Building a Persistent Object Store using the Java Reflection API

Arthur H. Lee and Ho-Yun Shin

Programming Systems Laboratory
Department of Computer Science
Korea University
Seoul, Korea
+82-2-3290-3196 (phone)
+82-2-925-3215 (fax)
{alee, hyshin}@psl.korea.ac.kr

ABSTRACT
In this paper we describe a persistent object store designed and implemented to test and evaluate the merits and shortcomings of the Java Reflection API. Our goal is to design a persistent object store that supports transparent object persistence. Our premise was that we should be able to achieve this goal if the reflection API is designed with enough features and flexibility. It turns out that most of the needs for object persistence was nicely met by the current reflection API, but we also found that some could not be done with the current API. We describe in detail the merits and shortcomings seen from the viewpoint of an object persistence implementor.

Keywords
Java, reflection, object persistence

INTRODUCTION
Using persistent object stores in dealing with data management has advantages over using other traditional data management systems such as relational data base management systems (RDBMS) [Loom95]. With a persistent object store (POS) the client code accessing the data managed by the persistent object store is written in an object-oriented language as opposed to the data definition languages (DDLs) and data manipulation languages (DMLs) in the case of a relational DBMS. However, the client code using persistent objects still has to write a fair amount of boring code to benefit from the objects being saved and loaded [Banc88, Care89, Deux90, Kim95]. It is even worse when this kind of meta information has to evolve over time and it does in most cases. That is, the class definitions often change and the objects saved in the store have to evolve to match the changes in the definitions. Unfortunately the recent applications demand more changes in their data being used, and the problem is getting worse.

With the reflection API supported by Java, this problem can be simplified a great deal and we describe a design and implementation of a persistent object store using the Java Reflection API.

THE JAVA REFLECTION API
Using the Java reflection API included in the Java Development Kit (JDK) version 1.1 or higher, a programmer can obtain meta information on the Java objects at run-time [Flan97]. That is, the programmer can access the information on a class definition including the fields and methods of the class. Additionally, it supports method invocations and accessing field values as long as it does not break the Java security boundary. All these extra capabilities are possible because this kind of meta information is maintained by the Java run-time system. What is better, the Java run-time elements are all objects themselves with their meta information available for access as well, thus allowing consistent access patterns.

The Java Reflection API maintains a metaobject for each of the main elements of the language such as a class, method, and field. All this centers around the class called `Class`. There is one instance of the class `Class` for each class loaded in the run-time environment. User code can get to the metaobject by invoking the `getClass` method given an object. Once a metaobject is obtained, a number of methods are available for accessing the state information on the metaobjects.

User code can access the fields or the methods of an object via field objects or method objects. For example, the name and the value of a field can be obtained through the field object. Similarly, the name of a method can be obtained and the method can actually be invoked through the method object.

The `Class` class supports `getMethods, getMethod, getDeclaredMethods, getDeclaredFields, getFields,` and `getField` for user code to call.

The Java Reflection API also supports a static class named `Array` for the arrays. There is no instance of `Class` for an array, but the `Array` class handles the task of maintaining the meta information for an array.

BASIC IDEA

In the traditional persistent object stores client code must explicitly provide the meta information for persistent objects. Based on the information, the persistent object store manages the objects by saving and loading each field of persistent objects as needed.

When the meta information is available with the help of the reflection API, client code can access objects in a persistent object store without having to write any extra code to specify the meta information for the persistent objects. The client code can be much simpler and some of the performance issues can be dealt with by the underlying persistent object store. A similar idea using the metaobject protocol of Common Lisp Object System but with a different design can be found in [LZ97].

DESIGN AND IMPLEMENTATION

Overview

We tried to focus our design of the persistent object store on easy programming interface and lean design targeted for applications requiring small to medium amount of data, say about 50,000 objects of typical sizes. Our design is intentionally simple in functionality still providing almost transparent persistence for easy interface on the part of client code.

Sequential access files are adopted as the structure of secondary storage system in our design. All persistent objects are saved and loaded in batch mode. With our goal mentioned above, this design is acceptable.

Our system consists of the `PersistentRoot` class, POS Indexing Manager, and Object Storage Manager. The `PersistentRoot` class is viewed as the user interface for object persistence. All persistent classes must inherit the `PersistentRoot` class, thus our system supports object persistence by inheritance. The POS Indexing Manager maintains an internal structure for objects. This also handles the search function for programmers. The Object Storage Manager is invisible to the client code, and plays the role of connecting persistent objects to the secondary storage system.

The Life Cycle of an Object in the POS Environment

Perhaps it would help the reader understand the overall design a little easier if we describe the life cycle of a persistent object. Along the way, we will indicate where the Java Reflection API plays what role.

When we define a persistent object, we would indicate that the object is persistent and provide the necessary meta information for persistence. In our design, however, all a persistent class has to do is inherit the `PersistentRoot` class. Then the meta information that we need to save an object can be obtained through the Java reflection API at run time.

Once an object is created from a persistent class, to the client code the object seems mush the same as an ordinary object in the way it is accessed. However, the system performs some extra work internally. A persistent object created from a persistent class that inherits the `PersistentRoot` class is registered in the POS system as persistent by the constructor of the class. A unique object identifier (OID) is also assigned at this point. At some point while a persistent object is being accessed, the client code invokes the `saveAll` method to save all the persistent objects to the POS. This in turn calls the `save` method for each object registered in the POS. A `save` method at this point has to know what part of the object has to be saved, i.e., some meta information is needed. Ordinarily this would be provided by the client code, but it is obtained through the Java reflection API in our system, thus reducing much of the unnecessary overhead of the client code. Through the Java reflection API, we can access the meta information and the value of a field. So, we can save the current state of an object by accessing the fields declared to be `public` or `protected`. The current Java reflection API does not allow accesses to the fields that are declared `private` other than the name of the field. So, in our current design we had to devise an *ad hoc* mechanism to compensate this situation. If a field of an object has an object reference as a part of its value, the reference is converted into a logical identifier (OID) before it is saved.

Once `saveAll` is completed, the memory copy of the saved objects behave the same as they did before they were saved. Figure 1 summarizes the creation and saving phases of a persistent object using the reflection API. The portion marked by the dotted box depicts how the reflection API is used to obtain the meta information.

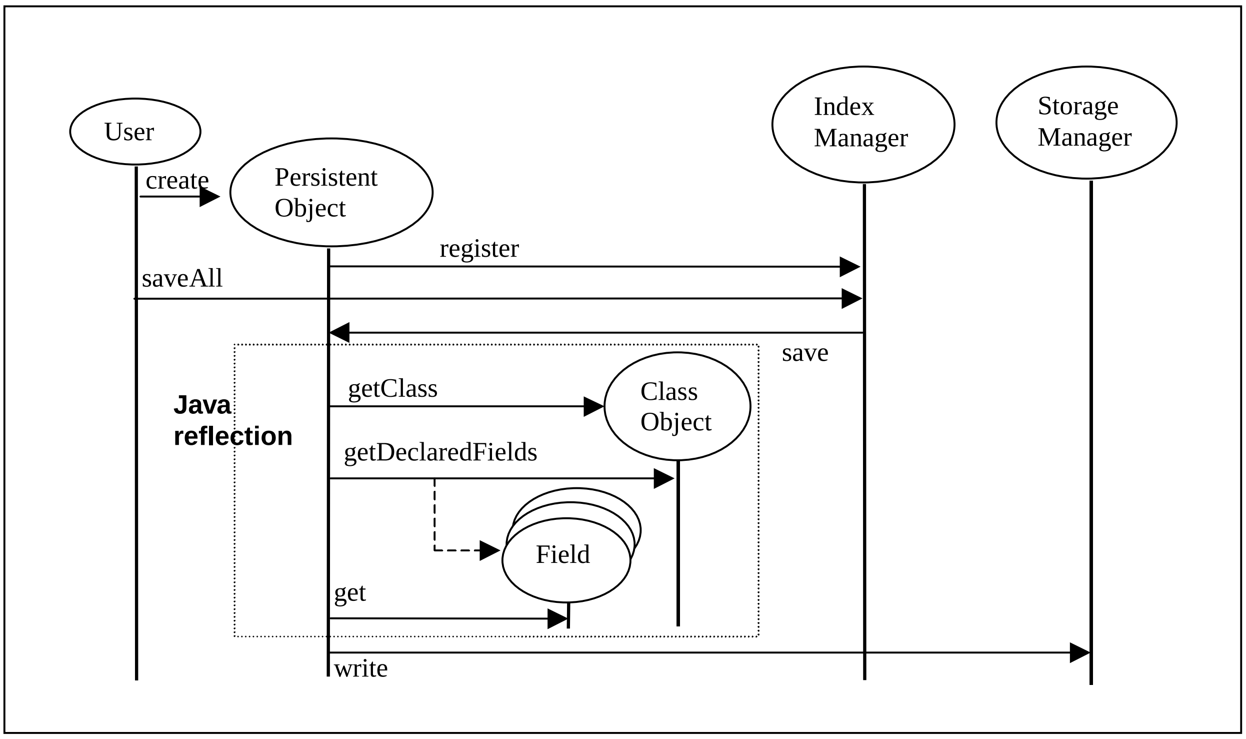

Figure 1 : Saving a persistent object using the Java reflection API

When an application is started anew or a running application needs to load saved objects, client code can invoke the `loadAll` method. The `loadAll` method reads in the saved objects one at a time and decides the class from which the object was originally created, and then creates a new copy in the memory. At this point all the meta information needed to create the object can be obtained from the Java reflection API. Once an object is created this way, the values of the fields can then be instantiated by invoking the `load` method of the object. A logical ID saved as a part of a field value is converted back to the physical address as objects are loaded. A similar figure to Figure 1 could be drawn for the loading phase of a persistent object's life cycle.

`PersistentRoot` class

The `PersistentRoot` class acts as the programmer's interface to the persistent object store. All the objects created from a class that inherits the `PersistentRoot` class will be persistent. In most cases, user code does not have to do any special work other than inheriting the class to define persistent objects.

Three main operations of the `PersistentRoot` class include the `save` method, `load` method, and the constructor. The constructor registers persistent objects to the indexing manager. This way, all persistent objects are implicitly registered during its instantiation process. Each persistent object is assigned a unique object identifier (OID).

The `save` method saves the current state of a persistent object to the secondary storage. It first obtains the meta information on the class definition of the run-time object from the reflection API. Based on the field information from the meta data, the value of each field is saved. Since the meta data are used through the reflection API, this piece of code in the save method does not have to hard code the field names. All the field values of atomic data types are saved as such and object references are saved using the OID, the logical address, of the persistent object. All the referenced objects are also saved, thus all the objects reachable from a persistent object are saved.

The `load` method loads a saved object from the secondary storage into memory. By the time a persistent object is loaded, the class definition for the saved object should already be in the Java run-time system. So, an object is created for the saved object and the fields are instantiated with the state that was saved in the store. The object references that were converted from the physical addresses to the logical addresses, OID's, are reverted back to new physical addresses in memory as the objects are loaded, thus the OID's are swizzled into physical object references again [Moss91, Loom95]. If a field in a class is declared to be `private`, then there is a problem with this strategy. We will elaborate on this point in the next section.

POS Indexing Manager

The Indexing Manager is designed with a B+ Tree. Each persistent object is inserted into the tree and its OID is used as the key for indexing the object. This is transparent to the client code. Two useful methods, `saveAll` and `loadAll`, are provided to the client code to save all the objects as a snapshot or load all the objects from file containing a snapshot. The `saveAll` method saves all the objects registered in the Indexing Manager by invoking the `save` method on each object.

Similarly, `loadAll` loads all the objects by loading each object in a file. For each object, a new instance of target object is created and the `load` method is invoked on that newly created object to read the state of the object saved, thus recovering a saved state by instantiating a new instance. In the case of a non-memory-resident object reference in the object currently being loaded, a temporary empty object is created and referenced. Since this temporary empty object has its OID, it can later be instantiated with the state saved as long as the save was done right.

Another functionality of the Indexing Manager is provided by the `search` method, it searches an object by stringfied field name and its value that is passed as a parameter. This operation is done easily again using the Java reflection API.

Object Storage Manager

The Object Storage Manager is an interface that links a memory resident object and the secondary storage. It provides a simple API which reads a block of data from and writes to a persistent storage file. This manager is also transparent to the client code.

LIMITATIONS OF THE JAVA REFLECTION API IN IMPLEMENTING A PERSISTENTOBJECT STORE

Due to the protection mechanism adopted by Java, the reflection API can not access the fields and methods declared to be `private`. Therefore, the design we described so far can not allow classes with `private` fields. Rather than restricting the semantics of class fields, we decided to come up with an *ad hoc* solution for now. We added two methods to the `PersistentRoot` class, `userSave` and `userLoad`, to solve this problem. The `userSave` method is invoked before invoking the `save` method on a persistent object. For private fields in a persistent object, values of the private fields are moved by the `userSave` method to the `userObjects` field declared to be `public` in the `PersistentRoot` class whose type is an array of Java `Object` class. The state of all the private fields can then be restored by the `userLoad` method which is invoked right after the usual `load` method is invoked as a persistent object is loaded from the store.

This sort of ad hoc solution is not desirable and we suggest extending the semantics of the Java reflection API so that private fields can also be accessed. However, this has to be done without compromising the Java security model. One possible solution might be to allow a class definition to specify what can be accessed by the reflection API. This is not a perfect solution, but at least the security can be maintained by the client code that defines the class. This level of modification in the reflection API would have solved the problem that we faced in our design without having to resort to the *ad hoc* solution that we adopted.

CONCLUSION AND FUTURE WORK

In most persistent object stores, a client programmer must specify the meta information for the persistent objects or use a preprocessor to support object persistence. This additional programming is not desirable and becomes the source of maintenance nightmare. To eliminate these difficulties, we designed and implemented a persistent object store using the Java reflection API. With this design a client programmer to the POS does not have to provide the meta information for the persistent objects. This design makes dealing with class evolution easier as well.

However, we encountered a difficult problem using the reflection API to implement a persistent object store. The Java security system does not permit the reflection API to access fields and methods declared to be `private`. Short of being able to change the Java implementation, we devised an *ad hoc* solution. This solution requires two extra methods for each persistent class: one for saving objects and the other for loading objects. This forced us to compromise on some of our design goals, e.g., transparent object persistence is somewhat compromised.

We are investigating the Java security environment in relation to the Java reflection API. It would be nice to find a way to allow more access to the object data including the fields declared to be `private` by the reflection API without sacrificing the security concerns.

Although not confirmed, it seems that even the fields declared private can be accessed through the Java reflection API on the JDK 1.2 version according to [Oaks98].

REFERENCES

[Banc88] F. Bancilhon. et. al. “The Designing and Implementation of O_2, an Object-Oriented Database System,” *OODBS*, 1988.

[Care89] Michael J. Carey, et. al. “Storage Management for Objects in EXODUS,” *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, eds., ACM Press, 1989.

[Deux90] O. Deux, et. al. “The Story of O_2,” *IEEE Transactions on Knowledge and Data Engineering*, Vol 2, No. 1, March 1990.

[Flan97] David Flanagan, *Java in a nutshell*, O’Reilly, pp. 219-226, 1997.

[Kim95] Won Kim, *Modern Database Systems*, ACM Press, pp. 175-202, 1995.

[LZ97] Arthur H. Lee and Joe Zachary, “Adding Support for Persistence to CLOS via its Metaobject Protocol,” *Lisp and Symbolic Computation: An International Journal*, Vol. 10, No. 1, pp. 39-60, 1997.

[Loom95] Mary E. S. Loomis, *Object Databases – The Essentials*, Addison-Wesley, 1995.

[Moss91] J. Eliot B. Moss, “Working with Persistent Objects: To Swizzle or Not to Swizzle,” *IEEE Transactions on Computers*, 1991.

[Oaks98] Scott Oaks, *JAVA Security*, O’Reilly, p.431, 1998.