Proceedings of the NACCQ 2000
Wellington NZ www.naccq.ac.nz

Persistent Object Storage in the File System: How it works and its use as a database teaching tool

Paul Charsley
Information Systems Section

UCOL Palmerston North
P.Charsley@ucol.ac.nz

ABSTRACT

The Java™ language provides a mechanism known as serialization that enables Java objects created by an application to be stored in files. A serialized object can be converted back into its original object state through the reverse process of de-serialization. This very simple storage mechanism is being used by the author as part of an application framework which can be used by software developers to build low cost business systems. The main problem with using serialization is that all of the features that are needed for a robust storage mechanism need to be implemented on top as part of the application framework. These features include support for transactions, rollbacks, commits, locks and the generation of unique object ids. This paper examines the author's approach to all these problems and discusses how the final product will be used as a teaching tool to aid in the understanding of database technology.

Definitions

Serialization – The ability to write the complete state of an object to an output stream. De-serialization – The ability to recreate a serialized object. Class – A collection of data and methods that operate on that data. Interface – A collection of empty methods that are implemented by Classes. Object – An instance of a Class.

1. INTRODUCTION

Most business applications involve the permanent storage of data. The majority of current business applications are permanently linked to a specific storage mechanism which cannot easily be changed without redesigning the application itself. This problem together with the fact that most new applications are object oriented has brought about a need for software (a persistence framework) that decouples object oriented applications from the mechanism used to store their data.

Most of the work in this field is aimed at large companies that need to access data from their legacy relational databases and as such, is too expensive for smaller businesses. The complexity of this type of software has also lead many practitioners and consultants to advise businesses to buy rather than develop their own persistence framework. Ambler (1998) puts this succinctly when he says: "Building a persistence layer is spectacularly hard" and goes on to say "so do a little shopping first!". However, some work has been done, see Charsley (1999), in the development of public domain software that meets these needs and this paper deals with one aspect of this.

One technique that can be used to permanently store objects in Java applications is serialization. The serialization of objects into files provides a cost free

storage system that can function on any operating system. Unfortunately, unlike a database, the file system does not provide standard database features such as transactions, rollbacks, commits, record locking and unique ids so that all of these features will need to be implemented.

Although complex to design and construct, the author believes that a persistence framework that uses file serialization would be an invaluable tool for small businesses. In addition, it would provide a very useful platform for the teaching of database principles and object oriented design.

2. GENERAL PRINCIPLES

This section discusses some of the known problems that must be dealt with when building persistent mechanisms. It looks at how other researchers and developers have tackled them and goes on to describe a recommended approach when file serialization is used.

2.1 Mapping objects to their physical storage medium

Most of the generally recognised strategies for mapping the data in an object oriented application to the physical storage mechanism relates to the use of relational databases. The simplest technique, and that used by Vadaparty (1999) involves mapping classes directly to database tables. Class attributes are mapped to database columns so that a table row maps directly to an instantiation (object) of a class. This sounds simple in practice, but what about inheritance? Should one table represent all classes in the hierarchy or should a different table be used for each class? Ambler (1998) describes three possible solutions:

- Use one table for an entire class hierarchy
- Use one table per concrete class
- Use one table per class

The above techniques all have their particular advantages and disadvantages. Using one table for the entire class hierarchy can provide faster data access but is inefficient. On the other hand, one table per class implies slower data access but a more efficient use of relational database tables. One of the advantages of using the file system to store serialized object data is that it can be used to model the hierarchical class structure of the object oriented Java business system (Java only supports single inheritance). As an example, consider an application with a class called Person. If two classes called Student and Lecturer inherit the attributes of Person, then the Student and Lecturer serialized objects would be located in a sub directory of the directory containing the Person objects.

The other issue to be addressed is that of the data structure to be used to store the serialized objects. A good candidate is the Java Hashtable. This data structure provides the ability to store any object along with a key value that can be used to retrieve it. Of course it must be recognized that the data structures provided with the standard Java classes will never be optimized for efficient data access and providing features such as indexing would require a specialist data structure to be designed.

One big advantage of using serialization to store data is that once the data is retrieved (de-serialized) it is already in a data structure suitable for manipulation by the application. Data restored from a relational database normally has to be restored in a Vector or Hashtable before the business application can make use of it. This is almost always a very inefficient operation, Stanchfield (2000), and copying data from a data store to a data structure before using it is one of the main reasons for poor performance in these types of applications.

2.2 Design patterns

Design patterns are proven solutions to standard problems. The pattern concept was described nicely by Alexander (1977) – “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”. A persistence framework will contain many standard design problems and the rest of this section describes two common patterns and their use.

2.2.1 The Peer pattern

This pattern is used heavily in the Java core libraries. The philosophy behind the pattern is one of extracting a very complex portion of a classes behaviour into a separate “peer” class. Each class in a business system that is designated as persistent (all instantiations need to be saved to a data store) will need to have a means of saving itself. It makes sense to move the implementation specific details of saving to a separate peer class and have the business domain class simply call methods in its peer when it needs to perform a “persistence” operation.

2.2.2 The Singleton pattern

The purpose of this pattern is to "ensure a class only has one instance, and provide a global point of access to it", Gamma, E. et al (1995). By making the singleton classes constructor method private and using a public method to access a single instantiation of the class it is possible to ensure that following the initial instantiation, all subsequent calls use the same instance.

This nicely meets the requirements of a global object ID that can be incremented and used to assign integer id's to each new object created. The Java code for such a class is shown below:

```
public class OID {
private static OID instance;
public static int id;

private OID() {
super();
}

public static synchronized OID getInstance() {
if (instance == null)
     instance = new OID();
return instance;
}

public synchronized int getId() {
     ++id;
      return id;
}
}
```

3. DATABASE ISSUES

The following issues are by no means a complete set of all the problems that need to be resolved. However, they are indicative of the complexity of the problem. For a more comprehensive list see Ambler (1998).

3.1 Locks

The ability to lock a set of objects that a particular user is updating or deleting is fundamental if a business system is to support multiple users accessing data simultaneously. Reese (1997) introduces the concept of a Lockholder who owns a lock object that is passed to every business object that the Lockholder might change.

A business application for an academic records system might instantiate a Person object for each user that logs in to the system. In order to "become" a Lockholder, the Person object would need to "implement" a Lockholder Interface. In Java, an Interface is simply a set of empty methods and any object that wishes to implement an Interface has to provide method bodies for all of them (it implements the methods).

A unique lock object would be created for the Lockholder, which would be passed as a parameter to the business object every time the Lockholder tries to take possession of an object. If another Lockholder attempts to modify an object held by that Lockholder, a lock error would be thrown since a comparison of the lock already associated with the object and that owned by the new Lockholder would reveal that they were not the same.

3.2 Transactions

Transaction management is an essential feature for any database management system. Most operations that are performed on an object oriented business system involve making changes or creating more than one object at a time. We have already seen how a lock object and a Lockholder interface can be used to prevent different users from making changes to the same object at the same time, we now need a technique to perform operations on different objects as a single operation.

Reese (1997) uses the concept of a Transaction object that is held by the Lock object. The Transaction object is aware of all the business objects that have been locked by the Lock object. When changes are made to business objects, the Transaction object only commits these changes to the data store if all of them were successful.

4. THE PERSISTENT STORAGE FRAMEWORK AS A TEACHING TOOL

One of the problems of trying to teach database principles with standard database development tools such as Oracle, Progress and Access is that these tools were not designed with teaching in mind. Although all database development environments implement transactions and record locking it is not possible to see how they do it.

An Object oriented storage framework that implements all important database functions as objects would enable students with an understanding of object oriented principles to better grasp the fundamentals of database principles. For example, it would be possible to examine in detail the transaction object and see what its

responsibilities are and with which other objects it collaborates in order to execute its functions.

5. CONCLUSION

An object oriented Java application framework was designed with the objective of providing software that could be used to aid small businesses in developing countries and New Zealand in the design and construction of business systems.

Using serialization to save objects to files stored in the file system was seen to be the simplest method of providing permanent storage for a business application's objects. However, it must be recognised that this technique does not scale well and will be used mainly when no commercial storage system is available or as a start off point when testing or introducing a new application.

Its use as a teaching tool is more obvious and the author sees it as part of a wider strategy to introduce object oriented techniques into our teaching methodology.

6. REFERENCES

Ambler, S. (1998), The Design of a Robust Persistence Layer for Relational Databases. <http://www.ambysoft.com/persistenceLayer.pdf

Ambler, S. (1998), Mapping Objects to Relational Databases. <http://www.ambysoft.com/mappingObjects.pdf

Charsley, P. (1999), Affordable business applications, The Proceedings of the 12th National Advisory Committee on Computing Qualifications, Dunedin, 4th - 7th July, 1999.

Vadaparty, K. (1999), Mapping Objects to Tables. The Journal of Object Oriented Programming, July/ August 1999, pp 45 - 47, 59

Stanchfield, S. (2000), Advanced Model-View-Controller Techniques. IBM Visual Age Developer Domain library. <http://www7.software.ibm.com/vad.nsf

Reese, G. (1997), Database Programming with JDBC and Java. O'Reilly. pp 80 – 82, 96 – 97.

Alexander C, Ishikawa S, Silverstein M, Jacobsen M, Fiskdahl-King I, Angel S (1977), A Pattern Language. Oxford University Press, New York.