



Sun Microsystems Laboratories

*Observations on Persistent Object Systems
from a (would be) Developer*

Mick Jordan

Sun, Sun Microsystems, Java, Java Blend, JDBC are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Eighth International Workshop on Persistent Object Systems

Outline

- **The Forest Project**
- **The Pain of "Do It Yourself" Persistence**
- **The Forces against Persistent Object Systems**
- **Comparing Persistent Object Systems**
- **Conclusions**

The Forest Project

JP - Software Development Environment (SDE) for Large Scale Java™ Systems

- **Configuration management, version management**
 - **Large-scale system building (software manufacture)**
 - **Integrated authoring (editing) tools**
 - **Based on the Vesta approach (Compaq SRC)**
-
- **PJama - Orthogonal Persistence for the Java language (OPJ)**
 - **persistence orthogonal to type**
 - **including Class and Thread**
 - **persistence by reachability**
 - **standard Java language rules for automatic storage management**
 - **persistence independence**
 - **code is unchanged, dynamic class loading unaffected**

SDE Infrastructure and Integration Mechanisms

- **Current Practice**

- file system - flat files, directories and (perhaps) symbolic links

- **Problems with File Systems**

- impedance mismatch
- loss of type safety
- lots of hard implementation problems
 - type mapping
 - concurrency control
 - storage management
- file manipulation is the "C programming" of persistent storage

- **Persistent Objects**

- in principle solve all of these problems
- so why are PO-based systems not widespread?

The JP Version Interface

```
interface Version extends Ancestry, Container, Naming,
    TimeStamped {
    Version checkout(String name, boolean reserve);
    Version advance(Mutability obj);
    Version branch(String name);
    Version checkin();
    Mutability getObject();
    Version addComment(Object comment)

    // other methods proving navigation, reflection, e.g.
    Version getMainBranch();
    String getName();
```


A Tale of Three Implementation Approaches

- **C++ and ObjectStore**
 - "Software Configuration Management in an Object-Oriented Database", COOTS95.
 - 950 lines of Objectstore-specific code
- **Tcl/Tk and the Unix file system**
 - JP-lite, used internally by group members
 - supported own development
 - supporting JP on PJama implementation
 - 1350 lines of Tcl
- **Orthogonal Persistence for the Java language (OPJ)**
 - 100% pure Java implementation
 - 550 lines of persistence independent code

Implementing the Version Interface: Design Issues

Design Issues

- **Object Storage Mechanism**
- **Concurrency Control**
- **Performance**
- **Scalability**
- **Portability**
- **Ease of Implementation**

Version: Simple File System Implementation

- **Design choices**

- map each `Version` node to a directory
- keep additional state in a hidden (dot) file
- lock entire tree when mutating

- **Problems**

- file systems have limited and different functionality
 - e.g. no symbolic links or no links to directories
 - directory scanning and open file calls are expensive
- programming is tedious
 - pathnames as object references, lots of pathname (string) manipulations
- manual (disk) storage management
- implementation is exposed
 - **BAD:** easy to violate system invariants
 - **GOOD:** enables (some) inter-operability with legacy applications

Version: More Complex File System Implementation

- **Design Choices**

- "Do it yourself" (DIY) object storage
 - define and implement a file format
 - manually map objects to byte sequences
 - keep leaf objects as files?

- **Problems**

- tedious, error-prone, manual programming
- concurrency control more difficult
- difficult to evolve the design
- must handle caching explicitly for scalability
- manual disk storage management even harder

Version Implementation: Does Object Serialization Help?

- **Virtues**

- easy to use
- can handle complex, fine-grained, object structures
- simple concurrency control, atomic update model

- **Problems**

- inherently not incremental
 - "big inhale/exhale" too expensive
- so data must be partitioned
 - have to worry about (too much) reachability
 - re-introduces concurrency control, atomic update problems
- types that do not implement `Serializable`
- does not handle `static` fields (lack of orthogonality)
- does not preserve behavior (method code)

Version Implementation: OPJ

- **No unnecessary programming**
 - focus on the essential logic
- **Full power of the Java language available**
 - arbitrary data structures
 - fine-grain concurrency control
- **Scalability**
 - transparent, incremental, data transfer, object caching implicit
- **(Disk) Garbage Collection**
 - normal Java language rules
- **Implementation encapsulated**
 - **GOOD:** hard to violate invariants
 - **BAD:** hard to integrate legacy applications

The Slippery Slope of DIY Persistence

- **Simple, manageable implementation (first system)**
- **Pressures arising from initial deployment**
 - scale
 - concurrency
 - performance
- **Complex, fragile implementation (second system)**
 - up to 30% of code handling persistence
- **But too much investment to abandon implementation**

Forces working against POS adoption

- **The Strongly-typed/Weakly-typed schism**
- **Ubiquity of File systems**
- **Immaturity of Persistent Object Systems**

Strongly-typed/Weakly-typed Data Schism

- **Strongly-typed data**
 - considered *essential* for POS
 - all access to the data checked by the type system
 - aspects of the data format hidden by the type system (meta-data)
- **Weakly-typed data**
 - permits multiple interpretations
 - data and meta-data format often explicitly defined and accessible
 - allows easy experimentation with new "type systems" (e.g. XML)
- **Current Status**
 - strong-typing the accepted norm for transient data in application programs
 - weak-typing the accepted norm for persistent storage (except DBMS)
- **Is this a temporary anomaly?**

Ubiquity of File systems

- **Comes free with the operating system**
- **Good enough for a lot of applications**
- **Denial of the "slippery slope" of persistence solutions**
- **Blind acceptance of the "read, compute, write" model**
- **Everything depends on the file system**
 - e.g. ODBMS, RDBMS configured via file system
 - the Java platform is dependent on the file system
- **The File/Directory abstraction is not fundamental**
 - it should be subsumed by a POS and provided as a core type

Immaturity of POS

- **Lack of an agreed, widely-deployed, standard model**
 - ODMG/OMG
 - Can the Java language fill this role?
- **Programming Complexity**
 - existing models require overly complex mappings to programming languages
- **Operational complexity**
 - too many tuning knobs - cf File systems
- **Fundamental deficiencies**
 - inadequate solutions for type (schema) evolution
 - inadequate models for sharing, concurrency control
 - inadequate model of distribution

Comparing Persistent Object Systems

- **Establish criteria for comparison**
- **Choose applications/benchmarks to measure criteria**
- **Work in progress**
 - **qualitative analysis only**

Criteria for comparison

- **Type Orthogonality**
- **Persistence Independence**
- **Reusability**
- **Support for Change**
- **Performance**
- **Scalability**
- **Transaction Support**
- **Operational Complexity**

Applications/Benchmarks

- **MQ: A simple message queue**
 - "minimal" test
- **OO7: Standard Benchmark from U. Wisconsin**
 - ported to the Java language by Laurent Daynès
- **JPCMS: Configuration Management Simulation**
 - exercises the actual JP configuration management system
 - simulates user-activity

Systems under Comparison

- **PJama**
- **Gemstone/J**
- **Object Design PSE**
- **Java Object Serialization**
- **Object-Relational Mapping Systems**
 - **Ardent Java Binding**
 - **Sun™ Java™ Blend™**

Type Orthogonality

- **What types can be made persistent?**
- **Only PJama aspires to complete Java language orthogonality**
 - current system fails to achieve it as `Thread` cannot be persistent
- **All the other systems mix in another object model**
 - ODMG, RDBMS influences
 - Transaction model pervades all persistence
 - Semantics often unclear
- **Class reuse limited by compatibility with mixin object model**
 - may limit use of Java platform classes
 - islands of reuse
- **Complete type orthogonality needs a platform-level solution**

Persistence Independence

- **Does code, source or binary, have to be modified?**
- **No system achieves complete persistence independence**
- **Bytecode post-processing is popular**
 - significantly affects software development process
 - may fail to deal with core classes or dynamically loaded classes
 - processed bytecodes are not reusable in standard Java environments
- **Managing *external* state must violate persistence independence**
 - new code must be written to restart correctly

Reusability

- **Is the code reusable in another context?**
- **Systems that process code limit reuse**

Support for Change

- **How easy is it to evolve applications?**
- **Evolution or Versioning?**
- **Significant impact on the POS implementation**
- **Limited support in systems compared**
- **Unclear how well Object-Relational Mapping systems will handle this**
- **Lack of support often cited as a reason for DIY persistence**

Performance

- **What is the impact on performance?**
- **Ways that performance can be affected**
 - execution time
 - persistent classes
 - non-persistent classes
 - memory usage
- **Measures of interest**
 - total increase over complete application execution
 - distribution of additional cost over time
 - orthogonality tends to distribute the cost more
- **Layered approaches can add a factor of ten to execution time**

Scalability

- **How many objects? What is the minimum footprint?**
- **Layered systems stress the standard storage management system**
 - particularly weak references
 - size of POS classfiles not a very useful measure
- **Java Object Serialization has inherent scaling problems**
- **Little focus on minimum footprint**

Transactional Support

- **Is a transaction API a fundamental requirement?**
 - all systems except PJama assume an external object store that is accessed by multiple processes (Java Virtual Machines)
 - => Java Virtual Machine not in complete control
- **In PJama persistence and transactions are separated**
 - transactions acts on a subset of the objects controlled by a VM
 - Java Virtual Machine is in complete control
 - transaction implementation can leverage orthogonal persistence
- **Transaction support is usually simple ACID model**
 - PJama is trying to provide additional models
 - e.g. nested, not strict isolation

Operational Complexity

- **How hard is it to install and deploy the system?**
- **Layered systems suffer combinatorial effects**
 - combination of JDBC™ driver, persistence layer and underlying DBMS
- **Some systems have a lot of knobs and dials**

Conclusions

- **One well-defined model with a direct implementation is preferable to a separate model/language mapping**
 - The Java language is good enough in the SDE application domain
- **Need direct operating system support for a POS**
 - subsume the file system
 - break the "read, compute, write" model
- **Current Java platform POS support is good and improving**
 - can we get orthogonal persistence for the Java language accepted?
 - need a good, well integrated, solution to the evolution problem
- **Will Object-Relational databases take over anyway?**