Semantics of Persistence in the Glib Programming Language

Daniel Gakh Libicki
Celequest Corporation
555 Twin Dolphin Dr.
Redwood City, CA 94065
dlibicki@celequest.com

Abstract

The cornerstone of object-oriented programming is the representation of data as a set of objects. In all of the widely-adopted languages that claim to support object-oriented programming, however, the lifetime of an object is bound by the lifetime of the process that instantiated it. In real applications, the lifetime of data is almost never related to the lifetime of the process that created it. This impedance mismatch necessitates a great deal of repetitive, error-prone labor. A true object-oriented design language must be a persistent language; in other words, the lifetime of an object must be independent of the lifetime of the process.

Many persistent languages have been developed in research settings. Most of these languages, however, have attempted to maintain backwards compatibility with some previous, non-persistent language, such as Modula-3 or Java. Glib, on the other hand, is a programming language designed from the outset to support object persistence. I propose that Glib's constructs are simpler and more powerful than those of its predecessors, and now that I have an OOPLSA poster displaying those constructs, you can judge for yourself.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Contructs and Features – *classes and objects, concurrent programming structures, constraints.*

General Terms Design, Reliability, Languages

Keywords Persistence, Transactions, Type Systems, Schema Evolution, Confinement, Object Queries, Modeling

1. Introduction

If you wanted to save an object in one run of a program and then retrieve it in a subsequent run, would you rather:

a) write the code to connect to a database, create and execute a SQL statement, and interpret a result set

b) write annotations all over your class and an xsd to go along with them

c) call List.add()

If you answered (c), then orthogonal persistence is for you. In particular, orthogonal, independent, transitive persistence. "Orthogonal persistence" means an object of any type can be persistent. "Independent persistence" means that code treats persistent and transient objects the same way – in fact, code is agnostic as to whether any given object is persistent or transient. "Transitive persistence" means that all objects reachable from a persistent object are persistent, so that persistence does not lead to data corruption.

In Glib, as in languages such as PJama (persistent Java) [2] and PM3 (persistent modula-3) [5], persisting an object is just another case of inserting an object into a data structure, as easy as calling List.add(). In non-research settings, on the other hand, where persistent languages have not been adopted, it is estimated that about 30% of the code of a typical application is dedicated to translating between the objects of the programming language and the mechanisms of a relational database [2]. This introduces a serious impedance mismatch and makes applications much more fragile.

Glib, unlike other languages, features local persistence. If you don't like global variables, you might like Glib's persistence design better than that of PJama, PM3, etc. Glib offers persistence, file I/O, security, and type casting, with a single simple construct, the Folder class of the Glib standard library.

2. Confinement

The lifetime of a Glib object is never bound by the lifetime of its creating process. By default, the lifetime of an object is bound by the lexical scope of its reference. If the reference to such an object is a local reference, then the lifetime of the object is bound by the control block or method where the reference is declared; if the reference to the object is a field of a composing object, then the lifetime of the field object is bound by the lifetime of the composing object.

References may be declared `outside`, allowing the objects they refer to be aliased. A type system governs the relationship between `outside` and default (or "inside") objects. Once an object can be aliased, its lifetime is not bound by anything. If an `outside` object becomes unreachable, it is garbage collected; if it becomes reachable from a persistent root, it becomes persistent.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

Many aspects of Glib semantics make programming with confinement easy and abstract. For instance, cloning an object is an intelligent, easy operation in Glib, whereas cloning an object in Java is prohibitively awkward.

3. Transactions

Glib features software transactional memory; threads are protected against each other through transactional concurrency control, like concurrent accesses to a database. Without transactions, programmers usually need to explicitly lock resources to avoid threads from improperly interfering with one another. Programming explicit locks is a perfect example of a repetitive, error-prone task that gets in the way of the real design.

One of the biggest disadvantages of explicit locks, however, is a disadvantage shared by many software transactional memory schemes. In these schemes, if you write a method without thinking about concurrency, your method is not safe to call in a concurrently executing system. In Glib, on the other hand, if you write a method without thinking about concurrency, it runs in a transaction, and it is safe to call in a concurrently executing system. If you think you can write a method that is thread safe even if it is not in a transaction, you can do that too. The only case in which a method may be thread-unsafe is if the programmer went to a lot of trouble to make it so, which is likely to be rare and documented.

In Glib's transactional semantics, deadlocks are entirely hidden from the programmer, so that the runtime environment can plausibly deny that they happen at all. Also, the exception-handling semantics of Glib leverage the transactional semantics. Much exception handling in other languages restores corrupted objects to a stable state; Glib can automate most of that task by simply rolling back the current transaction (up to the beginning of the try block) when an exception is thrown.

The transactional semantics of Glib bear some resemblance to the transactional semantics of EJB [4]. However, the semantics of Glib are much simpler than those of EJB. They are more powerful since they govern every object in the environment, while EJB transactions govern only the beans themselves. Also, misuse of the transactional constructs is trapped in Glib at compile time, while in EJB, misuse is trapped at runtime.

The state of concurrency control in the non-research languages is especially dismal. In Java, deadlock will cause a program to hang [3]. The compiler rearranges statements in ways that change the semantics of a multithreaded program. (In the discussion of this phenomenon, the designers of Java take the strategy of blaming the programmer, forgetting that safe languages are supposed to take responsibility for protecting their own abstractions.) Other ugly things can happen to data that is shared between threads.

4. Schema Evolution

Schema evolution has been called the hardest problem in object persistence [1]. As applications evolve, class definitions evolve, bugs are fixed, etc. When an object is shared between two code bases, the typechecks of the usage of the object are performed by code base currently using the object, but the behavior is defined by the code base that defined the object. Code bases must be updatable. An updatable code base does not just allow a convenient, abstract way to bring large sets of objects up to speed; it is the only way to fix bugs in persistent objects. If code bases were not updatable, persistent objects with a bug in the implementation of their methods at the time they were created would perpetually exhibit the bug in their behavior.

Luckily, a code base in Glib is represented by an object of the class Machine, which, like most classes in the Glib standard library, instantiates updatable objects. When old code in the Machine is replaced by new code, all currently running processes and persistent objects dependent on that Machine switch to the new code. To deal with incompatible class changes, the Machine class supports a message called `deprecate` by which the programmer can specify a relationship between a deprecated class and a class that replaces it.

5. Relational Queries

Most of the semantics of long-lived data are better represented by persistent objects than by relational tables. However, relational tables are good at satisfying a certain form of query efficiently. For applications that call for such queries, Glib offers a control structure that is a simple extension of Glib's `for` block.

Relational queries in Glib operate on the Glib standard library class `Database`, which is a subclass of `List`. A relational query takes a list of <iterator declaration, `Database`> pairs and evaluates the Boolean expression on the cross product of the databases. The body of the block is executed once for each element of the cross product for which the Boolean expression evaluates to true.

6. References

[1] R. C. Connor, Q. I. Cutts, G. N. Kirby, and R. Morrison, "Using persistence technology to control schema evolution", *Proceedings of the 1994 ACM Symposium on Applied Computing* (Phoenix, Arizona, United States, March 06 - 08, 1994). SAC '94. ACM Press, New York, NY, 441-446. DOI= http://doi.acm.org/10.1145/326619.326805

[2] Malcom Atkinson, Mick Jordan, Laurent Daynes, and Susan Spence, "Design Issues for Persistent Java: a type safe, object-oriented, orthogonally persistent system", *Seventh International Workshop on Persistent Object Systems (POS7)*, 1996.

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, *Java Language Specification Third Edition*, The Java series, Addison-Wesley, 2004.

[4] L.G. De Michiel, *Enteprise Java Beans Specification, Version 2.1*. Sun Microsystems, November 12th, 2003 http://java.sun.com/products/ejb/docs.html#specs

[5] Antony L. Hosking and Jiawan Chen. "Mostly-copying Reachability-based Orthogonal Persistence", *OOPSLA '97 Workshop on Memory Management and Garbage Collection*, October 1997.