IEEE TRANSACTIONS ON

KNOWLEDGE AND DATA ENGINEERING

A publication of the IEEE Computer Society

MARCH / APRIL 1999 VOLUME 11 NUMBER 2 ITKEEH (ISSN 1041-4347)

REGULAR PAPERS

CORRESPONDENCE

Dr. Naphtali Rishe
Florida International University
School of Computer Science
Southwest 8th St. and 107th Avenue
University Park
Miami, FL 33199

IEEE COMPUTER SOCIETY

http://computer.org • tkde@computer.org

Proceedings

Technology of Object-Oriented Languages and Systems

TOOLS 30

August 1-5, 1999

Santa Barbara, California

Edited by

Donald Firesmith, Richard Riehle, Gilda Pour, and Bertrand Meyer

Sponsored by

Interactive Software Engineering, Inc.

Los Alamitos, California

Washington • Brussels • Tokyo

Storing Java Objects in any Database

Raimund K. Ege
High Performance Database Research Center *
School of Computer Science, Florida International University
University Park, Miami, FL 33199
ege@cs.fiu.edu

Abstract

Typical Java applications involve access to a database system. Database systems store data according to their type system, even object-oriented databases generally have their own storage structures. It is therefore necessary to convert Java objects as they are stored in a database, and to re-convert them when they are read. Ideally, this should be done behind the scenes by a support package. Our paper presents an approach that automates the conversion process without involving pre- or post-processing of Java code: we use a reflection mechanism, where Java code is inspected at run-time and changed to include the convert and re-convert effort. The result is a flexible and transparent Java database access.

1. Introduction

Accessing a database system from the Java programming language is a common occurrence. For typical database systems, i.e. those based on relational technology, there are several packages available that enable Java to store and retrieve data. For object-oriented database systems it is desirable to allow Java to store and retrieve its objects directly, i.e. in a seamless manner. Several approaches to the integration of a Java program's object model with an object-oriented database have been proposed and demonstrated. Some advocate a strong and strict integration, some allow loose coupling. Either approach has advantages and disadvantages.

In this paper we will describe an API based on a novel approach that allows Java objects to be maintained by an object-oriented database system. Our approach employs a reflection mechanism that allows for an application program to explore itself and the database schema: it is neither strict nor excessively loose. We use reflection, i.e. self inspection and change of a running program, to map Java classes to database classes and to enable persistence. Reflection allows a program to inspect itself and to affect its own execution. In an object-oriented program that means that at run-time the class of an object is accessible, in our case, the classes of the Java application program as well as the classes of the database API.

And unlike other approaches that add persistence capabilities to Java our approach does not change the Java language nor does it require changes to the Java Virtual machine. The result is a very flexible and efficient style of Java database access.

This research was supported in part by NASA (under grants NAGW-4080, NAG5-5095, NAS5-97222, and NAG5-6830), NSF (CDA-9711582, IRI-9409661, HRD-9707076, and ANI-9876409), ARO (DAAH04-96-1-0049 and DAAH04-96-1-0278), AFRL (F30602-98-C-0037), BMDO (F49620-98-1-0130 and DAAH04-0024) DoI (CA-5280-4-9044), and State of Florida.

0-7695-0278-4/99 $10.00 © 1999 IEEE

This paper is organized as follows: Section 2 explores the issue of adding persistence to Java objects; Section 3 discusses other efforts that are related to our approach; Section 4 illustrates our reflection-based approach of adding persistence to Java objects and Java classes; Section 5 elaborates and gives examples of common classes, and gives an example of a program; Section 6 briefly outlines the implementation approach and reports on the status of our prototype [Ege et al., 1998a] which is based on SemODB, a semantic object-oriented database system [Rishe, 1992]. Finally, the paper concludes with our future plans.

2. Java Persistence

Both object-oriented programming languages and object-oriented database systems have been in use for some time. And from the early days there was the desire to combine both in a true object-oriented fashion. The programming language uses objects in its model of computation; the database stores objects: therefore, an application programmer interface (API) necessarily should be constructed around objects. Objects occur in two places: (1) the objects stored and retrieved from and to the database, and (2) service objects, that enable the typical database management capabilities, such as transactions etc.

It is classes that describe objects: the schema of the database is just a set of classes an OO program is also just a set of classes. Several approaches to their integration have emerged: from a loose integration, exemplified by the GemStone OODB for Smalltalk, to the ODMG[Catell et al., 1997] C++ binding, which represents an integration compromise to tight integration that is proposed for storing Java objects. In a tight integration, Java program objects can be made *persistent*, which causes them to be implicitly stored and managed by an OO database.

The issues involved in allowing Java objects to be persistent are (1) the selection/marking of classes that have persistent instances; (2) the creation of persistent objects; (3) the retrieval of objects from the database; (4) navigation from object to object; (5) updating objects; and also (6) some basic database housekeeping capabilities.

2.1. Persistence-Capable Classes

In order for an object to become persistent, its class needs to be enabled. A Java class has to be made persistence-capable, i.e. we allow that some of its instances reside in the object-oriented database. There are 2 ways to achieve it: *explicitly*, that is to require that the programmer mark persistence-capable classes; or via some *implicit* automatic or derived fashion.

Explicit marking can be done in several ways, e.g. by defining a class as a subclass of a database superclass. Any class in Java is already a subclass of Java class `Object`: the same concept applies now to such a database root superclass. We can now say that an persistent object handled by our Java API is an instance of that class. This class handles the basic correlation between the physical database object and its local Java counter part.

Another approach is more implicit: *any* object can be made persistent, in effect, all Java classes are therefore persistence-capable. The advantage of this approach is that even Java system classes, AWT classes, utility classes, etc. can have persistent instances. This concept is called *orthogonal* persistence. The programmer, of course, need not be aware of this.

The second issue of persistence in Java is how to make individual objects persistent. The desire is to stay minimally intrusive. The first option is to add a variation of a `new` operation to Java (e.g. `pnew`); or, to provide a mechanism to tell the database about those objects that are to be persistent; or, as a variation of that, enable a ODMG C++-style `set_object_name()` operation that declares root objects.

The third issue is how to retrieve objects from the database: 3 sub-options again: (1) allow some general query capability similar to a single SQL project, e.g. "select instance of class whose attribute has value"; (2) extend the Java programming language with a new looping construct that allows to scan instances of a class; or (3) retrieve specific objects that have been given a root name in ODMG C++ style: `lookup_object("root name")`. As Java objects are moved into the database, member fields that contain references to other Java objects need to be converted. A Java reference is only valid during a single run of a program. Therefore, they need to be converted into more general object identifiers that are valid, i.e. unique, in the context of the entire database. As objects are retrieved from the database these object identifiers will have to be reconverted into valid Java references. Once a persistent object has been retrieved from the database, other objects are reached by following its relationships, i.e. via object identifiers. Objects are automatically fetched as needed from the database. This concept is called *persistence by reachability*.

Updating objects is totally transparent to the programmer. The programmer invokes member functions and goes about the programming business as usual. Member functions can update member fields of persistent objects: that in effect is an update of a database object. The programmer does not have to insert any code to explicitly affect the update to the database. Of course, the final success of an update is dependent on the final success of the transaction.

2.2. Implementation Issues

The above mentioned issues present various degrees of complexity: how to resolve how to make a Java class subclass of the database root class; how to enable objects to be named; how to retrieve objects by name; how to enable traversal of persistent objects; and finally how to enable object update.

ODMG suggests 3 potential ways to address these "how-to" issues in Java: via a preprocessor to Java compilation; via a postprocessor of Java byte codes; or via a change to the Java Virtual machine.

In the preprocessor version, one would parse Java code to add the necessary snippets to enable persistence capability, e.g. make a class a subclass of a database root class. This approach would also allow to insert the necessary checks and additional code to enable the fetch, traversal and update of persistent objects.

In the postprocessor version, one would read and modify Java byte codes for the classes that are to be effected. Since Java byte codes are well defined and documented – tools exist to process them – this version has the same capabilities as the first version, but might even be easier to implement.

To change the Java virtual machine is not a complete solution: it needs to be made aware of objects that are persistent. One could add new byte codes: this approach is used in the PJama approach. However, in our opinion this runs against the spirit of Java: such a modified Java program will not run *everywhere*.

In summary, approaches one and two are doable, however, as a consequence they alter the Java compilation approach: a programmer needs to use a special version of the Java compiler.

Our approach attempts to stay away from changing the compilation approach and from changing the Java virtual machine. Instead we use Java's reflection capabilities, where we generate the necessary changes to the Java program automatically as it runs. The result is limited intrusion into the programmer's model and into the compilation model.

3. Related Work

Many different approaches have been suggested to enable database access from a Java program. Prominent, for example, is the Java Database Connectivity package (JDBC) [Hamilton and Catell, 1996]. Of course, it allows Java to access relational database system[1]. JDBC access can be bridged into ODBC access, or – as is becoming very common – direct drivers are provided by the database management system. Several extensions to this JDBC approach have been suggested: one is JRB (Java Relational Binding) which allows to manage database entities (transactions, queries, etc.) and to store and retrieve Java objects. JRB has been used as further basis for a Java Universal Binding (JUB) which hides the relational aspects of the underlying database system [Xhumari et al., 1997].

Several commercial vendors of ODBMS have provided Java bindings that follow the ODMG recommendation: ObjectStore[PSE, 1998] and GemStone/J[Gem, 1998] are 2 examples.

The most integrated approach is PJama[Atkinson et al., 1996]: here Java truly becomes persistent with minimal intrusion into the programming model, however, at the expense of requiring a non-standard Java compiler and significant changes to the Java Virtual Machine.

4. Reflection

Our approach to enable Java programs to store their objects in an object-oriented database is based on reflection. The term *reflection* means that a computer program is able to observe and/or change itself while it is running. A programming language becomes *reflective* if it enables its programs with reflection, i.e. it enables that the program code is changed by the program at run time [Stemple et al., 1993].

An object-oriented program is described via a set of classes. OO programming languages such as Smalltalk and Java represent these classes as objects and make them available at run-time. Smalltalk has a rich set of meta-classes that enable reflective programming. Java, on the other hand, does not directly support full reflection: its `java.lang.reflect` package allows a program to inspect the class of objects, its member fields and functions, etc., but it does not allow that they are changed. In order to achieve full reflection, access to the Java byte code is necessary. Once a Java class is available in byte code, it can be changed and the class reloaded using the standard Java class loader. Recreation of Java byte codes and dynamic loading can be done at run-time. While not elegant, this avenue in effect enables full run-time reflection for the Java programming language.

In the database context it is quite common to represent and store the schema with the rest of the data in the database and also to make it available to application pro-

[1] more precisely, it allows access to database that allow ODBC-style access, and of course, the common commercial OODBMSs have such interfaces.

grams[2]. Our prototype implementation uses SemODB (semantic object-oriented database system)[Rishe, 1992]. Its metaschema contains classes for the database, transactions, file storage, as well as classes "Category" and "Relation" that govern what is stored in the database. "Category" has instances for the classes whose objects populate the database. "Relation" captures the superclass/subclass relationships among classes and class associations. In the semantic object-oriented database approach, attributes are also captured as associations to basic primitive database types.

The Java meta schema on the other hand is quite simple: it contains classes `Class`, `Field` and `Method`. We also need to use the class `ClassLoader` since that enables the dynamic reload of classes at run time.

In addition to the meta classes, the Java program will contain application specific classes, and the database schema will contain content specific classes.

Our API now allows the 2 sets of classes to be reconciled automatically. As the programmers declares objects to become persistent, we inspect – via reflection – the details of the class, then check whether a corresponding class exists in the database, and then – again via run-time reflection – create a proxy class that serves as a bridge to enable Java objects to move into the database, and for database objects to move into the Java execution world. In this fashion, all Java classes mentioned in a program are made persistence capable, including the Java support classes from the standard JDK packages.

Some special cases need to be considered: (1) Java primitive types, e.g. `int`, `char` etc. are mapped directly into equivalent database primitive atomic types; (2) a Java class might not have a corresponding category in the database: we then create the category automatically based on the data members declared for the Java class.

In addition, we enable a caching mechanism, where the proxy classes are also stored with their byte codes in the database, which significantly improves the performance of those Java programs which use common classes. [Ege et al., 1998b]

5. Database Programming with Reflection

For further illustration lets consider a sample Java class (see Figure 1) which is defined without any database access in mind. This class is then used in Java programs that access the database:

5.1. To make an object persist

In order for a Java program to make arbitrary object persistent it will have to make that intention known. Our reflection package provides a helper class `DBAccess` which defines the member function `makePersist()`. This code example illustrates the use:

```
DBAccess db = new DBAccess("demo");
Person p = new Person();
db.makePersist(p, "first");
```

The member function inspects the class of its parameter in the Java program and in the database. If a corresponding database category is not found, a new category is created in the database based on the member fields of the Java class. In any case, a new instance of

[2]This is a form of reflection, full reflection would enable dynamic schema evolution.


```
import java.io.*;

class Person {
   String name;
   int age;
   Person [] children;
   void add(Person c) {
      Person [] list;
      if (children != null) {
         list = new Person [children.length + 1];
         for (int i=0; i<children.length; i++)
            list[i] = children[i];
      } else
        list = new Person[1];
      list[list.length-1] = c;
      children = list;
   }
   void print() {
      System.out.println(name + "(" + age + "):");
      if (children != null)
        for (int i=0; i<children.length; i++)
          children[i].print();
   }
}
```

Figure 1. Java Class Person

the category is created in the database. We now create a proxy class which will serve as an intermediary between Java objects and database objects. We add a new member field to the Java class to allow it to be associated with the proxy class. The Java class is further modified to become a subclass of the database root class, unless it is already a subclass of a class other than `Object`. In this case, its superclass will be modified to become of the database root class. The code of all member functions of the Java class is inspected to check for member field access: where it occurs checks are inserted to ensure related objects are fetched from the database as needed. Of course, if the Java class is already prepared in this way, then `makePersist()` will not repeat this setup task.

The second parameter to the `makePersist()` member function is used as root label for the object.

5.2. To retrieve an object

In order to retrieve an object from the database, we provide the member function `fetch()` in class `DBAccess`. This code example illustrates the use:

```
p = (Person) db.fetch("first");
```

It allows to find an arbitrary object in database that had been named previously via the `makePersist()` member function. It gets the database object, inspects its category, searches for the equivalent Java class. If the class is not found, then an exception is raised. It prepares an instance of that class by setting its member fields and converting database object identifiers into Java references. Like before, a proxy class is generated and associ-

```
import java.util.*;

public class DBwrite {
   public static void main(String[] args) {
      DBAccess db = new DBAccess("demo");
      Transaction tx = new Transaction(db);
      tx.begin();

      Person p = new Person();
      db.makePersist(p, "first");

      p.name = "John Doe";
      p.age = 31;
      Person c = new Person();
      c.name = "little guy";
      c.age = 1;
      p.add(c);

      tx.commit();
      db.close();
   }
}
```

Figure 2. Writing Java Objects

ated with the Java class of the retrieved instance. The Java class is also made persistence capable as above.

5.3. To navigate from object to object

Since object creation and retrieval inserted presence checks whenever member functions access member fields, objects can be fetched from the database as needed. Whenever an object is further retrieved it may be possible that it is an instance of a class that has not been encountered yet. We then create the necessary proxy class and modify its Java class as discussed above.

5.4. To update persistent objects

Before a transaction is committed we need to ensure that all persistent objects are stable, i.e. that all its dependents have been recognized and their classes been made persistence capable. The member function `stabilize()` serves this purpose. It is called whenever the programmer commits a transaction.

Figure 2 illustrates a Java program that writes objects into the database. Two Person objects are created, "little guy" as a child of "John Doe". Only "John Doe" is explicitly made persistent, "little guy" becomes persistent implicitly when he is added as a child. Once the transaction is committed, both objects are stored in the database.

Figure 3 illustrates a Java program that reads objects from the database, makes some changes, and then commits the transaction: "John Doe", the person object labeled "first" is explicitly fetched from the database; his child person object is implicitly directly accessible.


```
import java.util.*;

class DBread {
   public static void main(String args[]) {
      DBAccess db = new DBAccess("demo");
      Transaction tx = new Transaction(db);
      tx.begin();

      Person p = (Person) db.fetch("first");

      p.print();

      Person c = new Person();
      c.age = 0;
      c.name = "baby";
      p.add(c);

      tx.commit();
      db.close();
   }
}
```

Figure 3. Updating Java Objects

A new person object "baby" is created and added to "John Doe's" children array. All 3 objects are updated to the database upon the transaction commit.

In summary, the Java programmer will encounter limited intrusion when writing persistent programs. Figure 4 shows the UML class diagram for class `DBAccess`.

6. Implementation

The architecture of our prototype implementation [Ege et al., 1998a]. has the following components:

1. a Java front-end as described in Sections 4 and 5 of this paper;
2. a substrate, or facilitator, that enables our reflection-based persistence approach;
3. a communications protocol called Icecube, which enables objects to travel in a network of distributed clients and servers;
4. data base servers: either directly implemented database core engines, or proxy servers that interact with other database systems.

Components 2 and 3 are provided in form of a Java package. The package contains the classes `DBAccess` as well as the meta classes for the semantic object-oriented database system.

Another important element of the architecture is the Icecube communications link: Icecube enables a set of command that allow a Java application program to access the services of the server database system. Java persistent object access is translated into Icecube commands. The commands travel on the socket link to the database server. The database server receives the command through its engine interface (EI). This EI is implemented as

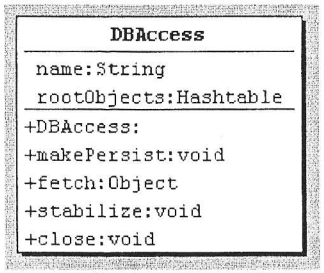

Figure 4. Database Access Class

a set of switch statements that make the function calls associated with the method invocation information (Action Code) of an incoming message. The database engine delivers its results through its EI. The EI packages the results into a message and sends it out through the socket.

Our architecture allows for multiple servers to connect to an Java API via our IceCube communications link. 2 servers are envisioned: both are based on the semantic binary database model [Rishe, 1992]: the first is implemented using Java; the second is represented with its C++ API, which maps itself into 2 available servers, one written again in C++, the other simulated on top of a relational database server.

7. Conclusion

In this paper we presented a flexible application programmer interface that allows Java objects to be stored in an object-oriented database system. Our approach uses a reflection mechanism, that inspects and changes a running Java program. Our approach requires limited intrusion into a normal Java program. Especially, it is not necessary to pre-process Java code or post-process Java byte codes. A standard Java compiler can be used. The code is able to run on any standard Java virtual machine.

Our future plans call for an investigation of using the reflection approach for other aspects of object-oriented database systems, such as schema evolution, query optimization, and load balancing.

References

[Atkinson et al., 1996] Atkinson, M., Daynes, L., Jordan, M., Printezis, T., and Spence, S. (1996). An orthogonally persistent java. *ACM SIGMOD Record*, 25(4):68–75.

[Catell et al., 1997] Catell, R., Barry, D., Bartels, D., Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., Strickland, H., and Wade, D., editors (1997). *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann.

[Ege et al., 1998a] Ege, R. K., Battikhi, Y., Pardo, P., Uppal, J., and Rishe, N. (1998a). A modular java api for object-oriented databases. *Proceedings of IEEE COMPSAC 98.*

[Ege et al., 1998b] Ege, R. K., Liu, J., and Lebedev, V. (1998b). *Using Java to add "Stored Procedures" to Databases.* FIU-HPDRC Technical Report.

[Gem, 1998] Gem (1998). *GemStone/J Programming Guide.* GemStone Systems, Inc., www.gemstone.com.

[Hamilton and Catell, 1996] Hamilton, G. and Catell, R. (1996). *JDBC: A Java SQL API.* Sun Microsystems, Inc., http://java.sun.com.

[PSE, 1998] PSE (1998). *ObjectStore PSE and PSE Pro for Java User Guide.* Object Design, Inc., http://www.odi.com/.

[Rishe, 1992] Rishe, N. (1992). *Database Design: The Semantic Modeling Approach.* McGraw Hill.

[Stemple et al., 1993] Stemple, D., Morrison, R., Kirby, G., and Connor, R. (1993). Integrating reflection, strong typing and static checking. In *Proceedings of the 16th Australian Computer Science Conference*, pages 83–92, Brisbane, Australia.

[Xhumari et al., 1997] Xhumari, F., dos Santos, C. S., and Skubiszewski, M. (1997). Java universal binding: Storing java objects in relational and object-oriented database. In *Proceedings of The Second International Workshop on Persistence and Java(tm) (PJW2)*, Halfmoon Bay, CA.