated with the information described above. Generated classes can be used when developing AMOS II database applications in Java. The source files includes methods for retrieving and manipulating persistent data. New database objects can be created using a constructor and a method is provided for deletion of objects.

3.4 Implementation

Amos Class Exporter extends the architecture of AMOS II. It is implemented using the AMOS II foreign function mechanism and is built on top of the existing AMOS II Java interface. ACE is completely built in Java and AMOSQL using Borland JBuilder 4 Professional Edition. JBuilder 4 is based on SUNs JDK 1.3. ACE is implemented with four classes as shown in figure 5.

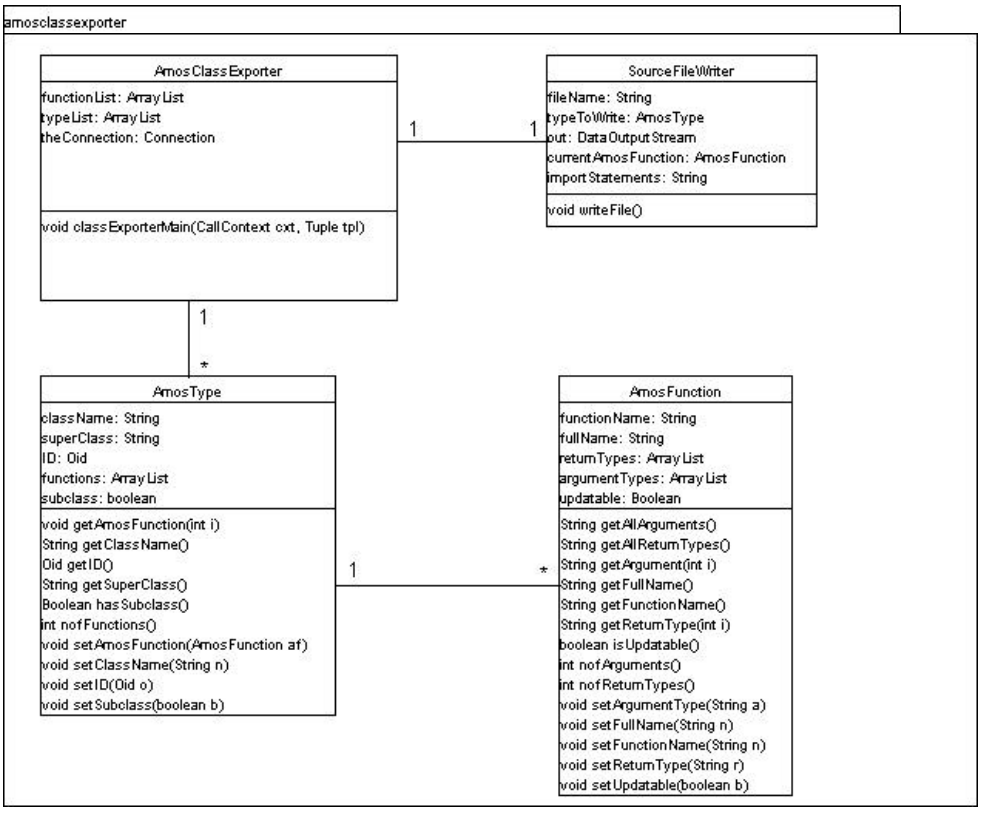


Figure 5: Class diagram of ACE

`AmosClassExporter` is the main class and contains the entry method, which is linked to AMOS II through the callout interface. This class retrieves the parameters sent from the database, creates data structures and populates them with data using meta-queries sent to AMOS II through the callin interface. Data about types and functions is stored in the classes `AmosType` and `AmosFunction` respectively. When all types and functions have been processed, `SourceFileWriter` generates Java source files for each type.

A call to a foreign function is always handled in the same manner by AMOS II independent of its implementation. For a foreign function written in Java, the method called has the following signature:

```
public void method_name(CallContext cxt, Tuple tpl)
```

AMOS II uses the `CallContext` object internally for managing the foreign function call and the parameter `tpl` is a `Tuple` object, which contains the arguments passed. The last position of this `Tuple` represents a possible return value. Thus, if a foreign function has two arguments and one return value the `Tuple` object has three positions, indexed from zero to two. [8]

ACE stores the vectors supplied as parameters in internal variables for further processing. This process is discussed in the next section.

3.5 How ACE works

This section illustrates with a step-by-step example how ACE works followed by a brief presentation on how the generated files are supposed to be used. The example is based on a database managing information about World Cup tournaments in soccer, taken from the AMOS II tutorial [19].

The schema is made up by seven types (entities): `Person`, `Referee`, `Player`, `Match`, `Team`, `Tournament` and `Country`. The entity-relationship schema of the database, pictured in figure 6, can be overviewed as follows. A country is host for a tournament a certain year. In each tournament there is a number of matches played, each between two teams participating in the specific tournament. A team represents a particular country and consists of a selected group of players from the team for each match. In a match, a team can make a number of goals. Players can make a number of goals in a tournament as well as a number of own goals in a certain match. Each match is refereed by some referee and has a number of spectators watching it. Referees and players inherit the properties from the abstract supertype person. All relations and attributes in the AMOS II database are represented as functions. How attributes and relations are defined for each type in this example is shown in table 1. [19]

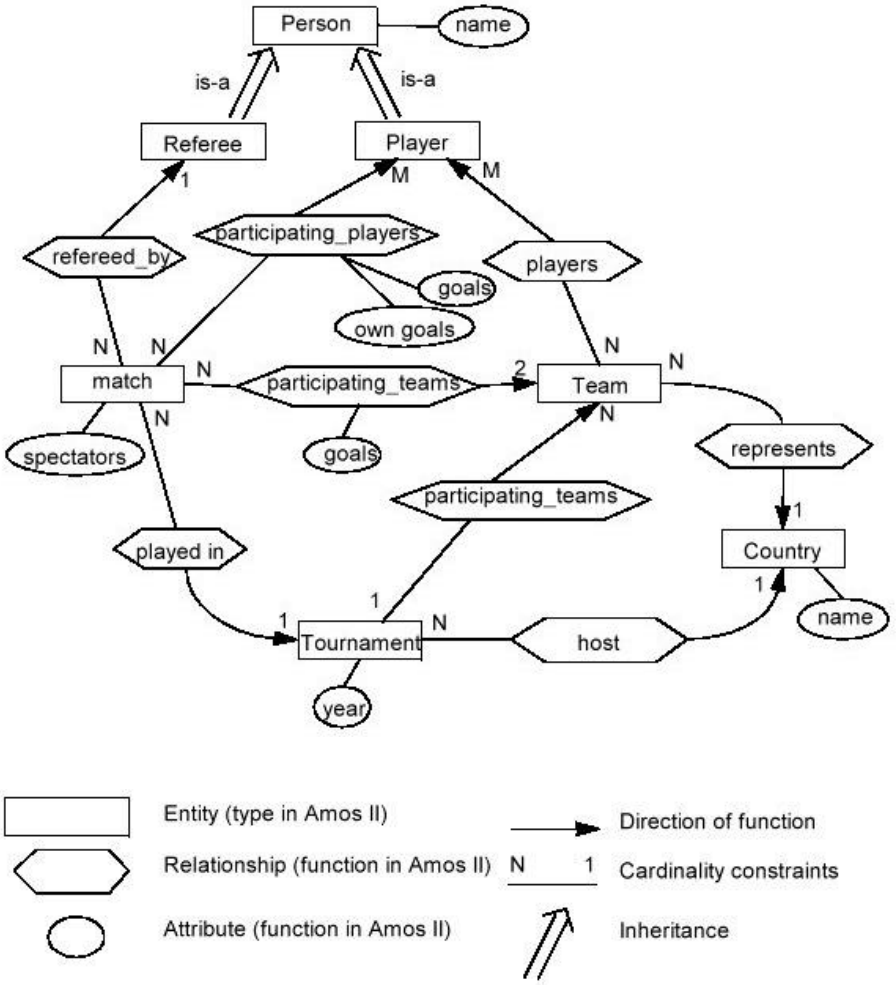


Figure 6: Extended entity-relationship schema, Soccer World Cup tournaments [19]

Types	Functions
TOURNAMENT	YEAR(TOURNAMENT) -> INTEGER
	HOST(TOURNAMENT) -> COUNTRY
	PARTICIPATING_TEAMS(TOURNAMENT) -> TEAMS
COUNTRY	NAME(COUNTRY) -> CHARSTRING
PERSON	NAME(PERSON) -> CHARSTRING
REFEREE (SUBTYPE OF PERSON)	
PLAYER (SUBTYPE OF PERSON)	GOALS(PLAYER, MATCH) -> INTEGER
	OWN_GOALS(PLAYER, MATCH) -> INTEGER
TEAM	REPRESENTS(TEAM) -> COUNTRY
	PLAYERS(TEAM) -> PLAYER
	GOALS(TEAM, MATCH) -> INTEGER
MATCH	REFEREED_BY(MATCH) -> REFEREE
	PLAYED_IN(MATCH) -> TOURNAMENT
	SPECTATORS(MATCH) ->INTEGER
	PLAYER_PARTICIPATIONS(MATCH) -> <PLAYER, INTEGER GOALS, INTEGER OWN_GOALS>

	PARTICIPATING_PLAYERS(MATCH) -> PLAYER
	TEAM_PARTICIPATIONS(MATCH) -> <TEAM, INTEGER GOALS>
	PARTICIPATING_TEAMS(MATCH M) -> TEAM
	GOALS(MATCH) -> INTEGER
	MATCHINFO(MATCH) -> <CHARSTRING C1, INTEGER G1, CHARSTRING C2, INTEGER G2, INTEGER Y>

Table 1: Definition of types and functions in the World Cup example [19]

The following sections provide a detailed description of the process when all types and functions in this example are sent to ACE. We have divided the processing made by the program into three logical phases: the receiving phase, the information-gathering phase and the write phase.

3.5.1 Receiving phase

ACE is, as mentioned previously, implemented as a foreign function written in Java. In order to call the program, the main method of ACE must be defined in the database. This procedure is referred to as creating a resolvent for a foreign function [8]. We assume that there is a function linked to ACE called `generate_interfaces` defined as below:

```
create function generate_interfaces(vector t, vector f) -> integer as foreign
“JAVA:AmosClassExporter/classExporterMain”;
```

In the call to ACE we supply all types and functions from the World Cup tournaments in soccer database. The call should be constructed as:

```
generate_interfaces({type person, type player…}, {function 1, function 2…});
```

The vectors supplied as parameters are retrieved by the `classExporterMain` method in a `Tuple` object. ACE checks if the arguments are valid, and copies the arguments into two internal variables, containing OIDs for types and functions respectively.

3.5.2 Information-gathering phase

The type and function vectors are now stored in internal variables and the information gathering concerning the data structures in the database can begin. This is done through issuing AMOSQL queries on meta-data, i.e. querying the schema by calling system functions in the AMOS II database. The first thing to be constructed by ACE is the functions sent from AMOS II. The program iterates through the function OIDs and issues database queries about each functions name, if the function can be updated, the number of arguments, return values and their respective data types. Gathered information for each function is stored as an instance of the class `AmosFunction`, an ACE data structure holding all necessary information of an AMOS II function. These data are

later used to build the Java structure. The code example below shows how ACE collects information about the data types for arguments and returns values of an AMOS II function.

```
Scan argResTypes;  //Results of the 'argrestypes' query.
Oid functionOid; //Oid used temporarily when processing functions.
AmosFunction af = new AmosFunction(); //New instance of the class AmosFunction
...
01 argResTypes = theConnection.callFunction("FUNCTION.ARGRESTYPES->
02 INTEGER.TYPE.INTEGER", functionOid);
03 while(!argResTypes.eos()){
04    if(argResTypes.getRow().getIntElem(2) == 0){
05          af.setArgumentType(convertType(argResTypes.getRow().getOidElem(1).
06          getName()));
07    }
08    else{
09          af.setReturnType(convertType(argResTypes.getRow().getOidElem(1).
10          getName()));
11    }
12    argResTypes.nextRow();
13 }
```

On the two first rows the actual database query is executed. All database queries in ACE are issued using the callFunction method in the fast-path interface (see section 2.3.2). The first argument in this example is the function to be called in AMOS II, while the second argument represents the argument to be passed to the AMOS II function. In this case the system function `AGRESTYPES` is used on a function OID to get the corresponding data types. The result of the call on the first two rows is a `Scan` object, which in turn contains one or several `Tuple` objects, each representing a data type string of an argument or a return value. In order to iterate through all the records a while loop is used. If the value of the number in position 2 in the current row (tuple) is zero, the tuple holds a data type of an argument, and if the number is one, it holds a data type of a result value. A method for converting AMOS II data types to Java data types has been implemented. The `convertType` method is used for this purpose. It provides conversion between the following data types:

AMOS II data type	*Java data type*
CHARSTRING	String
INTEGER	int
REAL	double
BOOLEAN	boolean
SURROGATE_OBJECTS (e.g. PERSON)	Surrogate_objects (e.g. Person)

After data type conversion has been made, the data types of the argument and the return type are added to the current instance of the `AmosFunction` class respectively. This is implemented using the `setArgumentType` and `setReturnType` method calls.

When the information needed to describe a function has been collected, the instance of the class `AmosFunction` is added to a list for subsequent matching between the function and the type it belongs to.

The next step in the information-gathering phase is processing the types sent to ACE. ACE iterates through all the types, creates an `AmosType` object for each type and collects

the information needed from the data structures in the database. AmosType is an ACE data structure holding necessary information of an AMOS II type used to build the Java class structure. Information stored for each type in AMOS II is the name of the class (type in AMOS II), the type OID, the name of the superclass, if the class has a subclass or not and the functions that belong to this class. An instance of the AmosType class with the name Common is created for the purpose of holding information about AMOS II functions that could not be derived to any specific type.

When a new instance of the AmosType class has been created, the previously populated list of objects from the AmosFunction class is iterated through. If a function has exactly one argument and the argument name is equivalent to the class name of the AmosType instance, the function is connected to that object. Functions with less or more than one argument are added to the instance with the class name Common. For example, the AMOS II function NAME(PERSON) -> CHARSTRING is linked to the AmosType instance with the class name Person, while the function GOALS(PLAYER, MATCH) -> INTEGER is linked to Common. An excerpt of the code for the actual connection of a function to an AmosType object is presented below.

```
at = new AmosType();
…
at.setAmosFunction(currentFunction);


public class AmosType {

     private ArrayList functions = new ArrayList(); //Functions of this class.

     public void setAmosFunction(AmosFunction af){
        functions.add(af);
     }
```

When a function has been connected to an AmosType object, the function is removed from the list of functions. The remaining functions, i.e. functions still in the list when all types have been iterated through, are linked to the Common instance. This approach also helps to improve performance since the number of iterations is reduced.

In order to handle inheritance and build the corresponding class hierarchy in Java, it is necessary to get the name of the nearest supertype for each type. It is also required to identify if the type has a subtype or not. This is implemented using a call to the AMOS II system function SUBTYPES with a type OID as parameter. If the Scan returned from this call is null, there is no subclass for that type:

```
if(theConnection.callFunction("TYPE.SUBTYPES->TYPE", at.getID()).eos()){
        at.setSubclass(false);
}
```

The code for obtaining the nearest supertype of a type is a somewhat more complicated algorithm, since it is not the nearest supertype in the database we ask for, rather, it is the nearest supertype in the collection of parameters sent to ACE. To get the correct supertype the AMOS II system function ALLSUPERTYPES is called on each type. This function returns a Scan object containing all the types above a type (i.e. all its supertypes), with the nearest supertype in the first Tuple. The result set is iterated

through and compared to the list of types sent to ACE. If no corresponding supertype is found in the list, the type is set to inherit from `Oid`. Functionality offered by the class Oid is necessary for all object proxies. Therefore, all generated classes inherit directly or indirectly from the class Oid. In the soccer tournament example, all types have `Oid` as their direct supertype, except `Player` and `Referee`, which inherit properties from the supertype `Person`. Another exception is the `Common` class, which has no inheritance.

3.5.3 Write phase

The last phase in the process is the creation of the Java source files for all the classes. The source files are written one by one to disk by the `SourceFileWriter` class. For each type supplied to ACE, one source file is created. By default, source files are written to the working directory of the current database. The generated Java source files contain import statements, constructors, get methods, set methods, a method for deleting objects and sometimes additional methods for add and remove operations. This varies according to how the type was specified in the database. Under the following headings a more detailed description of the source files is given.

Import statements and connection object
Import statements for the AMOS II callin interface and a static instance variable for the `Connection` object are inserted into all source files. All generated classes need to import the callin package, which is used in all methods for handling database calls. A `Connection` object in each class must be initialized when a new instance of that class is created.

Constructors
In order to manage inheritance, several different types and implementations of constructors exist. The class `Common` differs from all the other classes, as it does not inherit any properties from the class `Oid`. The `Common` class has only one constructor, which takes a `Connection` object as a parameter, which is used to initialize the instance, variable in this class. Each class except `Common` implements an empty constructor, used when retrieving existing objects from the database. Other kinds of constructors are illustrated below. Examples are taken from the soccer tournament database.

Class directly extending Oid:

```
public Person(Connection c) throws AmosException{
      super(c.getType("Person"));
      con = c;
}
```

The method call to `getType` retrieves an object proxy for the specified type, which is sent to the constructor of the Oid class in the AMOS II Java interface. With this construct, an object of the specified type is created in the current database.

Class with nearest superclass other than Oid:

```
public Player(Connection c) throws AmosException{
      super(c, c.getType("Player"));
```

```
      con = c;
}
```

Class with subclass and with Oid as the nearest superclass:

```
public Person(Connection c, Oid o) throws AmosException{
/* Constructor called from a subclass */
      super(o);
      con = c;
}
```

Class with subclass and with class other than Oid as the nearest superclass:

```
public Person(Connection c, Oid o) throws AmosException{
/* Constructor called from a subclass */
      super(c, o);
      con = c;
}
```

Retrieving database objects.
Each class implements a method for retrieving existing database objects, which takes an integer (the OID of the requested object) and a connection object as parameters. This method is implemented using a method in the AMOS II Java interface called `CopyProps`. This method clones the database object retrieved using the callin `execute` method and can be described as down-casting to a more specific type than the OID retrieved. All the necessary properties of the current object proxy are copied to the more specific object. The following lines of code illustrates an example:

```
public static Person getPersonElem(int i, Connection c) throws AmosException{
      Person retType = new Person();
      con = c;
      Oid o = con.execute("select #[OID " + i + "];").getRow().getOidElem(0);
      retType.CopyProps(o);
      return retType;
}
```

A class with a subclass, i.e. a superclass, also implements a method called `downCastConnect` (see below). This method is called from the subclass in order to initialize the `Connection` object in the superclass, which is the case when an existing subclass object is retrieved from the database. If the `Connection` object is not initialized, inherited methods in the subclass containing database calls will fail.

```
public static void downCastConnect(Connection c){
      con = c;
}
```

Thus, in a subclass to `Person`, e.g. `Player`, the following line is included in the `getPlayerElem` method:

```
Person.downCastConnect(c);
```

Methods for manipulation of database objects
For each AMOS II function, a maximum of four different methods are implemented in the Java class. For functions not defined as stored in the database, e.g. derived functions,

only a get method is written. The following methods are implemented for the stored AMOS II function `PLAYERS(TEAM) -> PLAYER` in type `Team`:

```
public PlayerScan getPlayers_() throws AmosException{
      return new PlayerScan(con.callFunction("TEAM.PLAYERS->PLAYER", this),
      con);}
```

Get methods are generated for all AMOS II functions. In this example, the get method retrieves the players of a `Team` object. The get methods can be viewed as a more natural way of querying an AMOS II system. A call to a get method on a specific instance of a class retrieves the same information as the corresponding AMOSQL query. The object returned from this operation, described by a result class called `PlayerScan`, is explained and further discussed in a section ahead.

```
public void setPlayers(Player a0) throws AmosException{

      Tuple arg = new Tuple(1);
      Tuple res = new Tuple(1);

      arg.setElem(0,this);
      res.setElem(0,a0);
      con.setFunction("Players",arg,res);
}
```

Set methods are generated for AMOS II functions that can be updated in the database. That is, functions defined as stored functions in the database schema. A set method can have a various number of arguments depending on the implementation of the AMOS II function. Arguments are named in a sequence, beginning with the name `a0` and with a greater number for each following argument. A set function is used for changing set-valued attributes of a type or updating relations between types.

```
public void addPlayers(Player a0) throws AmosException{

      Tuple arg = new Tuple(1);
      Tuple res = new Tuple(1);

      arg.setElem(0,this);
      res.setElem(0,a0);
      con.addFunction("Players",arg,res);
}
```

Methods with an add prefix in their name are used for add operations on a specific database object. A function in AMOS II can be defined to hold more than one value. Data are added to such functions with the generated add methods. For all functions defined as stored in the database an add method is generated. The method in the above example adds a player to a team. A team can have more than one player.

```
public void removePlayers(Player a0) throws AmosException{

      Tuple arg = new Tuple(1);
      Tuple res = new Tuple(1);

      arg.setElem(0,this);
      res.setElem(0,a0);
      con.remFunction("Players",arg,res);
}
```

The remove methods are the opposite of the add methods and they are used for removing values from a multi-valued function. Remove methods are generated for stored functions.

```
public void delete() throws AmosException{
      con.deleteObject(this);
}
```

For each class, except for the class `Common`, a delete method is generated. The `Common` class has no delete method since there is no representation of this object stored in the database schema. The delete method is implemented in the same way in all classes. It uses a method called `deleteObject` that deletes the object from the database.

Result classes

The get methods return different objects from the database depending on how they were defined. For each type of object returned a corresponding result class is implemented. This approach is taken in order to handle multiple results. A function returning a `CHARSTRING` in AMOS II could possibly return more than one value. In that case, a `StringScan` object is returned in Java. The `StringScan` result class is presented below:

```
import callin.*;

public class StringScan{

      private Scan theScan;
      private Connection con;

      public StringScan(Scan s, Connection c){

            theScan = s;
            con = c;
      }

      public String current() throws AmosException{

            if(!theScan.eos()){
                  return theScan.getRow().getStringElem(0);
            }
            return null;
      }

      public void nextRow() throws AmosException{

            theScan.nextRow();
      }

      public boolean eos(){

            if(theScan.eos()){
                  return true;
            }
            return false;
      }
}
```

A result class implementation has two instance variables. A `Scan` is used for managing the data retrieved from the database and a `Connection` object handles the current connection to the database where the queried objects exist.

Result classes can be viewed as a more specific implementation of the `Scan` object in the AMOS II Java interface. The underlying idea in the design of these result classes was to provide a unified way of accessing AMOS II data, irrespective of the actual data type returned. There is no difference in accessing string elements, soccer player elements or integers. The first element returned is always accessible using the `current` method.

A possible result of an AMOS II query is that no objects are returned. In this case, the Scan object internal to the result class, is empty. If the result set is empty, null is returned.

The specialized scan objects implement methods similar to the methods in the AMOS II Java API. If more than one value is returned from the database, the result scan contains multiple values. Values following the first value can be accessed with the `nextRow` method to advance one position forward in the scan followed by a call to `current`.

3.6 Usage of generated files

The following code examples gives a short illustration of how the generated classes can be used. `Tournament`, `Player` and `Person` are Java classes generated by ACE.

- Retrieval of existing database objects

The code below picks up existing objects from a saved database and operations are carried out on the objects.

```
Tournament t = Tournament.getTournamentElem(636, con);

TeamScan ts = t.getParticipating_teams_();
while(!ts.eos()){
      System.out.println("Test of getName on Player returned " +
      ts.current().getPlayers_().current().getName_().current());
      ts.nextRow();
}
```

- Creation of new database objects

Create a new `Player` object and set the name:

```
Player p1 = new Player(con);
p1.setName("Johan");
System.out.println("Name of p1 was set to " + p1.getName_().current());
```

100 persons are created and names are assigned:

```
Person pa[] = new Person[100];
String s = new String();
```

```
for (int i = 0;i < 100 ;i++ ) {
      pa[i] = new Person(con);
      pa[i].setName("Person" + (i + 1));
      s = s + pa[i].getName_().current() + " ";
}
System.out.println(s);
```

4 Interfaces compared to ACE

In this chapter, we compare our work with the existing AMOS II Java interface. Furthermore, differences between ACE and other binding solutions for the Java language, e.g. Java Blend, are discussed.

When comparing solutions for database programming interfaces one important aspect to discuss is the difference between a dynamic and a static interface. Both types of interfaces have advantages and disadvantages. A dynamic interface is more convenient to the user, due to its ability to react to changes in the current environment. This may have performance implications though, since a dynamic interface has to keep track of changes in e.g. a database schema. In a static environment, or at least in an environment where changes seldom are made, a static solution provides the same convenience to the developer. Performance of such an approach is more likely to be higher than a dynamic interface in a static environment. In environments that continually change, dynamic interfaces are more appropriate. A static solution in an often-changing environment means more work for the application programmer.

A dynamic interface adapts to the current situation with no intervention from the developer. For example, if the database schema is changed in the current session, a dynamic interface would adapt to the changes transparently, providing an interface to the latest version of the schema definition in the database. With a static interface, on the other hand, some kind of action needs to be taken by the programmer when changes occur or when the interface is used. One kind of intervention could be compilation of generated source files that constitute the interface. This approach is taken in our solution. When developing a completely dynamical interface, further investigation of different issues is needed. Due to time constraints, we chose not to investigate issues related to dynamic interfaces.

4.1 Differences between the AMOS II Java interface and ACE

Necessary in this context is a comparison with the existing AMOS II Java interface and ACE. In the underlying interface, upon which ACE is built, it is to some extent possible to manipulate database objects by Java methods. The functionality provided by the AMOS II Java interface is on a low level. Manipulation of database objects requires knowledge of functions associated with the current type. The existing interface also represents all surrogate objects as the generic class Oid, which leads to the loss of structure of data and relationships between objects. Our solution generates a complete set of source files for a type hierarchy in an AMOS II database, which can be used in the development of AMOS II database applications.

For each AMOS II function, get, set, add and remove methods are created. Methods for data modification, i.e. methods other than get methods, are generated only if necessary conditions are met. This provides a natural way of modifying persistent object-relational data through a simple and generic object-oriented interface. Relations between objects and inheritance are transferred to an equivalent structure represented in the Java language.

In Java, methods can be applied to different objects in a long sequence. If a method returns an object, a new operation (method) can be carried out on that object which in turn returns another object etc. With ACE, this brings new possibilities in accessing and modifying data in an AMOS II database. In fact, it provides a new approach of working with persistent data.

Applying Java methods in a sequence as described above, is the same thing as querying, or modifying, specific AMOS II database objects. The following example illustrates this functionality. It shows how to obtain the name of the referee for a specific match. The variable `con` is a `Connection` object, `mOid` and `rOid` are `Oids` for a match and a referee respectively and `m` is a reference to a `Match` object (see the soccer example).

With the AMOS II Java interface:

```
rOid = con.callFunction("MATCH.REFEREED_BY->REFEREE",
mOid).getRow().getOidElem(0);

String s = con.callFunction("PERSON.NAME->CHARSTRING",
rOid).getRow().getStringElem(0);
```

or

```
String s = con.callFunction("PERSON.NAME->CHARSTRING",
con.callFunction("MATCH.REFEREED_BY->REFEREE",
mOid).getRow().getOidElem(0)).getRow().getStringElem(0);
```

With ACE:

```
String s = m.getRefereed_by_().current().getName_().current();
```

In the above example the simplicity of ACE is clearly illustrated. Of course there is the intermediate step with creating the interface for the specific type but once this minor process is finished programming tasks are simplified. With the AMOS II Java interface, it is necessary for the user to keep track of the distinguished names of the AMOS II functions. This is unnecessary with ACE since the database calls, queries etc, constitute the implementation of methods in the generated class.

When creating new database objects from Java, ACE adds additional features not found in the AMOS II Java interface. Creation of database objects for a specific type is as simple as calling a standard Java constructor. With this construct, it is possible to create a large amount of database objects easily.

4.2 Comparison of ACE to Java binding solutions

ACE is in some ways similar to products such as Java Blend when comparing fundamental principles. For example, they both bridge the gap between a database schema and a programming language, in this case Java. ACE also differs from the other solutions in various ways. In this section, similarities and differences between related

technologies and ACE are discussed. Since the ODMG Java Binding and JDO are standards describing necessary properties for a system that handles object persistence, Java Blend is used as the primary frame of reference in this comparison.

Java Blend is an implementation of the ODMG Java Binding, which is also used as underlying principles of JDO. Java Blend is a development tool and a runtime intended for programmers writing Java applications that need to access data stored in relational databases. This is one of the most notable differences between ACE and Java Blend. Our solution is designed for an AMOS II object-relational database and handles the mapping needed between Java objects and objects in the database.

The perspective of the solution differs between products such as Java Blend and ACE. In our approach, necessary data is extracted from an existing database schema for creation of higher level Java interfaces using meta-data queries. Therefore, it is a prerequisite for ACE to work that a schema already has been defined. In Java Blend, the perspective is slightly different. With that product, it is also possible to create a database schema, although relational, from an existing Java hierarchy. Thus, our solution has a database-oriented perspective and solutions such as Java Blend are programming language-oriented.

Both ACE and Java Blend are layered on top of a call-level interface. ACE utilizes the AMOS II Java API and Java Blend relies on JDBC. Those lower level interfaces handle the database connectivity in both solutions. ACE is closer to a call-level interface than Java Blend. In ACE, methods of generated classes make direct calls to the underlying AMOS II database.

Java Blend is perhaps a more complete solution, especially in dynamic environments. It utilizes a caching mechanism as an intermediate layer between the Java application and the database, which takes necessary measures when data changes either in the database or in the Java application. Java Blend has support for concurrency, transaction control and schema manipulation from Java, functionality which is not implemented in ACE.

Our solution can take advantage of all the features that exist in an AMOS II system. With products based on relational technology, complex data structures are hard to model in a natural fashion. In an AMOS II database, it is easier to model real world entities more realistically. Combined with the mediator functionality in an AMOS II system, this brings interesting advantages to an interface such as ACE. Complex systems can be realized using the core features of object modeling and mediation in AMOS II and then interfaces can be generated to easily access and process the data retrieved by those applications.

The user interface of solutions similar to Java Blend is another point of view that can be taken when comparing it to ACE. Most of the commercial products, Java Blend and others previously mentioned, are based on a graphical user interface with which the user interacts. In these interfaces, the user selects for what entities mapping should be executed. At a glance, the user interface of ACE may seem simple, but a more thorough investigation reveals the advantages of this solution. Its tight integration with AMOSQL provides the user with the powerful language constructs available in this language. It is

easy for a proficient user of AMOSQL to write nested query statements that generates interfaces for selected types and functions. Seen from a performance perspective, command line user interfaces are often more effective, although they can be hard to understand at first.

5 Conclusion and future work

ACE is layered on top of the existing AMOS II Java interface and provides transparent integration of Java language classes with an AMOS II database. It handles mapping between a database schema and Java classes.

Our solution offers an easier way of writing AMOS II database applications in Java, as developers can write Java programs to retrieve data from an AMOS II database as semantically rich Java objects. These objects can be modified, which in turn modify the database objects. Modifying persistent data is therefore as simple as modifying any other Java data.

In the generated files, query results are handled in an analogous way. Retrieval of information on the database objects is accomplished using methods in the Java classes, since the database queries are coded in the Java language as part of the class. The use of methods in the generated Java classes can be viewed as a way of submitting queries to the database. The programming productivity increases by eliminating most of the need to write AMOSQL or understand the details of the database schema. It also reduces development time by eliminating the need to write, test and debug AMOS II Java interface code. An increased quality in future development of AMOS II database applications is supplied, since common coding mistakes can be avoided using automatic generation of Java source code. These improvements provide significant advantages over previous database programming solutions with the AMOS II Java interface.

In our work we have addressed problems related to the extraction of necessary information to build Java source files from an AMOS II database using meta-queries. This has concluded in a new approach on how to handle and use custom object proxies for dissimilar types defined in an AMOS II database. We have presented solutions for different problems that we have identified as related to the investigation of this issue. In this thesis, we have implied solutions for the following:

- Scan objects for a specific type. Previous to ACE, surrogate objects retrieved from an AMOS II database were represented using the generic class Oid. With classes generated by ACE, it is possible to represent these objects in a more natural way. Our solution is assembled by specialized scan objects, with methods for retrieving a materialization of objects as they were defined in the database.

- Mechanism for casting types to a more specific type. We identified the need for a programming construct to cast a type to a more specific one, usually referred to as down-casting. This method is used in the scan objects for a specific type and when existing objects are retrieved from an AMOS II database. For this purpose, the AMOS II Java interface was extended with a method called `CopyProps`, which clones properties of object proxies.

- Efficient handling of multi-valued functions. In the AMOS II Java interface, which ACE is built on top of, support for retrieving data in a collection object (scan) already existed. Hence, we found in our exploration of this issue that a proper solution to managing multi-valued functions would be to extend the functionality offered in

AMOS II Java interface. We implemented functionality for handling multiple values in each of the result classes for specific types. As in the scan objects, it is possible to iterate through an entire collection of objects with the use of the methods in the specialized scans.

- Easy access to the first element in a result set. Our solution to this topic was to implement support for effortless retrieval of the first element in a set of results gathered from an AMOS II database. To be more specific, we proposed a standardized way of accessing this element through a Java method called `current`, implemented in each of the result classes unique for each type generated by ACE.

- Management of methods that cannot be derived to a specific type. In the early stage of our investigation, we discovered a need for handling of methods that could not be derived to a type. We define methods that can be derived to a specific type as methods with exactly one argument. A method with less or more than one argument has to be placed in a class other than the files for the specific types. For that purpose, a class called `Common` is implemented.

- Null values handling. In the result classes for a specific type, the `current` method provides easy access to the first element. If no elements were returned from the database, an error would occur when trying to access these elements. We identified a simple solution to this problem. If the result of a query contains no objects, null is returned.

Several opportunities in improving the AMOS II Java interface exist. Consequently, we have identified some future directions in which our work could be continued. A major area in where our solution needs completion is the issue of concurrency and transaction control. In coming extensions to the AMOS II Java interface, possibly built with ACE as a basis, generated Java classes need mechanisms for managing transactions and concurrency problems.

The Java language does not support multiple inheritance, a characteristic supported by AMOS II type hierarchies. ACE handles this issue by ignoring the problem; the first supertype found in the hierarchy is used for inheritance. Further investigation of how this question can be addressed in a more complete way is needed. We suggest an analysis of how the interface construct in the Java language can be used to solve this problem.

A more practical and probably simpler issue, is the addition of Java Archive (JAR) features to ACE. When generating Java source files for extensive type hierarchies, a great number of files are created. Except one file for each supplied type, several files are generated for describing the result sets of the different types. Thus, an elegant solution would be the option to bundle created files in a JAR package. This requires an analysis of the order in which the generated files should be compiled. When the logical order has been determined, an implementation of a JAR archiving function should be reasonably simple.

In AMOS II, there are currently no methods available for determining if a function is defined as multi-valued or single-valued. For functions that can have only one value, Java methods for adding and removing values from the function is generated by ACE, even if this functionality is purposeless from a database perspective. We believe that this issue is easiest to address with an AMOS II system function, which allows for a user to decide if a function allows multiple values.

References

[1] Azavant, F, Sahuguet, A: “World Wide Web Wrapper Factory (W4F) User Manual”, University of Pennsylvania, USA, 2000

[2] Barry, D, Stanienda, T: “Solving the Java Object Storage Problem”, IEEE Computer, Vol. 31, No. 11, November 1998

[3] Centre for Objekt Teknology (COT): “Database Management Systems: Relational, Object-Relational, and Object-Oriented Data Models”, COT/4-02-V1.1, Denmark, 1998

[4] Committee on Innovations in Computing and Communications: “Funding a revolution: Government support for computing research”, Lessons from History, National Research Council, 1999

[5] Codd, E. F: “A Relational Model of Data for Large Shared Data Banks”, Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387

[6] Connolly, T, Begg, C, Strachan, A: “Database Systems: A Practical Approach to Design, Implementation and Management”, Addison-Wesley, Great Britain, 1996

[7] Eckel, B: “Thinking in Java”, Prentice-Hall, USA, 2000

[8] Elin, D, Risch, T: “AMOS II Java Interfaces“ , UDBL, Uppsala University, Sweden, August 2000

[9] Elmasri, R, Navathe, S.B: “Fundamentals of database systems”, Addison-Wesley, USA, 1997

[10] Flodin, S, Josifovski, V, Katchaounov, T, Risch, T, Sköld, M, Werner, M: ”AMOS II Users Manual”, UDBL, Uppsala University, Sweden, 2000

[11] Jordan, D: “An overview of Suns Java Data Objects specification”, Java Report, June 2000

[12] Josifovski, V, Risch, T: ”Comparison of AMOS II with Other Data Integration Projects”, Technical Report, EDSLAB, Linköping University, 1999, ”http://www.ida.liu.se/~edslab/amosII_comp.pdf

[13] McClure, S: “Object Database vs. Object-Relational Databases”, IDC Bulletin #14821E, 1997

[14] Melton, J, Michels, J, Josifovski, V, Kulkarni, K, Schwarz, P, Zeidenstein, K: “SQL and Management of External Data”, SIGMOD Record, Vol. 30, No. 1, March 2001

[15] Microsoft: “ODBC Programmers Reference”, http://msdn.microsoft.com/library/psdk/dasdk/odin8w4s.htm, 2001-04-29

[16] ODMG: “The Object Data Standard: ODMG 3.0”, Morgan Kaufmann Publishers, USA, 2000

[17] Risch, T: “AMOS II External Interfaces“ , UDBL, Uppsala University, Sweden, February 2000

[18] Risch, T: ”AMOS II White Paper”,
http://www.dis.uu.se/~udbl/amos/amoswhite.html, 2001-01-10

[19] Risch, T, Fahl, G: “AMOS II Introduction”, UDBL, Uppsala University, Sweden, October 1999

[20] Risch, T, Josifovski, V: ”Distributed Data integration by Object-Oriented Mediator Servers”, To be published in “Concurrency – Practice and Experience”, J Wiley & Sons, 2001

[21] Risch, T, Josifovski, V: ”Distributed Mediation using a Light-Weight OODBMS” , UDBL, Uppsala University, Sweden, 1999

[22] Risch, T, Josifovski, V, Katchaounov, T: ”AMOS II Concepts”, UDBL, Uppsala University, Sweden, 2000

[23] Sun Microsystems: ”Java Blend Overview”,
http://www.sun.com/software/javablend/overview.html, 2001-04-29

[24] Sun Microsystems: ”Java Blend White Paper”,
http://www.sun.com/software/javablend/whitepapers/index.html, 2001-03-20

[25] Sun Microsystems: “JDBC Datasheet”,
http://java.sun.com/products/jdbc/datasheet.html, 2001-04-22

[26] Wiederhold, G: “Mediators in the Architecture of Future Information Systems”, Stanford University, USA, 1991