Reusable Objects
David Morley, Stephen Chiu, Jason Robbins,
Tim Maddux, and Geoffrey Voelker

Submitted to
Technology of Object-Oriented Languages and Systems
International Conference & Exhibition
March 4-8, 1991

CNIT Paris
Reusable Objects
David Morley, Stephen Chiu, Jason Robbins,
Tim Maddux, and Geoffrey Voelker

Abstract

Object-oriented systems promote reuse of code, but no object-oriented system adequately supports management of the object graph from which reusable objects must ultimately be made. We describe an approach to object graph understanding, organization, and management through grouping objects into “object complexes” for a variety of purposes, foremost of which are object persistence and reuse. These reusable objects are large scale "pluggable" components which embody both state and behavior. We also describe the use of this approach in a large Smalltalk system which is being made into a product.

Keywords: Deep Copy, Object Reuse, Object-Oriented Database, Persistent Objects, Megaprogramming

Technical Problem

Object-oriented software enthusiasts usually tout “reusability” of code (through inheritance) among its most attractive features. But the reusability of objects is extremely problematic. Object oriented database proponents speak too glibly about storing and managing "objects" but fail to address the most difficult problem of making them reusable in any setting other than the one in which they are imbedded. The hierarchical data structures which these database systems support do not, by themselves, constitute practical reusable objects1.

In an object-oriented system, most individual objects are not very reusable by themselves but are reusable only as part of a larger “complex” of objects which comprise the programmer’s idea of the object being modelled. Individual objects are usually just a bundle of pointers to other objects. Most of the potentially reusable information (and behavior) is contained in the structure of the larger complex, not just in the atomic data at the “leaves” of the complex. An example will help clarify some of the issues and the forthcoming discussion.

Suppose we have an object graph which represents a car, and suppose that every part of the car is reachable from the root, called aCar. A simplified version of the object graph might look like Figure 1a. (In a real situation the object graph probably has many thousands of objects, and is rarely interesting to view at a low level as shown in the figure). One of the objects is called theEngine. The engine object is almost certainly of little interest by itself, being simply a bundle of pointers. But the engine object is probably the “root” of a larger complex of objects whose entirety would both functionally and conceptually be the engine “object”. In figure 1b we’ve draw a line around that larger complex. There are many situations in which we would want to treat this object complex independent of the car into which it is threaded. For example, we might want to reuse the engine in another car, or even in a completely different context like a boat or test bench. Since the engine consists of a complex of objects we will speak of the “engine object complex” or the “engine complex”.

Figure 1b illustrates an object complex conceptually, but there is no information in the object graph which describes this. The single engine object may define the root of an intended complex, but contains no information about the boundary of that complex. The

1. See the Composite Object Approach section in the sequel.
object graph is completely undifferentiated with respect to what constitutes important or central objects in the overall design. Even though the programmer imposes conceptual structure on his object graph, the system in no way supports it. Such support would be useful for deep copy, storing to disk, clustering, locking, versioning, and reuse.

Figure 1. The object graph contains theEngine object as the "root" of a "complex" of objects which (collectively) describe the programmer's conceptual idea of the engine (a). But theEngine object contains no explicit information regarding the boundary of that complex. In (b) we draw a line around the conceptual engine complex.

In the above example, we may want to reuse the engine complex in another car, or store it away in a database of engines. To do either of these things it is sufficient to be able to make a copy of the engine complex; we require a "deep" copy which starts at the root of the engine and terminates at its conceptual boundary. The technical problem which we address in this paper is essentially this deep copy problem and the attending problems of reuse of such copies: how can we specify the boundary, how can we copy everything up to that boundary, and how do we reconnect such an object complex into another system. A solution to this problem will realize the important goal of defining and managing reusable objects.

Such reusable objects (the copies mentioned above) may well reside in a database for organization and management purposes, and we will briefly discuss our plans for this; but it is important to distinguish this work from more conventional "object oriented database" topics such as [Kim 87][Dayal 87][Zdonik 90]. In our view, the conventional object-oriented database is essentially only a virtual memory system through which a large object graph is accessible. Such database systems do not adequately address the difficult issues of definition or management of reusable objects. The techniques in this paper should contribute significantly to that shortcoming of object oriented databases and to object oriented systems in general.

**Deep Copy**

We've been working in object oriented technology for the past 6 years at the Rockwell Science Center. Our larger applications have included robotics [Chiu 86], parallel processing [Morley 86], and currently an environment for factory automation. Deep copy has been a major problem in each of these projects, in a variety of guises. Today we face
very large and complicated Smalltalk object graphs which are rich descriptions of factory automation systems; each of these object graphs relates such things as hardware configurations, software modules, process emulators, networks, simulations, wiring diagrams, and even order forms for the required components. Until recently, these object graphs could be made persistent only in total; none of the components could be reliably cleaved away from the larger system for reuse, except through very ad hoc methods which were difficult to maintain.

Recently we have addressed the basic deep copy problem in a systematic way. But first, the alternative approaches.

The ad-hoc Approach

Our first few encounters with the deep copy problem led us along a path of seemingly least resistance. We have found many other object-oriented programmers also using this basic ad-hoc approach. Applied to the above example, this approach is roughly as follows.

Naively, we implemented a deepCopy method recursively at a high level of abstraction, letting each of the types which compose theEngine complex inherit that recursive method, and then specialize deepCopy at the boundary objects. In Object (the top of a single-inheritance tree of class definitions), we define the instance method

```
deepCopy
  If self has been touched by this method then return
  that previous copy.
  Assign newCopy to be shallowCopy of self.
  Assign all instance variables of newCopy to be
  deepCopies of all instance vars of self.
  Return newCopy.
```

Then we specialize deepCopy in classes whose instances occur at the intended boundary:

```
deepCopy
  Return EdgeObject newFor:self.
```

(The EdgeObjects for these boundary objects help us reconnect the copy of the engine complex into another Car, or other suitable complex of objects. An EdgeObject can be thought of as a kind of proxy for the object which was formerly on the boundary of the complex.)

There are several problems with this ad-hoc approach. First, class-based specializations are too coarse, because objects on both the interior and the edge of a complex may be of the same class. Also, since classes are reused in different contexts and different applications, these methods will collide in classes which are common boundaries for different contexts. And perhaps worst of all, the mechanism for describing a conceptual entity is spread all over the class lattice.

Code scattered all over the system like this is inherently brittle and hard to understand and manage. Indeed, the deepCopy facilities of our early systems were almost always in disrepair and difficult to fix. Waving our hands, we would say, "This is only a prototype. We'll fix this up if and when this becomes a product." That time has now come for our latest prototype, and the situation has not improved. See the Practical Experience section in the sequel.
The Composite Object Approach

Much of the object-oriented database literature addresses the problem of grouping objects for a variety of purposes, including reuse and clustering. Most approaches use various notions of “composite objects”, which describe object data structures as hierarchical. For us, the terms “complex object” and “composite objects” designate this approach.

One example of this approach is the LISP database system ORION [Kim 87][Kim 88] which specifies an “is-part-of” hierarchy in the class definitions by differentiating between five types of references between objects. This hierarchy essentially describes the shape of the object graph, at least for objects in the persistent state. But object oriented data structures defy a hierarchical mindset. A good object oriented programmer tries to define a data structure which is nearly isomorphic to his conceptual idea of the thing being modelled. Things in the real world are related to one another in complicated ways which are rarely hierarchical. Hierarchical data abstractions are a throwback to conventional systems which lack the power to manage a rich graph of objects. What programmers need in an object oriented database is the same modelling power which the object oriented languages enjoy. Unfortunately, the rich data structures found in OOPs have been difficult to make persistent in any reusable way because of their highly coupled nature.

Even in the case where non-component references can serve to cleave the composite object apart from a larger object graph, once again, the description of the boundary of a composite object is far removed from its root, causing the definition of a composite object to be spread throughout the class lattice. Furthermore, nothing in the object database literature really addresses how composite objects might be reconnected into different systems of objects.

Rumbaugh's Approach

[Rumbaugh 88] suggests maintaining relations between objects to control propagation of deep methods. Attributes of these relations can specify a variety of behaviors along the arcs of the object graph. This approach is attractive for many aspects of managing the object graph. Attributes are distributed in the object graph by a class-based mechanism which is dynamic in contrast to [Kim 87]. However, the prospect of an object on every relation, presumably monitoring every message, is daunting to say the least. Our approach will be seen to offer an opportunity to distribute such relations (and associated attributes) strategically in the object graph only at the desired locations, where they can be used in the same spirit as Rumbaugh's relations. Furthermore, our relations need only reside transiently in the object graph.

The description of an object complex so defined is spread throughout the object graph, instead of throughout the type or class lattice as in the previous alternatives, and is thereby still seemingly difficult to understand and manage.
Our Approach

Our goal is to decompose an otherwise undifferentiated object graph into complexes of objects that are naturally grouped by the programmer's conceptual model, and to describe these complexes and their interrelationships to form a high level abstraction of the object graph. See Figure 2. In this way, object complexes will be easily identifiable, easy to copy, and easy to recombine in other configurations; we will thereby achieve powerful reusability of object complexes.

![Diagram](image)

Figure 2. Our goal is to decompose the object graph (a) into "reusable objects" with well-defined interfaces (b). This will provide a high level description of the object

Our general approach is to define an object complex in a single object, to describe its interface for reuse in other contexts, and to build a high level database to organize and manage such "reusable objects".

Defining Object Complexes

Our approach is to localize the definition of an object complex in an object called a RecursionArbiter. A RecursionArbiter appears as an argument to a recursive method, deepTraverse:aRecArb (defined for Object), which traverses the object graph under its guidance. The RecursionArbiter detects boundary objects and public objects of the complex, and creates an auxiliary data structure (an Object Complex Descriptor, or OCD) which describes the complex for subsequent use. It thereby creates an intentional interface for the object complex and encapsulates its "implementation" (its objects), qualities which are crucial for effective reuse. See Figure 3. The RecursionArbiter and OCD are independent of the object graph, making them completely dynamic and allowing multiple, even simultaneous, organizations of the object graph.

---

2. The RecursionArbiter as an argument to deepTraverse is similar to [Bobrow 86]'s use of an object map as an argument to deepCopy. However, to define and manage an entire map of the (presumably dynamic) object complex would be far more difficult than what must be described for the RecursionArbiter. A programmer cannot be required to know the shape of a large object complex. See the Practical Experience section for why RecursionArbiters are easy to build without complete understanding of the object complex which it defines.

3. A public object is one which is pointed to by objects outside the desired complex. It is the receiver of pointers into the complex.
Figure 3. A Recursion Arbiter is programmed to detect public objects (inputs) and edges (outputs) of a complex. The "root" of a complex is a special kind of public object. An edge actually describes the pointer which crosses the boundary of the complex, so two dashed lines point from the edge to each end of that pointer. During traversal of the object graph, it creates an object complex descriptor (OCD) which describes the complex "from above". The underlying object graph is not altered.

The generic Recursion Arbiter responds to a number of directives, shown in Figure 4. A Recursion Arbiter for a specific kind of object complex can be subclassed from Recursion Arbiter and then specialized in the "initialize" method. For example, a Recursion Arbiter For Engine might define its initialize method as follows:

Recursion Arbiter For Engine <subclass of> Recursion Arbiter

initialize
  atClass: GasLine makePublic: #theGasLine
  atClass: Exhaust makePeerEdge: #myExhaust

When deepTraverse: (Recursion Arbiter For Engine new initialize) is sent to the Engine object, the graph will be traversed deeply, stopping and making an edge at any object of class Exhaust, and making any object of class GasLine into a public object. These edges and public objects will be maintained in an OCD for the complex so defined. More refined control can be achieved by supplying blocks of code to be evaluated in the context of the object being visited by deepTraverse:. (see the ifTrue: directives in Figure 4.)

An OCD describes the boundary of its object complex by maintaining public objects and edges (for boundary objects). Initially, an OCD is not part of the object graph being decomposed, but has pointers into that object graph as shown in Figure 3. However, OCDs respond to the message insertEdges which redirects pointers as shown in Figure 5. There are several applications for the capability of inserting edges into the object graph in this way, such as monitoring messages, debugging, virtual memory, translating messages to alternate forms, implementing the several kinds of relations of [Rumbaugh 88], etc. We have in mind a great variety of Edge and Public Object classes which implement these kinds of behaviors as well as "smart" edges which know how to reconnect into other complexes, etc. However, we concentrate on deep copy in this paper. It might be noted in the sequel that only the edges need to be inserted to perform the deepCopy, but we insert the public objects as well to support future uses.
Figure 4. The generic RecursionArbiter implements several directives (as methods) such as these. On receipt of one of these messages, a variety of control structures are created which are then used during deepTraverse: to detect the public objects and edges of the intended complex. The definition of a RecursionArbiter is completely dynamic and new directives may be issued even during execution of deepTraverse:

![Diagram](image)

Figure 5. An OCD responds to the message insertEdges which causes the public objects and edges to be inserted into the underlying object graph. The direct pointers into and out of the complex have been broken (illustrated by a broken arrow) and the edges and public objects have become part of the object graph (illustrated by solid arrows).

In order to copy the complex described by the OCD, we implement a relentlessly deep deepCopy method in Object and override deepCopy in Edge to terminate the recursion. For purposes of reuse, we also copy the OCD which describes the new complex as shown in Figure 6. For example, the new edges and public objects which are now part of the new object complex contain information which will assist in reconnecting the complex into another object graph.

After the copy has been made, the original OCD can be sent the extractEdges message to disengage itself from the object graph once again. The new OCD and object complex constitute a “reusable object” The OCD provides a “handle” to the underlying object complex, and can respond to a variety of messages, some of which will be described in the next section.
Currently we are developing another type of edge, one which represents a "shared" object. The engine complex might contain a pointer to its manufacturer, which we probably wouldn't want to copy yet not cleave away either. Such an object would be shared by both copies of the engine complex. Maintenance of shared objects requires database support which will be developed in a future paper.

This approach to deep copy is attractive because, as mentioned above, it localizes the definition and manipulation of an object complex. Deep copy is only one of the applications of this general technique of propagation control.

**Reusable Objects**

The copied OCD and its independent complex form what we call a reusable object. The interface to this reusable object is contained explicitly in the OCD, obviating the need for the programmer to understand the internal details (potentially very complicated) of the object complex itself. A reusable object therefore encapsulates the implementation of the underlying complex and presents a well-defined interface via its OCD. There is considerable opportunity here to provide interesting behaviors for reusable objects, both generically and specially by subclass.

Currently we have implemented only a small number of behaviors for reusable objects, all implemented by the OCD. The methods insertEdges, extractEdges, and copy were discussed in the previous section. Other examples are straightforward methods for storing and retrieving reusable objects to and from files. Eventually, database behaviors will also be implemented. Also, as mentioned briefly above, the RecursionArbiter associates a symbol with each public object and edge which it creates (see Figure 7), and the OCD responds to the methods

```
setEdge:aSymbol to:anObject   and
getPublicObj:aSymbol
```

which make the interconnection process a series of symbolic operations.
Figure 7. A symbol is associated with each public object and edge of the reusable object. This helps the programmer wire it up to other reusable objects or other parts of his system.

The interface so described suggests that OCDs may also play a design role for object oriented systems. The object complex which it describes can be thought of as one implementation of any number of complexes which might have the same interface. In the next section we will speak of OCDs being "bound" to specific object complexes, or "unbound" if they are serving only as a specification for an object complex implementation.

Reusing Objects

Reusable objects are truly "pluggable" components. Although they may not fully operate independently, they encapsulate much larger chunks of reusability than simple class descriptions. We envision the creation of many thousands of them, of many different types, organized in a database out of which application developers can more easily build large object graphs for their models. Although our current factory automation environment depends on the application to store, manage, and relink reusable objects as described to this point, we briefly describe our vision for the near future in this section.

A reusable object database will consist of three kinds of information structures: Reusable Objects, Maps, and Bindings. Maps are essentially design elements which describe, at a high level of abstraction, how components come together to form a larger system. See Figure 8. Each of the components in a map is essentially a specification of what is required of a reusable object which can be "bound" to it. For example, a reusable engine object complex must have inputs (public objects) and outputs (edges) as labelled in Figure 7. Of course, the specification should also describe the kinds of messages that will be generated by the outputs and those that must be serviced by the inputs. Other opportunities exist for "intelligence" at this level of abstraction, such as "link hints" which are methods of edges (or public objects) that respond to queries by public objects (or edges) regarding their interface characteristics. With such constructs, some linking of object complexes could be automatic, or could help the system designer.
Figure 8. A map is a design element which describes at a high level how reusable objects can be combined to generate a larger component. The OCDs in such a design would be unbound to any particular reusable object.

Given a high level design in the form of a map, there may be many combinations of reusable objects which can "instantiate" such a design. These instantiations will be given in the form of a "binding" which will bind each OCD in the design to a particular reusable object in the database. See Figure 9. When a particular binding is brought from the database into an application, the reusable objects will be wired together to form an object graph of conventional form4.

Figure 9. A database of reusable objects will store bindings which relate maps to reusable objects.

---

4. Actually, this scheme is a bit simplistic for the long term. Bindings should be reusable objects themselves. "Incomplete" bindings could be used to partially connect "atomic" reusable objects to form larger subsystems which have an interface consisting of those edges and public objects which the binding could "promote". In this way, nested designs could be created. This will be treated in more detail in a future paper.
Practical Experience

As mentioned in the introduction we are involved in implementing a factory automation system. This system was initially started two years ago as an experimental prototype; it has become very large and complex. Interest has recently grown in making this system a product, but it has several problems, foremost among which is the intricate interconnections within the object graph. Until recently, large designs which can be made with this system were difficult to reuse or even to make persistent. Sub-parts of the design were impossible to extract from the larger designs even for simple "cut and pastes" within the system. The ad hoc code that was implemented in an effort to accomplish these goals ultimately became the most brittle part of the system, and was in utter disarray.

Recently we implemented the ideas presented here for this "real" system with remarkable success. The RecursionArbiters have been much easier to write than any of us imagined (although need for a debugger became obvious and certain directives for collections should be implemented), and the interfaces provided by the OCDs transformed a seemingly hopeless situation into a relatively easy problem. Another striking aspect about this whole experience is the flexibility of decomposing the object graph by these mechanisms: repeatedly, as concepts or conditions have changed, the maintenance of the reusable objects has been very straightforward. Work is now going forward very rapidly in implementing cut & paste and persistent object facilities.

Why have the RecursionArbiters been so easy to develop? We have studied many object graphs in a variety of applications, and it turns out that the worst scenarios of object connectivity simply don't occur "in the large". What we find are object complexes that have dense interconnections, but rather sparse intraconnections. Furthermore, object complexes that occur in this fashion are rather natural to the application. This is to be expected because programmers cannot comprehend the entire object graph and tend to define manageable components of it which are fairly loosely coupled. On average, these natural complexes will have only 5 or 6 public objects and edges, and these points tend to be fairly well known to the programmer. Thus, it is natural to specify to the RecursionArbiter how to identify these public objects and edges as exceptions, rather than trying to describe everything within the complex as suggested in [Bobrow 86].

We have yet to fully implement in a practical setting the notion of maps and bindings as described above, but this work is forthcoming. Currently, reusable objects are maintained in files and the application environment is responsible for their interconnections.

Conclusion

The ability to decompose an object graph into complexes of objects allows us to understand and manage systems at a high level of abstraction. To the extent that the fine-grained object graph is isomorphic to the conceptual system being modelled, this high level abstraction is homomorphic to that system. This introduces a design paradigm in which systems can be assembled (and visualized) as large chunks of truly pluggable modules. Reusable object complexes are far easier to understand, manage, and manipulate than individual class definitions whose interconnectivity must be thoroughly understood in order to be used. As the complexity of systems increases, system developers will need ever larger reusable components to work with. Recent interest in large-grain modularity [Wegner 90] reflects this need.
References


