

The Java™ platform as a database

Mick Jordan

*Sun Microsystems Laboratories, M/S UMTV 29-112, 901 San Antonio Road,
Palo Alto, CA 94303 USA*

Email: mick.jordan@sun.com

Abstract

It is argued that the provision of orthogonal persistence for the Java™ platform (OPJ) provides a database suitable for a wide range of applications. The OPJ specification is briefly described and its properties discussed, including the unique support for persistence of code and execution state. It is shown how OPJ provides basic support for queries.

1 Introduction

In the call for papers (CFP) for this workshop it states that 'Real-world objects can easily be modeled in a Java program or applet. However, most applications require a persistent storage medium like a database'. The Forest project at Sun Microsystems Laboratories, in collaboration with the Persistence and Distribution group at the University of Glasgow, is tackling this need by developing orthogonal persistence for the Java platform (OPJ) [1]. OPJ is a persistence option that is partly characterized by there being no visible or separate database from the perspective of the programmer. OPJ is a *language-centric* approach to persistence, where the Java virtual machine is responsible for managing the entire storage hierarchy from the hardware registers to stable storage on disk or flash memory. This allows the programmer and the end-user to view the Java platform *as a Database*.

As we shall discuss below, this approach provides many benefits. However, many people would dispute whether such a Java platform really counts as a database, on the grounds that a database, by definition, provides additional features that are missing in the Java platform. In particular, a database is expected to provide a query facility, durability and, in general, support for controlled concurrent access by multiple users. However, as also noted in the CFP, the adoption of the Java platform 'can result in cultural changes in the way that things "get done"'.

The Java platform does, in fact, provide facilities that support or enable querying and multiple applications, and it is worthwhile to investigate how far these go towards fulfilling the requirements of modern applications. In particular, it is one of our goals of our research to discover whether it is possible to operate solely in a universe of persistent objects, where file systems and perhaps even relational databases are implemented on top of objects rather than vice versa as is typical today. To achieve this requires the successful integration of programming language, database and operating system technology, and is unusually challenging because it disrupts the status quo in these fields.

This paper briefly outlines the OPJ specification and contrasts it with other persistence mechanisms for the Java platform. We then discuss some of the implications of a single-language approach. This is followed by an analysis of the unique features of OPJ, namely the persistence of code and execution state. Then we discuss how queries can be handled in OPJ followed by our conclusions.

2 The OPJ Specification

The OPJ specification [14] is a ten-page document that includes a very small API, most of which is only needed for specialized code that deals with managing state that depends on values external to the Java virtual machine. The reason that it is possible for the specification to be this small is that our goal is one of minimal perturbation to the semantics of the Java programming language. Essentially, the OPJ model provides both the user and the programmer with the illusion of a continuously executing system, in the face of planned and unplanned shutdowns.

Orthogonal persistence [3] is characterized by three simple principles:

- **Type Orthogonality:** persistence is available for all objects irrespective of type.

- **Persistence by Reachability:** the lifetime of each object is determined by reachability from a set of *root* objects.
- **Persistence Independence:** code is identical whether it is operating on short-lived or long-lived objects.

The OPJ specification defines how these principles are applied to the Java programming language and the Java virtual machine. The specification derives from earlier research into persistence and the experience gained over several years with a succession of prototype implementations. It has also been influenced by other persistence models for the Java platform, but these models were most helpful in demonstrating the problems that arise from a lack of orthogonality rather than contributing directly to the specification. In general, our experience has led us to simplify the specification and move closer to the fundamental computational model of the Java programming language, rather than expanding the API. Consequently, in contrast to many APIs for the Java platform, the specification does not attempt to embrace a wide range of existing persistence mechanisms. Rather, in the same spirit as the Java specification [11] (JLS), it attempts to simplify a complex area by providing support at the Java platform level.

The general intent of the OPJ specification is that the complete computational environment of an executing OPJ virtual machine can be made persistent, in the form of a *suspended computation*, by an operation referred to as a *checkpoint*, and then *resumed* at some later time, possibly on a different machine. Except where explicitly provided for in the specification, it is intended that the checkpoint operation be transparent to an application. The specification does not prescribe the mechanism by which a checkpoint is achieved in order to provide flexibility of implementation. However, it is expected that the checkpoint operation will have a latency that is proportional to the amount of data modified since any previous checkpoint, independent of the total amount of reachable data.

The concurrency facilities of the Java programming language and the binding of code and data that are characteristics of object-oriented languages have had a profound effect on the evolution of the specification. Initially we took a fairly traditional “data-centric” approach, which is also common in most object-oriented database management systems. For example, we required applications to register persistent roots in a special table. As a result, the reachability rules for persistent objects were different from those for transient objects, for which the rules are essentially thread-based. Partly this was due to a concern about too many objects and classes becoming persistent. In practice, it turns out to be very hard to control persistence by reachability in this way. Furthermore, provided the orthogonality is complete, it is not of practical concern because, once the quantity of persistent classes and objects becomes significant, as it soon does for serious applications, transient objects are in the noise. This permits us to use the standard rules in the JLS for reachability, resulting in a smaller specification and a simpler world for the application programmer. It also results in a focus on the persistence of computation rather than just the persistence of data, as shown by the previous paragraph. However, as we discuss later, implementing persistence of computation (threads) remains an unrealized goal.

The area that we have not yet converged upon is the role of transactions [12] in an OPJ system. Our first paper [2] contained a proposal for a sophisticated extensible transaction model, that has not been implemented in any of our prototypes to date. One problem with adding transactions is the conflict between the implicit concurrency control that is typical in transactions and the explicit control that is provided by way of the `synchronized` statement in the Java programming language. A further problem is the semantics of transaction abort. Transactions are a semantically gray area in most of the persistence mechanisms for the Java platform, including the ODMG specification. For example, it is typical for only the state of persistent objects to be rolled back in the case of an abort. The programmer must handle the restoration of transient objects. This is unacceptable from the OPJ perspective because it violates orthogonality. In consequence, we chose not to include a transaction API in the initial OPJ specification. However, we do define that the behavior of the OPJ virtual machine as a whole must obey the ACID semantics so that, for example, a checkpoint is atomic and durable. It is also consistent in that the sense that it guarantees that the stored data conforms to the JLS (implicit constraints) and any implemented application invariants (explicit constraints).

The absence of a transaction API does not preclude concurrent applications in a OPJ system because of the provision of threads. However, there are some known problems with simulating multiple applications with threads, notably problems of unintentionally shared global state, thread termination, resource control and access control, some of which can be solved by a transaction framework. Currently, it is only possible to deploy applications that are known to be well-behaved in these respects. Our conjecture is that there are enough applications of this nature to justify our position, for example, essentially all personal productivity applications. Meanwhile, we are continuing to work on the extensible transaction framework in the context of a single virtual machine. One of the interesting research questions is whether the framework can be efficiently implemented on top of an orthogonally persistent Java platform, leveraging the persistence of threads.

3 The Single-Language Approach

The computer industry is very effective at generating new programming languages, but less effective at creating really good ones. However, occasionally a language comes along that hits a sweet spot and achieves a large and widespread following, like C or SQL [16]. For example, C is the de facto operating system interface language in use today. The existence of SQL is often argued as one of the reasons that relational databases became so successful.

The Java programming language has achieved similar success in a relatively short time. In the field of object-oriented languages, it is fast approaching dominance, despite, or perhaps because of, the fact that it is not a pure OO language like Smalltalk. Because of this dominance, it is reasonable to ask whether the Java programming language might become the de facto language for persistent objects, thus making redundant efforts like ODMG [4], which defines a separate language and programming model that must be mapped to a concrete programming language. We have argued elsewhere [15] against the complexities and inconsistencies of systems that are based on mapping between two languages. Our experiences with actual systems support this view. In practice the claimed transparency is incomplete and the devil is in the details. Ultimately the rationale for mapping disappears if the mapped-to language is both equivalent to the mapped-from language and also the dominant target. In the case of ODMG, several features of the target languages are removed from the mapping, at least in the support for persistence. For example, ODMG has no concept of a thread, and so threads cannot be made persistent.

The principal benefits of the single-language approach are the uniformity, consistency and high-leverage that it provides. The consistency attribute is particularly important in the persistence domain and is a major feature of traditional databases, as seen by it being on of the ACID properties. The Java programming language provides a basic layer of consistency by way of its strong type system and support for automatic memory management. We would argue that if automatic memory management, especially garbage collection, is beneficial for programming with transient objects, then it is even more beneficial for persistent objects. The implementation challenges are harder because of the increased scale and the performance characteristics of secondary storage. Perhaps because of the C++ legacy, many of the object-oriented database products do not support garbage collection and allow the explicit deletion of persistent objects.

The increased leverage arises from the ability to reuse existing software components, for example arbitrary third party Java classfiles. The OPJ principle of persistence independence is directed at supporting this kind of reuse. In OPJ, any class can be made persistence, with no changes to either the source code or the Java classfile. This allows the full power of the Java programming language to be utilized, and the code to be reused even in a non-persistent context. We have direct experience of taking classes from a third-party and immediately using them unchanged in an OPJ application. This capability of the Java platform is very valuable and should continue to be available in the domain of persistent applications. Many of the persistence mechanisms for the Java platform transform either the source code or the Java classfiles, which remedies the code unusable in a different environment. In our view, this violates the principle of persistence independence.

4 Persistence of Code and Execution State

Because in the Java programming language `Thread` and `Class` objects are exposed to the programmer as objects, it is possible to write code that uses them in ways that can be difficult for a persistence mechanism, for example storing a `Thread` or `Class` instance in a field of a persistent class.

4.1 Persistence of Code

We use “code” as a shorthand for all the metadata associated with a class, including the `Class` instance and the method bodies. Most persistence options remain very “data-oriented” in their approach to the persistence of code, disallowing the persistence of `Class` instances and separating the fields of an object instance from its methods in the persistent state. This is unfortunate because it puts at risk one of the benefits of the Java programming language, namely the type safety and consistency that derives from maintaining the binding between an object instance and its class. In contrast, OPJ maintains the binding because to not do so would violate the language semantics, given the model of continuous computation. Some systems do go as far as allowing the Java classfile contents to be stored in and reloaded from the database, but this is not enough to guarantee consistency on restarts, nor does it preserve object identity for `Class` objects. In fact the resumption semantics of OPJ set it apart from all other approaches. In non-OPJ systems, the programmer, and sometimes the user, understand that the program and the database are separate entities. The database only contains passive data which is somehow mapped into objects after the program “opens” it.

In OPJ, the database is opened by the OPJ virtual machine, and the program resumes execution from the last successful checkpoint. There is a special case where the database contains no active threads, in which case the database *is* passive, but the bootstrap classes and all their static variables are as they were at the time of the checkpoint. In this case, to activate the database, a new main class must be activated in a new thread of control, much like the normal Java virtual machine start-up.

4.2 Persistence of Threads

Persistent threads offer many benefits, not least the ability to resume a query that was in mid-progress when the system crashed. OPJ is the only Java platform that aspires to supporting persistent threads, but it is a major challenge to implement with current virtual machine and operating system infrastructure. It is relatively easy to achieve in simple, low-performance, virtual machines, for example, by executing the JavaInJava virtual machine [20] on top of an OPJ virtual machine. However, we have so far been unable to achieve persistence of threads in the context of production quality virtual machine technology. The problems are split about equally between the Java virtual machine and the underlying operating system's native thread support. In general, the Java virtual machine must maintain exact information about the contents of stack frames and registers, and the operating system must provide accurate and timely information about thread state. Current Java virtual machines generally support exact information up to what is required by garbage collection, and fall short of support for persistent threads. It is technically feasible to finesse the virtual machine aspect by checkpointing raw machine state, but this gives up on architecture-neutrality and prevents certain operations, such as evolution (4.3). Unfortunately, little can be done about an uncooperative operating system.

The inability to make threads persistent is directly related to whether OPJ can support multiple, independent threads (applications) in the same OPJ virtual machine. For example, imagine that one thread is midway through a synchronized block when another thread calls for a checkpoint. If threads, and their associated locks, can be made persistent, this is not a hazard. While it is true that the checkpoint will contain an object in an inconsistent state from a database perspective, the fact that the object is still protected by the lock and that the thread will resume and eventually release the lock causes no semantic difficulty. However, if threads are not persistent then the checkpoint will contain the object in an inconsistent state with no hope of this ever being remedied, which is disastrous. This is analogous to the reasons behind the deprecation of `Thread.stop` [19]. As a result, the absence of persistent threads requires that all checkpoints be globally coordinated, which is impossible if the threads are truly independent.

We believe that much of the complexity in current systems is related to the difficulty of restoring computational state in the event of a system shutdown. War stories about total system re-boot times after outages are becoming more common now that the reliability and availability of systems is exposed to the public through the Internet. Since most of these systems are dedicated to running some kind of database, it would seem that providing support for the whole computational environment including persistent threads, would be beneficial.

In principle persistent threads more easily support the implementation of triggers (events) and what the database community calls long-running transactions. Absent persistent threads, such facilities have to be implemented by ad hoc means and more complex resumption code is required to re-initiate the activity correctly.

4.3 Class Evolution

A consequence of supporting a persistent binding between code and data is that once a class is loaded and remains reachable [18], it cannot be altered using any mechanism in the current Java platform. For a persistent system this is an unacceptable state of affairs. There has been much research into schema evolution in object databases [9], [10], [17], but there is no consensus as yet on the correct set of mechanisms. For example, whether one should support multiple versions of classes. We chose not to include a specification for evolution in the initial OPJ specification, but we have implemented prototype off-line mechanisms [7], [8] that support most changes that a programmer might want to achieve. However, as yet, we do not have a large body of experience in using these mechanisms.

5 Support for Queries

The ability to submit queries through the standardized SQL language, regardless of the implementation of the underlying database, is one of the reasons for the success of relational databases. In particular, the ability to submit new queries that were not imagined when the database was originally created is extremely important. In programming language terms this is called extensibility or dynamic loading. A language that does not provide this capability is hampered in persistent application development, even if the language is orthogonally persistent, because the applications

are limited to a closed set of datatypes and queries. Fortunately, the Java programming language supports extensibility through its class loading mechanism.

The ability to load new classes at runtime provides a basic querying mechanism, provided that the database designer (i.e. application programmer) designed the (persistent) classes to reveal the appropriate information. We should note that the information hiding mechanisms in the Java programming language make it quite easy to deny access to relevant information, which can defeat subsequent queries. It is worth noting that class designers have more incentive than usual to plan ahead for reuse in the persistence domain.

It must be admitted that the Java programming language is not a user-friendly query language because its imperative nature makes it inherently hard to use in the same manner as SQL or OQL [4]. However, if the query generation is driven from a graphical user interface (GUI), as is often the case in a custom application, this becomes less of an issue. In any case, it would be possible to implement an OQL interpreter should a textual query language be required. Basic support for this is provided by way of the reflection support, which supports tool development in the face of extensibility of the set of loaded classes. The level of indirection provided by a GUI or query language interface allows certain optimizations to be provided when the code corresponding to the query is generated. Note that, unlike SQL query optimization which can leverage its simple data types, query optimization in an object-oriented context approaches general program optimization in its complexity. We believe that an orthogonally persistent system should be able to leverage the long-term binding between object instances and their methods to attack this problem by incrementally analyzing the system and building up useful, persistent, knowledge of its behavior. A similar approach is being taken to explicit constraints.

6 Conclusions

OPJ directly supports the persistence of the full computation state in a Java virtual machine, with practically no additional programming required. The ability to extend the computation environment by creating new threads and dynamically loading new classes provides basic support for queries and multi-user access. This permits a developer to view the Java platform directly as an object database for certain applications. Further work is needed in the area of multiple application support and evolution to extend this capability to the full range of applications.

7 Acknowledgments

The author is indebted to the entire OPJ team for their contributions to the OPJ specification, and especially to Malcolm Atkinson for his feedback on this paper.

8 Trademarks

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

9 References

- [1] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, S. Spence. An Orthogonally Persistent Java, *ACM SIGMOD Record*, 25, 4 (Dec 96), pp68-75.
- [2] M.P. Atkinson, L. Daynès, M.J. Jordan, S. Spence. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, *Proceedings of the 7th International Conference on Persistent Object Systems*, Cape May, New Jersey, May 1996.
- [3] M.P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems, *VLDB Journal*, 4(3), pp319-401, 1995.
- [4] R.G.G. Cattell (Ed.). *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997, ISBN 1-55860-463-4.
- [5] C.J. Date and H. Darwen: *Foundation for Object/Relational Databases - The Third Manifesto*, Addison-Wesley, 1998.
- [6] L. Daynès, M.P. Atkinson and P. Valduriez. Efficient Support for Customizing Concurrency Control in Persistent Java, *International Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa, India,

September 1996.

- [7] M. Dmitriev: The First Experience of Class Evolution Support in PJama, *The Third International Workshop on Persistence and Java*, Tiburon, CA, Sep 1998.
- [8] M. Dmitriev, M. Atkinson, Evolutionary Data conversion in the PJama Persistent Language. In the "Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases". In Association with 13th European Conference on Object-Oriented Programming, Lisbon, Portugal, June 14 - 18, 1999.
- [9] Fabrizio Ferrandina, Guy Ferran, Joelle Madec, Thorsten Meyer, and Roberto Zicari, Schema and Database Evolution in the O2 Object Database System. In Proc. of the 21th International Conference on Very Large Databases, pages 170-181, Zurich, Switzerland, September 11-15, 1995.
- [10] Fabrizio Ferrandina, Sven-Eric Lautemann, An Integrated Approach to Schema Evolution for Object Databases. In Dilip Patel, Yuan Sun, and Shishuma Patel, editors, Proc. of the 3rd International Conference on Object-Oriented Information Systems (OOIS), pages 280-294, London, UK, December 1996.
- [11] J. Gosling, W.N. Joy, G. Steele. *The Java Language Specification*, Addison-Wesley, 1996, ISBN 0-201-63451-1.
- [12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, ISBN 1-55860-190-2
- [13] M.J. Jordan, M.P. Atkinson. Orthogonal Persistence for Java - A Mid-term Report, *The Third International Workshop on Persistence and Java*, Tiburon, CA, Morgan-Kaufmann, Sep 1998.
- [14] M.J. Jordan, M.P. Atkinson. Orthogonal Persistence for the Java platform - Draft Specification, <http://www.sun-labs.com/research/forest/COM.Sun.Labs.Forest.doc.opjspec.abs.html>
- [15] M.J.Jordan, Observations on Persistent Object Systems from a (would-be) Consumer, *The Eighth International Workshop on Persistent Object Systems*, Tiburon, CA, Morgan-Kaufmann, Sep 1998
- [16] J. Melton and A. Simon, *Understanding the New SQL: A Complete Guide*, Morgan Kaufmann, 1993.
- [17] E. Odberg, Multiperspectives: Object Evolution and Schema Modification Management for Object-oriented databases, Department of Computer Systems and Telematics, Norwegian Institute of Technology, 1995.,
- [18] Clarifications and Amendments to the Java Language Specification, Sun Microsystems Inc., <http://java.sun.com/docs/books/jls/clarify.html>
- [19] Why is Thread . stop deprecated?, Sun Microsystems Inc., <http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html>
- [20] Antero Taivalsaari, Implementing a Java Virtual Machine in the Java programming language, Sun Microsystems Laboratories Technical Report, SMLI TR-98-64, 1998