An Orthogonally Persistent Java™

Atkinson, M.P.† , Daynès, L.†, Jordan, M.J.* , Printezis, T.† and Spence, S.†

† Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

* Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, CA 94043, USA.

Abstract

The language Java is enjoying a rapid rise in popularity as an application programming language. For many applications an effective provision of database facilities is required. Here we report on a particular approach to providing such facilities, called "orthogonal persistence". Persistence allows data to have lifetimes that vary from transient to (the best approximation we can achieve to) indefinite. It is orthogonal persistence if the available lifetimes are the same for all kinds of data. We aim to show that the programmer productivity gains and possible performance gains make orthogonal persistence a valuable augmentation of Java.

1 Introduction

The programming language Java™ [Gosling *et al.* 96] is achieving considerable prominence [Arnold & Gosling 96, van der Linden 96, Flanagan 96][1]. Its libraries and architectural neutrality encourage the use of Java for distributed and mobile computing [Straßer *et al.* 96]. However, other characteristics, such as: strong typing, single-inheritance, an object-oriented model, automatic space management, no explicit manipulation of pointers and validations to improve security, precision and productivity, are of particular interest when provisions for long-term data management are considered.

This paper reports on work which capitalises on those properties to provide Java with facilities to support long-lived systems in a way that improves application-programmer productivity. Our work is predicated on two assumptions:

- that Java will be used as the implementation language for many applications and
- that a significant proportion of these applications will require long-term data management.

The arguments for the popularity of Java have been made elsewhere, e.g. in the books cited above. We argue that it is self evident that longevity is important for many applications. Applications are built to service human requirements and most human activities that are worth such investment involve prolonged use. Even for games and entertainment, people expect their preferences to be remembered. In applications supporting planning, design or management, the human processes are prolonged because of the difficulty of tasks, the need to swap between activities, the need to consult and to collect information, the intrinsic pace of the external processes, etc. Often, the computational system will remain useful for as long as the artefact with which it was associated remains useful. Had Napoleon's engineers used CAD tools to design the sewers of Paris, we might assume that today's Parisian engineers would find access to the Napoleonic data and *programs* useful. Systems which require an intimate composition of long-lived data and programs are called "persistent application systems" (PAS) [Atkinson & Morrison 95].

1.1 Orthogonal Persistence

Orthogonal persistence is proposed to meet the needs of application programmers building new systems as parts of a PAS. They will typically be writing code by building up definitions that describe the form, factual data and behaviour of the system their PAS must service. Java is well designed to support that kind of object-oriented modelling. By providing *persistence* we allow components of these models to have whatever lifetime is required for an application, from microseconds to years. By requiring *orthogonality* we are simply stating that all possible components of such models should have the potential to have the same range of lifetimes. When objects are given the state which enables them to outlive a single transaction, we say they are *promoted.*

Orthogonally persistent Java (PJava) has been designed with the goal of optimising the work of these application programmers, both as components are constructed and during maintenance. Therefore, in PJava we deliberately choose to invest in implicit and incremental algorithms for data management, in order to simplify programmers' work and to relieve them of tasks. Over the years, such an approach has proved worthwhile for relational systems and we believe that in this context such automation will ultimately outperform *average* application programmer code and will immediately reduce production and maintenance costs. In particular, a full implementation of orthogonal persistence according to the following principles will facilitate class re-use significantly.

1.2 Three Design Principles

Orthogonal persistence is the provision of persistence for *all* data irrespective of their type. In PJava we interpret orthogonal persistence as support for the full range of lifetimes for all objects of whatever

[1] Java is a registered trademark of Sun Microsystems Inc. in the USA and other countries.

class[2]. All primitive data types are automatically included as they are either specialisations of class `Object`[3] or only occur as values of fields in classes. Objects of class `Class` can persist, either in their own right or because instances of a class they describe persist. In consequence all the code held in the methods of the class and all the meta data describing objects are able to persist.

Component lifetimes are constrained, so that if a component A is still required then any components needed to interpret A correctly must also continue to exist. The PJava system ensures this constraint, for example, it ensures that referends are retained for as long as a reference to them is held and that `Class` objects are retained for as long as that class appears in the definition of some other class or some instance of that class still exists. This leads to our next design principle.

Transitive persistence[4] requires that the lifetime of all objects is determined by their reachability. During the normal execution of a Java program, objects continue to exist if they are reachable from some variable. This is organised on a heap in standard Java [Gosling *et al.* 96] and is implemented using garbage collection. Lifetimes are terminated when objects are no longer reachable. This usually occurs because the last reference to an object is over-written.

In PJava, if objects are required to outlive a program execution, they can be identified as *persistent roots* or be referenced from some objects that have already been made persistent. All persistent roots are retained and every object reachable via a sequence of operations from those roots is retained. Space is recovered later by a disk garbage collector.

An implicit reference identifies the class of each object. Consequently, when the first object of any class is promoted, its `Class` object and the `Class` objects of all the classes used to define it, are also promoted. These `Class` objects may then refer to other objects, which will also need promoting. Of course, no object, including a `Class` object, is promoted if it is already in the persistent object store, so that common substructures are properly preserved. This preservation of the class objects with all their methods ensures that the code is available to properly interpret all of the objects stored in the stable store[5].

Persistence independence requires that it is indistinguishable whether code is operating on long-lived data, i.e. data originating from the stable store and outliving the program, or on data that is transient, i.e. data that is created in, and will only exist during, a conventional Java program execution. Compliance with the principle of persistence independence is particularly important for enabling software re-use. If code runs unchanged it is straightforward to re-use libraries of classes previously defined for transient applications in a persistent context.

The three principles of: *orthogonal persistence*, *transitive persistence* and *persistence independence* have been developed in 18 years of research into persistence [Atkinson & Morrison 95]. The type security of Java finally makes it possible to deploy them in a commercially significant language.

1.3 Structure of the Paper

The next section illustrates, via three program fragments, the use of PJava. Section 3 briefly discusses some implementation issues. Section 4 considers more sophisticated programming requirements. Section 5 compares the PJava approach with other approaches to providing persistence for Java. Section 6 presents the status of PJava and our plans for its development.

2 A Simple Example

A caricature of three PJava programs is presented to show an envisaged style of use of orthogonal persistence. The reader is reminded that Java development proceeds by defining each class in a separate file and that applications can be initiated by running any class that has a `main` method. We enter the scene where a programmer has already defined several Java classes and is now defining a class `SaveSpag` which has a `main` method that will create some persistent data.

2.1 Creating Persistent Data

The code outlined in Figure 1 is intended to create several complex data structures and to make the data structure referred to in the program as `sp1` persist for use by future programs.

A commentary on this `main` method is presented, keyed by line number.

1 The omitted code arranges to import classes defined elsewhere, which in this case would include classes `Spaghetti`[6] and `PJavaStore`.

4,5 Two normal transient data structures, `sp1` and `sp2` are created.

6,7 The object `sp1` is extended using the `add` method defined with the class `Spaghetti`.

8 Similarly, the object `sp2` has extra data added.

---

2 In the present prototype certain IO classes, AWT classes and Threads cannot persist. However, consistent treatment of these classes is planned.

3 This paper assumes knowledge of Java nomenclature; for precise definitions visit [Gosling *et al.* 96].

4 The original term "persistence by reachability" [Atkinson *et al.* 83] is superseded by this more suggestive ODMG term.

5 In most cases, the stable store will be held on disk.

6 So named to encourage the reader to think of it as involving a spaghetti of references between objects. It will use other classes in its definition. For example, it might specialise `Pasta` which itself specialises `Food`, and it might use `CylinderOnASpline` to model entangled strands of spaghetti, etc.

```
1  ...
2  public class SaveSpag {
3     public static void main (String[] args){ //start main method
4     Spaghetti sp1 = new Spaghetti(27);        //create a new object
5     Spaghetti sp2 = new Spaghetti(5);         //and another
6     sp1.add( "Pesto" );                       //modify sp1
7     sp1.add( "Pepper" );                      //modify it again
8     sp2.add( "Quattro Fromaggio" );           //modify sp2
9        try {                                  //catch store exceptions
10       PJavaStore pjs = PJavaStore.getStore(); //obtain a pers'nt. store
11       pjs.newPRoot( "Spag1", sp1 );          //make a persistent root
12       } catch (PJSException e){ ... }        //handle exceptions
13    }                                         //end of main
14 }                                            //end of SaveSpag
```

Figure 1: Creating a Persistent data structure

4-8 The resulting data structures may be arbitrarily complex, and may use a variety of other classes not directly apparent in this program.

10 PJava offers as an addition to the standard Java application programmers' interface (API) the class `PJavaStore` which provides access to the persistent facilities. If the program (in Fig. 1) is started with an existing persistent store, then the `getStore` method of `PJavaStore` will obtain an object that represents it.

11 The `newPRoot` method then sets up a binding between the `String` "Spag1" and the object `sp1` and records that as a persistent root. The object `sp1` is now identified as an object the programmer wants to make persistent.

13 This is the end of the `main` method of `SaveSpag`. When `main` is executed, the transaction implicitly associated with `main` will *commit* at this point. In consequence, the newly identified persistent objects and *all* objects newly reachable from persistent objects will be promoted to the persistent state. In this case that will be the object `sp1` and all objects made reachable from `sp1` during its initial construction and during the two `add` operations. The data structures associated with `sp2` (5&8) will not be promoted as they have not been made reachable from a persistent object. They are therefore discarded at this point. If this is the first instance of `Spaghetti` to be made persistent, then the `Spaghetti Class` object, all the classes used to define it and other objects it references will also be promoted to the persistent state.

9&12 Should an error occur, e.g. no store is associated with this program, then it will be converted into a `PJSException`. These two lines arrange to catch such exceptions. If an exception occurs within `main` that is not caught, then the transaction associated with `main` is automatically aborted and the state of the persistent store will be restored to its original state at the start of `main`'s execution.

2.2 Re-using Persistent Data

After the program in Figure 1 has run, the program `SpagShow` in Figure 2 would read the preserved data, display a representation of it, and leave the stored data unchanged. The new features introduced by this program are:

6 Here the API associated with a `PJavaStore` object is used to reclaim access to an object recorded as a root. The type of the object has to be re-established with a cast; a check is made to verify that the class referred to by the object in the stable store is exactly the same class as `Spaghetti` refers to in the program.

```
1  ...
2  public class SpagShow {
3     public static void main (String[] args){ //start main method
4        try {                                  //catch store exceptions
5        PJavaStore pjs = PJavaStore.getStore(); //obtain a pers'nt. store
6        Spaghetti sp = (Spaghetti)pjs.getPRoot( "Spag1" );
7        sp.display();
8        } catch (PJSException e){ ... }        //handle exceptions
9     }                                         //end of main
10 }                                            //end of SpagShow
```

Figure 2: Using Preserved Data Structures

```
1  ...
2  public class EditSpag {
3     public static void main (String[] args){ //start main method
4        try {                                  //catch store exceptions
5        PJavaStore pjs = PJavaStore.getStore(); //open a persistent store
6        Spaghetti sp = (Spaghetti)pjs.getPRoot( "Spag1" );
7        sp.add( "Garlic" ); sp.stir( 5 );      //modify using methods
8        sp.userArrangement();                  //and input from a user
9        } catch (PJSException e){ ... }        //handle exceptions
10    }                                         //end of main
11 }                                           //end of EditSpag
```

Figure 3: Updating a Persistent data structure

7 The code in the method `display` of class `Spaghetti` will traverse whatever objects it needs that are referenced from `sp`. The PJava system automatically faults them in from the stable store and arranges that they behave *exactly* as if they had been created with the same values during this execution.

9 If the code executed has made no changes to any of the traversed persistent data structures this will behave as a read only transaction.

2.3 Amending Persistent Data

7 In Figure 3, the operations `add` and `stir` will again cause whatever objects they need to be faulted in. Substructures that are not needed will not be faulted in to unnecessarily occupy space in the object cache. These operations will alter existing structures in `sp` and possibly other objects reachable from `sp`, and may make further objects reachable from it.

8 The method `userArrangement` is presumed to allow a user to make arbitrary changes. Again this will modify values in the data structures reachable from `sp` and possibly add, remove or replace objects reachable from it.

10 As this transaction commits, the changes are automatically preserved. `PJavaStore` internal code atomically and recoverably writes back all persistent objects that have been changed and then promotes all new objects that are reachable from these updated objects to the persistent store.

2.4 Discussion

The most important observation to make is that most programming is de-coupled from considerations of persistence and that the programmer writing the above programs does not have to control or modify any of the code in other classes. In the above example, the coding of the class `Spaghetti` and the coding of the classes `Pasta` and `Food` it specialises, etc. would have taken place without knowledge that that code was to be used in a PAS. Similarly, the PAS programmer was not required to inspect or modify this code in any way in order to use it. Indeed PJava uses unchanged class files produced by the standard Java compiler. As a consequence, very little of the code written actually uses the persistence API explicitly. In one recent example, there were only 18 lines of code explicitly using the PJava API in an application using 95 classes with a total of 7690 lines of code [Jordan 96].

Another consequence of persistence independence is that the strictness of type checking should be undiminished when long-lived data is used. This is achieved in PJava by ensuring that the class information about an object is associated with it in the stable store. The cast, as a binding is formed between program and an object in the stable store (line 6 in each of the last two examples), then verifies that the type expected in the program (here `Spaghetti`) is compatible with the type of the object being retrieved. Thus the type safety of Java is not weakened at this point of binding and the standard type-checking regime applies throughout.

These properties allow Java programmers to focus on their application, i.e. on building an object-oriented model of some system and then to utilise persistence with virtually no perturbation to their code and no loss of safety. That is, they use persistence without being distracted by it and without it obscuring the application code. Further details of this conceptual simplification may be found in [Atkinson & Morrison 95] where it is argued that a major factor is a reduction in the number of mappings a programmer must manage.

Further details of the design of PJava are in [Atkinson *et al.* 96] and an initial appraisal of its utility and performance is given in [Jordan 96].

3 Implementation Issues

PJava is implemented without any change to the Java language, the Java core classes or the Java compiler. The mechanisms for persistence have been exposed via an additional API, predominantly methods of the class `PJavaStore`. A purpose-built persistent object store holds Java objects on disk and manages space allocation, recovery and transactional updates. The automatic object faulting, the promotion to persistence and recoverable, transactional operation have been achieved by modifying the Java virtual machine (VM). The major change is to add an object cache that is made to look to the rest of the VM very much like Java VM's garbage collected heap.

In general, algorithms are incremental. For example, on start up, an initial region of the persistent object store (POS) is pre-loaded into the object cache. After that, the object faulting mechanism relies on detecting the difference between a reference to some object already

in cache, and some object in the stable store. This mechanism is invoked only when a pointer is de-referenced for the first time. A buffer pool is used to reduce disk transfers.

Similarly, the promotion algorithms are also incremental. The candidates for promotion are found by treating all the mutated pointers in objects in the cache as roots of the *new* reachability sub-graph. Further information about the current implementation can be found in [Atkinson *et al.* 96, Daynès *et al.* 96, Daynès 96, Printezis 96].

4 More Sophisticated Persistence

It is anticipated that many programmers will be satisfied with a very simple model of persistence, such as that illustrated in section 2. This is analogous to experience with relational databases, where most of the user requirements and most of the workload can be satisfied by simple flat transactions. In the present prototype, the only additional mechanism is a global checkpoint, `stabilizeAll`, that establishes intermediate recovery points.

However, some PAS programmers may require more control. For example, they may wish to checkpoint intermediate results within a specific transaction, build long-running transactions, utilise nested transactions, etc. Our approach to this, which is still under development, will ensure that the simple use is not sacrificed to meet these more complex requirements. To this end, programmers will be provided with two additional APIs corresponding to two levels of understanding of transaction management.

The first API will provide ordinary application programmers with a functional view of transaction management. This API consists of a hierarchy of transaction classes rooted by the class `TransactionShell`. Each specialisation of `TransactionShell` implements a given transaction model. Transactions are executed by creating instances of these transaction classes and parameterising them with any object implementing a `Runnable` interface. These `Runnable` objects constitute the body of transactions, while the instances of transaction classes constitute a shell that enforces a transactional behaviour.

The second API will provide an implementation view of transaction management. This API concerns the expert programmer who wishes to augment the set of available transaction models in order to satisfy new needs. This internal API consists of building blocks which may be used to implement specialisations of the class `TransactionShell`.

5 Related Work

Two other approaches to providing long-term data management for Java are currently popular:

1 interfaces to an existing data model, and

2 object serialisation.

These are familiar from previous languages and each has its strengths. They are not mutually exclusive and many large applications may use combinations of these approaches together with orthogonal persistence.

5.1 Interfaces to Standard Databases

Two themes are developing within this approach:

- connections to relational databases, and
- connections to OODB.

The former is exemplified by the JDBC interface [Hamilton & Cattel 96] which provides a traditional, embedded SQL-like interface from Java to relational databases. It clearly has the advantage of capitalising on established database systems and of allowing Java applications to inter-work with other applications via a relational system. It has the drawback that there is an extensive dissonance between Java's object-oriented data model and the relational model. This introduces a conceptual load on application programmers as they have to maintain a mapping between the two models. The execution of this mapping imposes translation costs. Managing the interface between Java code and the relational system requires explicitly programmed operations. This obscures the application code, eliminating the possibility of persistence independence, and militating against code re-use.

For some applications, these problems may be ameliorated by storing serialisations (see below) of complex Java structures in relational BLObs[7] provided cross-references between these objects in different BLObs are not required [Reinwald *et al.* 94]. A system motivated as a mechanism for communication between Java systems [Wollrath *et al.* 96] uses a similar mechanism, as it will store data as tuples which can be transactionally added, read and removed from a specified collection. The values may include arbitrarily complex serialisations. The principle difference is that tuples are identified by a much simplified query notation, presumably for efficiency reasons.

Connections between existing OODBMS and Java programs are being developed by most of the OODB vendors, e.g. [ODI 96], and the ODMG is engaged in developing a standard for its Java binding [Cattel 96]. Compared with the relational case, there is a closer resemblance between Java's type system and the data models in these systems. In consequence, they can perform more sympathetic mappings between Java and their databases. Some mapping is still necessary as their data models are already decided and do not necessarily match precisely the types in Java. In most cases, Java data structures will map easily to their structures, but not all their structures map directly to Java. In consequence, access to legacy data from Java may be more complex than storing new Java data in the OODB. We can expect *orthogonal* and *transitive persistence*. As in PJava, this avoids the danger of dangling persistent references and implies that the allocation and recovery of disk space is fully automated.

It is likely that many vendors products will deviate from the ideal of orthogonal persistence by disallowing

7 Binary Large Objects.

persistence for class objects (and hence Java programs). It also remains to be seen the extent to which the interface between Java and the OODB requires annotation and extra code. Many may deviate from persistence independence by requiring explicit lock and transaction management throughout the code. A few may even require notification of updates.

5.2 Object Serialisation

Object serialisation [Riggs *et al.* 96, Sun 96a] is available for Java and is the foundation of the Remote Method Invocation (RMI) technology [Sun 96b]. This technique was previously called "pickling", e.g. for Modula-3. It takes a reference to an object and provides a series of bytes that represents that object *and all objects reachable from it.* This sequence of bytes can then be transferred to another machine or stored in a conventional file or DB BLOb. Subsequently, it can be reloaded and a *new* object for each object serialised is constructed. These new objects have an equivalent type and state to the original objects that were serialised, however, they have a different identity. The result is that successive serialisations will no longer share common substructures. Furthermore an application programmer has to explicitly ensure that the classes used in the original data structure are all available at the destination. Large serialisations take a considerable time, because of the translations involved. Consequently, application programmers have to get involved in partitioning the data structures pertinent to their application into units. If they can't organise such a partitioning, they suffer from the "big inhale" problem. Even when their program only needs to load some small subset of data to work, or at least meet an end-user's initial needs, they have to suffer the delay while *all* the data is loaded and translated.

If PAS programmers make partitioned serialisation work to avoid these problems, they still have to program concurrency control, consistency between partitions, structure sharing, references between partitions, atomicity, etc. These are operations that we believe the underlying persistent technology can and should support and automate.

Serialisation is convenient to use; unless a non-standard treatment is required, it is fully automatic — essentially it is driven from class definitions and the application only has to request that an object be serialised, for it and all objects reachable from it, to be serialised. Serialisation is well adapted for shipping data between processes and sites, as in RMI, and can be used to record snapshots of data. It does not seem to be a good mechanism for persistence, for two reasons:

1 it does not scale well, as it isn't supported by incremental algorithms, and

2 it violates the principle of persistence independence, as the change of identity and loss of sub-structure sharing mean that programmers have to cope with changed semantics after serialisation.

5.3 Orthogonal Persistence

Moss and Hosking have provided a classification system for those designing and providing orthogonally persistent versions of Java [Moss & Hosking 96]. Here we focus only on the systems which we know have an operational implementation.

An orthogonally persistent Java has been produced by Garthwaite and Nettles [Garthwaite & Nettles 96]. This work is oriented towards high-throughput transaction processing in Java, and their implementation assumes that the whole database can be loaded into RAM at application-boot time. It includes a modification to the language, but as all methods called from within that transaction construct are then executed transactionally, it can call unmodified methods. Therefore, it is able to adhere to all three of the principles we consider important.

Dearle's group have made Java persistent by running it on top of their persistent operating system, Grasshopper [Dearle *et al.* 96]. This approach immediately satisfies our principles, and has the obvious advantage of requiring the least possible modification of application code. Its principal drawback is that persistent operating systems are not yet widely available.

The binding between Java and relational databases, built on top of JDBC, has been announced by O_2 Technology [Souza dos Santos & Theroude 96]. This removes some of the clutter from application code and simplifies the task of automatically generating a relational database and interface code for a package of classes selected via a configuration file that a PAS requires persistent. The system does not appear to facilitate interfaces to existing databases. As all classes can be mapped to the RDB, this persistent Java is orthogonally persistent. It does not, however, support transitive persistence, and as the application programmer has to notify the underlying program of such things as "an object has been updated", it does not provide persistence independence. A system announced by Baan seems to have a similar approach [Baan 96].

Another experiment with Java also sets a high premium on adding persistence and transactions without any changes to the application code [Wu & Schwiderski 96].

6 Summary and Future Directions

We have reported on the initial version of PJava, an orthogonally persistent version of Java. Three principles are important in the design and implementation of PJava, these are:

orthogonal persistence every component of an application should have the same rights to longevity;

transitive persistence all data needed to correctly interpret an object must be retained for as long as that object exists; and

persistence independence code should operate unchanged with exactly the same semantics whether it is operating on short-lived or long-lived data.

These three principles are motivated by the goal of maximising application programmer productivity. If any of them are not fully met, the application programmers have to pay the cost by writing more complex code. For example, if some class cannot be made persistent, violating orthogonal persistence, programmers end up coding translations to and from data structures that can be made persistent (we have had to do this with our current prototype for AWT classes). This distracts the application programmer, clutters the code and introduces errors associated with inconsistent translations. If persistence isn't transitive, application programmers get bogged down in a mire of dangling reference problems.

The principle of persistence independence is particularly important for code re-use. It ensures that code already written does not have to be modified to work in a persistent context. It therefore guarantees that code does not become cluttered with non-functional annotations and that application programmers can concentrate on building a correct set of object definitions.

These principles have been used as the basis for comparing PJava's approach with other persistence mechanisms available for Java.

6.1 Future Directions

These are divided into two parts: plans for PJava and issues that need further consideration regarding persistence and Java in general.

PJava is a vehicle for experiment and several lines of experiment and development are expected. We expect to learn from various internal uses of PJava at Sun and in Glasgow, including the Forest project [Jordan & Van Der Vanter 95], and from as many external uses as possible[8]. We already have several sets of measurements and test loads that show reasonable performance [Jordan 96] and have encountered several cases where using persistence has made significant gains over non-persistent approaches, either by avoiding translation costs or by avoiding a "big inhale". There are, however, opportunities to significantly improve the present implementation.

Currently, the cache management algorithms and persistent object store design are being revised to support much larger transactions and increased object populations. The flexible transaction model will then be investigated [Daynès 96, Daynès *et al.* 96]. At the same time we plan to develop incremental concurrent disk garbage collection [Printezis 96]. Our approach to distribution is intended to limit commitments to those feasible in an indefinitely distributed system [Spence & Atkinson 97, Spence 96].

Our principles led us to take the route of storing the meta data and code in the same store as the data it describes. It is quite feasible with Java to store methods in byte code form because of its architectural neutrality. The co-location of code, meta data and objects in one stable store has potential advantages, such as being able to retain code optimisations for future execution, and being able to evolve code and data in step. However, there are several technical challenges to be overcome before these advantages can be realised.

It is also interesting to consider the general future of orthogonally persistent Java. In addition to its role enhancing application programming productivity, it could also fulfil another role. Orthogonal persistence includes into the programming model much of the external activity currently achieved via files and databases. This allows that activity to be described by an abstract computational model and thus extends the language's architectural and context independence. This model could then be applied to mobile as well as stationary code. Code in transit could carry with it, logically at least, a persistent context. All operations within this context are constrained by Java's standard safety rules. Consequently, the mobile persistent unit might be a useful construct for organising security, sandboxing, accountability and ownership. Security is not reduced by persistence, unless there are covert channels to access the persistent store. These may be prevented by operating system protection regimes and encryption. With this proviso, persistence independence guarantees that all the protection currently afforded by Java within programs is undiminished and is now available for everything in a stable store.

The interaction between a mobile persistent unit and the environments it visits might be described separately. These persistent units could provide a solution to the problem of arranging that several applets (servelets or aglets) may need to work on the same data, and that users want to own and carry *their* data about with them. The utility of orthogonally persistent Java contexts in these roles is a topic ripe for exploration.

Acknowledgements

The work at Glasgow on PJava is supported by a grant from Sun Microsystems Inc. and by grant GR/K87791 from the British Engineering and Physical Sciences Research Council. Many offered useful comments on earlier drafts, including: Richard Connor, Dag Sjøberg Cathy Waite and Kathy Zuckerman.

Bibliography

Arnold & Gosling 96 Arnold, K. and Gosling, J. *The Java Programming Language*, Addison Wesley, 1996, ISBN 0-201-63455-4.

Atkinson, *et al.* 83 Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R., An approach to Persistent Programming, *Computer Journal*, 26(4), 360-365, Nov. 1983.

[8] http://www.dcs.gla.ac.uk/pjava/ contains up-to-date information on PJava and how to obtain it.

Atkinson *et al.* 96 Atkinson, M.P., Jordan, M.J., Daynès, L. and Spence, S. Design issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. in *Proceedings of the Seventh International Workshop on Persistent Object Systems, Cape May, May 1996* (Connor & Nettles). http://www.dcs.gla.ac.uk/pjava

Atkinson & Jordan 96 Atkinson, M.P. and Jordan, M.J. *Proceedings of the First International Workshop on Persistence and Java, Drymen, Scotland*, Sept. 1996, Sunlabs Tech. Report. http://www.dcs.gla.ac.uk/rapids/events/pj1

Atkinson & Morrison 95 Atkinson, M.P. and Morrison, R. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

Baan 96 http://www.baan.com/2_News/Press_Releases/press21.htm.

Cattel 96 Cattel, R.G.G. (editor) *Object Database Standard : ODMG - 93, Release 1.2*. Morgan Kaufmann, San Fransisco, 1996.

Daynès 96 Daynès, L. A Flexible Transaction Model for Persistent Java. In [Atkinson & Jordan 96].

Daynès *et al.* 96 Daynès, L., Atkinson, M.P. and Valduriez, P. Efficient support for customising concurrency control in Persistent Java. In Bertino, E., Jajodia, S. and Kerschberg, L. (editors) *Proc. of the International Workshop on Advanced Transaction Models and Architectures (ATMA), Goa, India, Sept. 1996*, pages 216-233.

Dearle *et al.* 96 Dearle, A., Hulse, D. and Farkas, A. Operating system support for Java. In [Atkinson & Jordan 96].

Flanagan 96 Flanagan, D. *Java in a Nutshell*. O'Reilly & Associates 1996, ISBN 1-56592-183-6.

Garthwaite & Nettles 96 Garthwaite, A. and Nettles, S. Transactions for Java. In [Atkinson & Jordan 96].

Hamilton & Cattel 96 Hamilton, G. and Cattel, R. *JDBC: A Java SQL API*, June 1996. http://splash.javasoft.com/jdbc

Jordan & Van De Vanter 95 Jordan, M.J. and Van De Vanter, M. Software Configuration Management in an Object Oriented Database, in *USENIX conf. on Object Oriented Technologies*, Monterey, CA, June 1995. http://www.sunlabs.com/research/forest/COM.Sun.Labs.Forest.doc.coots_95.abs.html

Jordan 96 Jordan, M.J. Early Experiences with Persistent Java. In [Atkinson & Jordan 96].

Moss & Hosking 96 Moss, J.E.B. and Hosking, A.L. Approaches to adding persistence to Java. In [Atkinson & Jordan 96].

ODI 96 http://www.odi.com/products/pse.

Printezis 96 Printezis, T. Analysing a simple disk garbage collector. In [Atkinson & Jordan 96].

Reinwald *et al.* 94 Reinwald, B., Dessloch, S., Carey, M., Lehman, T. and Pirahesh, H. Making real data Persistent: initial experiences with SMRC. in *Proc. of sixth international workshop on Persistent Object Systems (Atkinson, M.P., Maier, D. and Benzaken, V. eds)*, Springer Verlag, 1995, ISBN 3-540-19912-8.

Riggs *et al.* 96 Riggs, R., Waldo, J. and Wollrath, A. Pickling state in the Java system. *Proceedings of the second international conference on object-oriented technologies (COOT'96), Toronto, Canada*, June 1996.

Souza dos Santos & Theroude 96 Souza dos Santos, C. and Theroude, E. Persistent Java. In [Atkinson & Jordan 96].

Spence 96 Spence, S. Distribution strategies for Persistent Java, In [Atkinson & Jordan 96].

Spence & Atkinson 97 Spence, S. and Atkinson, M.P. A scalable model of distribution promoting autonomy of and co-operation between PJava object stores. to appear in *Proceedings of the thirtieth Hawaii international conference on System Sciences, Jan. 1997.* http://www.dcs.gla.ac.uk/~susan/papers.html

Straßer *et al.* 96 Straßer, M., Baumann, J. and Hohl, F. Mole - A Java based Mobile Agent System. in *Proceedings of ECOOP'96 Workshop on Mobile Object Systems.*

Sun 96a Sun Microsystems Inc. Java object serialisation specification, draft revision 0.9. http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html, 1996.

Sun 96b Sun Microsystems Inc. Java remote method invocation specification, draft revision 0.9. http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html, May 1996.

van der Linden 96 van der Linden, P. *Just Java*. Prentice Hall 1996, ISBN 0-13-565839-X.

Wollrath *et al.* 96 Wollrath, A., Riggs, R. and Waldo, J. A distributed object model for the Java system. In *Proceedings of the second international conference on object-oriented technologies (COOT'96), Toronto, Canada*, June 1996.

Wu & Schwiderski 96 Wu, Z. and Schwiderski, S. Design of Reflective Java. Internal Report APM.1818.00.05, APM, Poseidon House, Castle Park, Cambridge CB3 0RD, United Kingdom, Sep. 1996.