



Sun Microsystems Laboratories

SML 99-0410

*PJama: Orthogonal Persistence
for the Java™ Platform*

Michael L. Van De Vanter

The Forest Project, Sun Microsystems Laboratories
<http://www.sunlabs.com/research/forest/>

September 1999

Outline

- **Motivation for Orthogonal persistence (OP)**
- **Principles of OP**
- **OP for the Java™ programming language (OPJ)**
 - **Language semantics**
 - **Checkpoints and transactions**
 - **Handling external state, AWT, RMI**
 - **Support for change (Evolution)**
 - **Performance**
- **OPJ status & PJama prototype**
- **Applications**
- **Future plans**



Motivation for Orthogonal Persistence

- **The majority of software applications manage long-lived data**
- **Two approaches are dominant today**
 - File systems
 - (Relational) databases
- **These solutions have drawbacks**
 - more than one programming model
 - complex interactions between programming models and subsystems
 - potential loss of type safety at boundaries
 - impedance mismatch with object-oriented models
- **OP addresses these drawbacks**
 - provides a single model for all aspects of application development

Relative Size of DBMS Markets in 2005¹

<i>Query</i>	<i>Relational DBMS</i>	<i>Object-Relational DBMS</i> <i>150</i>
	<i>100</i>	<i>Object-Oriented DBMS</i> <i>1</i>
<i>No Query</i>	<i>File System</i>	<i>Persistent Programming Languages</i> <i>?</i>
	<i>?</i>	<i>?</i>
	<i>Simple Data</i>	<i>Complex Data</i>

1. Derived from Object-Relational DBMSs, The Next Great Wave, Michael StoneBraker, Morgan Kaufmann, 1996

What is Orthogonal Persistence? - What and How

- **OP aims to provide a *single* model for:**
 - data structure definition and behavior
 - data and program storage
 - application development
- **Possible approaches**
 1. design a persistent computational model from scratch (Napier88, Tycoon)
 2. evolve an existing database model to encompass computation (SQL3)
 3. elide two independently developed models (ODMG Binding)
 4. evolve an existing programming language to accommodate persistence (OPJ)
- **We saw a good opportunity for approach 4**
 - type safety
 - automatic storage management
 - runtime extensibility

The Principles of Orthogonal Persistence

- **Type Orthogonality**
 - persistence is available to all objects regardless of type
- **Transitive Persistence**
 - objects lifetime determined by reachability from a designated set of root objects
- **Persistence Independence**
 - it is indistinguishable whether code is operating on short-lived or long-lived data

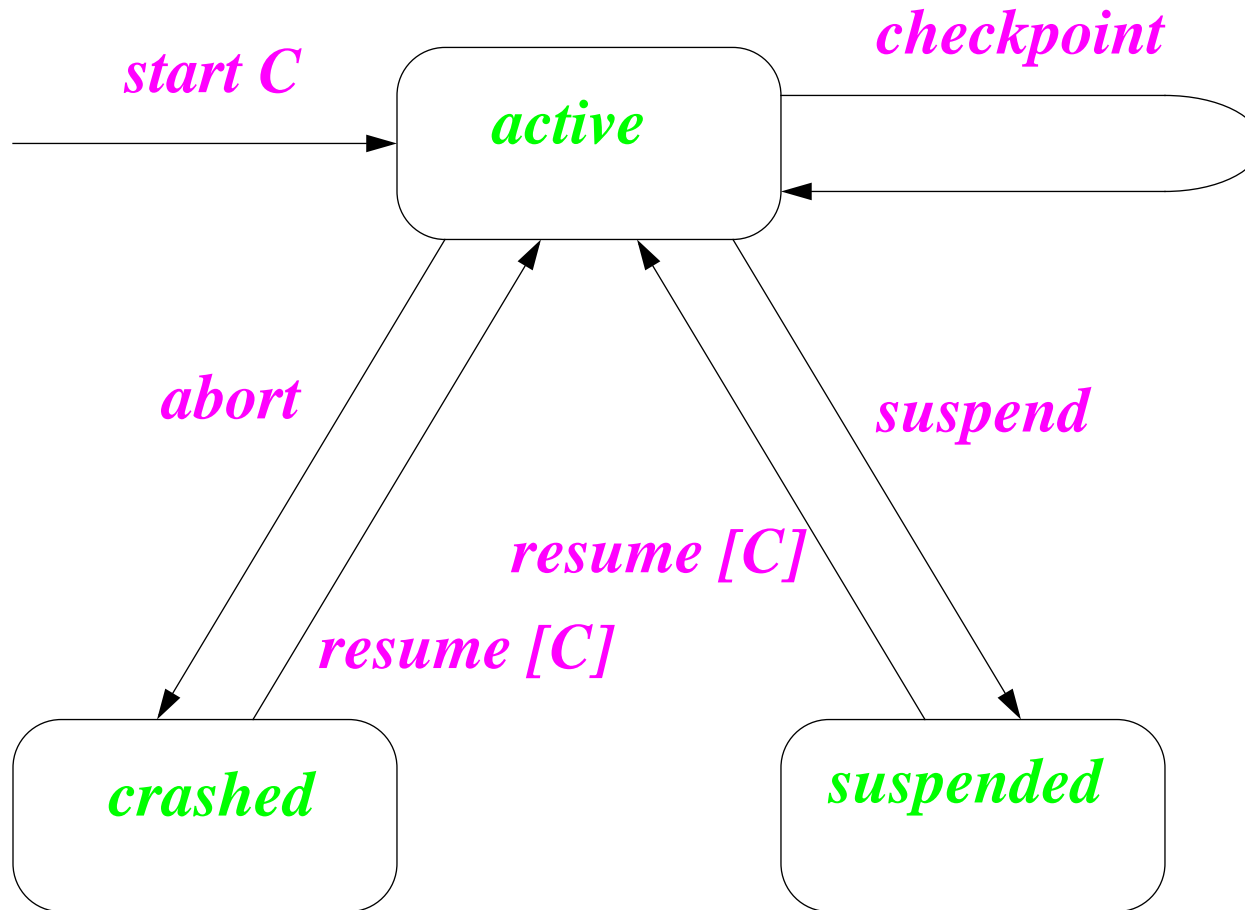
Observation

- The principles leave room for variation when applied to a *particular* language

Applying the Principles of OP to the Java programming language

- **Type Orthogonality**
 - classes requiring special support:
 - `Thread`, `Class`, `ClassLoader`, `Exception`
 - `static` variables
- **Transitive Persistence**
 - class reachability
 - how hard to try to guarantee referential integrity?
 - roots of persistence
 - explicit or implicit?
- **Persistence Independence**
 - is classfile processing in the spirit of OP?

OPJ Virtual Machine Life-Cycle



Transactional Properties

- **VM Transaction**
 - unit of computation between two checkpoints
- **VM Transactions conform to the ACID properties**
 - ***Atomic***: a VM transaction's changes to the computational state either succeeds completely or has no effect
 - ***Consistent***: A VM transaction is a correct transformation of the computational state with respect to the semantics of the programming language
 - ***Isolation***: Even though VM transactions may execute concurrently, it appears to each VM transaction, T, that others executed either before or after T, but not both.
 - ***Durable***: Once a VM transaction completes successfully, its changes to the computational state must survive system failures.

Limits of Orthogonal Persistence - Handling External State

- **External state not under the control of OPJ system**
 - affects many core classes, VM implementation, some user code
 - source of almost all user problems - exacerbated by transitive persistence
- **Correct handling requires programmer foresight**
 - code *cannot* be persistence independent
 - programmer must think in terms of *checkpoint/resume* not *start/finish*
- **Two techniques for handling external state**
 - the `transient` modifier plus explicit checks on (cache of) external state
 - but `transient` has been misused by serialization
 - callbacks on critical system events using standard event model
 - `RuntimeListener`, `RuntimeEvent` interface
 - `checkpoint`, `resume`, `abort`

Persistence and AWT/Swing

- **Native graphics system is an example of complex external state**
 - AWT peer state is external to JVM
 - hard to retrofit for OP
- **Swing simplifies the task**
 - components maintain their state in the language runtime, not in a peer
 - model-view separation, pluggable look and feel
 - provide helpful separation between GUI abstraction and device specifics
 - => abstraction persistent, device specifics transient
 - minimal changes needed for OP
- **Swing demo application**
 - code is unchanged - orthogonality
 - rapid restart to state when last executed

Persistence and RMI

- **RMI also maintains significant external state**
 - connection information, socket state, exported objects
- **PJRMi supports persistent RMI objects transparently**
 - persistence of server-side remotely-invokable objects
 - automatic re-exportation on first use after server resume
 - persistence of clients-side stubs
 - automatic re-establishment of connection on first use after client resume
 - easier to use than RMI Activation
- **Further work**
 - distributed checkpoints
 - transparent caching of remotely-invokable objects at client

Support for Change (Evolution)

- **Class evolution**

- all applications have this problem. However:
 - strong consistency guarantees of OPJ prevents ad hoc solutions
- PJama prototype provides a set of off-line tools
- OPJ needs runtime evolution facilities
 - extend the reflection interface?

- **Platform evolution**

- Virtual Machine, core classes
- classes that support the platform and the user
 - VM dependencies in persistent store representation
- classes with native methods
- Archive/Restore

Performance - Costs and Benefits

- **OPJ extends automatic storage management to long-term (disk) storage.**
 - continuous overhead of orthogonality versus bursty overhead of file or external database access.
 - significant engineering challenge to provide good performance
 - continues the trend of spending cycles to simplify software development
 - PJama best-case is currently a 10-15% slowdown
- **OPJ enables “real” OO programming**
 - “read, compute, write” model is a form of explicit cache management
 - can OPJ approach same performance level with automatic cache management?
- **Total system cost is an important measure**
 - crossing domain boundaries is very expensive!
 - are 3-tier architectures a virtue or just a necessity due to the lack of OP?



Performance - Optimization Opportunities

- **Reduce overhead caused by READ and WRITE barriers**
 - remove redundant barriers (cf array bound checks)
 - requires cooperation with object eviction manager
 - must ensure residency of objects while optimized code is executing
- **Exploit the longevity of OPJ applications**
 - a checkpoint/resume preserves language meta-data (e.g. class structure)
 - machine-independent optimizations still valid
 - e.g. inline analysis, virtual method selection
 - can optimize off-line
 - can preserve optimization analysis
 - using OPJ itself if JIT is written in the Java programming language
 - optimization of object structures
 - field layout
 - removing levels of indirection

Status of OPJ/PJama

- **OPJ submitted as a Java Specification Request**
- **PJama Classic (available externally under license)**
 - versions for JDK 1.1.7, JDK 1.2FCS
 - stores up to 2GB
 - explicit checkpoint model for persistence
 - persistent RMI, persistent Swing UI components
 - good off-line class evolution facilities
 - no persistent threads
- **PJama on Java 2 SDK Production Release for Solaris™ OS**
 - main goal: JIT integration
 - secondary goal: complete orthogonality, especially persistent threads
 - new persistent store layer; store sizes > 16GB



Applications using the PJama prototype

- **JP prototype software development environment**
 - scalable development approach with versioning and configuration management
 - CM repository stored in OPJ (no files)
 - builder (no *make*) and all tools run in OPJ
- **geographical information systems (terrestrial & marine)**
- **intelligent WWW proxies**
- **many others in research community around the world**
- **experience shows**
 - small amount of modifications necessary to port into OPJ
 - modifications do not grow with size of application
 - requires a concept shift in approach to application development

OPJ Continuing Research Topics

- **Storage Management**
 - garbage collection for multi-gigabyte stores
 - object eviction
- **Class Evolution**
 - how to integrate evolution with a live system?
- **Optimization**
 - exploit the benefits of persistence and reduce its costs
- **Beyond OPJ**
 - Extensible Transactions as application framework
 - beyond flat ACID transactions - nested, long-running, user-defined, etc.

Conclusions

- **OPJ works**
- **Handling external state is important because of the open nature of the Java platform**
- **More work is needed on the model for evolution**
- **OPJ + Extensible Transactions an exciting new platform**
- **Significant implementation challenges remain**

More Information

- **The Forest Project**

- <http://www.sunlabs.com/research/forest/>
- papers & public presentations
- download PJama prototype

- **Java Specification Request (JSR)**

<http://java.sun.com/aboutJava/communityprocess/jsr.html>