Persistency in a Dynamic Object-Oriented Programming Language

Heiko Kirschke*

University of Hamburg • Computer Science Department
Vogt-Kölln-Straße 30 • D 22527 Hamburg, Germany

New address: POET Software GmbH • Foßredder 12 • D 22359 Hamburg

The work described in this document has been carried out private and within Hamburg University. POET Software is not otherwise affiliated with the work presented here.

21. August 1995

The common approach used for storing longlived structured data is to employ a relational database which provides its users with types for structuring data and operations to manipulate them. Along with its data definition and manipulation language, a database often provides an interface to a general programming language. This interface is a possibility to access the database's functionality, but it neglects the concepts of the programming language using the database. A solution to this problem is to remove the dichotomy between a database and a programming language, i.e. to integrate the concepts of a database into the general programming language. This abstract presents some aspects of research done on persistency and explains shortly an implemented architecture for integrating persistency into the dynamic, object-oriented programming language Common LISP Object System (CLOS). Further details can be found in [Kir 94a, Kir 94b, Kir 95].

The background for this work was the field of model-based image analysis, which involves two kinds of data: On the one side, the raw image data (e.g. high resolution color images) and on the other side, models of the objects mapped to the images represented by highly interconnected data objects. This resulted in the following demands for a system capable of storing data:

- For storing the raw images, the system must be able to cope with very large data objects.
- For storing the object models, a storage system should be able to handle references between data objects appropriate.
- Since the programming language chosen for model-based image analysis was CLOS, the storage system's interface should be adapted both in its syntax and semantics as far as possible to CLOS. The development of the image analyzing software and the storage system was done in parallel, so the integration of the storage system into an existing CLOS application should be possible without big efforts.
- Besides simply storing data, the system should offer some database functionality, e.g. transactions, data locking and associative search.

A more personal goal was the time constraint imposed by the examination instructions of Hamburg University: A diploma thesis has to be finished after one year. I took this as a challenge to design, implement and document a real-working efficient system with an open interface for further extensions in this time interval.

The reason for selecting a dynamic programming language was that in most of them (esp. in CLOS), to each data object a class metaobject is associated which describes the data's structure and behavior. This description can be used to obtain all informations necessary for storing an object by a persistent system; in static systems, these informations are often not available at all or must be generated by dedicated compilers.

*The author's email address is mailto:Heiko.Kirschke@poet.de

1 Systems for persistent data

The classical approach to make data persist over time and to access it conveniently is to use a relational database. The formal basis of relational databases is the relational calculus defined by [Co 79].[1] His definition was aimed at abstracting from the details of the data's representation. The relational calculus has some mathematically provable, attractive properties, e.g. the existence of normal forms to avoid redundancies and anomalies, so a database obeying the relational calculus inherits these properties. Furthermore, there is no need for the notion of an explicit reference like in most other general programming languages; instead, a reference is established by a common value shared between the referencing objects.

Besides storing data, a real-world application also consists of methods working on the data; of course, Codd and others realized this and extended the relational calculus by relational algebra to comprise data manipulation too [Co 79, p. 400] [O'N 94, p. 44], but although his solution clearly has some theoretical advantages, it did not succeed in practice, perhaps because it was too formal to be developed into a general programming language. Instead, relational databases have been made available to general programming languages by more or less obscure interfaces, e.g. Embedded SQL [O'N 94, p. 201–283] or the LIBPQ interface of POSTGRES [Wen 93, p. 112–126] (Figure 1). Another way was taken by some database vendors: They extended their (relational) databases by more or less general programming languages whose development started as an ad-hoc solution, examples are SQL, NATURAL/ADABAS and ABAP-SAP/R3.[3]

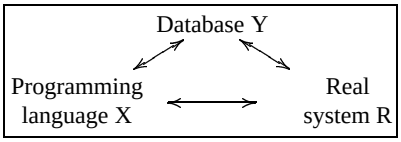

Figure 1: Modelling a real system[2]

Starting around the mid-70s, a new view to persistency was established. Instead of using a seperate subsystem, persistency was integrated into a general programming language; in the first attempts, an existing language was extended by dedicated constructs for defining types with its instances being persistent and additional control structures, e.g. selectors, iterators etc. [Sch+ 80]. Later on, general programming languages have been defined with persistency being a property of *each* object which can be represented [Cl 91, Mat+ 93]; persistency became orthogonal w.r.t. the type system (Figure 2). Practice also showed that it is desirable to keep the notion of an explicit reference between objects: Besides being a very efficient implementation of a relation between representations of real-world objects, it reflects this relation more direct than the relational calculus of sharing a common value.

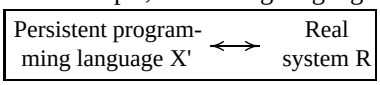

Figure 2: Modelling a real system[2]

2 Architecture concepts for persistent systems

From the various possible architectures for persistent systems, I decided to use the one shown in Figure3: An existing transient object system (CLOS) is extended by a new system which adds persistency. The application layer can use the transient system directly for its transient objects and the persistent system for its objects which should be held persistent. Building a persistent system on top of an existing transient system has the advantage that the persistent system can use all the services provided by the transient system. Especially for CLOS this approach yields a very elegant solution, since the persistent system can be integrated seamlessly into the overall architecture; this is accomplished by the Metaobject Protocol (MOP) of CLOS, which makes syntactic and semantic extensions for CLOS possible by means of CLOS itself (see [AMOP, CLtLII] for details).

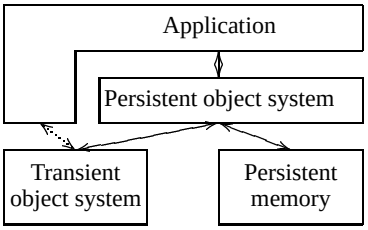

Figure 3: Persistent system architecture

One important question what kind of subsystem should be used for the persistent memory. Persistent systems using a relational database as a persistent memory can pass the rich data manipulation functionality to higher levels, but usually they suffer from the different type concepts of the relational database and CLOS. Since CLOS is a very type-rich language and relational databases normally offer only a subset of CLOS' types, these systems have to translate between both data representations; also, references between objects have to be transformed to the value-based approach of the relational calculus. A solution to this problem is to use a *persistent heap*

[1] The original definition dates back to 1970; the reference given here contains the original and some extensions to it.

[2] Figures from [Atk+ 89, p. 81]

[3] Although ADABAS and SAP/R3 are not relational databases, the above statement is also valid for NATURAL and ABAP, respectively.

as persistent memory; it works very similar to a transient heap, especially it has the possibility of establishing explicit references between the objects contained in it. These explicit references allow the lifetime of persistent objects to be defined in terms of reachability: The persistent system is allowed to delete a persistent object, if it can prove that it is no longer in the transitive closure of the references of a designated root object. The root object itself has an infinite lifetime.

Since a persistent heap offers a `low-level' untyped data representation, typing the objects in the heap must be done by higher layers, i.e. the persistent object system. Thereby, the type restrictions normally imposed upon the persistent system by using a relational database can be avoided, since the persistent system can simply adopt the typing concepts of CLOS.

3 Persistent LISP Objects

The evaluation of the concepts important for persistent object systems lead to the development of the Persistent LISP objects (PLOB!) system (Figure 4). The lowest level is the persistent heap POSTORE developed at the University of St Andrews [Br 92]; it implements the administration of persistent records. Its `typing' consists of dividing each record into a section with references to other persistent records and a value section, which is not further interpreted by POSTORE. The next layer C-PLOB! implements the overall database functionality of the persistent system in ANSI-C. Regarding data modelling, it imposes a refined typing onto the POSTORE records which can be characterized as a mapping of *each* CLOS type into an appropriate persistent type with the database specific extensions of object locking and transaction handling. Regarding data manipulation, this layer handles Two-Phase transactions and associative access to the objects contained in the persistent memory. The PLOB! layer written in CLOS is the link between the application layer (the `sessions' in Figure 4) and the persistent object system. Its task regarding data representation is to transform between the different representations used for objects in CLOS and in the lower levels of PLOB!. Its second task is to make the functionality of the C-PLOB! layer available to the higher application layer and to offer some more convenient concepts, e.g. persistent symbols for simple adressing of persistent objects. Care was taken that the integration of persistency into existing CLOS applications is possible. The highest session layer represents applications using PLOB!. Each session is an aggregation of a CLOS process and a transaction. The transaction is used by the process to make consistent changes to the persistent system, i.e. all changes to the states of persistent objects are embedded into transactions. Since PLOB! is almost type complete, the application layer gets persistency without restrictions.[4]

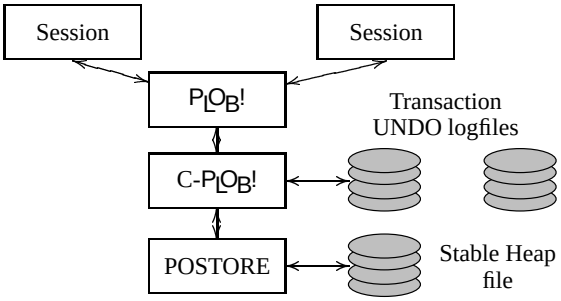

Figure 4: PLOB! layers

4 Conclusions

Practice showed that the system described in the last section proved very flexible and efficient. Its flexibility stems from its implemented orthogonal view of persistency w.r.t. the type system of CLOS; using the Metaobject Protocol of CLOS made the interface between the persistent system and an application very simple; indeed, persistency is transparent to the users of PLOB!.

Besides that meanwhile almost any database is decorated with the term `object-oriented' in some sense, the market of `real' object-oriented databases is growing. There are now available a number of commercial persistent systems for static programming languages (e.g. ObjectStore for C++) and for dynamic programming languages (among the here presented system e.g. Ithasca for CLOS and for C++). To my opinion, following the dynamic paradigm for persistent systems is more promising than concentrating on pure static systems, because the static systems can be subsummed under the dynamic systems. However, it must be noted that full dynamic systems are more difficult to optimize; a combination of both techniques might show the right way. Furthermore,

[4] The only exception is function code, which can not be stored because of relocation problems.

the concept of persistent objects with explicit references leaves the path shown by the relational calculus; from a very radical point of view, this could even be seen as a step back towards databases with complicated, graph-like and cyclic structures. Clearly, some theoretical research has to be done on this area, but it looks as if persistent object systems have much benefits w.r.t. general programming languages.

References

[AMOP] Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow: The Art of the Metaobject Protocol. MIT Press, Cambridge, Mass., 1991

[Atk+ 89] Malcolm Atkinson, Ronald Morrison: Persistent System Architectures. John Rosenberg, David Koch (Eds.): *Persistent Object Systems, Newcastle, Australia 1989*, p. 73–97, Springer-Verlag, Berlin, 1989

[Br 92] A. L. Brown: Stable Heap manual pages, File `postore/man/stable_heap.3p`, 25 Mai 1992

[Cl 91] Stewart M. Clamen: Data Persistency in Programming Languages, A Survey. Report CMU-CS-91-155, Carnegie Mellon University, Pittsburgh, 1991

[CLtLII] Guy L. Steele Jr.: Common LISP the Language, Second Edition. Digital Press, Bedford, Mass., 1990

[Co 79] E. F. Codd: Extending The Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979

[Kir 94a] Heiko Kirschke: Persistenz in objekt-orientierten Programmiersprachen am Beispiel von CLOS. Diploma thesis, Dept. of Computer Science, University of Hamburg, 1994

[Kir 94b] Heiko Kirschke: Persistent LISP Objects: User's Guide, Reference Manual. Dept. of Computer Science, University of Hamburg, 1994

[Kir 95] Heiko Kirschke: Persistenz in der objekt-orientierten Programmiersprache CLOS am Beispiel des PLOB! Systems. Report FBI-HH-B-179/95, Dept. of Computer Science, University of Hamburg, 1995

[Mat+ 93] F. Matthes, J. Schmidt: System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. P. P. Spies (Ed.): *Proceedings Euro-ARCH '93*, p. 301–317, Informatik Aktuell, Springer-Verlag, Berlin, 1993

[O'N 94] Patrick O'Neil: Database Principles, Programming, Performance. Morgan Kaufmann Publishers, San Francisco, Ca., 1994

[Sch+ 80] Joachim W. Schmidt, Manuel Mall: PASCAL/R Report. Report IFI-HH-B-66/80, Dept. of Computer Science, University of Hamburg, 1980

[Wen 93] S. Wensel (Ed.): The POSTGRES Reference Manual, Version 4.1. Report M88/20, University of California, Berkeley, 1993