Modules as Values in a Persistent Object Store[1]

Gilad Bracha

Horizon Technologies of New York, Inc.

gilad@cs.utah.edu

Charles F. Clark, Gary Lindstrom[2] and Douglas B. Orr

University of Utah

{clark,lindstrom,dbo}@cs.utah.edu

Abstract

We report on an object manager (OM) providing persistent implementations for C++ classes. Our OM generalizes this problem to that of managing persistent *modules*, where the *module* concept is an abstract data type (ADT). This approach permits a powerful suite of module manipulation operations to be applied uniformly to modules of many provenances, including non-class based entities such as conventional object files, application libraries, and shared system libraries. OMOS, a generalized linker and loader, plays a central role in our OM. Class implementations are represented by OMOS modules, which in turn are constructed from OMOS meta-objects encapsulating linkage *blueprints*. We cleanly solve the problems of (i) logically (but not physically) including executable object files in our OM, (ii) reconciling class *inheritance history* and *linkage history*, and (iii) supporting alternative implementations of a class, for client interoperability or version control.

Key words: Persistence, object stores, modules, dossiers, class implementations, dynamic linking and loading, functional interposition.

---

[1] This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[2] Primary contact author; Computer Science - 3190 MEB, Salt Lake City, UT 84112 USA, phone +1-801-581-5586, fax +1-801-581-5843.

1 Requirements for a Persistent Object Store

1.1 Motivations for Persistent Objects

Techniques for saving objects in long-term storage are the focus of vigorous scientific and commercial activity, arising from two motivations:

- Extending the software engineering (SWE) power and flexibility of object-oriented (O-O) programming systems, and
- Extending database technology to advanced applications such as computer aided design, multi-media data management, and large-scale scientific databases.

The first motivation is felt within the O-O programming language community, where the rapid advances of O-O programming methodology design are confronting its *semantic impedance mismatch* with byte-oriented file systems. Typically an O-O application system designer develops a customized object representation exploiting static typing, encapsulated data and functions, variable granularity data, and intricate interobject referencing. Unfortunately, this rich structure must be encoded and decoded when preserved in a flat, amorphous, and unencapsulated file. Systems permitting O-O language objects to be saved and restored without such loss of structure are termed persistent object stores (POS's).

The second motivation comes from the database community, where successors are being sought for the relational data model. Though uniform and mathematically elegant, the relational model is primarily motivated by *ad hoc* querying of schematically simple databases by short duration, highly concurrent transactions. Object-oriented databases (OODB's) [ABD+92] provide a modern alternative, where relations are replaced by collections of objects, in the O-O programming sense. Traditional concerns in the database community remain paramount in the design of OODB's, including atomicity and recovery, distribution, concurrency control, and optimized querying.

Users have good reason to hope that these two approaches will eventually converge, or even unify, into a pervasive O-O system framework offering comprehensive support of both O-O SWE and database requirements. However, there remain today sharp and weighty differences between these two worlds. One of the most vivid such differences lies in management of the functions (*method code*) associated with persistent objects. To an O-O programmer, the methods associated with an object are its *sine qua non* — and even, under *full encapsulation*, comprise the object's only external interface. Yet today's OODB's consider methods, and their association with stored objects, to be the application programmer's concern, much like query code.

We address here the conceptual, semantic, and implementation issues raised by the POS requirement that *method definitions comprising a class implementation must be permanently associated with its persistent instances.* This requirement poses several specific questions, including:

1. How should the class implementation of a persistent object be represented?
2. How can a persistent object and its class implementation be kept consistent?
3. How can an application program deal effectively with objects of classes unknown to it at the time of its compilation?
4. How does managing class implementations relate to managing the *object files* in which their implementations reside in a traditional system linking sense?
5. Can this long-lamented overloading of "object" (class instances vs. object files) be turned to advantage as the basis for a single, broadly useful architecture for managing program modules, their descriptions and their instances?[1]

1.2 The MSO Object Manager

Although much remains to be investigated, designed, implemented and validated, we believe we have preliminary answers to the above questions and others that arise when class implementations are required to be persistent. Our insights have been formulated while designing an object manager (OM) for the Mach Shared Objects (MSO) project. Although the MSO OM will ultimately provide more services than is customary for a POS (e.g. rudimentary querying and concurrency control), we will view it as a POS for the purposes of this paper. The MSO OM is being developed under the following requirements:

1. The primary function of the MSO OM is to provide a persistent store for C++ and Common Lisp Object System (CLOS) application program objects.
2. The MSO OM will be layered between application systems and modern operating systems of the Unix family, and run efficiently on current hardware. Hence any architectural "object orientation" will rely on software implementation only.

[1] For the purposes of this paper, we assert the following lexical equivalences: *declaration* $\equiv$ *specification*, and *definition* $\equiv$ *implementation*. By *description* we mean either of these two, in a generic sense.

3. The MSO OM must not invalidate use of existing compilers, programming environment tools (e.g. make) and file systems. In particular, executable code associated with persistent objects will not be physically stored in the MSO OM.

4. The MSO OM will use at its lowest level a storage manager that (i) stores and retrieves uninterpreted byte-oriented records; (ii) provides an object identifier (OID) naming service for these records, and (iii) implements rudimentary transaction control (e.g. object locking).

5. In addition, several pragmatic constraints are exerted by MSO's primary client, Alpha_1, a large computer aided geometric design and manufacturing system, including: (i) a high degree of compiler and hardware platform independence; (ii) minimal constraints on application programming style, and (iii) preservation of high computational speed on loaded objects.

The MSO OM design thus far has focused on support for C++ application programs. Consequently, the main thrust of this paper concerns providing persistence support for C++ class implementations.

2 Describing Stored Objects

It is well known that run-time objects describing classes are necessary for *fully polymorphic manipulation* of objects, e.g. copying, browsing, and secure casting. In particular, such descriptions are required for a POS to implement fully polymorphic load and store functions. This section discusses how both class specifications and implementations are represented in our system.

2.1 Representing Class Specifications

We begin by reviewing how specifications of statically loaded classes can be represented by *dossier* objects, as conceived by Interrante and Linton [IL90]. Their work is motivated by providing type descriptions at run time. Types fundamentally deal with compatibility of object interfaces, while classes are program modules from which objects can be instantiated. In C++ classes *are* types — which facilitates compilation but segregates classes which could be considered type equivalent.

All objects of a given class share the same dossier, which in turn is an instance of class dossier. Our dossiers comprise a type identifier, instance size, and information about the class's data members, member functions, and direct base classes. Each data member is represented within the dossier by a structure containing that member's name, its position within the declaration of the class, and a reference to the dossier for that data member's type.

The information about each member function includes that function's name, its arguments with references to the dossiers for their types, and a reference to the dossier for the return type. Each dossier of a class which uses inheritance contains only the information contained within that class's declaration. Hence, data members and member functions declared in base classes are not replicated within the dossier of a derived class. Since a dossier contains method specifications but not method implementations (code pointers), the dossier of a class corresponds to the class's *specification*.

2.2 Representing Class Implementations

Many current OODB systems associate with a stored object its class's symbolic name as a class implementation key. This device is acceptable, provided that any application program loading an object (i) ensures that class name uniqueness guarantees class implementation uniqueness, and (ii) will have already statically loaded that object's class implementation. In practice, these conditions are too severe. In particular, we may desire to: (i) dynamically link and load method implementations, (ii) access objects of unknown classes [DSS90], and (iii) provide alternate implementations of a class for interoperability (across compilers and hardware platforms) or versioning (e.g. debugging) purposes.

Dossier objects could be extended to record the method implementations of the classes represented by its objects. Static type checking would dictate that these be stripped of type information by `void`-casting, but this would be a benign loss since such information is already encoded in method type descriptors. Nevertheless, little purpose would be served by this extension for statically created dossiers for objects of known classes, since all methods must be already defined. The key issue is how to represent a class implementation associated with stored objects.

Since a class implementation is an extension of a class specification, we can extend class `dossier` to represent class implementations as well. Storing a persistent object $\mathcal{O}$ then entails (i) storing $\mathcal{O}$'s dossier as a persistent object augmented to include method implementations, (ii) storing $\mathcal{O}$ itself, with a data member `OID* my_dossier` set to the OID of its stored dossier. However, several questions remain:

1. How are method implementations represented in a stored dossier? Clearly, a memory address

at which the implementation was once loaded is inappropriate.

2. Loading an object should load its dossier, if not already loaded. How do we know if the dossier is already loaded?

3. If a dossier is dynamically loaded, how do we locate and map in its method implementations as executable code?

4. Dossiers record the *inheritance* history of classes. This is central to the logical structure of O-O systems, but separate from the management of class descriptions as SWE artifacts, which concerns *linkage* history of separately compiled files. How can these two dimensions be reconciled?

5. Since a dossier is a persistent object itself, does it possess a dossier itself, and if so, what is its function? (And ever upward!)

We assert that these questions should *not* be answered within the limited context of a particular O-O programming language (e.g. C++), lest overly specific mechanisms result. Rather, we believe that a comprehensive solution should be sought in the more general context of storing program modules as persistent values.

3 Classes as Modules

A module is a familiar program structuring notion involving encapsulating functions and data, controlling visibility, presenting a typed interface to clients, admitting instantiation, etc. For our purposes, there are two highly pertinent kinds of modules:

- The *class* notion, as formulated in O-O languages (especially C++), and
- The *object file* notion, as produced by language processors and manipulated by system utilities such as linkers and debuggers.

These two notions of module share many characteristics, but are very different in their senses of composability. Classes are composed by inheritance, a semantically rich but language-specific operation, while object files are composed by semantically simple linkage operations with universal applicability within families of language processors emitting a standardized object file format.

As suggested in §2.2, a class implementation stored as a persistent dossier object must live on the boundary of these two worlds, being a persistent object itself, while providing directions for locating method implementations in object files which are not stored in the POS. Rather than being a source of confusion, we believe this dual object nature can enrich the POS to manage both kinds of objects in a compatible and architecturally unifying manner.

The basis lies in recent work [BL92, Bra92] formalizing the module notion as an abstract data type (ADT), customizable to these two senses of module (class and object file), and many others. This is distinct from the familiar notion that modules (e.g. classes) can be used to introduce user-defined abstract data types. Rather, the concept of module *itself* is cast as an abstract data type, in the same sense one might characterize *stack* as an abstract data type.

A module is a self-referential scope, with some names bound to values (i.e. defined) and some names simply declared. Modules have no free variables, and we assume that declarations are sufficient for external access, thereby permitting separate compilation of modules. The module ADT provides a constructor function make_module, through which individual modules (cf. classes) are created. Once created, a module can create instances (cf. objects) via an instantiate function, specified (but perhaps not implemented) by the module ADT. Most importantly, the module ADT defines language-independent combinators applicable to any well-formed module. Metaphorically, these combinators permit the deft reshaping and fitting together of modules, much as a jigsaw tool may be used to craft the pieces of a wooden puzzle. In this spirit, we dub the applicative language of module ADT operators Jigsaw. Sample Jigsaw module combinators are shown in Fig. 1.

Viewing class implementations as modules enlarges the charter of a POS to be a persistent store for modules, their instances, and their descriptions. This three level perspective, developed in detail in §4, resolves two vital issues:

- A class implementation is stored as a dossier object extended to include a reference to the *object file module* in which its method implementations are found, and
- Dynamic linking and loading of class implementations (even unknown ones) involves simply the run-time merge of the executing program (viewed as a module) with the object file module containing the class implementation.

Name	Sample Module
$\mathcal{O}_1$	{int x; fun f (int y) = g(g(y)); fun g (int z) = z+x}
$\mathcal{O}_2$	{int x = 13; fun q (real z) = z*z}
$\mathcal{O}_3$	{int y = 15; fun g (int w) = w-y}

Operation	Result
$\mathcal{O}_1$ copy f as h	{int x; fun f (int y) = g(g(y)); fun g (int z) = z+x; fun h (int y) = g(g(y))} *(A definition copy is added)*
$\mathcal{O}_1$ freeze g	{int x; fun f (int y) = g(g(y)); fun g (int z) = z+x } *($\mathcal{O}_1$ is unchanged, but* g *becomes non-rebindable)*
$\mathcal{O}_1$ hide g	{int x; fun f (int y) = g'(g'(y)); fun g' (int z) = z+x} *(Component* g' *is not externally visible)*
$\mathcal{O}_1$ merge $\mathcal{O}_2$	{int x = 13; fun f (int y) = g(g(y)); fun g (int z) = z+x; fun q (real z) = z*z} *(Declarations and definitions collected & matched; conflicts disallowed)*
$\mathcal{O}_1$ override $\mathcal{O}_3$	{int x; fun f (int y) = g(g(y)); int y = 15; fun g (int w) = w-y} *(Merge with conflicts resolved in favor of right operand)*
$\mathcal{O}_1$ rename g to h	{int x; fun f (int y) = h(h(y)); fun h (int z) = z+x} *(Declaration and all uses consistently renamed)*
$\mathcal{O}_1$ restrict g	{int x; fun f (int y) = g(g(y)); fun g : int $\rightarrow$ int} *(Declaration stripped of its definition)*
$\mathcal{O}_1$ show f	{int x'; fun f (int y) = g'(g'(y)); fun g' (int z) = z+x'} *(Complement of* hide *—* x' *and* g' *are hidden)*

Figure 1: Sample Jigsaw operators

Level	C++	Dossiers	Jigsaw	OMOS	Shared Libraries
Meta	—	meta-dossier	module ADT	meta-object	—
Module	class	dossier	module	object	public data & functions
Application	object	—	instance	—	private data

Figure 2: Three levels of objects.

4 Three Levels of Persistent Objects

Thus far we have argued that (i) persistent objects need persistent descriptions, (ii) those descriptions should be persistent dossier objects, (iii) persistent dossiers should include class implementation information, and (iv) class implementation information should be obtained from object files, viewed as modules. From these specific decisions emerge three general criteria for the design of a comprehensively useful POS:

1. The POS should recognize the role of some objects as descriptions of collections of other objects.

2. A POS should not be tailored to a particular O-O language, or even to O-O languages in general. Rather, it should facilitate uniformity through generalizing its task to representing (i) an abstract notion of modules, (ii) individual modules, and (iii) instances of those modules.

3. This abstract module notion should embrace not only class implementations but comparable forms of modularity such as abstract classes, object files, system libraries, and application libraries, especially in *class framework* form [JR91].

Indeed, the converging technologies of O-O languages and shared system libraries was an original stimulus for the MSO project. Seeley [See90] establishes broad correspondences between these two software structuring concepts, including code reuse, dynamic function dispatch, and visibility control. Jigsaw clarified this relationship, which we recognize as a three-layer object management architecture that reappears in many software domains (see Fig. 2). The column labeled *OMOS* refers to MSO's generalized linker and loader, which incorporates the module ADT and Jigsaw operator suite. OMOS is discussed in detail in §5.

We now briefly consider the connections across each of these three layers.

1. **Application level:** At the bottom level we have objects in the traditional O-O language sense.

C++: These of course are class instances.

Dossiers: Due to their descriptive nature, dossiers are meaningful only at the class or module level, or above.

Jigsaw: As described in §3, module instances correspond to class instances.

OMOS: Modules can provide instantiate functions, allocating run-time data structures akin to C++ objects, but such functions have no special status to OMOS.

Shared libraries: Shared libraries generally permit the allocation of private data (e.g. random number seeds), allocated by init functions, much like user-level instantiate functions in OMOS.

2. **Class level:** At the intermediate level, we have objects serving as descriptions shared by collections of application objects.

   C++: Clearly, this is a class.

   Dossiers: A dossier object represents a class.

   Jigsaw: These are modules, resulting from the make_module constructor of the module ADT.

   OMOS: This is an object in the system programming sense, i.e. an object file.

   Shared libraries: The central idea behind shared libraries is to share public data and functions.

3. **Meta level:** At the top of our three-level architecture we have module descriptions. Since we truncate our meta-tower here, all objects at this level must be self-describing (i.e. have fixed format).

   C++: C++ has no corresponding notion. However, if we were dealing with Smalltalk or CLOS, this entry would be "metaclass".

   Dossiers: A meta-dossier is the dossier of the dossier class. Its role is described in §7.2.

   Jigsaw: This is the module ADT itself.

   OMOS: Objects in the OMOS world are described by *meta-objects*, which export a construct function delivering objects (i.e. modules).

   Shared libraries: The need to manage the creation and sharing of system libraries was an early motivation for OMOS.

```
(constrain '(> 0x60000000)
    (show "_open|_close|_read|_write|_ioctl"
        (merge /ro/lib/libc/open.o
               /ro/lib/libc/close.o
               /ro/lib/libc/read.o
               /ro/lib/libc/write.o
               /ro/lib/libc/ioctl.o)))
```

Figure 3: An OMOS module blueprint.

5 Defining and Using Modules

The MSO OM thus uses a generic notion of module to represent both class implementations and more general executable program units. We now explain the role of OMOS, our generalized linker and loader, in (i) manipulating modules for both these purposes, and (ii) as an MSO OM client itself. Throughout this section the reader should bear in mind the terminology of Fig. §2 — especially that an OMOS *object* is a *module* level entity, akin to a C++ class.

5.1 Modules As Persistent Objects

The OMOS Object/Meta-Object Server [OM92, OMHL93] was created to support and exploit the use of modules as persistent objects. OMOS deals with objects (modules) in two forms:

- Relocatable program *fragments*, e.g. compiler emitted object files, and
- *Meta-objects* which intentionally describe modules constructible according to *blueprints* applying Jigsaw operations on modules under specified address mapping constraints.

As described in §3, the dossier representing a class implementation obtains its function definitions from a persistent object representing an OMOS object. Since the MSO OM cannot store compiled code directly, that object is in fact a surrogate — the OMOS meta-object from which the specified module can be constructed on demand. OMOS meta-objects are defined using a Lisp-like blueprint language, illustrated in Fig. 3. The Jigsaw expression in the blueprint of a dossier's meta-object thus specifies the linkage history of its associated class implementation.

As a full-featured system loader, OMOS must also deal with client address space management. Execution of module operations is accomplished in conjunction with a constraint system that controls placement of objects within an address space. OMOS is responsible for assigning virtual addresses to the machine instructions that make up a set of object methods. OMOS retains flexibility by allowing rebinding of method virtual addresses as needed. In general, OMOS constraints will encourage use of the same virtual address bindings for all instances of a given set of methods. In this way, all clients mapping the methods in will (tend to) share the same physical memory. If, however, a given region is already occupied in a user's address space, OMOS will select another region for the mapping. Bound instances of methods are cached to avoid unnecessary repetition of symbol binding.

Hence OMOS is a "loader with a memory" — which is very helpful both within a client session (e.g. in support of dynamic relinking, see §6.2), and across client sessions (e.g. to suppress relinking and relocation of frequently used class implementations). Moreover, the constraint system allows the bulk of an object's clients to physically share the same set of methods, as is done by shared libraries. The use of a constraint system retains the flexibility of modern shared library schemes without incurring the overhead or added complexity necessitated by the use of position independent code.[2]

5.2 Classes As Persistent Objects

Before an object of a class $\mathcal{C}$ can be stored in the MSO OM, an implementation of $\mathcal{C}$ must be *registered.* A class implementation is a stored object $\mathcal{D}$ of class dossier, i.e. a class specification object, augmented to include two additional data members:

- OID* implementation, which refers to the OMOS module from which its functions can be loaded, and
- OID* meta_dossier, which refers to a fixed-format object $\mathcal{M}$ specifying the compiler and hardware conventions ("execution environment") under which this class implementation was produced.

The OMOS module referenced by implementation gathers (e.g. from various object files) the definitions comprising the class implementation. Typically, this gathering will be defined by an OMOS blueprint — hence the module is logically (intentionally) defined via an OMOS meta-object, rather than explicitly manifested at class registration time. Registering a class implementation entails

[2] Position independent code is not precluded by this scheme. Use of PIC code will render OMOS' relocation operations trivial, streamlining some phases of OMOS' execution. In general, OMOS performs these functions in the background, so this time savings is not a critical factor.

associating OID($\mathcal{D}$) with the key [$\mathcal{C}$, OID($\mathcal{M}$)] in a class name server provided by the MSO OM.

The execution environment information in a meta-dossier permits MSO OM client programs to store and retrieve objects in an interoperable manner, despite compiler and processor architecture differences. These characteristics are summarized in a shorthand fashion, via a small set of literal values describing compiler version, hardware platform, etc. (more on this in §7.2). An MSO OM client application has available as global read-only data the OID of the meta-dossier characterizing its execution environment. This leads to a rudimentary policy for validating dynamically loaded class implementations. The OID of the meta-dossier of the new class implementation is compared with the client's meta-dossier OID, and an exception is raised if the two disagree. More adaptive policies are sketched in §7.2. The means by which dossier objects are originally created is described in §7.1.

6 Linking and Loading as Module Manipulation

The use of the Jigsaw module operations within OMOS firmly casts program linking operations within the module manipulation model. The Jigsaw operations extend traditional linker operations to allow sophisticated module manipulation. Under OMOS, these operations apply to both static and dynamic linking and provide functionality such as function overriding and interposition. However, the Jigsaw formulation goes beyond conventional linking, by supporting (i) submodules (nested scopes), and (ii) typed module interfaces. OMOS does not currently implement these aspects of Jigsaw. Instead, OMOS relies (as do all existing linkers) on C++ *name mangling* to encode type information in external names. Since classes are types in C++, this also accomplishes flattening of class scopes. An extra advantage is compatibility with modules produced by other language processors, e.g. ordinary C compilers.

6.1 Static Linking and Loading

As mentioned earlier, the primary function of a static linker is to associate references to interfaces with their definitions. We can see that this operation corresponds largely to the module merge operation. OMOS implements module manipulation facilities that are more sophisticated than simple merging through the use of other Jigsaw module operators, individually or via blueprinted combinations. The override operation, for example, permits precedence when combining scopes.

This feature allows modules to be combined in a fashion that simulates single inheritance — which drives home our point that inheritance is simply a form of module combination, and should not be the exclusive purview of O-O languages. The hide operation, another Jigsaw operation supported by OMOS, reduces the amount of information exported by a module. OMOS uses this feature to provide different views of a shared library to its clients. Hiding symbols not used by the client speeds subsequent linking and prevents interfaces that may be defined but not referenced from interfering with definitions found later in other modules.

To illustrate the degree of sophisticated module manipulation supported by OMOS, consider the technique of *function interposition*. Here we wish all references to a function f to be rebound to an interloper function which takes its place. In turn, the new function can make references to the prior definition of f (as well as itself, in the general recursive case). In this fashion, new definitions of routines can be made to replace old ones, or inserted "between" the references to the routine and its original definition.

Function interposition is a very powerful system configuration device. OMOS uses it internally when performing program execution monitoring. For a set of routines that are to be monitored, OMOS creates interposing *wrapper functions* whose job it is to first log information about the call to the routine, then pass the call on to its original destination. The logging is completely hidden from the application; the only operations needed to add it to the system are the Jigsaw operations applied to the module namespace. Interposition is also very handy for pointer "swizzling," i.e. converting OID's to virtual addresses [Mos92].

Function interposition (by various names) is a familiar construct in O-O programming. There, it is common for an overriding method to refer to the method being overridden. For example, Smalltalk provides access to the overridden method through the pseudo-variable super. This feature is semantically identical to function interposition.

By our analysis, interposition is beyond the capability of conventional linkers since it fundamentally relies on renaming. Within OMOS, the general case of interposition can be achieved by application of the Jigsaw operations copy-as, override, and hide. To illustrate, suppose we wish all invocations of a function f to be redirected to a new definition of f using (in the general case) both its own definition, which it refers to as f, and the existing definition of f, which it refers to as f_{old}. Specifically, suppose the existing (loaded) module is

$$\mathcal{O} = \{\ \mathsf{g} = \alpha(\mathsf{f},\ \mathsf{g}),\ \mathsf{f} = \beta(\mathsf{f},\ \mathsf{g})\ \}$$

1. **(Copy-as)** $\mathcal{O}_1 = \mathcal{O}$ copy f as f_{old}

   Result: $\mathcal{O}_1 = \{\ \mathsf{g} = \alpha(\mathsf{f},\ \mathsf{g}),\ \mathsf{f} = \beta(\mathsf{f},\ \mathsf{g}),\ \mathsf{f}_{old} = \beta(\mathsf{f},\ \mathsf{g})\ \}$

2. **(Override*)** $\mathcal{O}_2 = \mathcal{O}_1$ override $\mathcal{O}'$

   Result: $\mathcal{O}_2 = \{\ \mathsf{g} = \alpha(\mathsf{f},\ \mathsf{g}),\ \mathsf{f} = \gamma(\mathsf{f}_{old},\ \mathsf{f}),\ \mathsf{f}_{old} = \beta(\mathsf{f},\ \mathsf{g})\ \}$

3. **(Hide)** $\mathcal{O}_{interpose} = \mathcal{O}_2$ hide f_{old}

   Result: $\mathcal{O}_2$ with f_{old} removed from its interface, as desired.

Figure 4: Functional interposition via Jigsaw operations

and the module containing the interposing f is

$$\mathcal{O}' = \{\ \mathsf{f} = \gamma(\mathsf{f}_{old},\ \mathsf{f})\ \}$$

where for brevity we omit the declaration of f_{old}. The desired interposition result is

$$\mathcal{O}_{interpose} = \{\ \mathsf{g} = \alpha(\mathsf{f},\ \mathsf{g}),\ \mathsf{f}_{old} = \beta(\mathsf{f},\ \mathsf{g}),\ \mathsf{f} = \gamma(\mathsf{f}_{old},\ \mathsf{f})\ \}$$

where f_{old} is hidden (removed from the interface of $\mathcal{O}_{interpose}$). We of course wish to preserve the original interface to f, so that subsequently merge'd modules see the interposed f, and, indeed, cascaded interposition works as expected. $\mathcal{O}_{interpose}$ can be obtained by the Jigsaw operator sequence shown in Fig. 4.

6.2 Dynamic Linking and Loading

As suggested in §6, dynamic linking and loading can be viewed as module manipulation where one operand is already loaded and undergoing execution. This variation poses pragmatic considerations in the use of Jigsaw module operations beyond the static case described in §6.1. A major concern is whether symbols to be rebound in the executing module have references in *data segments* or *code segments* (*text* in Unix jargon). Due to limitations on execution state modification, including instruction cache clearing, it is impractical to modify symbol bindings in text segments. This restriction is accommodated cleanly by viewing symbols referenced from the executing module's text segments as having been subjected to Jigsaw's freeze operation (see Fig. 1). This renders their bindings *read-only*, but does not hide them, so references in the dynamically loaded module can be bound to them.

In the simplest case, dynamic linking and loading involves a run-time merge operation between loaded module and the module being loaded. More sophisticated effects can be obtained through more powerful Jigsaw operators such as override, and compound operations such as interposition. Frozen symbols can be manipulated through operations that only affect the loading module's symbolic interface (e.g. hide and rename-to). Operations that cause binding or rebinding of symbols, such as restrict, override and merge, may not be applied if the binding of a frozen symbol would be affected.

The asymmetry between execution of Jigsaw operations on frozen vs. unfrozen bindings can be thought of as a matter of "views." A module with frozen bindings cannot change how it views the world, since it cannot change bindings to the interfaces it references. It can, however, experience a change in how it is viewed from the outside, since the names and values of the symbols (interfaces) it exports can be changed. Modules with unfrozen bindings modules (such as a module being dynamically loaded into a running program) can change either how they view the world or how they are viewed by the world.

It is only possible for a program to change the definition of a class on the fly (as in the case of class evolution, see §10.1) if all changes are made through unfrozen bindings, such as those residing in data segments. C++ systems generally allocate virtual function tables in data segments; hence references to symbols denoting virtual functions may be rebound. In particular, the starred step in the interposition example of Fig. 4 (involving override) cannot be performed unless the references being rebound are in data segments. Thus we can expect that dynamic interposition of C++ functions only to be possible on virtual functions. Note that a rebindable symbol may have multiple occurrences (e.g. entries in several virtual function tables). The state-saving capabilities of OMOS accommodate this fact by saving relocation information sufficient to locate and rebind all data segment references to a symbol.

7 Generating Dossiers

7.1 Dossiers for Application Classes

We now turn to the question of how dossier objects are created within an O-O software development process. Recall that the MSO OM's primary function is to provide a persistent store for C++ and CLOS application program objects without invalidating existing compilers, programming envrionment tools (e.g. make) and file systems. The scheme for creating dossier objects must fit within these constraints.

In the development of a C++ class, the specification for that class is finalized when the implementation of the class is compiled. At this stage, in current C++ development environments, the implementation for that class is then fixed. Our dossier objects are created during this stage of the development process. Dossiers may be generated by either a preprocessor or directly by a modified compiler. Linton et al. used a preprocessor. We describe here how a modified C++ compiler may be used for these purposes. As the compiler processes an input file, it builds descriptions of, among other things, the specification and layout of types defined therein. These descriptions contain all the information necessary for the creation of the dossier objects. After parsing a class definition, the dossier for that class is emitted by the compiler as static data to be linked into client applications.

As stated in §2, run-time class descriptions are necessary for fully polymorphic manipulation of objects. Consequently, we offer access to the dossier for a particular object at run-time. This access is provided through the virtual function dispatch table (vtable) for the class. Each vtable is expanded to include a reference to the appropriate dossier. Classes without virtual functions, and thus no vtable, are modified to have an empty vtable; this vtable is then expanded to include a reference to the dossier for that class. The inclusion of a vtable pointer within a class instance may violate C compatibility — a C++ class without virtual functions no longer has a memory layout compatible with a identically declared C structure. Although this incompatibility may be significant for some existing applications, thus far it has not proved to be a problem.

Loading an object is a two step process. First, the member data associated with an object is loaded into the memory space of an application. Then the object must be *configured* for use within that particular application's address space. Specifically, the virtual function dispatch table and virtual base class pointers must be initialized. To simplify this stage of reconstructing an object, special configuration functions are generated by the modified compiler. A configuration function is generated for each class, and is emitted along with the dossier for that class.

7.2 Meta-Dossiers

In §5.2 we describe how the OID of a meta-dossier can be used as a key to ensure execution environment compatibility between the creator and users of a persistent object. We now briefly sketch how we envision using an object's meta-dossier to govern its adaptation to varying execution environments. Again, the module ADT viewpoint plays a fundamental role.

As a persistent object (rather than simply an OID-based key), a meta-dossier summarizes the conventions observed in compiling the class implementations it describes. This will contain:

- Certain literal attributes, e.g. compiler = "GNU C++ version 2.2.2", processor = "HP 730", endian = "big", alignment = "word", etc.

- An OID referring to an OMOS module with a fixed set of object and class conversion functions (compiled, of course, under this execution environment). For example, there could be a function build_vtbl(d), which constructs a virtual function table consistent with its execution environment, given a dossier d (and d's meta-dossier) constructed under a different environment. In a sense, these functions would be meta-dossier driven generalizations of compiler-specific actions such as the configuration function described in §7.1. If relatively few execution environments are supported, and the set evolves slowly (as we expect), then these functions can exploit a hand-written *ad hoc* solution for each case.

8 Implementation Status

8.1 Dossiers for Application Classes

The compiler used in this experiment, the GNU C++ compiler version 2.2.2, is distributed with a partially implemented facility for generating type descriptors and providing them at run-time (for garbage collection). We have modified and extended this facility to provide the functionality described in §7.1. The dossiers produced are based on the object layout and parse trees generated during compilation. The compiler produces dossiers for all classes, including those that use multiple inheritance and virtual base classes. It also produces dossiers for classes defined using templates. Currently the dossiers created by this compiler are both compiler and platform specific. The information they contain includes what we have described for the dossiers and meta-dossiers, together in one object.

This modified compiler has been used to build the InterViews [LCV87] library. The InterViews library, with associated applications, contains over 45,000 lines of C++ code and makes extensive use of class inheritance and virtual functions. A text editor distributed with the library was also built with the modified compiler, and this application ran without error. The success of this regression test demonstrates that our scheme for generating and accessing type information at run-time is applicable to large, multi-file applications with libraries, and is compatible with existing operating system and X-Window libraries. The addition of virtual function table pointers to objects has, thus far, caused no compatibility problems with existing code or libraries.

8.2 OMOS

OMOS has been implemented as an operating system server using a set of C++ objects. An installation of OMOS exists on the Intel 386 platform running under the Mach operating system. Operating system dependent features are isolated, thereby easing the port of OMOS to any system with modern virtual memory and inter-process communication facilities. Recently, the object file manipulation aspects of OMOS (the other major porting obstacle) have been recast using the BFD object file back-end from the Free Software Foundation. BFD provides a high degree of object file format independence, which should greatly facilitate porting to other platforms.

OMOS provides module storage, linking, and loading facilities. The OMOS constraint system permits automatic sharing of modules between clients (à la shared libraries). In addition, OMOS provides facilities for automatic monitoring of program execution, and transparent reordering of program binaries to improve program paging behavior. A file-based version of OMOS (known as OFE) exists which acts as a superset of the Unix system linker, ld. OFE can be used to manually manipulate object files using the Jigsaw operation suite, as well as to manually reorder program binaries.

9 Semantic Issues

Storing and retrieving class implementations as modules opens many opportunities for broading POS utility. However, it also poses vexing issues that reveal subtle difficulties in adding persistence to O-O languages designed for manipulating transient objects.

1. What should be the role of user-defined constructors when an object is loaded from the POS? The only general answer is "none", since the retrieved object is not being constructed (at least not from whole cloth). In any case, given overloaded class constructors, it is impossible to determine which one might be appropriate to execute. Yet, existing user code may rely on constructors to maintain application specific invariants when an object "comes into existence".

2. The static data members of a class clearly need to exist whenever an object of that class is loaded. But when are they created and initialized, and are they persistent? The latter issue is especially troublesome when static member data is mutable. Such data could reside within the dossier object, but again, when should they be constructed? — when the class is registered?

What about multiple implementations of the class for differing execution environments — should the static data be shared or separate?

3. How late should the binding be between class names and class implementations? We suspect that one implementation per class per execution environment is much too limiting, even if load time specialization (e.g. interposition) is supported. Instead, a systematic approach to managing class versions and consistent class libraries should be devised.

10 Continuing Work and Longer Term Implications

10.1 Continuing Work

As outlined in §8, a basic implementation of OMOS, and a dossier-creating C++ compiler, already exist. Their integration with the basic object store remains to be done. In addition, class implementation registration and meta-dossiers are only in the design stage.

Another dimension of the MSO OM still under development is support for persistent CLOS objects, and multi-lingual objects. We believe the module ADT viewpoint will again permit flexible and highly appropriate abstractions, but this remains to be established.

We hope to extend the MSO OM to support three advanced effects:

1. *Multiple class implementations:* As remarked above, we believe multiple class implementations to be an important POS feature. Architecturally, there is no problem having a retrieved object can bring in its own implementation — indeed, that is what happens in the standard dynamic loading case. However, some means must be found for an already loaded implementation of that class not to become invalidated. As a simple avenue, we are considering extending the role of meta-dossiers to group logically consistent collections of class implementations, in the same sense that they currently group class implementations that share an execution environment. This turns the name server described in §5.2 into a rudimentary class version manager.

2. *Class evolution:* This is a generalization of multiple class implementations, whereby data and function members are added to a class definition, presumably in a monotonic manner [GS87]. The technical complications of this are myriad, and generally compiler specific. However, we hope to accommodate this in simple cases, perhaps as a generalization of multiple class

implementations.

3. *Object promotion:* Here an existing object is summarily converted to belong to a different class, typically a subclass of its original class. In effect, this is retroactive inheritance, on a per-object basis. Again, we believe the necessary technical devices exist within our POS architecture, but semantic and compiler-specific pitfalls abound — for example, do we execute a constructor for the new class?

10.2 Longer Term Implications

In this section we consider two directions in which this work could proceed in the longer term. The first direction concerns factoring the module ADT implementation out of OMOS, and the second deals with linguistic extensions made possible by our system.

The notion of an abstract data type representing modules pervades this paper. We have argued that such an abstraction arises naturally in the design of a POS, and we have incorporated it in our design and implementation. In our case, the ADT is implemented by OMOS. Yet the module ADT is a useful concept independent of persistent stores. Ideally, we should implement the ADT as a separate entity.

Specifically, we suggest implementing the module ADT as a *framework* in Johnson's sense [JR91]. In such a framework, notions such as modules and instances would be represented by abstract classes. The framework would likely provide concrete subclasses representing particular realizations of these notions. An application like OMOS would then extend or modify the framework to suit its particular needs. For example, the address loading constraints employed by OMOS are specific to linking, and would be added as an extension of the basic module ADT.

There are other valuable uses for such a framework. It may serve as a basis for the implementation of a family of interoperable language processors that share a common notion of object. This is not unlike the use of standard calling sequences to help ensure interoperability of procedural languages. Of course, a standard format for objects could be used by unrelated implementations as well. The advantage of the framework is that it implements the standard in a manner accessible to various compilers, saving implementation effort and encouraging adherence to the standard.

Another direction is to extend the semantics of C++. Jigsaw operators allow for richer behaviors than those of classes in C++. One example is the introduction of mixins [BC90].

The facilities introduced for *dynamically* applying Jigsaw operators allow inheritance at run time. These facilities could be employed by the modified C++ compiler itself, to support delegation. The difficult part here is deciding exactly how such features should be incorporated linguistically into C++; as described here, we already have an applicable implementation.

11 Related Work

This project builds on work ongoing elsewhere in many related areas. We briefly mention only a few here.

- *Basic object stores:* Several language-neutral object stores have been described, including ESM [CDRS89], Cricket [SZ90] and Mneme [Mos90].
- *C++ based OODB's:* Two well-known examples are ObjectStore [LLOW91] and ODE [AG89].
- *OODB toolkits:* Here OODB extensibility is the primary objective — examples include EXODUS [CDG$^+$90] and ObServer [HZ87].
- *Persistent language extensions:* The earliest persistent higher level language was PS-Algol [A$^+$82]. Recent persistent Lisp systems include PCLOS [Pae88] and MetaStore [Lee92].
- *Multi-language object systems:* The RPDE3 [HO92] system represents important new work in multi-lingual objects.
- *Dynamic extensions to C++* Several papers have exploited dynamic loading to stretch the bounds of C++ static type checking, e.g. [DSS90] and [GS87].
- *Reflective systems:* Class descriptions and related meta-representational issues are explored in Smalltalk80 [GR83], CLOS [KdRB91], and 3-Lisp, a fully reflective language [Smi82].

Acknowledgements

We acknowledge the many contributions to this work by members of the MSO OM project, including Robert Mecklenburg, Mark Swanson, Robert Kessler, Jay Lepreau, Guru Banavar, and Peter Hoogenboom.

References

[A+82] Malcom P. Atkinson et al. PS-Algol: An Algol with a persistent heap. *ACM SIGPLAN Notices*, pages 24–30, July 1982.

[ABD+92] Malcolm Atkinson, Françcois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Building an Object-Oriented Database System*, chapter 1, pages 3–20. Morgan Kaufmann, 1992.

[AG89] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment: The language and data model. In *Proc. Int'l. Conf. on Management of Data*, pages 36–45, Portland, Oregon, May-June 1989. ACM-SIGMOD.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. OOPSLA Conference*, Ottawa, October 1990. ACM.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20-23 1992. IEEE Computer Society.

[Bra92] Gilad Bracha. *The Programming Language* Jigsaw*: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, March 1992. 143 pp.

[CDG+90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS extensible DBMS project: An overview. In *Readings in Object-Oriented Databases.* Morgan-Kaufman, 1990.

[CDRS89] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage management for objects in EXODUS. In *Object-Oriented Concepts, Databases, and Applications*, chapter 14, pages 341–369. Addison-Wesley, 1989.

[DSS90] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding new code to a running C++ program. In *USENIX Proceedings C++ Conference*, pages 279–292. USENIX Association, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[GS87] Philippe Gautron and Marc Shapiro. Two extensions to C++: A dynamic link editor and inner data. In *USENIX Proceedings and Additional Papers C++ Workshop*, pages 23–34. USENIX Association, 1987.

[HO92] William Harrison and Harold Ossher. Attaching instance variables to method realizations instead of classes. In *Proc. International Conference on Computer Languages*, pages 291–299, San Francisco, CA, April 20-23 1992. IEEE Computer Society.

[HZ87] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM TOIS*, 5(1):70–85, January 1987.

[IL90] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *USENIX Proceedings C++ Conference*, pages 233–240. USENIX Association, 1990.

[JR91] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91-1696, University of Illinois at Urbana-Champagne, May 1991.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[LCV87] Mark A. Linton, Paul R. Calder, and John M. Vlissides. InterViews: A C++ graphical interface toolkit. In *Proceedings of the USENIX C++ Workshop*, page 11 pp., Santa Fe, NM, November 1987.

[Lee92] Art Lee. *The Persistent Object System MetaStore: Persistence Via Metaprogramming.* PhD thesis, University of Utah, June 1992. 171 pp.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[Mos90] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, 1990.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[OM92] Douglas B. Orr and Robert W. Mecklenburg. OMOS — an object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society.

[OMHL93] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the Hawaii International Conference on System Sciences*, page 10pp., January 1993.

[Pae88] Andreas Paepcke. PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Berlin, 1988. Springer-Verlag.

[See90] Donn Seeley. Shared libraries as objects. In *Proc. USENIX Summer Conference*, Anaheim, CA, June 1990.

[Smi82] B. Smith. Reflection and semantics in a procedural language. Laboratory for Computer Science TR-272, MIT, 1982.

[SZ90] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, MA, August 1990. U. Wisc. Tech. Rpt. 956.

Last revised December 16, 1992