

ObjectStore PSE: a Persistent Storage Engine for Java

**Gordon Landis, Charles Lamb,
Tim Blackman, Sam Haradhvala,
Mark Noyes, and Dan Weinreb
Object Design, Inc.**

Object Design, Inc. markets and sells *ObjectStore PSE for Java*, a lightweight *Persistent Storage Engine* for Java. PSE is an entry-level product for users who require persistent Java object storage in client, server, servlet, or applet environments. Typical PSE applications run the gamut from applets that need to store user configuration information locally, servlets that need to store user access history, GUI-based user-directed web spider applications that need storage for their results, and financial applications that need to store up to tens of megabytes of cached data on a client host.

This paper describes the design goals of PSE, as well as some of the implementation details, user and customer experiences with it, and future directions.

Goals

The goal of this project was to build a persistent storage and database access system for Java, with the following characteristics:

Transparent access to persistent data.

- *Fetch/Store.* The operations to fetch persistent data from a database and store modifications back to the database should be automatic and largely transparent.

Tight integration with the Java language and environment.

- *Object identity.* Object identity should be preserved across the transient/persistent boundary. That is, it must not be possible to get two different representations of the same object, nor to get two different references that denote the same object yet are not equal.
- *Memory management / persistence:* Java is a garbage-collected language, so an object *exists* if it is reachable from another (reachable) object. Similarly, an object *persists* if it is reachable from another (reachable) *persistent* object. We call this *persistence by reachability*, or *transitive persistence*.
- *Single type system.* There should be a single type system for both transient and persistent storage. It should not be necessary to translate between the types stored in the database and the types used in the runtime environment. Rather, the type system of the Java language should be directly supported by the persistence mechanism (including the ability to directly store built-in Java types).
- *Multi-threaded applications.* Java encourages multi-threaded programming so it is important for PSE to support thread safe operations. At a minimum, the entry-points must be safe against simultaneous calls from different threads. Furthermore, threads must be able to cooperate in their accesses to persistent data, or act independently from one another.
- *Compatibility with Java environment.* The persistence mechanism must not interfere with Java's goals of portability to any VM or platform, simplicity of code distribution, or applet execution.
- *Support for reuse of existing class libraries.* The development environment and tools should enable existing class libraries to be made persistent with little or no source modification.

Capable of supporting multiple back-end storage systems with different scalability characteristics.

There are currently three back-end storage systems in use with this API: PSE, PSE Pro, and ObjectStore. These span the range from a simple, low-end, single user persistent store, to a high-end multi-user client-

server DBMS. They share a common front end, such that the same user code (both Java source and compiled byte-code) can access either a PSE database or an ObjectStore database without modification or recompilation. This paper focuses on the PSE and PSE Pro engines.¹

The primary design goals for the PSE storage engine are:

- *Small footprint*: The target size for the uncompressed runtime .zip file is 300k bytes. This allows applets to download the PSE classes quickly. Minimum database size as well as storage overhead must also be small.
- *Portability*: To maximize its portability, PSE must be pure Java, and must not rely on specialized Java environments (such as a modified VM). Nothing should prohibit PSE's use in a variety of application scenarios, including applets, applications, and servlets.
- *Random Access to Persistent Data*: PSE must provide random access to persistent data, and must only read or write those objects which are read or modified during a transaction. This differs from Object Serialization, which is an "all or nothing" access method.
- *Atomic Transactions*: Even at the low end of the storage system functionality spectrum, it is a requirement that PSE must support ACID transaction semantics: atomic, consistent, isolated, and durable². This allows applications to maintain data integrity as well as provide simple "undo" capabilities through a transaction abort function. PSE must protect persistent data from system failure.
- *Multi-user*: PSE need only support coarse-grained locking for multi-user access; more fine-grained concurrency is provided by other storage systems in the ObjectStore product suite.

Note that it was not a goal to make PSE a full-function multi-user database system; rather, the goal was to make an easy to use, small-footprint storage environment, which would be API compatible with the full-function ObjectStore DBMS.

The following sections describe the implementation of PSE, and the level to which it has succeeded in meeting these goals.

API Transparency

For an entry-level database system such as PSE to be widely accepted, it must be easy to use, and it must impose as few implementation burdens as possible on the programmer. Ideally, programming for persistent storage should not be much different from programming for transient storage. In particular, the user should not have to explicitly read objects in from the database before using them, or write objects out when they are changed; instead, the database system should perform these tasks automatically. In most cases, the user should not even have to be aware of the fact that the objects being manipulated are being read from and written to a database.

We considered three main implementation options for such a transparent interface to persistent storage:

- changes to the Java Virtual Machine itself
- pre-processing of the Java source files
- post-processing of the class byte-code files

¹ Although there are small differences between them, for the purposes of this paper, PSE and PSE Pro may be considered the same.

² Transaction durability is one area where PSE and PSE Pro differ slightly. While both guarantee the durability of updates in the event of a process crash, only PSE Pro guarantees durability in the event of an operating system crash.

The first approach, changes to the Java Virtual Machine, was considered in some detail. While there could be a number of highly desirable aspects to such a solution (assuming it could be implemented in a way that did not compromise the performance of applications that do not make use of persistent storage, and could cover all of the corner cases), we nonetheless rejected it as unfeasible. Given JavaSoft's understandable reluctance to make sweeping changes to the VM, especially changes that affect the fundamental operations of object access and update, we felt that it was not practical to depend on changes to the standard Java VM definition. Even if it were possible to define a set of changes that was simple to implement, and that did not adversely affect the performance of mainstream Java program execution, it appeared that it would take a long time and a great deal of effort to overcome the inertia to change generated by the need for a stable VM definition on which to base silicon implementations. We also rejected the option of making our own, non-standard VM changes, both because we were concerned about the performance impact on transient code, and (perhaps more importantly) because this would violate our goal of portability.

The second approach, a source-code pre-processor, was also considered. It was rejected because it appeared to entail many of the same difficulties as post-processing. In fact, because the class file format is simpler than the source code format, a pre-processor is a larger undertaking than a post-processor. A pre-processor would also violate our goal of supporting the re-use of existing class libraries in their compiled form (that is, without access to the Java source code).

Given that the Java portability and code-distribution benefits are based upon the standardization of the byte-code format and structure, a post-processing technique seemed most consistent with the goals of Java and of PSE. Finally, we felt that post-processing might have wider utility, if compilers from other languages into Java byte-codes were to become available. This technique is feasible because the Java byte-code format is stable (unlikely to change much over time), simple (far simpler than .o file formats, for example), and well specified.

The transparent access to persistent storage provided by PSE (and the ObjectStore Java interface) is achieved by a class file post-processor called *osjcfp* (for ObjectStore for Java Class File Post-processor)³. The classfile post-processor accepts a program's .class files as input, and produces new annotated .class files as output. By setting the CLASSPATH appropriately, the annotated .class files produced by the post-processor form the basis of the user's program.

Class file post-processing performs two main functions.

- *Schema Generation*: It examines the fields of the targeted class files and determines the field types and locations. Using this information, it then generates auxiliary "ClassInfo" files (for example, the post-processor would generate a *PersonClassInfo.class* file to correspond to a persistent *Person.class*). ClassInfo files are registered with PSE at run-time, and contain methods that allow PSE to interrogate the number and type of fields in the corresponding class.⁴ A *ClassInfo* file has no corresponding source file; the post-processor generates the byte code directly.
- *Code Annotation*: In order for PSE to know when data is being read from or written to the database, byte codes are inserted into a class's methods. These byte codes cause the *COM.odi.ObjectStore.fetch* and *COM.odi.ObjectStore.dirty* methods to be invoked, to read data from the database into the object's fields, or mark an object as modified. These annotations ensure that a persistent object is fetched from the database before any attempt is made to read the object; and that any updates to the object are noted so that they may be

³ Our implementation supports manual code annotation as well, but most customers prefer the simplicity and safety of the post-processor.

⁴ Some of the ClassInfo code that the post-processor currently generates can be simplified or even replaced by the use of the introspection API provided in the JDK 1.1.

written to the database by *Transaction.commit*. Optimizations are employed to deal with cases such as repeated use of the same object within a single method, or within a loop.

The class file post-processor can make a class *persistence-capable* (both of the above operations are performed), or it can make a class *persistence-aware* (only the code annotation phase is performed). A persistence-capable class may have both transient and persistent instances: persistence is determined on an instance-by-instance basis, based on reachability (described in the next section). A persistence-aware class, on the other hand, can manipulate persistent objects, but cannot itself have persistent instances⁵.

The calls to *fetch* and *dirty* that are inserted by the post-processor check the object's state to determine what (if any) actions need to be taken. If the object in question is persistence-capable, but not actually persistent, then no action is required by either *fetch* or *dirty*. If the object is persistent, then it can be in one of three states:

- *Hollow*: This means that the contents of the object have not been read in from the database. Both *fetch* and *dirty* need to read the data from the database to initialize the fields of the object.
- *Active*: This means that the contents of the object have been read from the database, but have not been modified in the current transaction. While *fetch* does not need to do anything, *dirty* needs to mark the object as modified so that its contents will be written back to the database when the transaction commits.
- *Modified*: This means that the contents of the object have been read in from the database, and have been updated in this transaction. Neither *fetch* nor *dirty* need to do anything.

For instances of most classes, *fetch* and *dirty* check the state of the object by inspecting fields of the object that were inserted by the classfile post-processor. These fields, defined on the base class *COM.odi.Persistent*, store object state information⁶. When an object is read from the database by the *fetch* method, the object's state is initialized, and the object moves from the *hollow* state to the *active* state. At this point, any objects that are referenced by the object (and that have not already been accessed previously) are allocated in the Java virtual memory space, and are initialized to the *hollow* state. Hollow objects are placeholders for persistent objects, but their fields are not filled in from the database until they are actually needed. Any attempt to access a field of one of these hollow objects in the *fetch* operation being called on that object in turn.

Persistence-capable types that are not classes, such as array types and primitive types (e.g., int, boolean, long, etc.), are dealt with specially, as are their corresponding wrapper classes (*java.lang.Integer*, *java.lang.Boolean*, *java.lang.Long*), and the class *Java.lang.String*. All of these types are automatically persistence-capable. However, because they do not inherit from *COM.odi.Persistent*, their object state must be inferred differently. In the case of the primitive types, only their values are stored in fields of other objects, so they do not themselves have identity or separate representation in the database; they are read in when their containing object becomes *active*. The immutable types (the primitives wrapper classes, and *Java.lang.String*) do have separate identity in Java, but because they cannot be modified once they are created, PSE gives them a simpler treatment, and reads them in when any object that points to them is

⁵ Note that a class only needs to be persistence-aware if it *directly* manipulates the fields of a potentially persistent instance. Any class, even one that has not been post-processed at all, can manipulate persistent instances by calling their methods. In this case, because only the class itself directly manipulates its instances' fields, the object is strictly encapsulated, and code that uses the class need not be aware that instances may be persistent.

⁶ In the current release of PSE, the classfile post-processor inserts *COM.odi.Persistent* into the inheritance hierarchy of any persistence-capable class, so that these fields will be available. While this is generally automatic and transparent to the user, it nonetheless violates our goal of a single type system. An updated version of the PSE classfile post-processor will be available shortly that does not require this, but instead provides persistence support through the *COM.odi.IPersistent* interface.

initialized. Thus, there can never be *hollow* (or *modified*) instances of one of these types, there can only be *active* instances.

Arrays have identity and are mutable, and so have all the same possible states as class instances. However, because they cannot inherit from *COM.odi.Persistent* (or implement *COM.odi.IPersistent*), their state fields must be stored out-of-line, in a separate data structure that PSE maintains. Note that this means that access to a persistent array object is more expensive than access to a normal class object, because the *fetch* and *dirty* methods must look up the array in a separate table to determine its current state.

Two other classes bear special mention: *java.util.Hashtable* and *java.util.Vector*. These utility classes can be useful in a variety of database applications, so we generated persistent versions named *COM.odi.util.OSHashtable* and *COM.odi.util.OSVector*, which are shipped with PSE. We re-implemented and renamed these classes, rather than simply post-processing them with the same package and name, for three reasons. First, these classes are used both in the Java VM and in the PSE implementation in ways that would result in bootstrapping difficulties if they were persistence-capable. Second, these classes are widely used both in the Java VM itself and in application code, so the added expense of using persistence-capable classes in transient contexts could have had a performance impact. Finally, while we could have simply post-processed the classes to make them persistent, we chose instead to modify the implementations to make them perform better in persistent contexts.

Memory Management

Another aspect of transparency is the handling of memory management and object management. Java is a garbage-collected language. At the very least, therefore, a persistence interface should not interfere with the (transient) garbage collection built into the Java environment. Ideally, such a system should also extend this garbage collection support by implementing transitive persistence (or persistence by reachability), and garbage collection of persistent space.

PSE supports a model of *persistence by reachability*. When an object is stored in a database, the transitive closure of all persistence-capable instances that are reachable from that object is also stored. If a persistent object refers to other persistence-capable objects, then PSE will migrate the referenced object into the database when the transaction is committed. For example, if a linked list consists of one or more chained *ListElement* instances (and *ListElement* has been made persistence-capable of *osjcfp*), then by storing the head of the chain in the database, all of the *ListElements* on the chain will also be stored.

An object becomes persistent, therefore, by one of the following methods:

- *Root Value*: If an object is referred to by a database root, it becomes persistent. Roots are the mechanism for naming objects in a PSE database. All root objects are persistently reachable, so these objects comprise the roots of the persistent reachability graph. There will generally be at least one root in a database since without any, there would be no way to navigate to any of the stored objects.
- *Reference from another persistent object*: If an object that is already persistent is modified to refer to an object that is not yet persistent, then the referenced object will become persistent when the transaction is committed.
- *Explicit migration*: An object can be explicitly migrated by a call to *COM.odi.ObjectStore.migrate*.

Two special cases must be considered when inspecting the fields of an object to compute transitive closure of reachable objects. First, references through fields that are declared with the Java *transient* modifier are ignored when the transitive closure is computed. The value of a transient field on a persistent object is set to its default value when the object is first read from the database (hooks are available that enable the application to reconnect a persistent object to other objects in the transient environment when an object is

fetches). The second special case is static fields. These are also ignored, because we treat static fields as fields of the Class object (which is an element of the transient runtime environment) rather than as fields of instances of the class. A warning is issued by the class-file post-processor if a static field is declared to be of a persistence-capable type, telling the user that if the static field is intended to denote a persistent object, then it is necessary to manually initialize the field, for example by making it a root object.

If a persistent object refers (through a field that is neither static nor transient) to an object that is not persistence-capable, then a runtime exception is thrown by *Transaction.commit*.

PSE must maintain the correspondence between an object in the Java Virtual Machine, and an object stored in the database. In general, the application programmer should not have to think about the fact that there are two copies of the object (several cases where this fact does become apparent through the PSE API are described in the next section, *Transactions*). A single persistent object should never be manifested in the application space as more than one Java object; nor should a single Java object be stored in the database as more than one persistent object. Therefore, if a persistent object is referenced by two different paths, the resulting references obtained must be equal. For example, if the same object is stored as the value of two different roots, *root1* and *root2*, the following expression will evaluate to true:

```
db.getRoot("root1") == db.getRoot("root2")
```

Similarly, if a transient object is reachable from two different persistent objects, only one persistent copy of it is stored in the database. PSE accomplishes this with an internal *Object Table*, which maintains a bi-directional mapping between a persistent object in the database and the Java object representing the persistent object. When a persistent object is referenced, before a new Java object representing that object is created for the application, the object table is checked to see if one already exists. If an entry in the Object Table is found using the object's persistent ID as a key, the corresponding element in the table is returned to the user rather than instantiating a new object. If the object is not found in the Object Table, then a new hollow object is created and inserted into the table.

The object table has entries for all persistent objects that have been accessed in the current transaction. In early releases of PSE, therefore, the object table became an obstacle to effective garbage collection of the Java VM. In the current release of PSE, however, the object table is implemented with weak references (on those Java VM implementations that support them).⁷ This allows the object table to be garbage collected, and any entries for objects to which the user program is not holding references are cleaned up.

There are two places where our goal of transparency and object identity is not fully met, however. The first is a relatively minor issue, alluded to in the previous section: PSE/PSE Pro does not preserve identity for certain objects that are instances of the Java wrapper classes, because it is more efficient to store these objects as values rather than as objects. Only *Longs* and *Doubles* are stored as separate distinct objects. Because identity is not preserved, programs that use object identity to compare wrapper class objects, while perhaps not good programming style, work differently when used with persistent objects. For example, this method is incorrect if it is applied to persistent Integer objects:

```
boolean comparePersistentIntegers(Integer x, Integer y) {  
    return (x == y);  
}
```

Instead, it should be written as:

⁷ A weak reference is a Java reference to an object that allows the referenced object to be garbage collected, provided that there are no other references (except weak ones) to that object. It is a concept taken from Smalltalk implementations. The JDK 1.1 for Windows and Solaris, and the SDK 2.0 from Microsoft, support weak references without the use of non-Java (i.e. native) code.

```
boolean comparePersistentIntegers(Integer x, Integer y) {
    return x.equals(y);
}
```

The second limitation in PSE's transparency implementation is more serious: the current release does not include a garbage collector for persistent space. This means that users must explicitly delete objects to remove them from the database. A persistent GC is under development and will be released shortly.

Transactions

All access to persistent storage is mediated by a transaction. By default, once the transaction ends, any persistent objects become inaccessible. Any attempt to use a reference to a persistent object outside of any transaction will cause *NoTransactionInProgressException* to be thrown. At the start of the next transaction, of course, persistent objects become accessible again. An application can then navigate back to any objects of interest by starting from a database root; or it can simply continue to refer to any persistent objects that were read in the previous transaction (provided of course that the application held on to references to these objects). In this second case, we say that the references to persistent objects have been *retained* across the *Transaction.commit*. In either case, the objects will be read back in from the database if they are needed. This behavior is a natural extension of the notion of object identity: a reference to a persistent object will continue to refer to the same object from one transaction to the next.

PSE also provides options for not retaining references (that is, for discarding object identity for persistent objects at a transaction boundary, by explicitly flushing the PSE object table); as well as for retaining not only the references to persistent objects (the object identity), but also the contents of persistent objects. If references are not retained, then any attempt to use a reference to a persistent object in a subsequent transaction will cause *ObjectNotFoundException* to be thrown (of course, it will still be possible to re-navigate to the object from a database root). If on the other hand both the references and the contents of persistent objects are retained, then the contents of any objects that were readable in the previous transaction remain readable even after the transaction commits.

If a Java application or applet has multiple threads, then they may need to either cooperate, or act independently from one another. PSE supports both modes of operation through a transaction *session*. If a thread needs to access a PSE database, it must call *COM.odi.ObjectStore.initialize*. There are two different overloads of the initialize method: the first overloading creates a new session, which acts independently from all other threads. That is, the thread is *isolated* from threads in other sessions, in the sense that the PSE concurrency control mechanism guarantees that any transaction in this session will be unable to see changes-in-progress from other sessions' transactions, and vice-versa.

The second overloading of the *ObjectStore.initialize* method takes a *java.lang.Thread* argument, and causes the calling thread to join the session of the argument thread. In this case, the threads are said to cooperate, and they can see changes in progress to persistent data. In fact, one thread could start a transaction and make some changes, the other thread could then call *Transaction.commit* to commit the changes to the database.

Example

The following simple program consists of two classes, a *UserConfig* class, which represents a simple user configuration, and a *Browse* class, which makes use of the *UserConfig* to retrieve and update a user's configuration. The *UserConfig* class has two fields, a vector of *java.lang.String*, and an *int* that holds an offset into the vector of *Strings*. Several methods are defined on *UserConfig*, including a default constructor for the class, a method that returns the last URL string in the vector, and a method that sets the last URL string in the vector. (A more complete definition might have other methods, to iterate over the entire history, prune selective portions, re-sort, etc. These functions are omitted for simplicity.)

```

import COM.odi.*;
import COM.odi.util.OSVector;

class UserConfig {
    int lastVisitedURLOffset = -1;
    OSVector history = new OSVector();

    String getLastVisitedURL() {
        if (lastVisitedURLOffset == -1) {
            return null;
        }
        else {
            return (String) history.elementAt(lastVisitedURLOffset);
        }
    }

    void setLastVisitedURL(String URL) {
        history.addElementAt(URL, ++lastVisitedURLOffset);
    }
}

public class Browse {
    static Database database;

    public static void main(String argv[]) {
        ObjectStore.initialize(null, null);

        database = Database.open("users.odb", ObjectStore.OPEN_UPDATE);

        Transaction.begin(ObjectStore.UPDATE);

        UserConfig config = findConfig(System.getProperty("user.name"));
        System.out.println("Last URL is " + config.getLastVisitedURL());

        <do some stuff>

        config.setLastVisitedURL(newURL);

        Transaction.current().commit();
    }

    static UserConfig findConfig(String name) {
        try {
            return (UserConfig) database.getRoot(name);
        }
        catch (DatabaseRootNotFoundException e) {
            UserConfig result = new UserConfig();
            database.createRoot(name, result);
            return result;
        }
    }
}

```

The program first initializes PSE and opens a database; then it calls *findConfig()* to look up or create a new *UserConfig* object representing the currently logged in user. Next, the program calls *UserConfig.getLastVisitedURL()* to retrieve and display the last element of the vector in the *UserConfig*

object. After doing some other unspecified operations, the *Browse.main()* method updates the "last URL" by calling *UserConfig.setLastVisitedURL()*. The final act of *Browse.main()* is to call *Transaction.commit()*, which writes any changes back to the database.

This program illustrates a few of the fundamental features of PSE: roots, API transparency, and transactions. Once a root has been retrieved, all subsequent access to persistent data is completely transparent. In the example above, the *UserConfig* class does not require any keywords, or source code modifications in order to store instances persistently in the database. Nor does the code that reads and writes the elements of a *UserConfig* (specifically, the *getLastVisitedURL()* and *setLastVisitedURL()* methods) require any special user annotations (e.g. fetch or dirty calls). The code creates and operates on persistent instances of *UserConfig* just as it would on transient instances.

The example illustrates the use of transactions as a scoping mechanism for all access to persistent data. Transactions are atomic all-or-nothing units of work. A transaction is started by calling *COM.odi.Transaction.begin()* with a transaction type (Update or Read-only) as argument. Persistent data may not be read from or written to the database unless a transaction of the proper type is in progress. Transactions are committed by calling *COM.odi.Transaction.commit()*, or they are aborted (rolled back) by calling *COM.odi.Transaction.abort()*. Transactions may not be nested in PSE. For simplicity, the example above has only a single transaction, which spans virtually the entire program execution. In a more realistic example, of course, smaller segments of processing would be bracketed by *Transaction.begin()* and *Transaction.commit()* calls.

If the *Browse* class accessed any of *UserConfig*'s fields directly, it would have to be made persistence-aware. As it is written, however, all accesses to persistent data are through method calls on *UserConfig* so only *UserConfig* needs to be post-processed. That is, because the *UserConfig* class encapsulates all access to its fields, and only exposes methods, there is no need to make any user of this class persistence-aware. Users of the class need not know whether a particular instance is transient or persistent, or even whether the class itself is persistence-capable.

Performance

In this section we present a simple implementation of the *oo1* benchmark⁸ using ObjectStore PSE for Java. Briefly, *oo1* has two classes, *Part* and *Connection*, which represent parts in a design and the connections between them. Each part has 3 connections to other parts. In each *Part* instance, we used an instance of the *OSVector* class to reference the 3 inbound and outbound connections. Each *Part* instance has the following fields: *id* (int), *type* (String), *x* and *y* (int), *time* (int), and *from* and *to* (*OSVector*) fields. Each *Connection* instance has a *type* (String), *length* (int), and *source* and *target* (*Part*) fields. For each *Part*, 90% of the *Connections* are randomly selected among the 1% of the parts with *id* values "closest," while the remaining 10% of the *Connections* are made to any randomly selected *Part*. We ran the benchmark with 1000, 2000, and 5000 *Part* objects (3000, 6000, 15000 *Connection* objects, respectively).

A Pentium Pro 200 Mhz machine with 32mb of memory was used. Two runs were made, one with the Sun JDK 1.1.2, and the other with the Sun JDK 1.1.1 JIT. For each of the three benchmark runs, the entire database was created in a single transaction. The create times shown below are from the start of the transaction to the end of the commit call, and include the database creation time.

Size	Create Time (secs) JDK 1.1.2, no JIT	Create Time (secs) JDK 1.1.1 with JIT
1000	13.8	9.25
2000	28.2	16.8

⁸ R.G.G. Cattell, J. Skeen, "Object Operations Benchmark," ACM Transactions on Database Systems, 17(1), March 1992.

5000 88.0 60.7

The "lookup" part of the benchmark builds a random array of N part id's, where N is the total number of parts in the database. Then, within a single transaction, it looks up each of those parts in the database, recording the time spent doing so. Since each part is stored in an OSHashtable, the lookup is a simple OSHashtable.get() call using the part id. It repeats the N lookups 10 times so that the effects of caching may be seen. The table below shows the "cold" time, which is the average time for the first N lookups, as well as the "warm" time, which is the average time per lookup for the fastest set of N lookups

Size	Cold time per lookup (msec)		Warm time per lookup (msec)	
	JDK 1.1.2	JDK 1.1.1 with JIT	JDK 1.1.2	JDK 1.1.1 with JIT
1000	1.31	.796	.141	.094
2000	1.38	.852	.148	.094
5000	1.54	1.03	.150	.103

Customer Feedback

PSE was made generally available in October 1996. It can be downloaded for free from Object Design's web site (www.odi.com). At this writing it receives more than 80 downloads per day. It is bundled with the Netscape Communicator 4.0 browser, the Symantec Cafe environment, the Asymetrix Supercede IDE, Borland's Latte environment, Natural Intelligence's Roaster IDE as well as other Java-based products. Because it is also distributed through other sources from which we do not receive download information (e.g. www.microsoft.com) we do not have complete information on the total number of copies in the field.

The deployed applications using PSE that we are aware of include:

- *US Open Golf Results*: During the US Open live commentary and results of each hole were stored in a PSE database as audio byte streams. These results were made available via telephone to press and other interested parties. This was a multi-threaded application.
- *Web Spider Applications*: Several user-directed web spider applications that use PSE to store results are known to exist. These programs generally query a user for a starting URL and then go out seeking and displaying information from that URL and the links it contains. The user can save information from the web page, or the URL itself, in PSE.
- *Servlet Applications*: PSE is being used in servlet applications for applications ranging from storing user configuration information to storing newspaper content for redistribution via the web. The size of the data ranges from a few thousand bytes to tens or hundreds of megabytes.

Because PSE was designed to be an entry-level persistence engine, it was an important goal that it be easy to use, and require little support and maintenance. The number of downloads that we have seen, and the high level of traffic on the [pse-java-discussion](http://pse-java-discussion.majordomo.org) majordomo list (set up for the purpose of discussing PSE usage and issues), together with the low level of support events (fewer than 5 per day) indicate to us that this goal has been achieved, and that most users are able to use the product successfully with little or no assistance.

There have been, however, several problem areas in the product, which have led to user difficulties and support events. These include:

- *classpath and classfile re-writing*: The most common source of confusion for new users centers around the use of post-processed classes in the development and runtime environments. Because post-processor reads in .class files and writes out new ones into another directory hierarchy, the runtime classpath must specify the annotated .class file

hierarchy ahead of the original "source" hierarchy (otherwise the classes will not appear to be persistence-capable or persistence-aware at runtime). Some users have tried to correct this by copying the annotated .class files back into the source hierarchy, which unfortunately can cause further problems if they later modify, recompile, and attempt to re-post-process these files. To address this program, we are considering modifications to the post-processor to be able to annotate .class files "in-place".

- *classfile post-processing mechanics*: The classfile post-processor technology requires that all .class files for a program be post-processed at once. (Although this is not a strictly correct statement of the requirement, for many simple applications this is the effective result.) If all of the classes are not postprocessed *en masse*, then persistence-aware classes may not be aware of all of the persistence capable classes that they may encounter, and runtime errors can result, because persistent objects may be accessed without the appropriate *fetch/dirty* calls. We're considering the addition of runtime checks to detect if a user has made this mistake.
- *integration with RMI*: Both PSE and RMI are most easily used by employing inheritance from a special base class (*COM.odi.Persistent*, in the case of PSE, and *java.rmi.server.UnicastRemoteObject*, in the case of RMI). This means that using the two systems together requires some extra work. It is possible to make a class persistence-capable in PSE without inheriting from the Persistent base class, but only by manually annotating the class (the automated generation of persistence-capable classes by our post-processor currently inserts this base class). It is also possible, of course, to use RMI without inheriting from *UnicastRemoteObject*, but again this requires extra coding to accomplish. We are in the process of updating our post-processor to implement persistence through the *IPersistent* interface, rather than the *Persistent* base class, to simplify integration with RMI and other packages.

Future

Future enhancements to the product might include the following:

- *Associative Queries, Indexes*: While PSE supports the JGL (Java Generic Library from ObjectSpace, Inc.) it provides no built-in support for queries or indexes. Many of our users who have asked us for this believe that it is important enough to warrant the significant increase in footprint that it implies.
- *Schema Evolution*: Applications that have deployed with PSE to date are relying on dump/reload mechanisms for dealing with schema evolution. Built-in support for schema evolution is one of our most common enhancement requests.
- *Persistent Garbage Collection*: We intend to release an updated version of PSE that includes a persistent garbage collector in the near future, as discussed above in the section on ***Memory Management***.

Conclusions

The simplicity and flexibility of Java makes it a compelling choice for new application development in a wide variety of product domains. Our goal in the development of PSE was to extend the Java environment with a persistence capability that is easy to use and flexible in terms of underlying storage system implementation. The most challenging aspects of this development have been in building our implementation of transparent persistence in such a way that it melds cleanly with the existing Java environment and tools (including unmodified Virtual Machines), in the creation of a single API that could serve as the front end for a set of storage systems of quite different implementation and capability. Despite these challenges, we feel that we have been largely successful in our fundamental goals.