First International Workshop on Persistence and Java, Mick Jordan and Malcolm Atkinson (Eds.)
SMLI TR-96-58, Sun Microsystems Laboratories, Nov. 1996

Approaches to Adding Persistence to Java

Position Paper for the
First International Workshop on Persistence and Java
Drymen, Scotland, September 1996

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610, USA
moss@cs.umass.edu

Antony L. Hosking
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
hosking@cs.purdue.edu

Abstract

We describe and name a range of approaches to adding persistence to the Java programming language. Java is interesting in this regard not only because of the current excitement over it. Some relevant properties of Java include: its blend of static and dynamic features, its incorporation of object code into the environment, its offering of automatic storage management, its standardization of the object code format, its broad (but not exclusive) use of object orientation, and its use of a standard library. In considering approaches to adding persistence to Java, we offer a preliminary evaluation of the advantages and disadvantages of the approaches, and describe some directions we are pursuing in our own developments. We hope our descriptions and evaluations will be useful to others in understanding the attributes of systems and designs to be discussed at the workshop, or considered thereafter.

Keywords: persistence, Java, transparency, byte codes, orthogonality

1 Introduction

The Java programming language [Gosling et al. 1996a; Arnold and Gosling 1996] has incited much discussion and activity of late, not the least in the community of those interested in persistent systems and programming languages, because the advent of Java offers a rare opportunity to bring persistence into mainstream use. But there are many ways one might add persistence to Java, having various technical and non-technical properties. Here we enumerate some of the possibilities and offer a preliminary description of their characteristics. We are not trying to be entirely comprehensive, and focus more on techniques that offer high degrees of *transparency* and *orthogonality*.

By transparency we mean that a program that manipulates persistent (or potentially persistent) objects looks little different from a program concerned only with transient objects.[1] Complete transparency cannot typically be achieved, and even if it can, it may not be desirable, since one usually wants to offer a degree of control to the programmer. A case in point is that in using a transaction mechanism one must generally specify at least the placement of transaction boundaries (begin/end). We would not call a technique transparent if it required different expression for the usual manipulation of persistent and non-persistent objects, i.e., for operations such as method invocation, field access, parameter passing, etc.

By orthogonality we mean the usual [Atkinson et al. 1983]: that persistence is a property independent of type.[2] Further, we take orthogonality to mean that any allocated *instance* of a type is potentially persistent, so that programmers are not required to indicate persistence at object allocation time. Again, perfect

[1] [Atkinson and Morrison 1995] define the term *persistence independence* for what we call transparency.

[2] [Atkinson and Morrison 1995] term this *data type orthogonality*.

orthogonality may not be achieved and may not be desirable. For example, some data structures refer to strictly transient entities (e.g., open file channels or network sockets), whose saving to persistent storage is not even meaningful (they cannot generally be recovered after a crash or system shutdown). Whether thread stacks and code persist as part of the store is a trickier question. As in most languages, these objects are not entirely first class in Java, and supporting persistence for them can also be challenging to implement. In any case, there is certainly room for different positions on transparency and orthogonality all to be reasonable, depending on one's objectives for the resulting systems.

The remainder of the paper essentially elaborates in more detail a range of possibly reasonable positions one could take concerning orthogonality and transparency, for Java in particular (though indeed many of the possibilities would apply to any language with at least some characteristics in common with Java: object-orientation, static type checking, standardized virtual machine [Lindholm and Yellin 1996] and libraries [Gosling et al. 1996b; Gosling et al. 1996c], dynamic code loading, etc.).

2 Orthogonality: Persistence model

Orthogonality is concerned with what entities can and do persist. We assume that for Java we desire at least that heap objects reachable from some kind of persistent roots will persist.[3] Thus, we arrive at the following choices for the model of persistence:

Persistence model choice 0 (M0): *Is persistence by reachability, or by some other means of designating which objects are to persist?*[4] Since we are primarily concerned with persistence by reachability, we do not consider alternatives here. We note that it may be desirable to allow programmers to specify some objects that should not persist, some types whose instances should never persist, or some edges that should not be followed in tracing a persistence reachability closure. These are possibly important refinements to the basic model, but do not reflect a fundamentally different choice.

As previously mentioned, threads and their state (including stacks), and code loaded into the system might or might not be stored. This leads to the following choices:

Persistence model choice 1 (M1): *Is (object) code persistent?* Including code has a degree of intuitive appeal in that it is more complete and elegant. However, it leads to duplication (code will often still live out in file systems: we cannot yet assume that everyone is working in a wholly integrated environment), it may cause a store to grow more than desired, and as the Glasgow group has found, it leads to "interesting" issues concerning its relationship to code loading in existing virtual machines [Atkinson et al. 1996]. *Not* including code is simpler in some ways, but leads to the problem of perhaps needing to find code and load it dynamically when objects are fetched from the store and acted upon, and the running program does not have the necessary classes loaded yet. It is not all that difficult to envision keeping track of code file names and loading code from a file system, but for safety one will need some kind of check that the code properly corresponds to the saved objects. Exactly what is the most appropriate check is a topic open to investigation and debate, especially in light of possibly evolving systems.

Persistence model choice 2 (M2): *Is program execution state persistent?* Having program and thread state persist is also intuitively appealing, but, unlike code, for which there is a standard definition of its format, and which is almost first class, stacks are not first class objects in Java. Among other things, this implies that a persistence implementor will need to dig more deeply into any existing virtual machine to support persistence of execution state. Some aspects of execution state, such as open file channels, etc., may

[3] This principle actually combines both transparency and orthogonality, but in any case, we take it as a basic goal.

[4] [Atkinson and Morrison 1995] speak of the principle of *persistence identification* for this choice.

not be sensible to save.[5] Even trickier is that if a saved state is one from which continuation will always lead to program failure, one would obtain a nasty kind of infinite loop, from which it would be difficult to recover.[6]

Persistence model choice 3 (M3): *What is the transaction model?* This choice addresses what becomes persistent at what time, who gets to see it when, etc. It is an area that we will not elaborate much, since it is somewhat independent of the issues on which we prefer to concentrate here. The Glasgow team has chosen to design a system supporting extension with new concurrency control and recovery models; we do not yet grasp the exact range of possibilities, but it is aimed at offering rich and flexible functionality, on the premise that for broad practical use of persistence such capabilities will be necessary for acceptance of persistence into mainstream programming. One might also argue that programmers need very simple models, especially in order to deal with potentially subtle "new" concepts such as persistence. This debate will undoubtedly rage for some time. For the time being we are content to note that the transaction model affects performance, how deeply one must dig into a virtual machine, overall difficulty of implementation, and ease of programming.

3 Transparency: Persistence implementation

It is interesting to note that our definition of transparency above is framed essentially in terms of the source language, leaving open a variety of avenues to achieving transparency. However, Java is somewhat unusual in that we are dealing not only with a language, but with a standardized virtual machine interface (byte codes and their meaning). This gives rise to a more subtle range of possible choices as to how to implement the persistence model defined by one's persistence model choices (e.g., as described in the previous section). In our categorization we choose to describe choices according to their transparency properties *and* according to what software artifacts are needed or affected. One set of choices has to do with whether the language and/or the compiler are affected:

Persistence transparency choice 1 (T1): *Is the source language changed?*

Persistence implementation choice 1 (I1): *Is the Java compiler changed?* If the source language changes, then we need to change the compiler, or provide a preprocessor from the changed language into standard Java. One can avoid changing the language by making extensions in other parts of the system and/or by making requisite features available without language extensions (e.g., via new library interfaces, etc.). Even if one avoids changing the language it may be reasonable to use a Java-to-Java preprocessor to add persistence; it is not as transparent or pleasant for source level debugging, but may reduce effort elsewhere and/or increase portability.

We observe that the Object Data Management Group has proposed to the Java language designers that a **pragma** feature be added, which would allow arbitrary text to be attached to method and field declarations, and possibly to classes. This text would be passed through the compiler unchanged, and associated with the constructs in the Java byte code file. Such a pragma feature opens a wide range of possibilities for adding persistence related annotations *without* modifying the Java compiler (see also [Moss and Hosking 1994]).

[5] They may show up as heap objects though, hence we support refinement of pure persistence by reachability to support some programmer control.

[6] In fairness, the problem is more general, in that one can always save a state that is essentially corrupted by a buggy program, and that results in failures at a later time.

Persistence transparency choice 2 (T2): *Is the object language changed?* Regardless of the appearance and processing of the Java source code, if we can manage to output standard Java byte codes, we obtain immediate portability benefits. Of course, this begs the question of whether the standard byte codes have the same semantics in the overall persistent implementation of Java as they do in the usual non-persistent Java system, but that is a separate choice.

We observe that just as one might use a source code pre-processor, one might use a byte code *post*-processor. This would take in standard Java byte codes and rewrite into standard Java byte codes, but perhaps adding instructions to check for object residency, to note modifications to objects, etc. These would be associated with some or all field accesses and/or updates. It is believed that the Java byte code format has enough information to support this transformation. This technique is a bit like using object code editing on standard workstations, to add instrumentation or other functionality. Cattell et al. have devised a post-processing approach for which they are applying for a patent [Cattell 1996].[7]

Persistence implementation choice 2 (I2): *Is the Java interpreter and run-time system modified?* Obviously any approach must somehow augment the underlying implementation, but it *may* be possible to do so without changing the interpreter available from Sun (or any other system), by loading additional Java libraries and arranging for them to be called in the right places. It is likely that such an approach will have limitations, e.g., it will be hard to support code and thread persistence that way, so this choice interacts with choices previously described. In fact, it might be more appropriate to state this choice not as a yes/no question, but in this way: *In what way is the Java interpreter and run-time system modified to support persistence?* We also note that we are using the term "interpreter" broadly, to include systems with dynamic compilation to native code, direct hardware execution of Java byte codes, etc.

A possibly important aspect of this implementation choice is whether one will immediately derive advantage from, and portability to, improved Java interpreters. If one uses approaches that rely on modifying the interpreter, then the anticipated arrival of dynamic optimizing compilers for Java, of Java machines, and the like, will be a barrier to acceptance and future use of persistent Java system. On the other hand, modifying the virtual machine may give the best achievable performance for persistent Java.

4 Our approach

Since we intend to work from a base of our own existing software, namely a Persistent Smalltalk system,[8] we aim to make minimal modification to that system, rather than minimal modifications to existing Java processors. This leads to the following provisional choices:

Choice		Our decision
M0	How are persistent objects identified?	By reachability
M1	Does code persist?	No
M2	Does execution state persist?	No
M3	What is the transaction model?	Not yet worked out
T1	Is the source language changed?	No (but will use pragmas if available)
I1	Is the Java compiler changed?	No
T2	Is the object language changed?	No (but will consider postprocessing)
I2	Is the Java virtual machine changed?	Yes, new interpreter

[7] We are indebted to Cattell for the characterization of Java persistence techniques into the categories of (source) preprocessors, (object, i.e., byte code) postprocessors, and enhanced virtual machines. This position paper is essentially an elaboration and expansion of Cattell's categories.

[8] [Hosking and Moss 1993a; Hosking and Moss 1993b; Hosking et al. 1993; Hosking 1995; Hosking and Moss 1995]

We should note in addition that our style of research includes building a range of variants and evaluating their relative performance, so we will likely look at a range of optimizations performed at the byte code level and internal to the interpreter (cf. [Hosking 1996]), as well as a range of interpreter implementation choices (byte code, threaded code, native code, various object faulting techniques, various update detection techniques). We anticipate using the UMass Language Independent Garbage Collector Toolkit [Hudson et al. 1991] and the Mneme persistent object store [Moss and Sinofsky 1988; Moss 1989; Moss 1990], as we did with Persistent Smalltalk.

5 Conclusion

We have offered a preliminary list of choices by which persistent Java systems may be characterized, which we hope will be useful to those working in this area as they compare various approaches. Because Java involves more than a standard source language, but also a standard object code format, virtual machine specification, and library, the situation is more complex than that of other languages, and the issues of orthogonality and transparency of persistence present interesting considerations.

References

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. Addison-Wesley.

ATKINSON, M. P., BAILEY, P. J., CHISHOLM, K. J., COCKSHOTT, P. W., AND MORRISON, R. 1983. An approach to persistent programming. *The Computer Journal 26*, 4 (Nov.), 360–365.

ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. 1996. Design issues for Persistent Java: A type-safe object-oriented, orthogonally persistent system. See [Connor and Nettles 1996].

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *Int. J. Very Large Data Bases 4*, 3, 319–401.

CATTELL, R. G. G. 1996. Personal communication.

CONNOR, R. AND NETTLES, S. (Eds.) 1996. *Proceedings of the International Workshop on Persistent Object Systems*, Cape May, New Jersey. Morgan Kaufmann.

GOSLING, J., JOY, B., AND STEELE, G. 1996a. *The Java Language Specification*. Addison-Wesley.

GOSLING, J., YELLIN, F., AND TEAM, T. J. 1996b. *The Java Application Programming Interface*, Volume 1: Core Packages. Addison-Wesley.

GOSLING, J., YELLIN, F., AND TEAM, T. J. 1996c. *The Java Application Programming Interface*, Volume 2: Window Toolkit and Applets. Addison-Wesley.

HOSKING, A. L. 1995. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. Ph.D. thesis, University of Massachusetts at Amherst. Available as Department of Computer Science Technical Report 95-02.

HOSKING, A. L. 1996. Residency check elimination for object-oriented persistent languages. See [Connor and Nettles 1996].

HOSKING, A. L., BROWN, E., AND MOSS, J. E. B. 1993. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the International Conference on Very Large Data Bases*, Dublin, Ireland, pp. 429–440. Morgan Kaufmann.

HOSKING, A. L. AND MOSS, J. E. B. 1993a. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, pp. 288–303.

HOSKING, A. L. AND MOSS, J. E. B. 1993b. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, pp. 106–119.

HOSKING, A. L. AND MOSS, J. E. B. 1995. Lightweight write detection and checkpointing for fine-grained persistence. Technical Report 95-084 (Dec.), Department of Computer Sciences, Purdue University.

HUDSON, R. L., MOSS, J. E. B., DIWAN, A., AND WEIGHT, C. F. 1991. A language-independent garbage collector toolkit. Technical Report 91-47 (Sept.), Department of Computer Science, University of Massachusetts at Amherst.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

MOSS, J. E. B. 1989. Addressing large distributed collections of persistent objects: The Mneme project's approach. In R. HULL, R. MORRISON, AND D. STEMPLE (Eds.), *Proceedings of the International Workshop on Database*

Programming Languages, Salishan Lodge, Gleneden Beach, Oregon, pp. 269–285. Morgan Kaufmann. Also available as COINS Technical Report 89-68, University of Massachusetts.

MOSS, J. E. B. 1990. Design of the Mneme persistent object store. *ACM Transactions on Information Systems 8*, 2 (April), 103–139.

MOSS, J. E. B. AND HOSKING, A. L. 1994. Expressing object residency optimizations using pointer type annotations. In M. ATKINSON, D. MAIER, AND V. BENZAKEN (Eds.), *Proceedings of the International Workshop on Persistent Object Systems*, Workshops in Computing, Tarascon, France, pp. 3–15. Springer-Verlag, 1995.

MOSS, J. E. B. AND SINOFSKY, S. 1988. Managing persistent data with Mneme: Designing a reliable, shared object interface. In K. R. DITTRICH (Ed.), *Proceedings of the International Workshop on Object Oriented Database Systems*, Volume 334 of *Lecture Notes in Computer Science*, Bad Münster am Stein-Ebernburg, Germany, pp. 298–316. *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.